CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

DATE: February 26, 1982

TITLE: BASIC Version 3 Reference Manual

PUBLICATION NO.: 19983900

REVISION: H

REASON FOR CHANGE:

Revised at PSR level 552 to reflect BASIC 3.5, which supports NOS Version 2, and to clarify use of the RESTORE statement with the SET statement, use of a carriage return as a delimiter, and use of format 2 of the PRINT and PRINT USING statements when files are connected to the terminal. This revision also includes a new appendix on in-line editing commands, two new compile time diagnostics, and miscellaneous changes.

INSTRUCTIONS:

This revision supersedes all previous editions.

19983900

CONTROL DATA CORPORATION

# BASIC
# VERSION 3
# REFERENCE MANUAL

**CDC® OPERATING SYSTEMS:**
**NOS 1**
**NOS 2**
**NOS/BE 1**

# INDEX TO BASIC STATEMENTS AND FUNCTIONS

## STATEMENTS



## FUNCTIONS

19983900

CONTROL DATA CORPORATION

# BASIC
# VERSION 3
# REFERENCE MANUAL

**CDC® OPERATING SYSTEMS:**
**NOS 1**
**NOS 2**
**NOS/BE 1**

# REVISION RECORD

| Revision | Description |
|---|---|
| A (06/23/75) | Original printing. |
| B (11/05/75) | Includes corrections to revision A and user information pertaining to the Network Operating System/Batch Environment (NOS/BE Version 1.0). |
| C (02/15/76) | Includes minor editorial changes to revision B, plus modifications for the following new features: CHAIN Statement, user number function, file number 0, trace option, comments at end of source lines, positioning beyond bad input items, and improved field length management. |
| D (05/23/78) | Revised to include new features upgrading the products to BASIC Version 3.2, PSR level 472. These consist of the IF...THEN...ELSE statement and the capability to handle large strings. |
| E (11/10/78) | Revised to include new features upgrading the product to BASIC Version 3.3, PSR level 485. These consist of the RPT function and the ON ATTENTION statement. |
| F (07/20/79) | Revised to reflect BASIC 3.4. The changes and additions include substring addressing; CYBER Interactive Debug facility; eight new string functions (LPAD$, LTRM$, LWRC$, ORD, POS, RPAD$, RTRM$, and UPRC$); alphabetic characters in file name must be uppercase; CALL statement limitation with IF...THEN...ELSE; operating system terminology; and miscellaneous changes. This printing obsoletes all previous editions. |
| G (10/31/80) | Revised to conform to the American National Standard for Minimal BASIC (ANSI). Changes and additions include new statements OPTION and RANDOMIZE; subscript and index rounding; FOR...NEXT control variable value; handling of unquoted strings; new RND and DET function forms; default array base 0 (zero); formatting of large integers; new TAB features; ASCII default collating sequence; print comma spacing control; redimensioning result matrices; reading numeric data as string data; INPUT validation; other miscellaneous changes; and appendixes explaining guidelines for a possible CDC merge to ANSI standard BASIC and the difference between BASIC 3.4 (last revision) and BASIC 3.5 (this revision). Released at PSR level 528. This printing obsoletes all previous editions. |
| H (02/26/82) | Revised at PSR level 552 to reflect BASIC 3.5, which supports NOS Version 2, and to clarify use of the RESTORE statement with the SET statement, use of a carriage return as a delimiter, and use of format 2 of the PRINT and PRINT USING statements when files are connected to the terminal. This revision also includes a new appendix on in-line editing commands, two new compile time diagnostics, and miscellaneous changes. This printing obsoletes all previous editions. |

REVISION LETTERS I, O, Q, AND X ARE NOT USED



Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

# LIST OF EFFECTIVE PAGES

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

| Page | Revision |
|---|---|
| Front Cover | - |
| Inside Front Cover | H |
| Title Page | - |
| ii | H |
| iii/iv | H |
| v | H |
| vi | H |
| vii thru xii | H |
| xiii | H |
| 1-1 thru 1-17 | H |
| 2-1 thru 2-8 | H |
| 3-1 thru 3-5 | H |
| 4-1 thru 4-9 | H |
| 5-1 thru 5-14 | H |
| 6-1 thru 6-6 | H |
| 7-1 thru 7-26 | H |
| 8-1 thru 8-13 | H |
| 9-1 thru 9-7 | H |
| 10-1 thru 10-6 | H |
| 11-1 thru 11-5 | H |
| 12-1 thru 12-11 | H |
| A-1 thru A-4 | H |
| B-1 thru B-5 | H |
| B-6 | G |
| B-7 | H |
| B-8 thru B-11 | G |
| B-12 | H |
| C-1 | H |
| C-2 | H |
| D-1 thru D-5 | H |
| E-1 | H |
| E-2 | H |
| F-1 | H |
| F-2 | H |
| G-1 | H |
| H-1 thru H-3 | H |
| I-1 thru I-3 | H |
| Index-1 thru -4 | H |
| Comment Sheet | H |
| Mailer | - |
| Summary Card - Front | H |
| Summary Card - Back | H |
| Inside Back Cover | H |
| Back Cover | - |

# PREFACE

This manual describes the BASIC Version 3.5 language which operates under control of the following operating systems:

NOS 1 and NOS 2 for the CONTROL DATA® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

NOS/BE 1 for the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

Any reference to NOS refers to either the NOS 1 or NOS 2 operating system. In all instances where the two operating systems differ, NOS 1 or NOS 2 is specified.

CDC offers guidelines for the use of the software described in this manual. These guidelines appear in appendix E. Before using the software described in this manual, the reader is strongly urged to review the content of this appendix. The guidelines recommend use of this software in a manner that reduces the effort required to migrate application programs to future hardware or software systems.

BASIC 3 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. The CDC CYBER Interactive Debug (CID) facility can be used in interactive mode to debug a BASIC program.

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 similar to English 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 multiple-line and multiple-argument user-defined functions

Facility for calling BASIC and non-BASIC subroutines

Facility to chain to other BASIC programs

Matrix I/O for 1- and 2-dimensional numeric and string arrays

Output format determination, including various commercial formats

Manipulation of coded and binary files, including random access for binary files

Error detection and processing during program execution

Facility to trace program flow

Facility to debug a program (CYBER Interactive Debug)

This document is intended to describe these and other BASIC features to both the nonprogrammer and the experienced programmer. The information in this manual is provided in three major parts:

Section 1 is a primer or introduction to the BASIC language directed at the nonprogrammer. Appendix H contains sample BASIC programs.

Sections 2 through 12 include reference information that expands on section 1 information and is directed at the experienced programmer. Appendixes A through D and I support and summarize information in these sections.

Appendix E contains general feature use guidelines to ensure ease of migration to future hardware or software systems and appendix F contains an overview of the differences between this version of BASIC (BASIC 3.5) and the previous version (BASIC 3.4). Appendix G summarizes those features that are described in the American National Standard for Minimal BASIC as implementation-defined.

You can find additional pertinent information in the Control Data Corporation manuals. The NOS Manual Abstracts and the NOS/BE Manual Abstracts are pocket-sized manuals containing brief descriptions of the contents and intended audience of all NOS and NOS/BE manuals and all the product set manuals of these two systems. The abstracts manuals can be useful in determining which manuals are of greatest interest to a particular user. The Software Publications Release History serves as a guide in determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software.

The manuals are listed alphabetically in groupings that indicate relative importance to the readers of this manual.

The following manuals are of primary interest:

| Publication | Publication Number | NOS 1 | NOS 2 | NOS/BE |
|---|---|---|---|---|
| Network Products Interactive Facility Version 1 Reference Manual | 60455260 | X | | |
| Network Products Interactive Facility Version 1 User's Guide | 60455250 | X | | |
| NOS Version 1 Reference Manual, Volume 1 of 2 | 60435400 | X | | |
| NOS Version 2 Reference Set, Volume 3 of 4, System Commands | 60459680 | | X | |
| NOS/BE Version 1 Reference Manual | 60493800 | | | X |

The following manuals are of secondary interest:

| Publication | Publication Number | NOS 1 | NOS 2 | NOS/BE |
|---|---|---|---|---|
| CYBER Interactive Debug Version 1 Reference Manual | 60481400 | X | X | X |
| CYBER Loader Version 1 Reference Manual | 60429800 | X | X | |
| INTERCOM Version 5 Reference Manual | 60455010 | | | X |
| NOS Time-Sharing Version 1 User's Guide | 60436400 | X | | |
| NOS Time-Sharing Version 1 User's Reference Manual | 60435500 | X | | |
| NOS Version 1 Manual Abstracts | 84000420 | X | | |
| NOS Version 2 Manual Abstracts | 60485500 | | X | |
| NOS/BE Version 1 Manual Abstracts | 84000470 | | | X |
| Software Publications Release History | 60481000 | X | | X |
| Text Editor Reference Manual | 60436100 | X | X | |
| XEDIT Version 3 Reference Manual | 60455730 | X | X | |

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

# CONTENTS

## APPENDIXES



## INDEX

## FIGURES

## TABLES

# NOTATIONS

Certain notations are used throughout this manual. The notations and their meanings are:

| | |
|---|---|
| ... | Horizontal ellipses indicate repetition. |
| .<br>.<br>. | Vertical ellipsis indicate program lines not shown. |
| UPPERCASE | Uppercase text in examples of terminal dialog indicates terminal output. Uppercase words in statement and command formats must appear exactly as shown. |
| Lowercase | Lowercase text in examples of terminal dialog indicates user input. Lowercase words in statement and command formats indicate values or options supplied by the user. |
| Δ | Delta indicates a space (blank). |
| Ⓒ | Carriage return denotes the transmission key on the keyboard. |
| Shading | Shading indicates Control Data extensions to the language described in the American National Standard x3.60-1978, BASIC. Language and processing features that are in the standard, but are implementation-defined, are not shaded. |

Examples of actual terminal sessions appearing in this manual were produced on a class 1 terminal. The format of these terminal sessions might differ slightly from the formats appearing at your terminal.

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. A program can perform a simple task, such as adding or subtracting two numbers, or printing a single letter or digit. However, a program usually performs a more complicated task. 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 by 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 program operation are input, computation, and output. The process of a program performing tasks in a computer 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 computer's architecture. Writing a program by using computer instructions in the form used directly by the computer (machine instructions) is tedious and time-consuming. In order 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 John G. Kemeny and Thomas E. Kurtz at Dartmouth College.

This section describes the process of writing and executing a BASIC program by solving a sample problem. The BASIC statements used in solving the problem are explained. This section is intended for nonprogrammers. This section provides the information necessary to write BASIC programs and understand the more detailed descriptions of the BASIC language provided in the sections that follow this section.

### STATEMENT OF THE PROBLEM

The following general description outlines a manufacturing system problem that is to be solved by using BASIC. In this 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. The total cost incurred per year is T = F + C*V. If the revenue per unit made (and sold) is R per unit, then the total annual revenue is R1 = R*V. 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.

For example, 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. Using this data, answer the following questions:

1. What is the break-even point?
2. If the predicted sales are 25000 units, what is the expected profit or loss?
3. What is the expected profit or loss for sales of 50000, 75000, and 100000 units?

The BASIC program in figure 1-1 answers questions 1 and 2 of the problem. The solution to question 3 is provided later in this section.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO 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 POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P;"VOLUME =";V;"UNITS"
220 GOTO 240
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
240 END
```

Figure 1-1. Break-Even Program

## ANALYSIS OF STATEMENTS

Each line of a BASIC program is called a statement; each statement must begin with a line number. Line numbers normally indicate the sequence in which the computer is to execute the statements. The following statements are used in the break-even program shown in figure 1-1.

### REM Statement

Figure 1-2 shows a segment of the break-even program that contains the REM statement. The REM statement allows the user to insert remarks. These remarks increase readability and comprehension in a program; they have no effect on the program during execution. A maximum of 150 characters can be included in a REM statement.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
    .
    .
    .
```

Figure 1-2. REM Statement Lines

Figure 1-2 shows the use of the REM statement to identify the type of program, the variables used, and the variable identifiers. These identifiers are used later in program computations.

### LET Statement

The LET statement specifies that the variable (quantity that can vary during execution of the program) to the left of the equals sign be set to a value (the value is the formula or expression to the right of the equals sign).

Examples:

Constant Value Assignment - Statements 20 through 50 of the program in figure 1-3 assign values to variables F, C, R, and V, which are used later in computing the break-even point. The values for F, C, and R represent dollars and the value for V represents units.

Formula Value Assignment - In the program in figure 1-4, 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. The value of the variable or expression to the right of the equals sign becomes the value of the variable to the left of the equals sign. BASIC conforms to the normal algebraic rules for order of arithmetic computation. (See Arithmetic Expressions in this section.)

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

Figure 1-3. LET Statement Lines (Constants)

```
    .
    .
    .
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
    .
    .
    .
```

Figure 1-4. LET Statement Lines (Formulas)

Statement 120 directs the computer to multiply V (25000) by C (10) and add the product (250000) to F (1000000) equaling 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: print out a value; print a message; print a combination of a value and a message; and print a blank line. BASIC normally separates an output line into five print zones, each 15 characters long. 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.

Statement 080 in figure 1-5 illustrates the assignment of a value to a variable by using the LET statement. Statement 090 illustrates the use of the PRINT statement to print an identifying label and the derived value.

```
    .
    .
    .
070 REM COMPUTE BREAK-EVEN POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
    .
    .
    .
```

Figure 1-5. PRINT Statement Lines

Statement 080 directs the computer to subtract C from R (30-10) and, by using the remainder (20) as a divisor, divide F (1000000) by 20. 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 labeled VOLUME UNITS. When executed, this PRINT statement in figure 1-5 produces:

BREAK-EVEN POINT= 50000 VOLUME UNITS

### IF, GOTO, and END Statements

In the sample program (figure 1-1), if sales volume V is greater than the break-even volume, a profit is earned. If the sales volume is less than the break-even volume, a loss is incurred.

The IF statement at line number 200 in figure 1-6 directs the program execution to the statement at line number 230, if the condition V is greater than V1 is met. The IF statement directs execution to the statement at line number 210, if the condition is not met. Line 200 illustrates how execution sequence by line number can be altered.

```
    .
    .
    .
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P,"VOLUME =";V;"UNITS"
220 GOTO 240
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
240 END
```

Figure 1-6. IF, GOTO, and END Statement Lines

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

In figure 1-6, the PRINT statement at line number 210 is executed because V = 25000 and V1 = 50000. After executing the PRINT statement, the computer 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.

### BREAK-EVEN PROGRAM AND OUTPUT

Figure 1-7 shows the break-even program and the output that answers questions 1 and 2. After the program is entered into the computer, the BASIC compiler is directed to compile and execute the program.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO 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 POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P,"VOLUME =";V;"UNITS"
220 GOTO 240
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
240 END


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


BREAK-EVEN POINT= 50000 VOLUME UNITS
LOSS = $ 500000              VOLUME = 25000 UNITS
```

Figure 1-7. Break-Even Program and Output

## EXPRESSIONS IN BASIC

An expression can be simple, that is, consisting of one term (A); or complex, that is, consisting of two or more terms connected by operators (A+B-C). Expressions evaluate to a single value, which can be used later in computation, or can be used in determining program execution sequence. (See line number 200.) There are three types of expressions in BASIC: arithmetic, relational, and string. Arithmetic and relational expressions are discussed in the following paragraphs and in section 2; string expressions are discussed in section 2 of this manual.

## ARITHMETIC EXPRESSIONS

Arithmetic expressions are formed from numeric variables, numeric constants, function references, and arithmetic operators. The arithmetic operators allowed for BASIC are shown in table 1-1.

TABLE 1-1. ARITHMETIC OPERATORS

| Symbol | Meaning |
|---|---|
| ^ or ** | Exponentiation ( ↑ on some teletypewriters) |
| / | Division |
| * | Multiplication |
| + | Addition |
| - | Subtraction |

NOTE

The circumflex (^) is the preferred character symbol for exponentiation. See Future System Migration Guidelines, appendix E.

In the sample break-even program, operators (+, -, *, and /) are used in line numbers 080, 120, 150, and 180. 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; multiplication and division; addition and subtraction. Evaluation proceeds from left to right through an expression. 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.

Within a number in BASIC, commas cannot be used to separate decimal groupings. For example, ten million is written 10000000, not 10,000,000.

A numeric variable (such as F, C, R, or 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.

BASIC provides several mathematical functions that can be requested within an arithmetic expression such as SIN (sine), COS (cosine), and SQR (square root). Functions are described in section 5.

## RELATIONAL EXPRESSIONS

Relational expressions are formed by combining variables and/or constants into arithmetic expressions that are compared by using relational operators. Relational expressions are used in IF statements to compare two values. Table 1-2 illustrates the relational operators.

TABLE 1-2. 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 can be found in line number 200 of the sample break-even program. For more details and the rules for using relational operators, see section 2.

# DEFINING AND READING DATA

An efficient method of assigning values to variables is through the use of the READ and DATA statements.

## DATA AND READ STATEMENTS

In the break-even program, values are assigned to variables by using LET statements as shown in figure 1-8.

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

Figure 1-8. LET Statement Value Assignment

Statements at line numbers 020 through 050 can be replaced with the following:

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

The DATA statement creates a block of data that is internal to the program. Within the DATA statement, values must be separated by commas. In the above program, the DATA statement precedes the READ statement; however, this is not required. The DATA statement can 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; for example, F = 1000000, C = 10, R = 30, and V = 25000. This method is more efficient from the programmer's standpoint because only the associated DATA statements need to be changed for added or different data. Figure 1-9 illustrates the use of the READ and DATA statements in the break-even program.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO F, C, R, V
035 DATA 1000000,10,30,25000
037 READ F,C,R,V
060 REM
070 REM COMPUTE BREAK-EVEN POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P,"VOLUME =";V;"UNITS"
220 GOTO 240
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
240 END


When executed, this program produces:


BREAK-EVEN POINT= 50000 VOLUME UNITS
LOSS = $ 500000              VOLUME = 25000 UNITS
```

Figure 1-9. Break-Even Program With READ and DATA Statements

# LOOPING IN BASIC

We are frequently interested in solving a problem in which a specified sequence of statements is executed a number of times. Each time the sequence 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, question 3 requests the profit or loss for sales of values 50000, 75000, and 100000 units. To solve questions 1 and 2 of the problem for these four values, a loop is inserted using the IF statement (line number 104 in figure 1-10) and the GOTO statement (line number 236).

In figure 1-10, V is assigned the initial value of 25000 (line number 102). The statement of line number 104 then compares V to 100000. If V is greater than 100000, control is transferred to line number 240 and the loop ends. If V is not greater than 100000, line numbers 110 through 236 are executed in the normal sequence. The statement at line 235 increments V by 25000, and the statement at line 236 transfers control back to line 104. The statement at line number 104 compares the new value of V to 100000 to determine whether or not to execute the loop again. Looping continues until V is greater than 100000.

For each value of V, the values of T, R1, and P are computed, and LOSS or PROFIT is printed depending on the value of V; this completes the execution of the loop in the break-even program.

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

## FOR AND NEXT STATEMENTS

The sample program in figure 1-11 shows a loop created by using the FOR statement (line number 101) and the NEXT statement (line number 235).

The FOR statement establishes the first value of V (25000), the final allowable value of V (100000), and the step value of (25000). Statements between the FOR statement and the NEXT statement are repeatedly executed until V is greater than the final allowable 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 multidimensional structure. A 1-dimensional array, or list, is called a vector and a 2-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 can also be used.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO F, C, R
035 DATA 1000000,10,30
037 READ F,C,R
060 REM
070 REM COMPUTE BREAK-EVEN POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
102 LET V = 25000
104 IF V>100000 THEN 240
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P,"VOLUME =";V;"UNITS"
220 GOTO 235
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
235 LET V = V + 25000
236 GOTO 104
240 END
```

LOOP

When executed, this program produces:

```
BREAK-EVEN POINT= 50000 VOLUME UNITS
LOSS = $ 500000              VOLUME = 25000 UNITS
LOSS = $ 0   VOLUME = 50000 UNITS
PROFIT=$ 500000              VOLUME= 75000 UNITS
PROFIT=$ 1.00000E+6          VOLUME= 100000 UNITS
```

Figure 1-10. Break-Even Program With IF and GOTO Statements

Variables are used to name arrays. The individual elements of an array, identified by the use of subscripts, are called subscripted variables. The subscripts, one for each dimension of the array, are position indicators that locate elements within the array. Subscripts are separated by commas and enclosed by parentheses. The first matrix subscript designates a row; the second matrix subscript designates a column. Numbering of the elements begins with zero; the first element in the first row and the first column has subscripts (0,0).

Example:

In the following matrix, the element designated by A(1,2) is circled.

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

In the break-even program, where the profit or loss for four different sales volumes is computed, the values V, P, T, and R1 can be organized in array form, with each array containing four elements. For each volume (V), an associated revenue (R1), cost (T), and profit (P) are computed.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO F, C, R, V
035 DATA 1000000,10,30,25000
037 READ F,C,R,V
060 REM
070 REM COMPUTE BREAK-EVEN POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
100 REM
101 FOR V = 25000 TO 100000 STEP 25000
110 REM COMPUTE TOTAL COST
120 LET T=F+C*V
130 REM
140 REM COMPUTE TOTAL REVENUE
150 LET R1=R*V
160 REM
170 REM COMPUTE PROFIT/LOSS
180 LET P=R1-T
200 IF V>V1 THEN 230
210 PRINT "LOSS = $";-P,"VOLUME =";V;"UNITS"
220 GOTO 235
230 PRINT "PROFIT=$";P,"VOLUME=";V;"UNITS"
235 NEXT V
240 END


When executed, this program produces:


BREAK-EVEN POINT= 50000 VOLUME UNITS
LOSS = $ 500000              VOLUME = 25000 UNITS
LOSS = $ 0      VOLUME = 50000 UNITS
PROFIT=$ 500000              VOLUME= 75000 UNITS
PROFIT=$ 1.00000E+6          VOLUME= 100000 UNITS
```

Figure 1-11. Break-Even Program With FOR and NEXT Statements

In the sample program (figure 1-12), the DIM statement is used to specify each array as containing four elements (line numbers 039, 040, 041, and 042); however, the use of this statement is not required. To specify an array of up to eleven elements, only the selected variable name and associated subscripts are required. The advantage of using 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 eleven array elements by BASIC. If the array is to contain more than eleven elements, the DIM statement is required. See section 3 for additional information pertaining to the DIM statement.

The DIM statement in line number 039 of figure 1-12 reserves space for an array named V. The amount of space reserved is determined by the bound specifier; the bound for array V is 3. This means that the largest subscript for array V is 3 and that array V has four elements: V(0), V(1), V(2), and V(3) because a count of the elements begins with zero (0). (See figure 1-13.) Arrays P, T, and R1 in figure 1-12 are also four-element arrays. A count of the elements can also begin with 1. See the OPTION statement described in this manual.

Figure 1-14 shows the method used for placing data into the array. The variable I is used to initialize the volume array V. The variable I is set to the value of zero, and is incremented within the FOR loop (line number 102) by 25000 for each increment of J. The variable J is a subscript used to address the individual elements of array V; when J is zero, 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.

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

Figure 1-12. Break-Even Program With DIM Statements

| element 0 | element 1 | element 2 | element 3 |
|---|---|---|---|

Figure 1-13. Array V

After completing the loop between line numbers 101 and 183 (figure 1-14), all of the arrays contain the results of the computation. The PRINT statements in lines 201, 202, 203, and 204 (figure 1-15) print the individual elements of each array. The program output displays the contents of each array as shown in figure 1-16.

```
    .
    .
    .
095 REM INITIALIZE ARRAY V, COMPUTE P,T,R1
096 LET I = 0
101 FOR J = 0 TO 3
102 LET I = I + 25000
103 LET V(J) = I
130 REM
140 REM COMPUTE TOTAL COST
141 LET T(J) = F + C * V(J)
160 REM COMPUTE TOTAL REVENUE
161 LET R1(J) = R * V(J)
170 REM COMPUTE PROFIT/LOSS
181 LET P(J) = R1(J) - T(J)
183 NEXT J
    .
    .
    .
```

Figure 1-14. Placing Data Into Arrays

```
    .
    .
    .
201 PRINT " VOLUME",V(0),V(1),V(2),V(3)
202 PRINT " REVENUE",R1(0),R1(1),R1(2),R1(3)
203 PRINT " COST",T(0),T(1),T(2),T(3)
204 PRINT " PROFIT",P(0),P(1),P(2),P(3)
240 END
```

Figure 1-15. PRINT Statements for Array Elements

## TERMINAL INPUT AND OUTPUT (I/O)

Sometimes it is desirable to enter data while a program is executing. For example, if the break-even problem is 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 while the program is executing.

The INPUT statement is used in a BASIC program when entering data from the terminal keyboard. When the INPUT statement is executed, a displayed ? asks for data. Execution stops until the requested data is entered. Data entered through the terminal keyboard is assigned sequentially to variables listed as INPUT statement arguments.

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

```
BREAK-EVEN POINT= 50000 VOLUME UNITS
 VOLUME          25000          50000          75000          100000
 REVENUE         750000         1.50000E+6     2.25000E+6     3.00000E+6
 COST            1.25000E+6     1.50000E+6     1.75000E+6     2.00000E+6
 PROFIT         -500000         0              500000         1.00000E+6
```

Figure 1-16. Break-Even Program With DIM Statements Output

Figure 1-17 illustrates the break-even program using the INPUT statement. The values of variables F, C, and R are to be input. The PRINT statement at line number 015 prints a message on the terminal indicating the values and the sequence of the values to be input. The output of this statement is followed by the question mark and the result of the INPUT statement line 036 is shown in figure 1-18.

Note that only two values were entered and that the NOT ENOUGH DATA diagnostic was issued; the data was then reentered.

The program output is shown in figure 1-18. Revenue, cost, and profit were computed on the basis of data entered at the terminal. Refer to section 7 and appendix D for more information pertaining to input and output.

```
001 REM THIS IS A BREAK-EVEN PROGRAM
002 REM THE FOLLOWING VARIABLES ARE USED
003 REM FIXED ANNUAL COST          F
004 REM VARIABLE COST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VOLUME               V
007 REM BREAK-EVEN POINT (VOLUME)  V1
008 REM TOTAL COST                 T
009 REM TOTAL REVENUE              R1
010 REM PROFIT/LOSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TO F, C, R
015 PRINT "INPUT:FIXED COSTS  VARIABLE COSTS  REVENUE PER UNIT"
036 INPUT F,C,R
038 REM DEFINE ARRAYS FOR V, P, T, R1
039 DIM V(3)
040 DIM P(3)
041 DIM T(3)
042 DIM R1(3)
060 REM
070 REM COMPUTE BREAK-EVEN POINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN POINT=";V1;"VOLUME UNITS"
095 REM INITIALIZE ARRAY V, COMPUTE P,T,R1
096 LET I = 0
101 FOR J = 0 TO 3
102 LET I = I + 25000
103 LET V(J) = I
130 REM
140 REM COMPUTE TOTAL COST
141 LET T(J) = F + C * V(J)
160 REM COMPUTE TOTAL REVENUE
161 LET R1(J) = R * V(J)
170 REM COMPUTE PROFIT/LOSS
181 LET P(J) = R1(J) - T(J)
183 NEXT J
201 PRINT " VOLUME",V(0),V(1),V(2),V(3)
202 PRINT " REVENUE",R1(0),R1(1),R1(2),R1(3)
203 PRINT " COST",T(0),T(1),T(2),T(3)
204 PRINT " PROFIT",P(0),P(1),P(2),P(3)
240 END
```

Figure 1-17. Break-Even Program With INPUT Statement

```
INPUT:FIXED COSTS  VARIABLE COSTS  REVENUE PER UNIT
? 1000000,10
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  36
? 1000000,10,30
BREAK-EVEN POINT= 50000 VOLUME UNITS
 VOLUME          25000          50000          75000          100000
 REVENUE         750000         1.50000E+6     2.25000E+6     3.00000E+6
 COST            1.25000E+6     1.50000E+6     1.75000E+6     2.00000E+6
 PROFIT         -500000         0              500000         1.00000E+6
```

Figure 1-18. Break-Even Program With INPUT Statement Interactive Input/Output

# USING BASIC UNDER NOS AND NOS/BE

The previous paragraphs describe BASIC statements and the organization of these statements into a BASIC program. The following paragraphs describe the procedures for entering a program into a computer and for executing 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 teletypewriter (TTY) or cathode ray tube (CRT) terminal. See section 12 for a description of BASIC program card deck structures and batch mode operations.

BASIC runs under both the NOS and NOS/BE operating systems. Its usage under NOS is described in the following paragraphs; its usage under NOS/BE is described later in this section. See sections 10 and 11 for more detailed information.

If operating from a terminal, the program must be written into a file, as shown in the examples that follow, and must be executed from the file. To correct a line, reenter the line number, followed by the corrected line. To delete a line under NOS, enter the line number and press the transmission (carriage return) key. To delete a line under NOS/BE, enter DELETE, the line number, and press the transmission (carriage return) key. New lines can be added freely.

## NOS

BASIC programs can be run from a time-sharing terminal under NOS through Interactive Facility (IAF) or the Time-Sharing System. Login procedures for IAF and the Time-Sharing System differ. The procedures are described in the following paragraphs.

To initiate the login procedure, establish physical connection between the terminal and the computer. The method used to establish this connection varies depending on the type of terminal being used and the type of coupling between the terminal and the computer. Connection methods for IAF are described in the Network Products Interactive Facility reference manual (NOS 1 sites) and Volume 3 of the NOS 2 reference set (NOS 2 sites). Connection methods for the Time-Sharing System are described in the NOS Time-Sharing User's reference manual.

### Login, Execution, and Logoff Procedures for the Interactive Facility

The login procedure for the Interactive Facility (IAF) begins with the system printing the following three lines at the terminal. The second line of this message is dependent on the installation.

```
yy/mm/dd.  hh.mm.ss  termname
CDC NOS
FAMILY:
```

When this occurs, perform the following steps:

1. Enter the family name on the same line. If the family name is the default family for the system, press the carriage return. Certain installations do not request a family name.

   The system responds:

   ```
   USER NAME:
   ```

2. Enter the user name on the same line. The user name consists of alphanumeric characters assigned by the installation.

   The system responds:

   ```
   PASSWORD:
   ```

3. Enter the password. The password must consist of up to seven alphanumeric characters. To provide a greater measure of security, overtyping is done on hardcopy terminals.

   If the family name, user name, and password are not acceptable, the system responds:

   ```
   IMPROPER LOGIN, TRY AGAIN.
   FAMILY:
   ```

   If the family name, user name, and password are acceptable, the system responds:

   ```
   termname - APPLICATION:
   ```

   The termname on this line is the same as that on the first line of the login sequence and can be disregarded.

4. Select the Interactive Facility by entering:

   ```
   IAF
   ```

   Under NOS 1, if validation is given to access the Interactive Facility, the system responds:

   ```
   TERMINAL:            nn, NAMIAF
   RECOVER/CHARGE:
   ```
   or
   ```
   TERMINAL:            nn, NAMIAF
   RECOVER/SYSTEM:
   ```

   where nn is the terminal number. Remember this number because it can be used for recovery.

   Under NOS 2, if validation is given to access the Interactive Facility, the system responds:

   ```
   JSN:  zzzz, NAMIAF
     CHARGE NUMBER:
   ? XXXXXXXXX
   ```
   or
   ```
   JSN:  zzzz, NAMIAF

    READY.
   ```

   where zzzz is the job sequence name. Remember this name because it can be used for recovery.

5. If RECOVER/SYSTEM (NOS 1) or READY (NOS 2) is printed, the login procedure is complete; any valid command can be entered.

   If, under NOS 1, RECOVER/CHARGE is printed, type CHARGE followed by the assigned charge number and project number on the same line:

   ```
   CHARGE,chargeno,projectno
   ```

   The system responds by printing:

   ```
   READY.
   ```

   The login procedure is now complete.

   If, under NOS 2, CHARGE NUMBER is printed, type the assigned charge number in the area that has been blacked out. The system will respond:

   ```
   PROJECT NUMBER:
   ? XXXXXXXXXXXXXXXXXXXX
   ```

   Type in the assigned project number in the area that has been blacked out.

   If the charge number and project number are valid, the system responds by printing:

   ```
   READY.
   ```

   The login procedure is now complete.

6. Enter the desired subsystem by typing:

   ```
   BASIC
   ```

   Because all interactive programs run under NOS reside as files, the system queries the applicable file type by responding:

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

7. Submit the appropriate file status: lfn is the local file name.

   ```
   OLD,lfn
   ```

   Indicates the file previously created and available.

   ```
   NEW,lfn
   ```

   Indicates a new file.

   ```
   LIB,lfn
   ```

   Indicates a file from the system library.

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

   ```
   lfn NOT FOUND, AT nnnnn.
   ```

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

   ```
   ERROR IN ARGUMENT
   ```

   Correct the file name.

   If the file name entered contains too many characters, the system responds:

   ```
   ILLEGAL PARAMETER
   ```

   Correct the file name.

   After the system finds the specified file, it responds:

   ```
   READY.
   ```

   The example in figure 1-19 illustrates a sample login for both NOS 1 and NOS 2.

8. Enter the new BASIC program. Each line must begin with a 1- through 5-digit line number, and end with a carriage return. BASIC statements need not be typed in correct order; the BASIC subsystem automatically sequences the statements according to line number. The NOS edit facility, XEDIT, can be used to enter a new BASIC program or change an existing file. See the XEDIT reference manual for use of this facility.

9. To execute the program, type:

   ```
   RUN
   ```

   This command initiates compilation and execution of the BASIC program. If there are compilation or execution errors, the appropriate error messages will be displayed.

10. When a run is completed, the following options are available:

    Continue processing (build and execute new programs; modify existing program and rerun; or rerun the same program).

    or

    Terminate the terminal session with the following command:

    ```
    BYE
    ```

All files not saved (see appendix D, Indirect Access Permanent Files) are released.

Under NOS 1, the following is printed:

```
xxxxxxx    LOG OFF    hh.mm.ss.
xxxxxxx    SRU        s.sss    UNTS
```

xxxxxxx  Indicates the user name.

s.sss  Indicates the total number of system resource units used under this charge and project number.

Under NOS 2, the following is printed:

```
UN=xxxxxxx    LOG OFF          hh.mm.ss.
JSN=zzzz      SRU        s.sss     UNITS.
```

xxxxxxx  Indicates the user name.

zzzz  Indicates the job sequence name.

s.sss  Indicates the total number of system resource units used under this charge and project number.

```
NOS 1 Login:

    82/01/08. 10.50.14. T128
    CDC NOS 1
    FAMILY:
    USER NAME: xxxxxxx
    PASSWORD:  xxxx
    T128    - APPLICATION: iaf
    TERMINAL:     61, NAMIAF
    RECOVER/ CHARGE: charge,xxxx,xxxxxxx
    CHARGE,xxxx,xxxxxxx.
    /basic
    OLD, NEW, OR LIB FILE: new,ex4


     READY.



NOS 2 Login:

    82/01/08. 10.42.16. T143A
    CDC NOS 2
    FAMILY:
    USER NAME: xxxxxxx
    PASSWORD:  xxxx
    T143A   - APPLICATION: iaf
    JSN: AADI, NAMIAF
      CHARGE NUMBER:
    ? XXXXXXXXX
      PROJECT NUMBER:
    ? XXXXXXXXXXXXXXXXX
    /basic
    OLD, NEW, OR LIB FILE: new,ex4


     READY.
```

Figure 1-19. NOS Login Examples

## Login, Execution, and Logoff Procedures for the Time-Sharing System

The login sequence for the Time-Sharing System begins with the system printing the following three lines at the terminal. The second line of this message is dependent on the installation.

```
yy/mm/dd.  hh.mm.ss.
CDC TIME-SHARING SYSTEM NOS
FAMILY:
```

When this occurs, perform the following steps:

1. Enter the family name on the same line. If the family name is the default family for the system, press the carriage return. If your installation does not use family names, a family name is not requested.

   The system requests:

   ```
   USER NUMBER:
   ```

2. Enter the user number on the same line. The user number consists of up to seven alphanumeric characters assigned by the installation.

   The system requests:

   ```
   PASSWORD:
   ```

3. Enter the password. The password must consist of up 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, enter a carriage return.

   If the family name, user number, and password are not acceptable, the system responds:

   ```
   IMPROPER LOGIN, TRY AGAIN.
   FAMILY:
   ```

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

   ```
   TERMINAL: nnn,TTY
   RECOVER/CHARGE:
   ```

   or

   ```
   TERMINAL: nnn,TTY
   RECOVER/SYSTEM:
   ```

   The nnn indicates the particular terminal number being used. (These responses are installation-dependent.)

4-6. These steps are the same as steps 5 through 7 of the previous description of Login, Execution, and Logoff Procedures for the Interactive Facility.

   The example in figure 1-20 illustrates a sample login.

7-9. These steps are the same as steps 8 through 10 of the previous description of Login, Execution, and Logoff Procedures for the Interactive Facility.

```
81/07/31.      13.19.28.
TIME  SHARING  SYSTEM
FAMILY:
USER NUMBER:xxxxxxx
PASSWORD
xxxx
TERMINAL:  60,TTY
RECOVER/SYSTEM:  basic
OLD,NEW,OR  LIB  FILE:  new,ex4

READY.
```

Figure 1-20. Sample Timesharing Login

## Sample Terminal Session

The sample program in figure 1-21 was run at a terminal under the NOS 2 IAF System. Responses entered at the terminal are in lowercase letters. Press the transmission (carriage return) key after typing in each response.

```
/basic                                    <- Requests BASIC subsystem.
OLD, NEW, OR LIB FILE: new,ex4            <- Creates new file EX4.

 READY.

100 print "type a number"
110 input x
120 let f=1
130 for i=1 to x
140 let f=f*i                             <- Enters BASIC program.
150 print "factorial ";x,"is ";f
160 goto 110
170 end
list                                      <- Lists BASIC program.

100 PRINT "TYPE A NUMBER"
110 INPUT X
120 LET F=1
130 FOR I=1 TO X
140 LET F=F*I
150 PRINT "FACTORIAL ";X,"IS ";F
160 GOTO 110
170 END

 READY.
alter,160,/110/100/                       <- Changes statement 160 to correct error.

160 GOTO 100

 READY.
run                                       <- Compiles and executes program.

 FOR WITHOUT NEXT  AT  130                <- BASIC issues diagnostic.
BASIC COMPILATION ERRORS

 RUN COMPLETE.
145 next i
115 if x=0 then 170                       <- Correct program and rerun.
run

TYPE A NUMBER
? 3
FACTORIAL  3   IS  6                      <- Program requests input and prints output.
TYPE A NUMBER
? 0
 RUN COMPLETE.
list                                      <- Lists BASIC program.

100 PRINT "TYPE A NUMBER"
110 INPUT X
115 IF X=0 THEN 170
120 LET F=1
130 FOR I=1 TO X
140 LET F=F*I
145 NEXT I
150 PRINT "FACTORIAL ";X,"IS ";F
160 GOTO 100
170 END

 READY.

save,ex4                                  <- Makes file EX4 permanent.

 READY.
```

Figure 1-21. IAF System

In figure 1-21, the program is saved as a file named EX4. The program in this file is stored as an indirect access permanent file which can later be accessed by use of the OLD command (as shown in figure 1-22). At this time, add, delete, or change program statements as shown in figure 1-23. (See appendix I for an explanation of the editing commands used in figure 1-23.)

In figure 1-23, the REPLACE command replaces the old version of EX4 with the updated version. If logoff of the system had occurred before replacing EX4, the corrected version would have been lost while the old version of EX4 remained intact.

For a detailed description of the NOS commands used in figure 1-21, as well as other available NOS commands, see the IAF reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the NOS Time-Sharing User's reference manual.

## NOS/BE

To access a central computer from a terminal, establish physical connection 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. See the INTERCOM Version 5 reference manual. When connected to the terminal, the system responds:

```
CONTROL DATA INTERCOM 5.n
DATE              mm/dd/yy
TIME              hh.mm.ss

PLEASE LOGIN
```

When this occurs, perform the following steps:

1. Log in to the system by entering:

   ```
   LOGIN
   ```

   The system responds:

   ```
   ENTER USER NAME-
   ```

2. Enter the user name followed by a carriage return. The user name can be any combination of up to ten letters or digits and must not be followed by a period.

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

   ```
   xxxxxxxxxx ENTER PASSWORD-
   ```

   At a 200 User Terminal (200 UT) or any display terminal, the system responds:

   ```
   ENTER PASSWORD-
   ```

3. Enter the password followed by a carriage return. A password is any combination of up to ten letters or digits that must not terminate with a period. On a teletypewriter (TTY) listing, the system preserves privacy by allowing the password to be entered over ten character spaces that have been blacked-out by overprinting.

   When the user name and password are accepted, the time logged in and the user id (a 2-character user code), followed by the equipment number (multiplexer equipment status table ordinal) and the port number logged in, are displayed at the terminal, as shown below:

   ```
   19/07/79      LOGGED IN AT 17.47.26
                 WITH USER-ID AB
                 EQUIP/PORT 52/03
   ```

4. After a successful login the system responds:

   ```
   COMMAND-
   ```

   Enter the text edit mode by typing

   ```
   EDITOR
   ```

   The system indicates text edit mode by displaying two consecutive periods.

```
old,ex4 <------------------------ Makes a copy of file EX4 accessible.

 READY.
list <------------------------ Lists BASIC program on file EX4.

100 PRINT "TYPE A NUMBER"
110 INPUT X
115 IF X=0 THEN 170
120 LET F=1
130 FOR I=1 TO X
140 LET F=F*I
145 NEXT I
150 PRINT "FACTORIAL ";X,"IS ";F
160 GOTO 100
170 END

 READY.
```

Figure 1-22. OLD Command Accesses Permanent File Under NOS

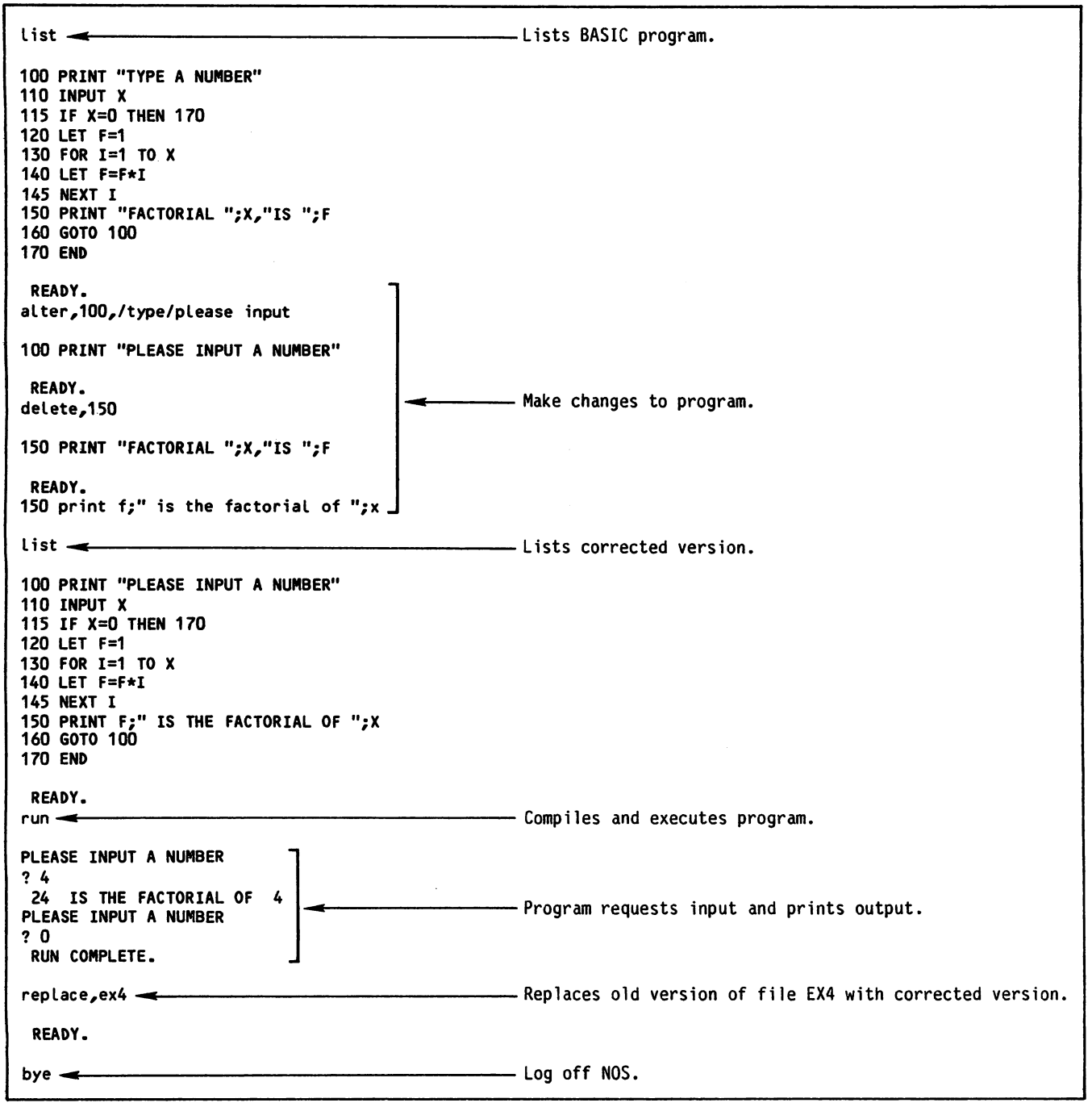


Figure 1-23. Editing a Program Under NOS

5. Once in text edit mode, enter the command

   FORMAT,BASIC

   When this command is entered after the two periods, a format specification is automatically established at the terminal that permits lines to be entered in BASIC language format. The comma is optional.

6. Enter the BASIC program statements (line number followed by BASIC statement).

   After the first line, the two period prompts are not given; continue inserting statements. Each line must begin with a 1- through 5-digit line number and end with a carriage return. BASIC statements need not be typed in correct order because the EDITOR automatically sequences them according to line number.

7. Once the entire program is entered, compile and execute the program by typing:

   RUN,BASIC

   After the program compiles and executes, the appropriate error messages are displayed if program errors occur. The comma is optional.

8. When the run completes, select one of the following options:

   Continue processing (build and execute new programs; modify and rerun existing programs; or rerun the same program).

   or

   Terminate the terminal session by entering the BYE or BYE BYE command. When the BYE or BYE BYE command is entered, the system is returned to command mode from EDITOR mode. The BYE command does not save the EDIT file. (See the INTERCOM Version 5 reference manual.)

   The system responds with:

   COMMAND-

   At this time, enter the LOGOUT command to release any local files created under EDITOR. Only files that are permanent are retained after logout. Disassociation from NOS/BE occurs until a subsequent LOGIN command is entered. NOS/BE displays the date and time logged out. LOGOUT is not allowed when operating under control of the EDITOR. (Leave EDITOR via the END or BYE command.)

   For example if the command LOGOUT is entered, the system responds:

```
CPA            6.377 SEC.      6.377 ADJ.
CPB             .000 SEC.       .000 ADJ.
SYS TIME                       7.774
CONNECT TIME      0 HRS.       19 MIN.
  10/21/79   LOGGED OUT AT 08.43.09.
```

   Logout time is given in hours, minutes, seconds (24-hour clock); CP time is given in seconds. Disconnect the terminal from NOS/BE by turning it off, or by hanging up the data set receiver.

## Sample Terminal Session

After logging in, create and execute BASIC programs. The sample BASIC program in figure 1-24 illustrates how to run a BASIC program under NOS/BE. The program was entered at a TTY terminal. After typing each response, press the carriage return key.

```
COMMAND- editor                          Enter EDITOR.
..format,basic                           Request BASIC program format.
..10 print "type a number";
20 input x
30 let f=1
40 for i=1 to x                          Enter BASIC program.
50 let f=f*i
60 print "factorial ";x,"is ";f
70 goto 110
80 end
run,basic                                Compile and execute the program.

FOR WITHOUT NEXT  AT  40                 BASIC issues diagnostic.
 BASIC COMPILATION ERRORS
..55 next i                              Correct errors.
25 if x=0 then 80
run,basic                                Rerun program.


TYPE A NUMBER ?3
FACTORIAL  3  IS 6                       Program requests input and prints output.
TYPE A NUMBER ?0
..list,all,sup                           List program; sup suppresses additional line number prefixes.


10 PRINT "TYPE A NUMBER";
20 INPUT X
25 IF X=0 THEN 80
30 LET F=1
40 FOR I=1 TO X
50 LET F=F*I
55 NEXT I
60 PRINT "FACTORIAL ";X;" IS";F
70 GOTO 10
80 END
..save,basprog                           Make edit file a local file named BASPROG.
..store,basprog,jones                    Make local file BASPROG permanent.
 CT ID=    JONES PFN=BASPROG:
 CT CY= 001 SN=PFQSET  00000064 WORDS.:
..end                                    Exit EDITOR.
```

Figure 1-24. BASIC Program Under NOS/BE

Using the SAVE command to save file BASPROG allows the file to be reserved for later use during the terminal session (for example, before logging out). To save the file permanently, it must be stored as a permanent file using the STORE command. (Some accounting information might be necessary before saving a file with STORE. Check site procedures.) To retrieve and execute this program later, the command sequence in figure 1-25 must follow the user login sequence.

The FETCH command retrieves the file previously made permanent and tells EDITOR that BASPROG is to be the edit file. The commas are optional. The RUN command compiles and executes the program.

```
COMMAND-fetch,basprog
COMMAND-editor
..format,basic
..edit,basprog
..run,basic
```

Figure 1-25. Retrieval and Execution Example

For a more detailed description of INTERCOM EDITOR commands used in this example, as well as other available commands, see the section on Terminal Operation under NOS/BE and the INTERCOM Version 5 reference manual.

This section describes the BASIC language structure, and explains the elements of the language. The language elements include: numeric data consisting of integer, decimal and exponential constants; string data consisting of alphanumeric text with or without quotation marks; variables representing values that are not fixed; and operators of the language, expressions, and function references.

## BASIC LANGUAGE STRUCTURE

A BASIC program is comprised of statements that define the type of operations performed and the types of data manipulated by the program. The statement lines are written by using characters from the BASIC character set. The following paragraphs define the BASIC character set, the structure of a BASIC statement, and the structure of a BASIC program.

### CHARACTER SET

The characters listed in table 2-1 can be used to form BASIC statements. 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

A BASIC statement can be in the form of an executable statement that specifies a program action (LET X=10) or a nonexecutable statement that provides information necessary for program execution (DATA 1,3,5). All BASIC statements have the following common characteristics:

Each statement begins with a line number. Line numbering must range from 1 to 99999.

Each statement must be completed on a single line. Statement continuation onto another line is not allowed.

Generally, blanks within a BASIC statement have no meaning. However, there are specific instances in which blanks are significant, such as in strings. Blanks should only be used to separate elements of the BASIC language; for example, they should not be embedded within line numbers, keywords, constants, or variable names. See Future System Migration Guidelines, appendix E.

A BASIC statement, including blanks, line numbers, and tail comments, can be a maximum of 150 characters.

The character apostrophe (') marks the beginning of a tail comment except when it appears in a string, an image statement, or a DATA statement. Tail comments serve only as documentation except for being included in the 150 character statement limit.

TABLE 2-1. BASIC CHARACTER SET

| Symbol | Description |
|---|---|
| A thru Z | Letters (uppercase) |
| + | Plus |
| - | Minus |
| * | Asterisk |
| / | Slash |
| ( | Left parenthesis |
| ) | Right parenthesis |
| $ | Dollar |
| = | Equal |
| : | Colon |
| ' | Apostrophe |
| 0 thru 9 | Numerals |
| Δ | Blank† |
| , | Comma |
| . | Period |
| " | Quote |
| ^ | Circumflex†† |
| < | Less than |
| > | Greater than |
| ? | Question mark |
| ; | Semicolon |
| # | Number |

†Refer to appendix E for recommendations tions for the use of blanks.

††Up arrow (↑) on some terminals.

### PROGRAM STRUCTURE

A BASIC program is a group of statement lines arranged according to the following general rules:

Program statements must be in line number order when the program is compiled. If entering program lines in the BASIC subsystem under NOS or using the EDITOR command FORMAT,BASIC under

NOS/BE, the program statements need not be entered in line number order because they are automatically sorted. See the Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the INTERCOM Version 5 reference manual for information about sorting line numbers before execution.

Executable and nonexecutable statements can be intermixed. In the following example, a nonexecutable statement is the DATA statement at line number 110, and an executable statement is the IF statement at line number 100. These executable and nonexecutable statements are explained in more detail later in this manual.

```
100 IF A=B THEN 110
110 DATA 10,20,30
120 READ C,D,E
130 END
```

An END statement must have the highest line number in the source program.

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

# 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 of the numeric constant types has specific rules that govern its use, the following rules apply to all three constant types:

A comma cannot be used to delimit placement over the one-hundredth place, such as thousands and millions.

When a numeric constant is not signed explicitly by a negative or positive sign, the constant is assumed to be positive.

Any number of digits can appear in a numeric constant; a maximum of 14-digit accuracy is used in computation. The CYBER 170 Model 176 uses a method different from other CYBER models when rounding the results of division. The difference is in the 15th digit of accuracy, but can become apparent when several divides and multiplies are done in succession (as in the case when matrix inversion is followed by matrix multiplication).

Whether integer, decimal, or exponential, the absolute value of a constant must be in the range 3.13152 times $10^{-294}$ to 1.26501 times $10^{322}$. To compile a program containing constants with values above this range results in the diagnostic ILLEGAL NUMBER. Constants with values below this range are treated as zeros.

### Integer Constants

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

Examples:

```
-49
+123456789
25000
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
50.5
1.91632614
147.2
.0000001
+3025.098
```

### Exponential Constants

The representation of very large or very small numbers is simplified by using exponential constants. For example, to write ten billion in its full form requires 11 digits (10000000000); however, ten billion can also be represented as 1.0 times $10^{10}$.

In BASIC, this exponential form is expressed by 1.0E10. The 1.0 is the significand and the 10 is the exponent. The E means times ten to the power of.

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

A number, the significand, must precede the E. The significand can be any valid integer or decimal constant.

The exponent (number that follows the E) is an integer constant with a positive or negative sign. If a sign is absent, a positive sign is assumed. If the exponent is too large to be represented in the computer, a diagnostic is issued.

Decimal points are not permitted in the exponent.

Examples:

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

## STRING CONSTANTS

A string is a collection of alphabetic, numeric, and special characters. In BASIC, these characters are usually set off by quotes from the rest of the program; this is called quoted text. Strings that are not set off with quotes, called unquoted strings, are permitted, but they can only be used in DATA statements or as input data.

Rules:

A string enclosed in quotes consists of all characters between quotes, including blanks.

The maximum length of a string depends on the mode: normal or ASCII. In normal mode, the maximum length is 131070 characters; in ASCII mode, the maximum length is between 65535 and 131070 characters, depending on the number of escape code characters in the string. See appendix A.

A zero-length string, also called a null string, is represented by a pair of quotes ("").

Any character can be used in quoted strings.

The character quote (") must be used as a pair of quotes (""). An embedded quote uses two pairs.

Examples:

```
"PART 25"
"THIS IS A TEST"
"An""embedded""quote"
```

The outside quotation marks are not part of the string constant. See DATA statement under I/O Statements and Functions, section 7, for an example of unquoted strings.

# VARIABLES

Variables represent values that 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 and can be simple or subscripted.

## SIMPLE VARIABLES

Simple variables can be either numeric or string. These two types of simple variables are described in the following paragraphs.

### Numeric

A simple numeric variable represents a numeric value. It is named by a single alphabetic character or a single alphabetic character and a 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:

Numeric variables represent only numeric data.

Numeric variables are preset to zero before the program executes.

The absolute value of a numeric variable must be in the range of 3.13152 times $10^{-294}$ to 1.26501 times $10^{322}$.

If a value smaller than the minimum is assigned, the variable is set to zero.

If a value greater than the maximum is assigned, a fatal diagnostic is issued.

### String

String variables represent alphanumeric text and are named with a 2- or 3-character identifier. The first character must be alphabetic, the optional second character must be a digit (0 through 9) and, in either case, the last character must be a dollar sign ($). For example:

```
A$
B$
Y$
A1$
B9$
Y3$
```

The value represented by a string variable is a string of characters. Internally, each character is represented by one or two 6-bit numeric codes. (See appendix A.) Each character has a code value that represents a position in the collating sequence. The characters at the beginning of the alphabet have code values that 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$.

The string represented by a string variable can contain from 0 through 131070 6-bit characters or from 0 through 65536 12-bit escape code (ASCII) characters. The maximum for a string containing both 6- and 12-bit characters (the usual case when operating in ASCII mode) lies somewhere between 65535 and 131070 characters depending upon the number of 12-bit escape code characters.

The memory space allocated to each string is determined by the length of the string. The minimum is one computer word; the maximum is 13108 computer words. The one-word minimum space is allocated by the BASIC compiler for every string variable mentioned in the program. The remaining words are allocated and de-allocated dynamically at execution time.

## SUBSCRIPTED VARIABLES

Subscripted variables represent one value in an array of values. There are two types of subscripted variables: numeric and string. Numeric subscripted variables are formed by a simple numeric variable followed by a subscript list; string subscripted variables are formed by a simple string variable followed by a subscript list. A subscript list consists of one to three numeric expressions bounded by parentheses. (See figure 2-1.) Rules for the values of subscripted variables are the same as for simple variables.

| NUMERIC SUBSCRIPTED VARIABLES |
|---|
| A(0) |
| B2(3) |
| B(5,10) |
| A(B2(3)) |
| X(1,N+M,A(3)) |
| **STRING SUBSCRIPTED VARIABLES** |
| B$(4) |
| L$(1,J+3) |
| C$(1,J+3,A(1)) |

Figure 2-1. Numeric and String Subscripted Variables

Rules for subscripted variables are listed below:

BASIC permits 1-, 2-, or 3-dimensional arrays. In BASIC, array dimensions can be declared implicitly by using subscripted variables.

Unless an array has been explicitly defined by a DIM statement, as described in section 3, the first subscripted variable that references an element in an array automatically defines the array as containing 11 elements (0 through 10) in each dimension. Thus, a 1-dimensional array has 11 elements; a 2-dimensional array has 11 times 11 (or 121) elements, and a 3-dimensional array has 11 times 11 times 11 (or 1331) elements.

A subscript value greater than 10 requires a DIM statement. If a maximum subscript value of less than 10 is desired, a DIM statement can be used. (See section 3.)

Subscripted variables with one subscript refer to elements in 1-dimensional arrays; subscripted variables with two subscripts refer to 2-dimensional arrays; subscripted variables with three subscripts refer to 3-dimensional arrays.

A subscript can be any arithmetic expression. The subscript used is the value of the expression rounded to an integer.

Simple and subscripted variables with the same name can be used in the same program.

The lower limit on subscripts is zero. However, this limit can be changed to one by using OPTION BASE 1. (See OPTION statement in section 3.) OPTION BASE 1 instructs the system to start array subscripting with element 1, rather than the default element 0. Thus, when OPTION BASE 1 is in effect, automatically-defined 1-dimensional arrays contain 10 elements (1 through 10), automatically-defined 2-dimensional arrays contain 100 elements, and automatically-defined 3-dimensional arrays contain 1000 elements.

Once an array is defined in a BASIC program, the number of array dimensions cannot be changed. For example, T(5) and T(2,3) cannot be used in the same program. However, the number of elements within a particular dimension can be changed if the total number of elements in the resulting array is less than or equal to the total number of elements in the original array. For example, array T(2,3) could be redefined as T(3,2).

## SUBSTRING ADDRESSING

Substring addressing specifies a portion of the value associated with a simple or subscripted string variable. Substring addressing can be used anywhere that simple or subscripted string variables can be used. Substring addressing is achieved by adding a substring qualifier after a simple or subscripted variable. (See figure 2-2.)

The substring qualifier specifies the portion of the value of the string variable from its mth through nth character.

| sv (m:n) | |
|---|---|
| sv | Indicates the string variable. |
| (m:n) | Indicates the substring qualifier with m and n as numeric constants, variables, or expressions. |

Figure 2-2. Substring Addressing Format

The m and n represent the positions in the string. The substring indices are first rounded to integers, then the following rules are applied:

If m<1, then m is considered to equal 1.

If m>the length of the string, then the substring addressed is the null string immediately following the last character of the string.

If n>the length of the string, then n is considered to be equal to the length of the string.

If n<m, then the substring addressed is the null string preceding the mth character of the string.

If A$ contains ABCDEF, then the following is true:

A$(1:4) represents ABCD.

A$(0:3) represents ABC (rule 1).

A$(4:8) represents DEF (rule 3).

A$(4:4) represents D.

A$(3:2) represents the null string between B and C.

A substring variable can be used anywhere that a string variable can be used.

# EXPRESSIONS

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

## ARITHMETIC EXPRESSIONS

Arithmetic expressions consist of a series of numeric operands and operators. Operators can be any arithmetic operator listed in table 2-2; operands can be any numeric constant, simple or subscripted variable, numeric function reference, or any expression enclosed in parentheses. A function reference is a notation for activating a predefined algorithm. If arguments are required by the function, the arguments are evaluated and passed to the function. The function then calculates and returns a result based on the arguments. The returned value is used in place of the function reference. BASIC provides several built-in functions and allows you to write your own functions. See BASIC Functions in section 5.

TABLE 2-2. ARITHMETIC EXPRESSION OPERATOR HIERARCHY

| Hierarchy | Operator | Definition |
|---|---|---|
| 1 | ^ or ** | Exponentiation (Note: ↑ on some teletypewriters) |
| 2 | * and / | Multiplication and division |
| 3 | + and - | Unary + and - |
| 4 | + and - | Addition and subtraction |

### Rules for Writing Arithmetic Expressions

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

Only numeric operands and numeric operators can be used.

Two arithmetic operators cannot appear side by side; for example, X+-Y is not allowed. If a minus sign is used to indicate a negative value in an expression, parentheses must be used to separate the negative sign and associated operand from the remainder of the expression. For example:

Correct A*(-B)

Incorrect A*-B

Operators cannot be implied; for example, (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+SIN(X) (SIN is a built-in function)
-3.14*R^2
```

### Arithmetic Expression Evaulation

The rules for the evaluation of arithmetic expressions are as follows:

Expressions within parentheses are evaluated first.

Operations of higher precedence are performed before those of lower precedence. Precedence is determined by the hierarchy illustrated in table 2-2 from highest (1) to lowest (4).

Operations of equal priority or precedence are performed in order from left to right.

Table 2-3 illustrates some examples of arithmetic expression evaluation.

TABLE 2-3. EXPRESSION EVALUATIONS

| Expressions | Evaluation Steps |
|---|---|
| A+B*C/D^E | 1. D^E = a<br>2. B*C = b<br>3. b/a = c<br>4. A+c = d (final value) |
| A+(B-C)*3 | 1. B-C = a<br>2. a*3 = b<br>3. A+b = c (final value) |
| -2^2 | 1. 2^2 = a<br>2. -a = -4 (final value) |
| (-2)^2 | 1. -2 = a<br>2. a^2 = 4 (final value) |

## STRING EXPRESSIONS

String expressions consist of a series of string operands and operators. There is only one string operator available, string concatenation (+). String operands can be one of the following:

A string constant

A simple or subscripted string variable

A string function reference

A substring reference

The following are examples of string expressions:

```
"TEST1"
B$(1)+D$
B$(1:4)
```

### Concatenation

The format of a string concatenation is shown in figure 2-3. The concatenation operation causes the string to the right of the operator, $se_2$, to be appended or joined to the end of the string to the left, $se_1$.

| | |
|---|---|
| $se_1 + se_2$ | |
| se | Indicates string constant, variable, or expression. |

Figure 2-3. String Concatenation Format

The character + is either a concatenation operator or an arithmetic operator, depending on the context. It must be surrounded by string variables to be considered a concatenation operator. Any expression containing both string and arithmetic operands is illegal.

Only one string operator can be used, so there is no hierarchy of operations. Parentheses can be used to group expressions into subexpressions, but such groupings have no effect on the result.

The fatal error, STRING OVERFLOW, results if a concatenation operation produces a string longer than the allowable maximum. In normal mode, the maximum string length is 131070 characters. In ASCII mode, the string length maximum is 65535 to 131070 characters, depending on the number of escape code ASCII characters in the string.

The following examples illustrate string expressions and the string concatenation operator.

"ABC" + "DEF" evaluates to "ABCDEF"

If string A$ contains the string expression SUBSTRING EXPRESSION, then A$(1;10) + "ADDRESSING" evaluates to "SUBSTRING ADDRESSING".

## RELATIONAL EXPRESSIONS

There are two types of relational expressions: simple and compound. Simple relational expressions are formed by connecting two numeric or string expressions with a relational operator. Compound relational expressions are formed by connecting two simple relational expressions with a logical operator.

### Simple Relational Expressions

The format of a simple relational expression is shown in figure 2-4. The relational expression operators that can be used to connect numeric or string expressions are shown in table 2-4.

| | |
|---|---|
| $e_1$ op $e_2$ | |
| $e_1$, $e_2$ | Indicates numeric or string constants, variables or expressions. |
| op | Indicates relational operator. |

Figure 2-4. Format for Simple Relational Expressions

The rules for writing simple relational expressions are as follows:

Comparison of a string to numeric expressions is not allowed.

Only one relational operator is allowed in an expression.

Relational expressions can be used only in IF statements (section 4).

TABLE 2-4. RELATIONAL EXPRESSION OPERATORS

| Operator | Definition |
|---|---|
| = | Equal to |
| <> or >< | Not equal to |
| > | Greater than |
| < | Less than |
| > = or = > | Greater than or equal to |
| < = or = < | Less than or equal to |

The rule for evaluating simple numeric relational expressions is as follows:

The two arithmetic expressions are evaluated and then their resultant values are compared algebraically to yield a true or false value. If A = 2 and B = 3, the expressions in figure 2-5 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 |

Figure 2-5. Evaluating Simple Relational Expressions

The rules for evaluating simple string relational expressions are as follows:

Strings are compared character-by-character in left-to-right order. BASIC compares characters according to their position in the collating sequence. (See appendix A.) For example, A is less than B, since the numeric code is 65 for A and 66 for B.

ASCII is the default collating sequence used for all string comparisons in BASIC. OPTION COLLATE can be used to change the collating sequence to a collating sequence that is native to the character set being used. See the OPTION statement, and appendix A (describes the various character sets supported by BASIC).

Strings are equal if they have the same length and contain the same characters (including blanks) in the same order. Blanks are important when they are used in strings.

When strings are equal in length, the first pair of corresponding characters that are not equal determines the greater string. For example, ABXY is greater than ABCZ because the numeric code for X is greater than the numeric code for C.

When strings are unequal in length, but corresponding characters that can be compared are equal, the longer string is always considered greater. For example, ABX is greater than AB.

When strings are unequal in length, but one of the corresponding characters that can be compared when scanning from left-to-right is greater, the string with the first character of greater value is the greater string. For example, X7 is greater than X6543, and X76 is greater than X75123.

## Compound Relational Expressions

A compound relational expression is a sequence of simple relational expressions separated by logical operators. A compound relational expression evaluates to TRUE or FALSE. The format for the compound relational expression is shown in figure 2-6. The logical operator hierarchy is shown in table 2-5.

$r_1$ op $r_2$

| | |
|---|---|
| $r_1$,$r_2$ | Simple relational expression or compound relational expression. |
| op | Logical operator (AND, OR, unary NOT). |

Figure 2-6. Format for Compound Relational Expressions

TABLE 2-5. LOGICAL OPERATOR HIERARCHY

| Hierarchy | Operator | Definition |
|---|---|---|
| 1 | NOT | Logical negation |
| 2 | AND | Logical multiplication or logical intersection |
| 3 | OR | Logical addition or union (inclusive or) |

The rules for evaluating compound relational expressions are as follows:

Expressions within parentheses are evaluated first.

Operators of higher precedence (hierarchy) are performed before those of lower precedence. The hierarchy and definition of the logical operators are provided in table 2-5.

NOT is a unary operator and can appear to the left of any operand; however, it cannot appear as the only operator between two operands.

NOT can appear between the other logical operators (AND, OR) and an operand (for example, $r_1$ AND NOT $r_2$; $r_1$ OR NOT $r_2$).

In the truth table 2-6, the NOT (unary) operator is evaluated. The NOTp is the opposite of p. In the following examples, A=1 and B=2; thus, TRUE is printed for the first example, and FALSE is printed for the second example.

```
IF A<B THEN PRINT "TRUE" ELSE PRINT "FALSE"

IF NOT A<B THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

In the first example, it is true that A is less than B; in the second example, it is false that A is not less than B (A is less than B).

TABLE 2-6. NOT (UNARY) OPERATOR EVALUATIONS

| p | NOTp |
|---|---|
| FALSE | TRUE |
| TRUE | FALSE |

The logical operators AND, OR are defined in truth tables 2-7 and 2-8.

In the examples below, which illustrate the use of NOT, AND, and OR, if A=5, B=4, C=2, D=1, I=8, and J=4, the results are as follows:

```
NOT A>B AND C=D
```

Evaluates to false AND false, so the expression is false.

```
NOT (A>B AND C=D)
```

Evaluates to NOT false, so the expression is true.

```
I=J OR NOT J>I
```

Evaluates to false OR true, so the expression is true.

```
2*I=J^2 AND I<J
```

Evaluates to true AND false, so the expression is false.

TABLE 2-7. AND OPERATOR EVALUATIONS

| p \ q | FALSE | TRUE |
|---|---|---|
| FALSE | FALSE | FALSE |
| TRUE | FALSE | TRUE |

TABLE 2-8. OR (INCLUSIVE) OPERATOR EVALUATIONS

| p \ q | FALSE | TRUE |
|---|---|---|
| FALSE | FALSE | TRUE |
| TRUE | TRUE | TRUE |

This section describes the statements that are used for the following purposes:

Perform value assignment during program execution.

Choose the lower boundary of an array.

Choose the collating sequence to be used for string and function comparisons.

Define and allocate storage for arrays.

Terminate execution of a program.

Insert explanatory remarks into a program.

The tables in each category of statements summarize the effect and usage of each statement.

## VALUE ASSIGNMENT

The value of a variable can be assigned with the LET statement. For numeric variables, the present value is replaced by a new value. For string variables, the complete present value or a specified substring of the value can be replaced by a new value.

### LET STATEMENT

The LET statement assigns a value to one or more variables during execution of a BASIC program. The effect and usage of the LET statement is shown in table 3-1. The format of the LET statment is shown in figure 3-1. The use of the word LET is optional in the LET statement.

TABLE 3-1. VALUE ASSIGNMENT

| Statement | Effect | Usage |
|---|---|---|
| LET | Assigns a numeric or string value to one or more variables specified in the LET statement line. | LET B = 3+2<br>LET A1=A2=X+Y<br>C(4) = 20 |

When the LET statement contains a single variable (nv or sv) on the left-hand side of the equals sign, the value of the expression ne or se on the right-hand side of the equals sign is assigned to the variable. When the LET statement contains a series of equalities, each variable is assigned the value of the expression. Subscript expressions are evaluated prior to the assignment of the value, and all expressions are evaluated according to the rule of operator precedence. (See table 2-2 in section 2.) For examples, see figure 3-2.

1. LET nv=ne (or) nv=ne
   (or)
   LET sv=se (or) sv=se

2. LET $nv_1=nv_2=nv_3 \ldots =nv_n=ne$
   (or) $nv_1=nv_2=nv_3 \ldots =nv_n=ne$

   (or)

   LET $sv_1=sv_2=sv_3 \ldots =sv_n=se$
   (or) $sv_1=sv_2=sv_3 \ldots =sv_n=se$

nv Indicates a numeric variable (simple or subscripted). The string variables can also have a substring descriptor.

sv Indicates a string variable (simple or subscripted).

ne Indicates a numeric expression of any complexity.

se Indicates a string expression of any complexity.

Figure 3-1. LET Statement Format

```
      .
      .
      .
10 LET A1=X+Y
20 LET A2=A3=A4=X+Y
25 LET I=2+1
30 LET Z(I)=I=6
35 LET Z(I)=4
40 LET B$="TEST"
      .
      .
      .
```

Figure 3-2. LET Statement Examples

In figure 3-2, the LET statement at line number 10 assigns the value of the expression X+Y to the variable A1. The LET statement at line number 20 assigns the same expression value to each of the variables A2, A3, and A4. The LET statement at line number 25 assigns the value 3 to variable I. The LET statement in line number 30 simultaneously assigns the value 6 to variable I and Z(3). (The subscript is evaluated before any assignments occur; therefore, the value of I in Z(I) is 3.) The LET statement in line number 35 assigns the value 4 to Z(6). The LET statement in line number 40 assigns the character string TEST to the string variable B$.

Substring addressing can be used anywhere that string variables are used. Use the LET and the INPUT statements to replace, delete, extract, or insert substrings into or from a simple or subscripted string variable. Any length string (up to

the limits) can be inserted into a string by using a substring descriptor. A substring can be replaced by assigning a new value to that particular part of the string. A substring can be deleted by assigning a null value to it. The value of the original string can be lengthened or shortened with these insertion, deletion, and replacement operations. A variable containing a null string can be assigned a value by extracting a substring value from one string and inserting it into the null string. Figure 3-3 shows several examples of substring addressing; all the examples assume an original string variable value of ABCDEF.

**The following examples of substring addressing use an original string value of ABCDEF.**

| | |
|---|---|
| 20 LET A$(2:5)="XXXX" | Value XXXX replaces BCDE; value of string A$ becomes AXXXXF. |
| 215 LET C$(3:5)=" " | Null value replaces CDE; value of string C$ becomes ABF. |
| 110 LET B$(4)(2:0)="MM" | Value MM replaces the null string before B; value of subscripted string variable B$(4) becomes AMMBCDEF. |
| 30 LET Z$(1:3)=Z$(4:6) | Value DEF replaces the first three characters of string Z$; value of Z$ becomes DEFDEF. |
| 10 LET B$=A$(2:4) | A$ is the original string value of ABCDEF; B$ contains the null value; B$ is assigned the extracted value BCD. |

Figure 3-3. Substring Addressing Using LET Statement

# OPTION STATEMENT AND DIM STATEMENT

To choose a particular collating sequence for comparing strings and computing values, and to declare the base (origin) of all arrays, use the OPTION statement. To declare and allocate storage for 1-, 2-, or 3-dimensional arrays that are not the default size, use the DIM statement. See table 3-2 for a summary of the effects and usage of the OPTION and DIM statements.

## OPTION STATEMENT

Use the OPTION statement for two distinct purposes: to explicitly declare the lower boundary (or origin) of all arrays being used in the program to base 0 or to base 1, and to choose the collating sequence to be used in the program for comparing strings and for computing values of the CHR$ and ORD functions. If the OPTION statement is encountered during normal program execution, control passes to the next statement, with no effect on the program.

## OPTION BASE n

The OPTION BASE n statement explicitly sets the origin of all arrays to either 0 or 1. OPTION BASE n can appear only once in a program, and it must precede any DIM statement or any reference to an array. If OPTION BASE n is not specified, the lower boundary of all arrays is assumed to be base 0. The default for array subscripting starts with element 0.

In the following example, BASE n is declared as 1. Since the example specifies that subscripting starts with element 1, the DIM statement defines A as a 3 by 4 (or 12 element) array, and B as a 2 by 13 (or 26 element) array.

```
100 OPTION BASE 1
110 DIM A(3,4),B(2,13)
```

Using OPTION BASE 0 (the default) in the above example would cause the array A to be dimensioned as a 4 by 5 (or 20 element) array, and B to be dimensioned as a 3 by 14 (or 42 element) array. Other examples of using OPTION BASE n are shown under Matrix Statements in section 8. Figure 3-4 shows the possible formats for OPTION BASE n.

TABLE 3-2. OPTION AND DIM STATEMENTS

| Statement | Effect | Usage |
|---|---|---|
| OPTION | Can set the lower boundary of all arrays being used by the program to base 0 or to base 1. Also, this statement can select the collating sequence to be used for string comparison and for value computation of the CHR$ and ORD functions. | OPTION BASE 1<br>OPTION COLLATE NATIVE<br>OPTION COLLATE STANDARD |
| DIM | Defines and allocates storage for 1-, 2-, and 3-dimensional arrays. | DIM A(4,4), B(15) |

1. OPTION BASE n

2. OPTION COLLATE NATIVE
   (or)
   OPTION COLLATE STANDARD

3. OPTION BASE n, COLLATE NATIVE
   (or)
   OPTION BASE n, COLLATE STANDARD

   (or)

   OPTION COLLATE NATIVE, BASE n
   (or)
   OPTION COLLATE STANDARD, BASE n

n Indicates the origin to be set; it can be either 0 or 1.

Note

You should not use OPTION COLLATE NATIVE in normal mode. See Future System Migration Guidelines, appendix E.

Figure 3-4. OPTION Statement Formats

## OPTION COLLATE

The OPTION COLLATE NATIVE and OPTION COLLATE STANDARD determine the collating sequence used by a program for comparing strings and for computing values of the CHR$ and ORD functions. Figure 3-4 shows the formats for these two choices.

OPTION COLLATE STANDARD is the default collating sequence; it specifies that the ASCII collating sequence is to be used by the program for comparing strings and computing values of the CHR$ and ORD functions. Every character in the BASIC character set (as shown under BASIC Language Structure) is assigned an ASCII character code; the smaller the ASCII character code, the earlier the character appears in the collating sequence. This ordering is important in string comparison operations because BASIC compares characters according to their assigned numeric codes in the applicable character set. For example, A is less than B because the ASCII (or BASIC decimal) code is 65 for A and 66 for B. Table A-1 in appendix A provides a list of characters and their corresponding ASCII character codes.

OPTION COLLATE NATIVE instructs BASIC to select the collating sequence native to the character set being used by the program. The character set used by a program is determined by the AS parameter of the BASIC control statement. (See Batch Operations, section 12.) As shown in appendix A, the native character sets supported by BASIC can be classified as the ASCII character set or as the normal character set. The native collating sequence for ASCII character sets (described in appendix A as NOS ASCII 128-character set, NOS/BE ASCII 128-character set, and the Extended Character Set) is the same as for the standard collating sequence. The native collating sequence used for normal character sets (described in appendix A as CDC 63-character set, CDC 64-character set, ASCII 63-character set, and ASCII 64-character set) is display code. However, because of the anticipated changes in BASIC, it is recommended that OPTION COLLATE NATIVE not be used in normal mode. See the Future System Migration Guidelines, appendix E. BASIC treats display character codes in the same way as ASCII character codes. That is, the smaller the display character code, the earlier the character appears in the collating sequence. Table A-2 in appendix A provides a list of characters and their corresponding display character codes.

The COLLATE option can be used only once in a program. If the statement is not specified, OPTION COLLATE STANDARD is assumed by default.

## DIM STATEMENT

The DIM statement explicitly defines one or more arrays and allocates storage space for the named arrays. The format for the DIM statement is shown in figure 3-5.

DIM $m_1(nc_1, \ldots, nc_3), \ldots, m_n(nc_1, \ldots, nc_3)$

$m_1 - m_n$ Indicates numeric or string matrix identifier.

$nc_1 - nc_3$ Indicates one-to-three unsigned integers, separated by commas, that represent the maximum value of each subscript.

Figure 3-5. DIM Statement Format

Arrays require a DIM statement when a subscript value greater than 10 is needed. To save space, use the DIM statement to dimension an array with an upper subscript limit of less than 10. An array not previously defined by the DIM statement is implicitly declared to have one dimension (10) when an element is referenced by an array variable with one subscript; two dimensions (10,10) when the element is referenced by an array variable with two subscripts; and three dimensions (10,10,10) when the element is referenced by an array variable with three subscripts. In all cases, the maximum subscript for each dimension in implicitly declared arrays is 10.

Use DIM statements anywhere in a program, but define an array prior to usage of that array. See Future System Migration Guidelines, appendix E. However, an array variable cannot be declared in a DIM statement more than once in the same program. An array can be redimensioned when a matrix statement is executed. (See Redimensioning and Matrix Operations, section 8.) DIM is not executable; the program is not affected if DIM is encountered during normal program execution.

Arrays passed as arguments to the INV function are limited to 100 times 100 elements. (See INV function, section 8.) In all other cases, the number of dimensioned array elements is limited only by the amount of available memory. Figure 3-6 illustrates use of the DIM statement to define arrays and to reserve space for each of the declared array elements. The examples presented in figure 3-6 assume that subscripting begins with element 0.

- **100 DIM X$(5,5), B3(1,2), X1(50)**

  **This statement reserves space for:**

  **X$** A two-dimensional string array with 6 times 6, or 36 elements.

  **B3** A two-dimensional numeric array with 6 elements.

  **X1** A one-dimensional numeric array with 51 elements.

- **50 DIM G2(5,6,7), A0(9,2), P$(2,3)**

  **This statement reserves space for:**

  **G2** A three-dimensional numeric array with 6 times 7 times 8, or 336 elements.

  **A0** A two-dimensional numeric array with 10 times 3, or 30 elements.

  **P$** A two-dimensional string array with 12 elements.

**NOTE**

**Each element of a numeric array requires one computer word. Each element of a string array requires 1 + n computer words where n is a function of the number of 6-bit characters currently assigned to the string. If the number of characters is zero, n=0. If the number of characters is nonzero, n=INT ((number of 6-bit characters +11) /10) + 1.**

Figure 3-6. DIM Statement Examples

# PROGRAM COMMENTS

Program comments in a BASIC program are indicated by using the REM statement or by appending statements with tail comments. Table 3-3 summarizes the effects and usage of the REM statement and the tail comment.

## REM STATEMENT

The REM statement is used to insert explanatory remarks or comments into a program. REM is a non-executable statement and, therefore, has no effect on program execution. The format of the REM statement appears in figure 3-7. Figure 3-8 shows some examples of the REM statement.

If control reaches, or is transferred to, a REM statement, the next executable statement following the REM statement is executed. In the following example, if A is equal to 10, control is transferred to the REM statement and the next executable statement becomes 40.

```
10 IF A=10 GOTO 30
20 PRINT "A=AVERAGE"
30 REM TEST FOR SECOND AVERAGE
40 IF B=20 PRINT "B=AVERAGE2"
```

TABLE 3-3. REM STATEMENT AND TAIL COMMENT

| Statement | Effect | Usage |
|---|---|---|
| REM | Adds comments to a program without affecting execution. | REM SOLVE FOR Y |
| Tail Comment | Adds comments to the end of BASIC statements. | 'Beginning of subroutine A |

REM $ch_1 \ldots ch_n$

$ch_1 \ldots ch_n$ Any comment or explanatory character string within the 150-character total statement length limitation; comments can be continued on additional REM statements.

Figure 3-7. REM Statement Format

```
100 REM M EQUALS MASS IN GRAMS
110 REM V EQUALS VELOCITY IN CM/SEC.
120 REM T EQUALS KINETIC ENERGY
```

Figure 3-8. REM Statement Examples

## TAIL COMMENTS

An alternate form of a comment is the tail comment. A tail comment can be added to the end of any BASIC statement, except DATA and image, by adding an apostrophe (') before the comment. For example:

```
100 LET F = 1000 'F IS FIXED COSTS
```

An apostrophe always indicates the beginning of a tail comment, except when it appears in a quoted string, a DATA statement, or an image statement.

# PROGRAM TERMINATION

To terminate a program, use either the END statement or the STOP statement. Table 3-4 shows the purpose of these two statements.

## STOP STATEMENT

The STOP statement can be used anywhere in a BASIC program to cause an immediate exit from the program. When the STOP statement is encountered, program execution terminates at that particular point, and control is returned to the operating system. Figure 3-9 shows the format of the STOP statement.

TABLE 3-4. END AND STOP STATEMENTS

| Statement | Purpose |
|---|---|
| STOP | Terminates program execution. |
| END | Marks physical end of a source program and terminates execution. |

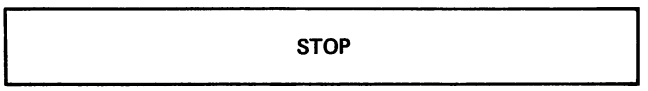


Figure 3-9. STOP Statement Format

The STOP statement is equivalent to an unconditional GOTO statement that specifies the line number of an END statement.

In the following example, the STOP statement causes program execution to terminate if A1 is less than zero; if A1 is greater than or equal to zero, program execution continues until the END statement is encountered.

```
100 IF A1<0 GOTO 120
110 IF A1>=0 GOTO 130
120 STOP
130 PRINT "VALUE IS SUFFICIENT."
.
.
.
999 END
```

## END STATEMENT

The END statement signals the end of a BASIC program; if control reaches the END statement during program execution, the program terminates as if a STOP statement had been executed. If used, the END statement must be the last statement in the program. The format of the END statement appears in figure 3-10.

The END statement is optional, but it should be used in programs because future versions of BASIC might require its use. See the Future System Migration Guidelines, appendix E.

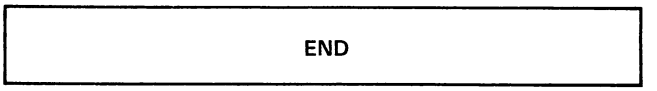


Figure 3-10. END Statement Format

This section describes control statements of the language that are used to change the sequence of execution of statements, to test and branch on a condition, to perform loops, and to monitor and control errors and interrupts.

## TEST AND BRANCH STATEMENTS

Testing and branching to certain points in a program is accomplished with the GOTO, the ON GOTO, the IF, and the IF...THEN...ELSE statements. Table 4-1 defines the test and branch statements and their effects in a program. Further details of these statements follow table 4-1.

### GOTO STATEMENT

The GOTO statement unconditionally transfers control from one point in the program to another, thereby interrupting the normal sequence of instructions. The format for this statement is shown in figure 4-1.

GOTO specifies that the statement at the referenced line number is to be executed next. Normal sequential execution continues from that point. If a GOTO statement references a nonexecutable statement, such as a DIM statement, execution continues with the first executable statement that follows the referenced nonexecutable statement.

Since the GOTO statement unconditionally causes control to be transferred to the specified line number, care must be taken that this does not set up an infinite loop.

For example, consider the program in figure 4-2. When this program is executed, it cycles continuously through lines 10, 20, and 30, and never reaches the END statement at line 40. It can be terminated only by interrupting the program. (See the NOS Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the INTERCOM Version 5 reference manual.) Inserting an IF statement before the GOTO (25 IF X=100 GOTO 40) provides an exit. When the value of X equals 100, the IF statement branches to line 40 and automatically terminates the program. The IF statement is described later in this section.

```
GOTO ln

ln          Indicates line number.
```

Figure 4-1. GOTO Statement Format

```
10 LET X=X+1
20 INPUT X
30 GOTO 10
40 END
```

Figure 4-2. Infinite Loop

TABLE 4-1. TEST AND BRANCH STATEMENTS

| Statement | Effect | Usage |
|---|---|---|
| GOTO | Unconditionally transfers control to a specified statement. | GOTO 50 |
| ON GOTO | Transfers control to one of a group of statements depending on the integer value specified in the ON GOTO statement. | ON A/3 GOTO 50,60 |
| IF | Tests a relationship or a group of relationships. If the test is true, control moves to a referenced program statement; otherwise, control falls through to the next executable statement. | IF A=20 THEN 80 |
| IF THEN ELSE | An extension of the IF statement that specifies an action for both true and false conditions. | IF A<0 THEN 50 ELSE 100 |

## ON GOTO STATEMENT

The ON GOTO statement provides for conditional branching depending on the value of an expression. The expression is evaluated and rounded to an integer value. Then control is transferred to $ln_1$ if ne is equal to 1; to $ln_2$ if ne is equal to 2; and so forth. If the value of the expression is negative, zero, or greater than the number of line numbers specified, an execution diagnostic ON EXPRESSION OUT OF RANGE is issued. Figure 4-3 illustrates the formats for the ON GOTO statement. The second format should not be used because it might not be supported in future versions of BASIC. See the Future System Migration Guidelines, appendix E.

1. ON ne GOTO $ln_1, ln_2, ln_3, \ldots, ln_n$

or

2. ON ne THEN $ln_1, ln_2, ln_3, \ldots, ln_n$

| | |
|---|---|
| ne | Indicates numeric expression. |
| ln | Indicates line number. |

Figure 4-3. ON GOTO Statement Format

In figure 4-4, SGN(A) can have the value -1, 0, or 1. The expression SGN(A)+2 can have the value 1, 2, or 3, and control transfers 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, and the next logical statements.

```
    .
    .
    .
095 ON SGN(A)+2 GOTO 100,110,120
100 LET A=A*A
105 GOTO 130
110 LET A=A*B
115 GOTO 130
120 LET A=A*B ^ 2
130 LET B=A+1
    .
    .
    .
```

Figure 4-4. Example of ON GOTO and GOTO Statements

## IF STATEMENT

The IF statement tests conditions and controls the sequence of operations. The formats for the IF statement are shown in figure 4-5. If the relational expression r is true, the program transfers control to the statement at line number ln, if format 1 is used, and executes statement stm, if format 2 is used. Do not use the format GOTO because it might not be supported in future versions of BASIC. See the Future System Migration Guidelines, appendix E. If the relation r is false, the next sequential statement is executed. Examples of simple IF...THEN clauses are shown in figure 4-6.

1. IF r THEN ln (or) IF r GOTO ln

2. IF r THEN stm

| | |
|---|---|
| r | Indicates simple or compound relational expression. |
| ln | Indicates line number. |
| stm | Indicates executable BASIC statement. |

Figure 4-5. IF Statement Format

```
20 IF 2*I >= J ^ 2-1 THEN 165
```

Assuming I = 8 and J = 4, the value 16 is compared to the value 15; the evaluation is true, the next statement executed is at line number 165.

```
15 IF I = J OR NOT J < I THEN 140
```

Assuming I = 8 and J = 4, the relation I = J is false. The relation J < I is true; however, NOT J < I is false. The compound relational expression evaluates to false (false or false is false) and the branch to statement 140 is not made.

```
25 IF A <> 0 THEN LET B = 0
```

This statement causes B to be set to 0 if A is not equal to 0. The next statement in sequence is then executed. If A = 0 the next statement in sequence is executed but the LET B = 0 is not.

Figure 4-6. IF Statement Examples

The stm parameter can contain any executable statement other than a FOR or a NEXT statement. The nonexecutable statements OPTION, DATA, DEF, DIM, END, FNEND, image, and REM are not allowed in the stm parameter.

Multiple IF...THEN clauses can be embedded within a single IF statement to perform various kinds of conditional tests, as shown in figure 4-7. The maximum number of IF...THEN clauses is governed only by the 150 character line width limitation. The IF statement in figure 4-7 contains two IF...THEN clauses to test for a zero value in each of the numeric variables A and B. If both A and B are zero, C is assigned the value 14. If neither A nor B is zero, C is not assigned the value 14.

When the IF statement contains multiple IF...THEN clauses, the clauses are tested consecutively, beginning with the first clause.

```
    .
    .
    .
030 IF A=0 THEN IF B=0 THEN LET C=14
    .
    .
    .
```

Figure 4-7. Nested IF...THEN Statement Example

## IF... THEN... ELSE STATEMENT

The IF...THEN...ELSE statement, an extension of the simple IF statement, enables execution to continue at a specified line number when the IF condition is false, as well as when it is true. The formats for this form of the IF statement are shown in figure 4-8. Do not use format 2 because it might not be supported in future versions of BASIC. See Future System Migration Guidelines, appendix E.

1. IF r THEN $ln_1$ or $stm_1$ ELSE $ln_2$ or $stm_2$
2. IF r GOTO $ln_1$ ELSE $ln_2$ or $stm_2$

| | |
|---|---|
| r | Indicates simple or compound relational expression. |
| ln | Indicates line number. |
| stm | Indicates executable BASIC statement. |

Figure 4-8. IF...THEN...ELSE Statement Format

In figure 4-8, if the relational expression r is true, the program transfers control to line number $ln_1$ or it executes statement $stm_1$, depending upon which was specified in the statement. If the relational expression is false, statement $stm_2$ or the statement at $ln_2$ is executed. The statements $stm_1$ and $stm_2$ can be any executable BASIC statements except FOR and NEXT. They cannot be nonexecutable statements such as OPTION, DATA, DEF, DIM, END, FNEND, image, or REM.

Figure 4-9 contains examples of the IF...THEN...ELSE statement.

```
1.  IF A<0 THEN 150 ELSE 160

2.  IF A$="STOP" OR A$="END" THEN STOP
    ELSE 100

3.  IF X=0 THEN LET Y=0 ELSE LET Y=Y/X

4.  IF A=0 THEN IF B=0 THEN PRINT 1
    ELSE PRINT 2 ELSE PRINT 3

5.  IF A=0 THEN IF B=0 THEN PRINT 1
    ELSE PRINT 2

6.  IF A=0 THEN GOSUB 500 ELSE IF B=0
    THEN GOSUB 600 ELSE LET B=3
```

Figure 4-9. IF...THEN...ELSE Statement Examples

Example 1 causes control to go to line number 150 when A is less than zero, and to line number 160 when A is not less than zero.

Example 2 causes the program to stop when the string variable A$ contains either STOP or END, and to transfer control to line 100 when A$ contains any other value.

Example 3 sets Y to zero when X is equal to 0, and sets Y to Y/X when X is not equal to 0. In either case, control falls through to the next sequential statement after Y has been assigned a value.

Example 4 causes the number 1 to be printed if both A and B are equal to zero, the number 2 to be printed if A is equal to 0 but B is not equal to 0, and the number 3 to be printed if A is not equal to 0 (the value of B is not tested in this case). When IF...THEN...ELSE statements are nested, as in this example, the inside ELSE belongs to the inside IF...THEN, the next level ELSE (the second one seen when reading from left to right) belongs to the next level IF...THEN, and the levels continue in this progression (similar to the way nested parentheses are paired). It is not necessary to have as many ELSEs as IF...THENs however, and outside IF...THEN...ELSEs are executed first.

Example 5 causes 1 to be printed if both A and B are equal to 0, 2 to be printed if A is equal to zero but B is not, and nothing to be printed if A is not equal to zero. In all cases, the next statement in sequence is executed next. There is no ELSE clause for IF A = 0 THEN. It is a simple IF...THEN with an IF...THEN...ELSE as its consequent statement.

In example 6, if A is equal to 0, the subroutine at line 500 is executed, and control returns to the next sequential statement following the IF. If A is not equal to 0, the relation B is equal to 0 is tested. If B is equal to 0, the subroutine at line 600 is executed, and control returns to the next sequential line. If B is not equal to 0, it is set to 3, and control falls through to the next sequential statement. Neither the subroutine at line 500 nor the subroutine at line 600 is executed in this case. In this example, the consequence of the ELSE in the first IF...THEN...ELSE is an IF...THEN...ELSE statement.

# LOOPING

Looping, the repetitive execution of the same statement or statements, can be efficiently controlled in BASIC with the FOR and NEXT statements. Table 4-2 summarizes these looping statements and their effect in a program.

## FOR... NEXT STATEMENTS

The FOR statement initiates repeated looping through the statements that physically follow the FOR statement, up to and including a corresponding NEXT statement. The FOR statement must appear as the first statement of the loop, and the NEXT statement must be the last statement of the loop. The format of the FOR...NEXT statements is illustrated in figure 4-10.

TABLE 4-2. LOOPING STATEMENTS

| Statement | Effect | Usage |
|---|---|---|
| FOR | Marks the beginning of a loop and initiates its execution. | FOR I=1 TO 10 |
| NEXT | Marks the end of the FOR loop; tests for end-of-loop condition and reexecutes or terminates depending on the results. | NEXT I |

1. **FOR snv = $ne_1$ TO $ne_2$ STEP $ne_3$**

   **(or)**

2. **FOR snv = $ne_1$ TO $ne_2$**

   **NEXT snv**

| | |
|---|---|
| snv | Indicates simple numeric variable (called the control variable; it must be identical in both statements). |
| $ne_1$ | Indicates any arithmetic expression (called the initial value). |
| $ne_2$ | Indicates any arithmetic expression (called the final value). |
| $ne_3$ | Indicates any arithmetic expression (called the step value). |

Figure 4-10. FOR...NEXT Statement Formats

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.

The STEP value can be positive or negative. For a positive STEP value, the initial value must be less than the final value upon entrance to the loop. Similarly, for a negative STEP value, the initial value must be greater than the final value. If either condition is not met, the loop does not execute, and control branches to the statement following the NEXT statement. Figure 4-11 illustrates a loop with a specified STEP value of +2. Execution of the loop in figure 4-11 causes the values 1, 3, 5, 7, 9, and 11 to be printed. Statements 20 through 30 are repeated six times, once for each value assigned to X.

```
    .
    .
    .
010 FOR X=1 TO 11 STEP 2
020 PRINT X
030 NEXT X
040 END
```

Figure 4-11. Loop With Specified STEP Value

The initial, final, and STEP expressions are evaluated only once (upon 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, however, can be changed by statements within the loop; its last value is always adjusted by the STEP value and is used in comparison to the final value, as shown in figure 4-12. Execution of the loop in figure 4-12 causes the values 2, 4, 6, 8, and 10 to be printed. Even though the FOR statement specifies that the control variable X be incremented by an implicit STEP value of +1 until it exceeds 10, the LET statement adds 1 to X, thereby causing the control variable to be incremented by 2 for each pass through the loop. Thus, the value of the control variable can be changed by statements within the loop.

```
    .
    .
    .
010 FOR X=1 TO 10
020 LET X=X+1
030 PRINT X
040 NEXT X
050 END
```

Figure 4-12. Control Variable Value Changed

After a loop has repeated itself the specified number of times, the final value of the control variable is the first value not used. That is, upon normally exiting from a loop, the control variable assumes its final value plus an additional STEP value (+1 when a STEP value is not specified). Using a control statement, such as GOTO, to prematurely terminate a loop causes the control variable to retain the value it has when the control statement is executed. Figure 4-13 shows the effect that a normal exit from a loop has on the control variable. The X in line number 120 assumes the value of 1, 3, 5, 7, 9, and 11, and the X in line number 140 assumes the value 13.

```
    .
    .
    .
110 FOR X=1 TO 11 STEP 2
120 PRINT X
130 NEXT X
140 PRINT X
150 END
```

Figure 4-13. Loop Exit Effect on Control Variable

Loops can be nested (loops specified within loops) to a maximum depth of 10, but the loops must not intersect each other. Examples of correct and incorrect looping are shown later in this section.

A loop can contain a GOTO statement 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. Do not transfer control into a FOR...NEXT loop. See Future System Migration Guidelines, appendix E.

Figure 4-14 shows the effect of the FOR statement on control variables. The loop initiated in line number 112 did not execute because the initial value is not greater than the final value, and the step value is negative. Figure 4-15 shows examples of correct and incorrect looping.

| Statement | Values |
|---|---|
| 110 FOR X = 2 to 4 | 2, 3, 4, 5 |
| 111 FOR G = 6 TO 3 STEP -1 | 6, 5, 4, 3, 2 |
| 112 FOR X = 5 TO 10 STEP -1 | 5 |

Figure 4-14. FOR...NEXT Statement Examples

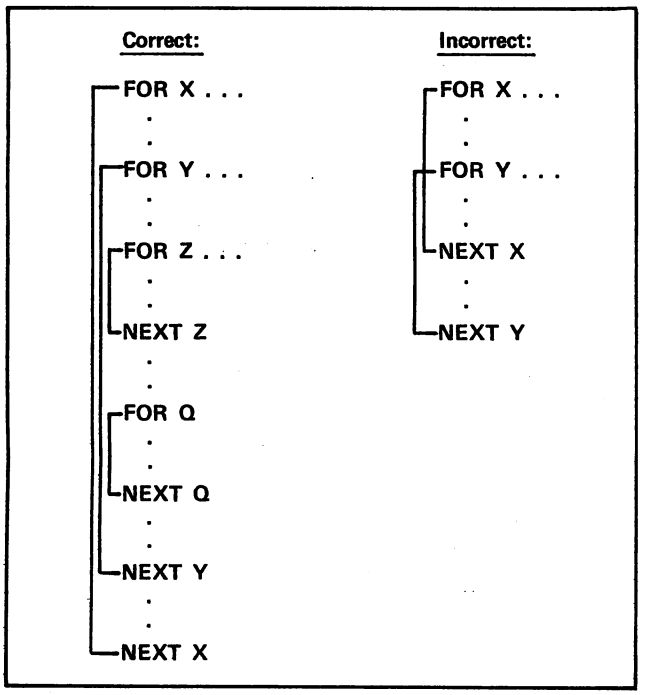


Figure 4-15. FOR...NEXT Loops

# ERROR AND INTERRUPT PROCESSING

Three statements (ON ATTENTION, ON ERROR, and JUMP) and four functions (ASL, ESL, ESM, and NXL) are used to detect and control processing errors and the terminal interrupts that occur during program execution. Table 4-3 defines these error and interrupt processing statements and functions and their effects in a program.

## ON ATTENTION STATEMENT

The ON ATTENTION statement establishes an address for interrupt handling if an interrupt occurs during program execution. It enables the program to respond to terminal interrupts. Figure 4-16 shows the format for the ON ATTENTION statement.

1. ON ATTENTION GOTO ln
2. ON ATTENTION THEN ln
3. ON ATTENTION

ln Indicates line number.

Figure 4-16. ON ATTENTION Statement Formats

Formats 1 and 2 in figure 4-16 specify that control is to be transferred to statement ln if a terminal interrupt occurs during execution. The ON ATTENTION statement is used in conjunction with the attention statement line (ASL) function.

The statements at ln can use the ASL function to determine where the terminal interrupt occurred. After the interrupt location is determined, appropriate action to process the terminal interrupt can be taken; execution can be reinitiated at any point in the program.

Normal terminal interrupt processing is suppressed and the program can gain control of interrupts by the execution of format 1 or format 2 of the ON ATTENTION statement. Normal interrupt processing is restored (ON ATTENTION turned off) if format 3 of the ON ATTENTION statement is executed, or if format 1 or 2 of the ON ATTENTION statement has been executed and a terminal interrupt, which transfers control to statement ln, is entered.

The first ON ATTENTION statement should appear at, or near, the start of the program. When an ON ATTENTION statement is encountered during compilation, each following statement is changed to test for interrupts before execution. During execution, interrupts are normally recognized at the start of a statement so that statements are not partially executed. The exceptions are terminal I/O statements. Interrupts are recognized when an INPUT statement is waiting for data. When an input request is interrupted before any values have been entered, none of the variables on the input list are modified. Interrupts are also recognized at each line of output. However, many lines of output processing usually occur before data appears at the terminal, therefore some data lines could appear after the terminal is interrupted.

TABLE 4-3. ERROR AND INTERRUPT PROCESSING (STATEMENTS AND FUNCTIONS)

| Statement | Effect | Usage |
|---|---|---|
| ON ATTENTION | Establishes an address for interrupt handling if an interrupt occurs. | ON ATTENTION GOTO 100 |
| ON ERROR | Establishes an address for error handling if an error occurs. | ON ERROR GOTO 100 |
| JUMP ne | Transfers control to a program statement depending upon the value of an expression. | JUMP 100 |
| ASL(x) | Returns the line number of the statement at which the last terminal interrupt occurred. | ASL(Y) |
| ESL(x) | Returns the line number of the statement that caused the most recent program execution error to occur. | ESL(Z) |
| ESM(x) | Returns the error number associated with the most recent program execution error. | ESM(C) |
| NXL(ne) | Returns the line number of the statement where the program execution is to resume. | NXL(25) |

Figure 4-17 shows how the ON ATTENTION statement can be used to control terminal interrupts. In the sample dialog of this figure, the programmer mistakenly types in 443, then presses the appropriate key to cause a terminal interrupt. The ON ATTENTION statement establishes line number 900 as the ON ATTENTION address, and control transfers to line 900, at which point the program offers a choice of recovery options. The programmer enters NEXT ITEM and, when the NEXT ITEM or zero is requested, the corrected value of 444 is entered and the dialog continues normally.

## ON ERROR STATEMENT

The ON ERROR statement is used for detecting and processing errors that occur during program execution. Figure 4-18 illustrates the three formats for the ON ERROR statement.

Formats 1 and 2 specify that control is to transfer to statement ln if an execution error occurs. The ON ERROR statement, in conjunction with the error statement line number (ESL) function (explained later in this section) and the error statement message (ESM) function (explained later in this section), allows the BASIC program to respond to execution errors.

After the error location and error type are determined, appropriate action to process the error can be taken; execution can then be reinitiated at any point in the program. The program should not give control to a CHAIN statement that chains to a copy of the original program. An endless loop can result if the error that caused the ON ERROR branch recurs each time the program is executed.

Normal error processing is suppressed (allowing the program to continue executing at the ON ERROR address) by the execution of format 1 or 2. Normal error processing is restored after format 3 of the ON ERROR statement has been executed or after format 1 or format 2 of the ON ERROR statement has been executed and control has been transferred to the specified statement ln as a result of an error.

It is possible to recover from any execution error; however some errors, such as time limit and operator drop, can only be recovered from one time. (See appendix B.) An example of the ON ERROR statement is shown later in this section.

## JUMP STATEMENT

The JUMP statement transfers control to the statement at the line number determined by the value of an arithmetic expression. The format for the JUMP statement is illustrated in figure 4-19.

```
    .
    .
    .
100 ON ATTENTION GOTO 900
    .
    .
    .
200 PRINT "ENTER NEXT ORDER NUMBER OR 0"
210 INPUT N
220 IF N=0 GOTO 500          '0 MEANS END OF ORDERS
    .
    .
    .
300 PRINT "ENTER NEXT ITEM NUMBER OR 0"
310 INPUT I
320 IF I=0 GOTO 400          '0 MEANS END OF ITEMS
330 PRINT "ENTER QUANTITY"
340 INPUT Q
    .
    .
    .
900 Z=ASL(0)                 'Z IS LINE NUMBER AT WHICH TO CONTINUE
910 ON ATTENTION GOTO 910    'RESET SO INTERRUPT WILL NOT CHANGE Z
920 PRINT "INTERRUPTED AT LINE";Z;", LAST ORDER ";N;", LAST ITEM";I
930 PRINT "TYPE STOP, NEXT ORDER, NEXT ITEM, OR CONTINUE"
940 INPUT Z$
950 IF Z$="STOP" THEN STOP
960 ON ATTENTION GOTO 900    'RE-ENABLE AT ORIGINAL LINE NUMBER
970 IF Z$="NEXT ORDER" THEN GOTO 200
980 IF Z$="NEXT ITEM" THEN GOTO 300
990 IF Z$="CONTINUE" THEN JUMP Z
995 GOTO 910                 'INVALID RESPONSE.  REPEAT QUESTION
999 END




    .
    .
    .
ENTER NEXT ITEM NUMBER OR 0
? 443
ENTER QUANTITY
? ATTN†
INTERRUPTED AT LINE 340 , LAST ORDER 6087 , LAST ITEM 443
TYPE STOP, NEXT ORDER, NEXT ITEM, OR CONTINUE
? NEXT ITEM
ENTER NEXT ITEM NUMBER OR 0
? 444
ENTER QUANTITY
? 2
    .
    .
    .
```

†The key that initiates an interrupt varies with the operating system and the terminal mode. Consult the appropriate reference manual for this information.

Figure 4-17. ON ATTENTION Statement Example

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

ln Indicates line number.

Figure 4-18. ON ERROR Statement Formats

JUMP ne

ne Indicates numeric constant, variable, or expression.

Figure 4-19. JUMP Statement Format

The expression is evaluated and rounded 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. Care must be taken not to jump into a FOR loop or function definition.

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

Figure 4-20 illustrates a program example using the ON ERROR and JUMP statements, and the ESL, ESM, and NXL functions (described later in this section). The following sentences explain the meaning of specific statements in this program. Execution of statement 100 suppresses normal error processing and ensures that on a subsequent error, control will be transferred to statement line number 160. If an error occurs in reading the data (line number 120), control is transferred to statement line number 160. Normal error processing is then reinstated. If another error occurs during further execution, the program aborts. In statement line numbers 160 and 170, value 120 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. The statement at line number 180 indicates that if an error occurred in statement line number 120, execution control is transferred to statement line number 210. A jump is made from statement line number 220 to statement line number 130 and normal execution continues. If another error occurs during execution, the job aborts.

## ASL FUNCTION

The ASL function returns the statement line number of the statement executing, or about to be executed, when the most recent terminal interrupt occurred. Figure 4-21 shows the format for the ASL function.

| ASL(x) | |
|---|---|
| x | Indicates a dummy variable. |

Figure 4-21. ASL Function Format

The ASL function returns a value of -1 if a terminal interrupt has not occurred since the most recent execution of an ON ATTENTION statement. Thus, when processing terminal interrupts, the function value should be saved before issuing another ON ATTENTION statement.

In the statement 100 LET A=ASL(x), A is assigned the line number of the statement that was to be executed when the terminal interrupt occurred. For an additional example of the ASL function, refer to the ON ATTENTION statement example (figure 4-17).

## ESL FUNCTION

The ESL function returns the line number of the statement that caused the most recent program execution error. The format for the ESL function appears in figure 4-22.

```
100 ON ERROR GOTO 160
110 PRINT "READ ERROR WILL BE PROCESSED BY PROGRAM"
120 READ X1,X2,X3
130 PRINT "VALUES READ WERE ";X1;",";X2;",AND";X3
140 STOP
150 REM ERROR PROCESSING ROUTINE
160 LET X=ESL(X)
170 LET Y=ESM(X)
180 IF X=120 THEN 210
190 PRINT "ERROR NOT IN STATEMENT 120"
200 STOP
210 PRINT "ERROR NUMBER #";Y;"DETECTED AT LINE #";X
220 JUMP NXL(X)
230 DATA 2.0,3.0,"STRING"
240 END


produces:


READ ERROR WILL BE PROCESSED BY PROGRAM
ERROR NUMBER # 126 DETECTED AT LINE # 120
VALUES READ WERE  2 , 3 ,AND 0
```

Figure 4-20. Example Using ON ERROR, JUMP, ESL, ESM, and NXL

| ESL(x) | |
|---|---|
| x | Indicates a dummy variable. |

Figure 4-22. ESL Function Format

The ESL function returns a value of -1 if an execution error has not occurred since the most recent execution of an ON ERROR statement. Thus, when processing errors, the function value should be saved prior to issuing another ON ERROR statement.

In the statement 100 LET A=ESL(x), A is assigned the line number of the statement that caused the error. For an additional example of the ESL function, see figure 4-20.

## ESM FUNCTION

The ESM function returns the error number associated with the most recent program execution error. (See table B-3 in appendix B for a list of error numbers associated with execution errors.) Figure 4-23 shows the format for the ESM function.

The ESM function returns a value of -1 if an execution error has not occurred since the most recent execution of the ON ERROR statement. Thus, this function value should be saved prior to issuing another ON ERROR statement.

In the statement 100 LET A=ESM(x), the error number of the error is assigned to A. For an additional example of the ESM function, see figure 4-20.

| ESM(x) | |
|---|---|
| x | Indicates a dummy variable. |

Figure 4-23. ESM Function Format

## NXL FUNCTION

This function returns the next line number of the statement that follows the line number specified in the argument of the NXL function. Use NXL to determine at what point program execution is to resume in the event of an error or interrupt.

For example, NXL(ESL(x)) returns the line number of the line following the one that caused the error. The format for the NXL function appears in figure 4-24.

| NXL(ne) | |
|---|---|
| ne | Indicates numeric constant, variable, or expression. |

Figure 4-24. NXL Function Format

The NXL function should not refer to a REM statement or to any nonexisting statement because such referencing can cause program execution to terminate (unless ON ERROR is in effect) and the diagnostic ILLEGAL LABEL to be issued. The function argument ne is evaluated and rounded to an integer value. For an example of the NXL function, see figure 4-20.

A function is a predefined algorithm. A function returns a value to the point of reference each time the function is invoked from an executing program.

Two kinds of functions are provided with BASIC: the predefined functions of the language, called built-in functions; and the functions that can be written by using the DEF and FNEND statements, called user-defined functions. The built-in functions are in the form of subset programs written to perform specific kinds of tasks.

The built-in functions and user-defined functions are classified as follows:

Built-in functions:

- Mathematical functions
- System functions
- String functions
- Matrix functions
- Error and interrupt processing functions
- I/O functions

User-defined functions:

- Single-line functions
- Multiple-line functions

Although all of the built-in functions are defined in this section, some of the functions are described in more detail in other sections of this manual. The seven tables in this section identify the built-in functions and indicate their functional classification. See the Summary Card at the end of this manual for a complete alphabetical listing of the built-in functions. (See the table of contents for specific section references.) The user-defined functions are described at the end of this section.

## REFERENCING A FUNCTION

Built-in and user-defined functions are referenced by specifying a function name followed by associated function parameters in parentheses. If no parameters are used in the function definition, no parameters are needed in the function reference. The form for a function reference is shown in figure 5-1.

function name $(e_1,e_2, \ldots ,e_n)$

e — Indicates numeric or string expression; parameter is optional.

Figure 5-1. Function Reference Format

The number and type of parameters (e) passed with a function reference must exactly correspond to the number and type of parameters expected by the function; for example, a string must be passed where a string is expected and a number must be passed where a number is expected. A diagnostic is issued if the type and number of parameters contained in the function reference do not correspond to those expected in the definition.

Built-in function parameters that are integer quantities use the value of the numeric expression rounded to an integer. User-defined functions cannot specify that a parameter is an integer. With user-defined functions, all numeric values are real numbers and the function either truncates or rounds values to integers, depending upon the written statement. Function reference parameters are evaluated and the values of the parameters are passed to the function. The function is then evaluated and the result is returned to the point of the function reference.

## MATHEMATICAL FUNCTIONS

Table 5-1 is an alphabetical list of the standard mathematical functions that can be referenced by a BASIC program. In this table, the function argument ne can be a numeric expression of any complexity and can include other function references.

Figure 5-2 shows an example of the ABS and the SQR mathematical functions. The absolute value of -71 is multiplied by the square root of 520.

```
10 LET C=ABS(-71)
20 PRINT C
30 LET D=SQR(520)
40 PRINT D
50 LET T=C*D
60 PRINT "ABS(-71)*SQR(520)=";T
70 END


produces:


 71
 22.8035
ABS(-71)*SQR(520)= 1619.05
```

Figure 5-2. ABS and SQR Functions Example

### RANDOM NUMBER GENERATION

The generation of pseudo random numbers is controlled by the RND function and by the RANDOMIZE statement. The RANDOMIZE statement overrides the predefined sequence of numbers generated by RND.

TABLE 5-1. MATHEMATICAL FUNCTIONS

| Function | Description |
|---|---|
| ABS(ne) | Finds the absolute value of ne. |
| ATN(ne) | Finds the arctangent of ne in the principal value range ($-\pi/2$) to ($+\pi/2$). |
| COS(ne) | Finds the cosine of ne; the angle ne is expressed in radians. |
| COT(ne) | Finds the cotangent of ne; the angle ne is expressed in radians. |
| DET(m) | Returns the determinant of matrix m. |
| DET | Returns the determinant of the last matrix inverted by the matrix function INV. |
| EXP(ne) | Finds the value of e to the power of ne. |
| INT(ne) | Finds the largest integer not greater than ne.<br>Example: INT(5.95) = 5 and INT(-5.95) = -6. |
| LGT(ne) | Finds the base 10 logarithm of ne; ne must be greater than zero. |
| LOG(ne) | Finds the natural logarithm of ne; ne must be greater than zero. |
| MAX($ne_1$,$ne_2$,...,$ne_{20}$) | Returns the maximum value from the list of parameters; 2 to 20 parameters can be used. |
| MIN($ne_1$,$ne_2$,...,$ne_{20}$) | Returns the minimum value from the list of parameters; 2 to 20 parameters can be used. |
| RND or RND(ne) | Returns a pseudo random number from the set of numbers uniformly distributed over the range of 0 to 1. See the description and examples in this section. |
| ROF(ne) or ROF($ne_1$,$ne_2$) | Finds the value of $ne_1$ rounded to $ne_2$ decimal places. If ne is omitted, then ne is rounded to the nearest integer. Some exact decimal fractions cannot be represented exactly in fixed word length binary computers. Results of the ROF function, therefore, may differ by one digit from results obtained from rounding by hand. |
| SGN(ne) | Interrogates the sign of ne and returns a value of 1 if ne is positive; 0 if ne is 0; or -1 if ne is negative. |
| SIN(ne) | Finds the sine of ne; the angle ne is expressed in radians. |
| SQR(ne) | Finds the square root of ne; ne must be $\geq 0$. |
| TAN(ne) | Finds the tangent of ne; the angle ne is expressed in radians. |

## RND FUNCTION

The RND function returns a pseudo random number from the set of numbers uniformly distributed over the range of 0 to 1. The formats for the RND function are shown in figure 5-3. Do not use the second format because it might not be supported in future versions of BASIC. (See Future System Migration Guidelines, appendix E.)

RND is equivalent to RND(0) in that it returns a value in the established sequence of pseudo random numbers uniformly distributed over the range of 0 to 1. Random numbers are returned in the same sequence each time the program containing RND is executed unless the RANDOMIZE statement is used to override the predefined sequence. RANDOMIZE affects RND(0). RND(ne) affects RND, if ne>0. The RANDOMIZE statement and its effect on random number generation is discussed in more detail later in this section.

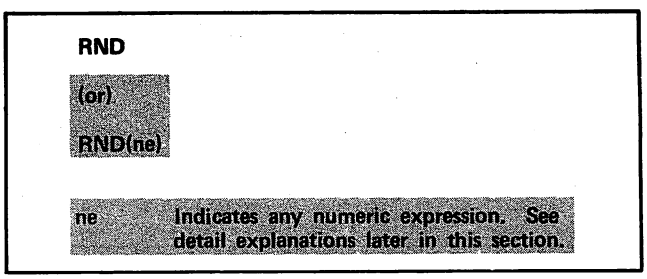


Figure 5-3. RND Function Format

An example of the RND function is shown in figure 5-4. The program was executed twice. The RND function twice returned the same set of pseudo random numbers. An example later in this section shows this same program with the RANDOMIZE statement that ensures that a different sequence of pseudo random numbers is generated each time the program is executed.

```
100 FOR T=1 TO 3
110 L=RND
120 E=RND
130 I=RND
140 PRINT L,E,I
150 NEXT T
160 END

produces:

 .580114        .950513        .786371
 .29762         .4537          6.26194E-3
 .275736        .305651        .689101

produces:

 .580114        .950513        .786371
 .29762         .4537          6.26194E-3
 .275736        .305651        .689101
```

Figure 5-4. RND Function Example

The value of ne in RND(ne) affects random number generation as follows:

ne>0 A random number sequence is initialized based on the value of ne, and the first number in the sequence is returned. Each reference to RND with ne 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 the ne>0 arguments are used. If ne>0, RND(ne) can affect RND without the argument.

ne=0 The next number in the established sequence of pseudo random numbers is returned. If the sequence was not previously established by an ne>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 you initialize the sequence with a different positive (>0) value each time the program executes. This can be done by using a first reference, such as RND(CLK(0)). CLK(0) returns the time-of-day. If ne=0, RANDOMIZE affects RND(ne).

ne<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 ne<0 return the next number in the sequence. A program that uses ne<0 returns a different value on each reference and a different sequence each time it is run. The sequence initialized by ne<0 is separate from the sequence controlled by ne>0, and ne=0 references to RND sequences.

## RANDOMIZE STATEMENT

The RANDOMIZE statement causes a new initial or seed value to be placed in the random number generator each time a program containing the RND function is run. The placement of this new value in the random number generator overrides the predefined sequence of pseudo random numbers generated by the RND function; therefore, the RND function returns a different sequence of values each time the program is executed. Figure 5-5 shows the format for the RANDOMIZE statement.


Figure 5-5. RANDOMIZE Statement Format

Figure 5-6 shows an example using RANDOMIZE to control random number generation. This program was executed twice. The RANDOMIZE statement causes RND to return a different sequence of values, unlike the example shown for the RND function that does not use RANDOMIZE (figure 5-3).

```
090 RANDOMIZE
100 FOR T=1 TO 3
110 L=RND
120 E=RND
130 I=RND
140 PRINT L,E,I
150 NEXT T
160 END

produces:

 .34368         .310629        .590422
 .993254        .237534        .876869
 .481367        .900958        .320888

produces:

 .463818        .82842         .977286
 .882296        .96833         6.09989E-2
 .630496        .41131         .654263
```

Figure 5-6. RANDOMIZE Statement Example

## SYSTEM FUNCTIONS

The built-in system functions are CLK$ (or CLK(x)), DAT$, TIM, and USR$. Table 5-2 defines the system functions, and figure 5-7 illustrates the use of the CLK$, DAT$, and TIM functions in a BASIC program.

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


produces:


CLK$ TIME OF 12.40.43.= 12.6786 IN CLK(X) TIME
 81/06/22.
TOTAL ELAPSED TIME IS .001
```

Figure 5-7. Program Using System Functions CLK$, DAT$, and TIM

TABLE 5-2. PREDEFINED SYSTEM FUNCTIONS

| Function | Description |
|---|---|
| CLK$† | Returns the time of day as a string constant in the form:<br><br>hh.mm.ss.<br><br>Example: 17.36.37. |
| CLK(x)† | Returns the time of day in hours and fractions of an hour in a 24-hour scale (x is a dummy argument).<br><br>Examples:<br><br>(1) Three minutes and 58 seconds past the hour of 9:00 is represented:<br><br>9.0611<br><br>(2) Midnight is represented:<br><br>0.00<br><br>(3) Noon is represented:<br><br>12.000<br><br>(4) 2:30 p.m. is represented:<br><br>14.50 |
| DAT$ | Returns the date as a string constant in the following form:<br><br>yy/mm/dd<br><br>Example: 74/08/11 |
| TIM(x) | Returns the total elapsed central processor time in seconds used-to-date in this job (x is a dummy argument). |
| USR$ | Returns the NOS 7-character user number. On NOS/BE, this function returns the string USERNUM. |

†Refer to appendix E for recommendations on the use of the CLK$ and DAT$ functions.

# STRING FUNCTIONS

The string functions provided by BASIC are used to create and/or manipulate character string data in specific ways. Strings can contain from 0 through 131070 characters in normal mode and from 0 through 65535 or 0 through 131070 characters in ASCII mode, depending on the number of escape code characters in the string. Table 5-3 is an alphabetical list of string functions.

## ASC FUNCTION

ASC returns the decimal code (ordinal position in the ASCII character set) of a character name or abbreviation in its argument. ASC returns the same result whether used in ASCII or normal mode.

The mode of the program is controlled by the AS (ASCII) parameter in the BASIC control statement and by the mode of the terminal. (See Batch Operations, section 12, and appendix A.) ASCII values range from 0 to 127. Character abbreviations are listed in table A-1 of appendix A.

The format for the ASC function is shown in figure 5-8. Use of the ASC function is not recommended; the ORD function should be used instead. For guidelines, see appendix E.

```
ASC(ch)
  or
ASC(abr)

ch      Indicates any character.
abr     Indicates abbreviation for an ASCII
        character name.
```

Figure 5-8. ASC Function Format

## CHR$ FUNCTION

This function returns the character corresponding to the decimal code (ordinal position in the collating sequence) specified in the function argument. The format for the CHR$ function appears in figure 5-9. The argument ne is evaluated and rounded to an integer.

TABLE 5-3. STRING FUNCTIONS

| Function | Description |
|---|---|
| ASC(ch) or ASC(abr) | Returns the ASCII code in decimal of the character in its argument. Use ORD instead.† |
| CHR$(ne) | Returns the character with the decimal code (ordinal position in the collating sequence) that corresponds to ne. |
| LEN(se) | Determines current length of string se. |
| LPAD$(se,ne) | Pads string se out to ne characters by adding spaces on the left of string se. |
| LTRM$(se) | Trims string se of all leading space characters. |
| LWRC$(se) | Returns a string consisting of the se string value with all uppercase letters replaced by their lowercase equivalents. |
| ORD(se) | Returns the decimal code (ordinal position) of a character in string se in the collating sequence being used. |
| POS($se_1$,$se_2$,ne) or POS($se_1$,$se_2$) | Returns the position of string $se_2$ within string $se_1$. The position search begins with character ne. If ne is omitted, 1 is assumed. |
| RPAD$(se,ne) | Pads string se to ne characters by inserting blanks on the right of string se. |
| RPT$(se,ne) | Returns the string created by repeating the se string ne times. |
| RTRM$(se) | Trims string se of all trailing space characters. |
| STR$(ne) or STR$(ne,se) | Converts numeric value to string representation. If present, se represents an image specification. |
| UPRC$(se) | Returns string se with all lowercase letters replaced by their uppercase equivalents. |
| VAL(se) | Converts string se to its numeric value. |

†See Future System Migration Guidelines, appendix E.

**CHR$(ne)**

ne — Indicates numeric constant, variable, or expression.

Figure 5-9. CHR$ Function Format

CHR$ is valid for all characters in the current collating sequence, including nonprinting characters. If the standard ASCII collating sequence is in effect, CHR$ character values exist for argument values 0 through 127; if the nonstandard display code collating sequence is in effect, CHR$ character values exist for argument values 0 through 63, or 1 through 63, depending on the character set being used by your system. (See appendix A.) OPTION COLLATE controls the collating sequence being used, and is described in the section on Fundamental BASIC Statements. A fatal error results if CHR$ argument values are outside the valid range of characters represented in the collating sequence.

This function returns the same result whether used in ASCII or normal mode if the standard ASCII collating sequence is in effect. The function returns different values if the nonstandard collating sequence is in effect. The mode of the program is controlled by the AS (ASCII) parameter in the BASIC control statement and by the mode of the terminal in the BASIC subsystem under NOS. (See Batch Operations, section 12, and appendix A.) In normal mode, 12-bit escape code characters do not exist. However, if the standard collating sequence is in effect, CHR$ returns a 12-bit escape code character for argument values of 0 through 31 and for values of 96 through 127, which are treated as two normal 6-bit characters when manipulated or printed. An example of the CHR$ function is shown in figure 5-10.

## LEN FUNCTION

The LEN function returns the current length, in characters, of the string specified by the argument in this function. Figure 5-11 shows the format for the LEN function, and figure 5-12 shows an example of how to use this function to return the length of the string S$.

```
10 REM 98 IS THE ASCII CODE FOR LOWERCASE B
15 OPTION COLLATE STANDARD
20 LET B$=CHR$(98)
30 PRINT B$;" IS";LEN(B$);"CHARACTER(S)"
40 END


In ASCII mode, produces:


b IS 1 CHARACTER(S)


In normal mode, produces:


^B IS 2 CHARACTER(S)


10 REM 98 IS THE ASCII CODE FOR LOWERCASE B
15 OPTION COLLATE NATIVE
20 LET B$=CHR$(98)
30 PRINT B$;" IS";LEN(B$);"CHARACTER(S)"
40 END


In ASCII mode produces:


b IS 1 CHARACTER(S)


In normal mode, produces:


 ILLEGAL CHR$ ARG  AT  20
BASIC EXECUTION ERROR
```

Figure 5-10. CHR$ Function Example

**LEN(se)**

| | |
|---|---|
| se | Indicates a string constant, variable, or expression. |

Figure 5-11. LEN Function Format

```
100 LET S$="543"
110 LET A=LEN(S$)
120 PRINT A
130 END


produces:


 3
```

Figure 5-12. LEN Function Example

## LPAD$ FUNCTION

The LPAD$ function pads a string out to a specified number of characters by inserting blanks on the left (called left pad). Figure 5-13 shows the format for the LPAD$ function.

**LPAD$(se,ne)**

| | |
|---|---|
| se | Indicates the string constant, variable, or expression; represents the string to be padded with spaces. |
| ne | Numeric expression that indicates the desired length of the resulting string; ne must be greater than or equal to zero. |

Figure 5-13. LPAD$ Function Format

In figure 5-13, if ne is not greater than the length of se, then se is the resultant string. If ne is in the form of an expression, the argument is evaluated and rounded to an integer. If ne is less than zero, the diagnostic ILLEGAL LPAD$ ARGUMENT is returned.

In figure 5-14, the LPAD$ function pads two space characters to the left of the string value for A$; the output from the PRINT statement shows the two spaces between 0 and 12345.

```
100 LET A$="1234"
110 LET B$=LPAD$(A$,6)
120 PRINT "0";B$;"5"
130 END


produces:


0△△12345
```

Figure 5-14. LPAD$ Function Example

## LTRM$ FUNCTION

The LTRM$ function trims the original string of all leading space characters (spaces on the left). The format for LTRM$ is shown in figure 5-15.

In figure 5-16, the LTRM$ function trims the leading space characters of string B$. The printout from the second PRINT statement has no spaces between the value 8 and the value 12345.

**LTRM$(se)**

| | |
|---|---|
| se | String constant, variable, or expression that represents the string to be trimmed. |

Figure 5-15. LTRM$ Function Format

```
100 LET B$="ΔΔ12345"
105 PRINT "8";B$;"5"
110 PRINT "8";LTRM$(B$);"5"
120 END


produces:


8ΔΔ123455
8123455
```

Figure 5-16. LTRM$ Function Example

## LWRC$ FUNCTION

The LWRC$ function returns the original string with all its uppercase letters replaced by their lowercase equivalents. The LWRC$ function is only useful in ASCII mode. In normal mode, the LWRC$ function returns the original string in its same form because no lowercase letters exist in the normal character set. The results of this function depend on the terminal you use. Figure 5-17 illustrates the format for the LWRC$ function.

In figure 5-18, the LWRC$ function returns the lowercase equivalent of the value of string A$. The returned value "file a" is shown below the original value "FILE A".

**LWRC$(se)**

| | |
|---|---|
| se | String constant, variable, or expression that represents the string to be converted to lowercase. |

Figure 5-17. LWRC$ Function Format

```
100 LET A$="FILE A"
110 PRINT A$
120 LET B$=LWRC$(A$)
130 PRINT B$
140 END


produces:


FILE A
file a
```

Figure 5-18. LWRC$ Function Example

## ORD FUNCTION

The ORD function returns the decimal code (ordinal position) of a character in the collating sequence being used. The character or abbreviation for a character name having its ordinal position returned is specified as a string in the function argument, as shown in figure 5-19.

**ORD(se)**

| | |
|---|---|
| se | Indicates a string constant, variable, or expression that can have the values of single characters, and the 2- and 3-character mnemonic abbreviations of the ASCII character set in appendix A. |

Figure 5-19. ORD Function Format

Decimal codes that represent ordinal positions range from 0 to 127 when the standard ASCII collating sequence is in effect, and from 0 to 63 or 1 to 63 when the nonstandard display code collating sequence is in effect (see the OPTION statement and appendix A). The diagnostic ILLEGAL ORD ARGUMENT is issued if the argument string contains more than one character and the string is not an abbreviation for a character name. Figure 5-20 shows an example of using the ORD function.

```
100 PRINT "PROGRAM FOR ORD FUNCTION"
105 LET A$="a"
110 LET A=ORD(A$)
115 PRINT "CHARACTER ";A$;" HAS ORDINAL OF ";A
120 PRINT ORD("LCA")
130 PRINT ORD("5")
140 PRINT ORD("BS")
150 END


produces:


PROGRAM FOR ORD FUNCTION
CHARACTER a HAS ORDINAL OF  97
 97
 53
 8
```

Figure 5-20. ORD Function Example

## POS FUNCTION

The POS function returns the starting character of one specified string within another specified string. The format for the POS function appears in figure 5-21.

**POS($se_1$,$se_2$,ne) or POS($se_2$,$se_2$)**

| | |
|---|---|
| $se_1$ | String constant, variable, or expression; represents the string to be searched. |
| $se_2$ | String constant, variable, or expression; represents the string to be found. |
| ne | Numeric constant, variable, or expression; represents the position at which to start the search. Optional for starting position 2. |

Figure 5-21. POS Function Format

The POS function returns the first character position of string $se_2$ within string $se_1$. The search for this first character position begins at position ne. Character positions are numbered from the left, and start with one.

If the value associated with $se_2$ does not occur within the designated portion of $se_1$, the function value will return zero. If starting position ne is omitted or less than 1, it is considered to be equal to 1. If ne is greater than the number of characters of $se_1$, the portion of $se_1$ being specified is the null string starting after the last character position of $se_1$. The numeric expression ne is evaluated and rounded to an integer.

Figure 5-22 shows a program using the POS function to determine the position of different string characters within the original string A$ that has the value OUTSTANDING.

## RPAD$ FUNCTION

The RPAD$ function pads a string to a specified number of characters by inserting blanks on the right (called right pad). Figure 5-23 shows the format for the RPAD$ function.

If the length argument ne is not greater than the string argument se, then se is the resultant string. The argument ne is evaluated and rounded to an integer. If ne is less than zero, the diagnostic ILLEGAL RPAD$ argument is returned.

In figure 5-24, the RPAD$ function pads string A$ with one blank in order to generate a 2-character string. The printout from the second PRINT statement has a space between ABCD and EF.

## RPT$ FUNCTION

The RPT$ function generates a string consisting of several repetitions of the argument string. The format for the RPT$ function appears in figure 5-25.

RPAD$(se,ne)

| | |
|---|---|
| se | A string constant, variable, or expression that represents the string to be padded with space characters. |
| ne | A numeric constant, variable, or expression that indicates the desired length of the resulting string; ne must be greater than or equal to zero. |

Figure 5-23. RPAD$ Function Format

```
100 LET A$="D"
110 PRINT "ABC";A$;"EF"
120 PRINT "ABC";RPAD$(A$,2);"EF"
130 END

produces:

ABCDEF
ABCDΔEF
```

Figure 5-24. RPAD$ Function Example

RPT$(se,ne)

| | |
|---|---|
| se | Indicates a string constant, variable, or expression representing the string that is to be repeated. |
| ne | Indicates a numeric constant, variable, or expression specifying the number of times string se is to be repeated. |

Figure 5-25. RPT$ Function Format

```
10 PRINT "POS FUNCTION PROGRAM"
20 PRINT
30 LET A$="OUTSTANDING"
40 LET A=POS(A$,"AN",2)
50 PRINT "THE POSITION OF 'AN' STARTING WITH CHARACTER POSITION 2 IS ";A
60 PRINT POS(A$,"ST")
70 PRINT POS(A$,"AN",15)
80 PRINT POS(A$,"T")
90 END

produces:

POS FUNCTION PROGRAM

THE POSITION OF 'AN' STARTING WITH CHARACTER POSITION 2 IS  6
 4
 0
 3
```

Figure 5-22. POS Function Example

If the repetition argument ne is greater than zero, the function returns a string consisting of ne occurrences of the characters in string se. If ne is zero, a null string is returned. If ne is less than zero, the diagnostic ILLEGAL RPT$ ARGUMENT is returned. The argument ne is evaluated and rounded to an integer. Figure 5-26 shows three examples of the RPT$ function.

## RTRM$ FUNCTION

The RTRM$ function trims the original string of all trailing space characters (spaces on the right). Figure 5-27 illustrates the format for the RTRM$ function.

In figure 5-28, the RTRM$ function trims the two trailing space characters of string A$. The printout from the second PRINT statement has no spaces between the 345 and the ABC.

## STR$ FUNCTION

The STR$ function converts a numeric value to a string representation. The format for this function appears in figure 5-29.

In format 1 of the STR$ function, the resultant string is formatted according to the image specified by se. The image se can contain alphanumeric constants and any specification control characters that are allowed in the image statement. (See I/O Statements and Functions, section 7, for a complete discussion of format images.) If se is absent, as shown in format 2 of figure 5-29, the string is formatted according to the standard rules for numeric output except no preceding or trailing blanks are included. (See I/O Statements and Functions, section 7, for a complete discussion of standard rules for numeric output.) Figure 5-30 illustrates two examples of the STR$ function.

## UPRC$ FUNCTION

The UPRC$ function returns the original string with all its lowercase letters replaced by the uppercase equivalents. The UPRC$ function is only useful in ASCII mode. In normal mode, the UPRC$ function returns the original string in its same form because there are no lowercase letters in the normal character set. The format for the UPRC$ function is shown in figure 5-31. Figure 5-32 illustrates an example of this function (all of the letters change except D).

**RTRM$(se)**

| | |
|---|---|
| se | Indicates a string constant, variable, or expression; represents the string with spaces to be trimmed. |

Figure 5-27. RTRM$ Function Format

```
10 LET A$="1Δ345ΔΔ"
20 PRINT A$;"ABC"
30 PRINT RTRM$(A$);"ABC"
40 END

produces:

1Δ345ΔΔABC
1Δ345ABC
```

Figure 5-28. RTRM$ Function Example

1. **STR$(ne)**

   or

2. **STR$(ne,se)**

| | |
|---|---|
| ne | Indicates numeric constant, variable, or expression. |
| se | Indicates string expression (image specification). |

Figure 5-29. STR$ Function Format

## VAL FUNCTION

The VAL function converts a string containing numbers to a numeric value. The VAL function is the inverse of the STR$ function. The format of the VAL function is indicated in figure 5-33. The string must be written in the form of a numeric constant. Examples of this function are illustrated in figure 5-34.

# ERROR AND INTERRUPT PROCESSING

Table 5-4 summarizes the functions used in error and interrupt processing. Further details on these functions are in section 4.

| | |
|---|---|
| 10 LET A$ = RPT$("*",132) | A$ is assigned the string consisting of 132 asterisks (*). |
| 20 IF B$ = RPT$("Δ",80) THEN 90 | Control is transferred to statement 90 if B$ consists of 80 blanks. |
| 05 LET C$ = RPT$("ABC",2) | C$ is assigned the string ABCABC. |

Figure 5-26. RPT$ Function Examples

TABLE 5-4. ERROR AND INTERRUPT PROCESSING FUNCTIONS

| Function | Description |
|---|---|
| ASL(x) | Returns the statement line number of the statement executing or about to be executed when the most recent terminal interrupt occurred. |
| ESL(x) | Returns the statement line number of the statement that caused the most recent execution error. |
| ESM(x) | Returns the error number of the most recent execution error. |
| NXL(ne) | Returns the next line number of the statement that follows the line number specified in the argument of this function. |

```
10 LET B$ = STR$(A(1,6))                    Assuming A(1,6) = 1234, execution of this statement
                                            assigns the string 1234 to B$.

20 LET A$ = STR$(I,"PRICE = $###.##")       Assuming I = 203.23476, execution of this statement
                                            assigns the string PRICE = $203.23 to A$.
```

Figure 5-30. STR$ Function Example

UPRC$(se)

| | |
|---|---|
| se | String constant, variable, or expression; represents the string to be converted to uppercase letters. |

Figure 5-31. UPRC$ Function Format

```
10 LET A$=UPRC$("Department 4")
20 PRINT A$
30 END

produces:

DEPARTMENT 4
```

Figure 5-32. UPRC$ Function Example

VAL(se)

| | |
|---|---|
| se | Indicates a string constant, variable, or expression. |

Figure 5-33. VAL Function Format

```
110 LET B9 = VAL(B$(1))                     Assuming that B$(1) contains a string 1234, then the numeric value
                                            1234 is assigned to B9.

100 LET X4 = 2*C4 + VAL("123.7")            }
                                            } Similarly for these two examples, numeric values are extracted and used
090 LET IF VAL(D$(I,J))< 24 THEN 291        } for arithmetic purposes or for comparison with a numeric constant.
```

Figure 5-34. VAL Function Examples

## MATRIX FUNCTIONS

Table 5-5 outlines the functions used to simplify the use of matrices. For more details regarding these matrix functions, see the section on Matrix Operations.

## I/O FUNCTIONS

Table 5-6 briefly describes the functions used in I/O operations. Further details of these functions are described in the section I/O Statements and Functions.

## USER-DEFINED FUNCTIONS

BASIC, in addition to providing built-in functions of the language, also permits you to define your own functions. User-defined functions can be written either as single- or multiple-line functions. When these functions are referenced, they return a value based upon the parameters passed by the function reference and the function definition. User-defined functions are referenced the same as built-in functions. See Referencing a Function.

TABLE 5-5. MATRIX FUNCTIONS

| Function | Description |
|---|---|
| IDN<br>(or) IDN($ne_1$)<br>(or) IDN($ne_1$,$ne_2$) | Returns an identity matrix (ones along the diagonal; zeros in the remaining areas). The result is a square matrix with n x n elements where n = $ne_1$ = $ne_2$ or n = dimension of array IDN if no ne is specified. |
| CON<br>(or) CON(ne)<br>(or) CON($ne_1$,$ne_2$) | Returns a matrix of all ones with dimensions of $ne_1$ x $ne_2$, ne x ne, or the dimension of the array to which CON is assigned. |
| TRN(m) | Returns the transpose of matrix m. |
| ZER<br>(or) ZER(ne)<br>(or) ZER($ne_1$,$ne_2$) | Returns a matrix of all zeros with dimensions $ne_1$ x $ne_2$, ne x ne, or the dimensions of an array to which ZER is assigned. |
| INV(m) | Returns the inverse of matrix m. |

TABLE 5-6. I/O FUNCTIONS

| Function | Description |
|---|---|
| LOC(ne) | Returns the current word position in the random file ne. |
| LOF(ne) | Returns the length in words of the random binary file ne. |
| TAB(ne) | Returns a string of blanks, which results in moving the print mechanism to print position ne. TAB can only be used with the PRINT statement. |

The DEF and FNEND statements are provided to write user-defined functions. To write a single-line function, only the DEF statement is used. To write a multiple-line function, the function definition must begin with the DEF statement and end with the FNEND statement. Any BASIC statement, except END and another DEF statement, can be located between the DEF and FNEND statements. Table 5-7 summarizes the effect and usage of the DEF and FNEND statements.

TABLE 5-7. USER-DEFINED FUNCTIONS

| Statement | Effect | Usage |
|---|---|---|
| DEF | Defines a function. | DEF FNA(X) = A+B+C |
| FNEND | Terminates definition of a multiple-line function. | FNEND |

## SINGLE-LINE FUNCTIONS USING DEF

The DEF statement is used to write a single-line user-defined function. A single-line function is a complete definition on one statement line. It can be in the form of a numeric function or a string function, and it can contain parameters (up to 20 parameters are allowed). The format for a single-line function appears in figure 5-35.

1. DEF FNa=ne
2. DEF FNa ($sv_1$,$sv_2$, . . . ,$sv_{20}$) = ne
3. DEF FNa$=se
4. DEF FNa$ ($sv_1$,$sv_2$, . . . ,$sv_{20}$) = se

| | |
|---|---|
| a | Any alphabetic character that uniquely identifies the function. |
| ne | Indicates numeric expression: |
| se | Indicates string expression. |
| $sv_1$ . . . $sv_{20}$ | Indicates simple variable numeric or string. |

**NOTE**

Formats 1 and 2 are for numeric functions; formats 3 and 4 are for string functions.

Figure 5-35. Single-Line Function Using DEF

The rules for writing a single-line function using DEF are as follows:

The variables sv are formal parameters. They can be used elsewhere in the program without affecting the function. Each formal parameter must be unique within the function. From 0 to 20 formal parameters are permitted.

The expression defining the function can include variables other than formal parameters. The current value of these variables is used when the function is evaluated.

The definition must be complete on one line.

A function can include references to built-in BASIC functions or other user-defined functions, but not to the function being defined; recursive definitions are not allowed and cause an error diagnostic to be issued at compilation time.

Although a user-defined function can be referenced before it is defined, this is not recommended. A compile time warning diagnostic is issued when this occurs (WARNING - FUNCTION REFERENCE BEFORE DEFINITION). See the Future System Migration Guidelines, appendix E.

Although a function can be redefined within a program, it is not recommended; a compile time warning diagnostic is issued when this occurs (WARNING - FUNCTION REDEFINITION). If a function is redefined, the definition used is the one on the highest line number before the line containing the function reference; for a function referenced at a line number before any definitions, the definition used is the one with the lowest line number after the function reference. See the Future Systems Migration Guidelines, appendix E.

Figure 5-36 shows three examples using the DEF statement to express a single-line function. In example 1 of figure 5-36, line number 10 contains the function definition, and line number 20 contains the function reference.

In example 2 of figure 5-36, 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. Note that the definition FNV 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.

In example 3 of figure 5-36, a DEF statement with no formal parameters is used to define the area of a circle having a radius of .2.

```
10 DEF FNA(M,N,O,P)=M+N+O+P
20 LET E=FNA(2,3,4,5)
30 PRINT E
40 END

produces:

 14


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","   VOLUME"
50 FOR R=.1 TO 1 STEP .3
60 PRINT R,FNC(2*R),FNA(R),FNV(R)
70 NEXT R
99 END

produces:

RADIUS          CIRCUMFERENCE     AREA           VOLUME
 .1              .628318         3.14159E-2     1.04720E-3
 .4              2.51327         .502654        6.70206E-2
 .7              4.39823         1.53938        .359188
 1               6.28318         3.14159        1.0472


10 DEF FNP=3.14159
20 DEF FNA(R)=FNP*R**2
30 PRINT "AREA=";FNA(.2)
40 END

produces:

AREA= .125664
```

Figure 5-36. Single-Line Function Examples Using DEF

## MULTIPLE-LINE FUNCTIONS USING DEF...FNEND

Multiple-line functions are defined through use of the DEF and FNEND statements. The function definition must begin with the DEF statement and end with the FNEND statement. Any BASIC statement, except for another DEF, can appear between the DEF and FNEND statements. A multiple-line function can be in the form of a numeric or string function, and it can contain parameters (maximum of 20 parameters are allowed). The format of a multiple-line function appears in figure 5-37.

1. DEF FNa

   LET FNa=ne

   .
   .
   .

   FNEND

2. DEF FNa ($sv_1$,$sv_2$, . . . ,$sv_{20}$)

   LET FNa=ne

   .
   .
   .

   FNEND

3. DEF FNa$

   LET FNa$=se

   .
   .
   .

   FNEND

4. DEF FNa$ ($sv_1$,$sv_2$, . . . ,$sv_{20}$)

   LET FNa$=se

   .
   .
   .

   FNEND

a Indicates any alphabetic character that uniquely identifies the function.

ne Indicates numeric expression.

se Indicates string expression.

sv Indicates simple, numeric, or string variable.

NOTE

Formats 1 and 2 are for numeric functions; formats 3 and 4 are for string functions.

Figure 5-37. Multiple-Line Function Format With DEF...FNEND

The rules for writing a multiple-line function using the DEF...FNEND statements are as follows:

The variables (sv) are formal parameters. They can be used elsewhere in the program without affecting the function. Variables used as formal parameters are local to the function; for example, changing the values within the function has no effect on variables of the same name outside the function.

When a function is referenced (described earlier in this section under Referencing a Function), parameter expressions are evaluated and their values are passed to the function. Therefore, changing the value of a formal parameter within a function only effects the local value of the parameter, not the value of the parameter expression or variable used in the expression outside the function.

The expression or expressions within the function definition can include other program variables in addition to the formal parameters; these are global variables and have the same values inside and outside the function.

Global variables can have their value changed by operations within the function definition. Unexpected results can occur if a global value that has been changed in a function is used in an expression containing the function reference.

A function can include references to built-in or other user-defined functions, but not to itself; recursive function definitions are not permitted.

It is illegal to reference a line number outside a function definition from within the definition, or to reference a line number in a function definition from outside the definition. An attempt to do so causes the compile time diagnostic TRANSFER OUT OF DEF or TRANSFER INTO DEF to be issued.

The function name must be assigned a value if the function is to return other than a 0 or null.

A function definition can appear after a function reference; however, it is not recommended. The warning diagnostic WARNING - FUNCTION REFERENCE BEFORE DEFINITION is issued. See Future Systems Migration Guidelines, appendix E.

A function can be redefined within a program; however, this is not a recommended programming practice. The diagnostic warning WARNING - FUNCTION REDEFINITION is issued. If a function is redefined, the definition used is the one on the highest line number before the line containing the function reference; or for a function referenced at a line number before any definitions, the definition used is the one with the lowest line number after the function reference. See Future Systems Migration Guidelines, appendix E.

Statements located between the DEF and FNEND statements can be any BASIC statement except END and another DEF statement.

The function name can be used as a local variable within a function definition (on the right side of the statement). However, such usage is not recommended because it might not be supported in a future version of BASIC. See Future System Migration Guidelines, appendix E.

Figure 5-38 shows two examples of writing and referencing multiple-line functions. The first example illustrates a string function; the second example illustrates a numeric function.

In example 1 of figure 5-38, the function FNR$ replaces J characters of A$, starting with character I, with the first J characters of B$. B$ is blank padded to J, if necessary.

In example 2 of figure 5-38, the function FNM uses its formal parameter N as a local variable. Changing its value in the function (line numbers 170 and 200) has no effect on the actual parameter M passed to the function at line number 220. Note also that the actual parameter need not be used in calculating the function result. The result in this example is calculated from global variables B(0),...,B(4).

```
100 DEF FNR$(A$,I,J,B$)
120 REM REPLACE J CHARACTERS OF A$ BEGINNING AT CHARACTER I
130 REM WITH THE FIRST J CHARACTERS OF B$
140 REM B$ IS PADDED TO LENGTH J IF NECESSARY
150 IF J<LEN(B$) THEN LET B$=RPAD$(B$,J)
160 LET A$(I:I+J-1)=B$(1:J)
170 LET FNR$=A$
180 FNEND
190 LET X$="ABCDEFGH"
200 LET Y$="12345"
210 PRINT FNR$(X$,3,4,Y$)
220 END

produces:

AB1234GH

100 DIM B(5)
110 PRINT "TYPE IN ANY 5 NUMBERS"
120 INPUT B(0),B(1),B(2),B(3),B(4)
130 LET M = 17
140 REM FUNCTION DEFINITION
150 DEF FNM(N)
160 LET FNM=B(0)
170 FOR N=1 TO 5
180 IF FNM>=B(N) THEN 200
190 LET FNM=B(N)
200 NEXT N
210 FNEND
220 LET Y=FNM(M)
230 PRINT "THE VALUE OF M IS UNCHANGED BY THE FUNCTION"
240 PRINT "IT IS STILL ";M
250 PRINT "MAXIM IS ";Y
260 END

produces:

TYPE IN ANY 5 NUMBERS
? 89,78,45,67,9
THE VALUE OF M IS UNCHANGED BY THE FUNCTION
IT IS STILL  17
MAXIM IS  89
```

Figure 5-38. Multiple-Line Function Examples Using DEF...FNEND

This section describes the statements used to write BASIC subroutines, link to external subprograms, and chain to other programs. Table 6-1 outlines the subroutine, subprogram, and chaining statements. Further details for these functions and statements follow the table.

## BASIC SUBROUTINES

When a particular part of a program must be performed more than once, it is useful to use a subroutine. Control can be transferred to a subroutine from the main program and, at the conclusion of the subroutine, be returned to the main program.

Within the main BASIC program, control can be transferred to BASIC subroutines. These subroutines are compiled along with the main program. The following paragraphs describe the method of calling subroutines using the GOSUB or ON GOSUB statements. The RETURN statement directs execution to the most recently executed GOSUB or ON GOSUB. The following rules must be followed when using these statements:

- Any number and type of BASIC statements are allowed in a BASIC subroutine.
- GOSUB statements can be nested to a depth of 40.
- Recursion is allowed; a subroutine can contain a call to itself.

Figure 6-1 illustrates a subroutine call and return sequence. Lines 150 through 220 contain subroutine A. The subroutine is called from line 60. After execution of subroutine A, control is transferred to line 70, and at line 80 execution is directed to line 230 (the end of the program) bypassing the subroutine statements.

### GOSUB STATEMENT

The simple GOSUB statement unconditionally transfers control to a line number that is the first statement of the subroutine. Figure 6-2 shows the format of the GOSUB statement.

**GOSUB ln**

| | |
|---|---|
| **ln** | **Indicates the line number of the first statement of the subroutine.** |

Figure 6-2. GOSUB Statement Format

Execution of the GOSUB statement and the RETURN statement (described later) can be described in terms of a stack of line numbers. The stack is empty prior to execution of the first GOSUB statement. Each time a GOSUB statement is executed, the line number referred to in the GOSUB statement is placed on top of the stack and execution of the program continues at this line number, which is the first statement of a subroutine. Each time a RETURN statement is executed, the line number on top of the stack is removed from the stack and execution of the program is continued at the line following the line number presently at the top of the stack.

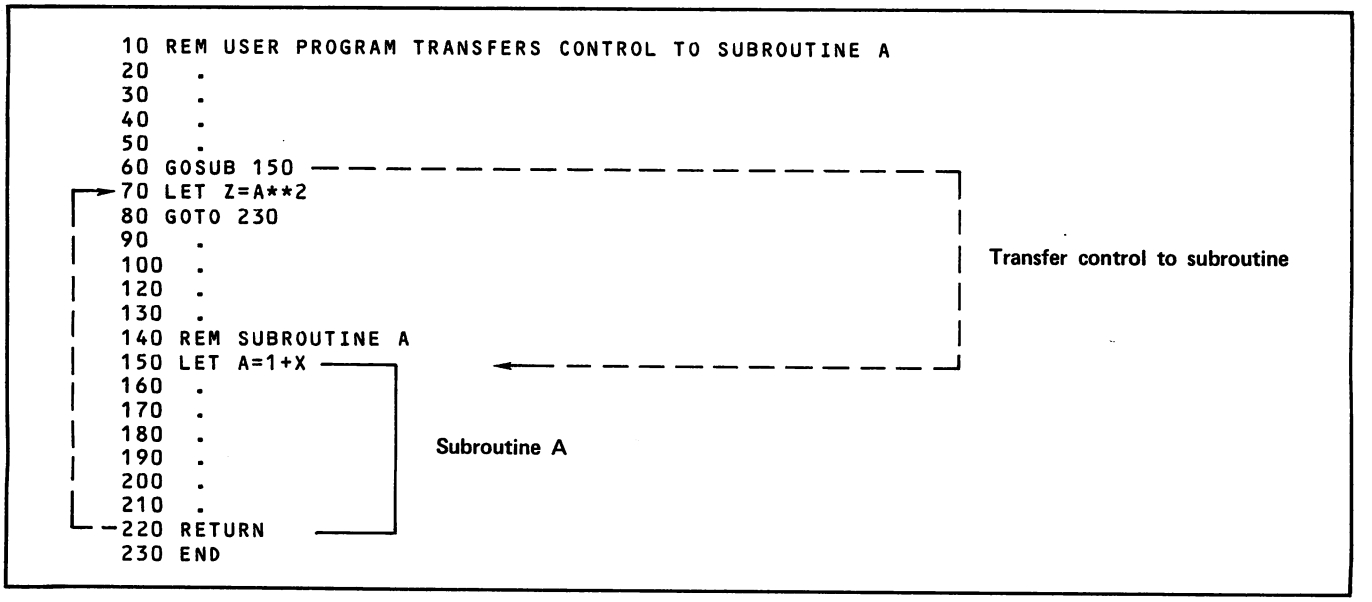


Figure 6-1. BASIC Subroutine and RETURN Statement

TABLE 6-1. SUBROUTINE, SUBPROGRAM, AND CHAINING STATEMENTS

| Statement | Effect | Usage |
|---|---|---|
| GOSUB | Transfers control to a BASIC subroutine. | GOSUB 150 |
| ON GOSUB | Conditionally transfers control to a subroutine depending upon the integer value specified in the ON GOSUB line. | ON X-Y GOSUB 20,30 |
| RETURN | Upon completion of the subroutine, returns control to the statement immediately following GOSUB or ON GOSUB. | RETURN |
| CALL | Permits execution to a separately compiled subprogram in non-BASIC language. | CALL SUB2 |
| CHAIN se<br>CHAIN #ne | Permits control to exit from a BASIC program by terminating the current program and initiating execution of another program. | CHAIN "CHESS" |

A GOSUB (or ON GOSUB) statement can be used within one subroutine to transfer control to another subroutine; these are nested subroutines. The GOSUB statement can be used 40 times in these nested subroutines. Each subroutine in nested subroutines is executed by the stack order, explained under the GOSUB statement. A GOSUB can be ended without a RETURN. For example, execution can be stopped inside a subroutine. An example of nested subroutines is shown in figure 6-3.

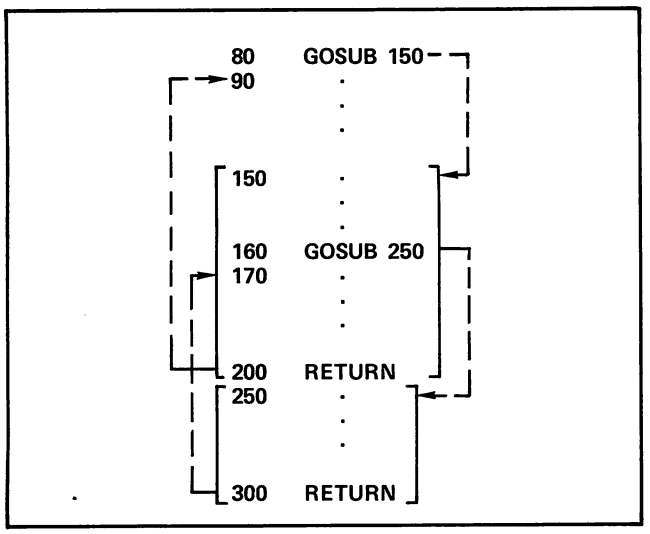


Figure 6-3. Nested Subroutines

## ON GOSUB STATEMENT

The ON GOSUB statement conditionally transfers control to one of n possible subroutines. The format of the ON GOSUB statement is shown in figure 6-4.

ON ne GOSUB $ln_1$,$ln_2$, . . . ,$ln_n$

| | |
|---|---|
| ne | Indicates arithmetic expression which is evaluated to determine point of transfer. |
| $ln_{1-n}$ | Indicates line numbers. |

Figure 6-4. ON GOSUB Statement Format

In figure 6-4, the expression ne is evaluated and rounded to an integer value; control transfers to $ln_1$ when ne is 1, to $ln_2$ when ne is 2, and so on. The line number of the ON GOSUB statement is recorded on the GOSUB stack (see GOSUB STATEMENT) so that RETURN statement can subsequently return control to the statement following the ON GOSUB statement. If the value of the expression ne is negative or zero, or greater than the number of specified line numbers in the ON GOSUB statement, program execution terminates, displaying the diagnostic ON EXPRESSION OUT OF RANGE.

Figure 6-5 illustrates the ON GOSUB statement. In the example, when the expression X-Y equals 1 or 3, control transfers to line 200. When X-Y equals 2, control transfers to line 250, and when X-Y equals 4, control transfers to line 300.

## RETURN STATEMENT

The RETURN statement is usually the last statement of a BASIC subroutine; however, it can be used anywhere and any number of times within the subroutine; RETURN directs the program to resume execution at the statement immediately following the most recently executed GOSUB or ON GOSUB statement. Figure 6-6 shows the format of the RETURN statement.

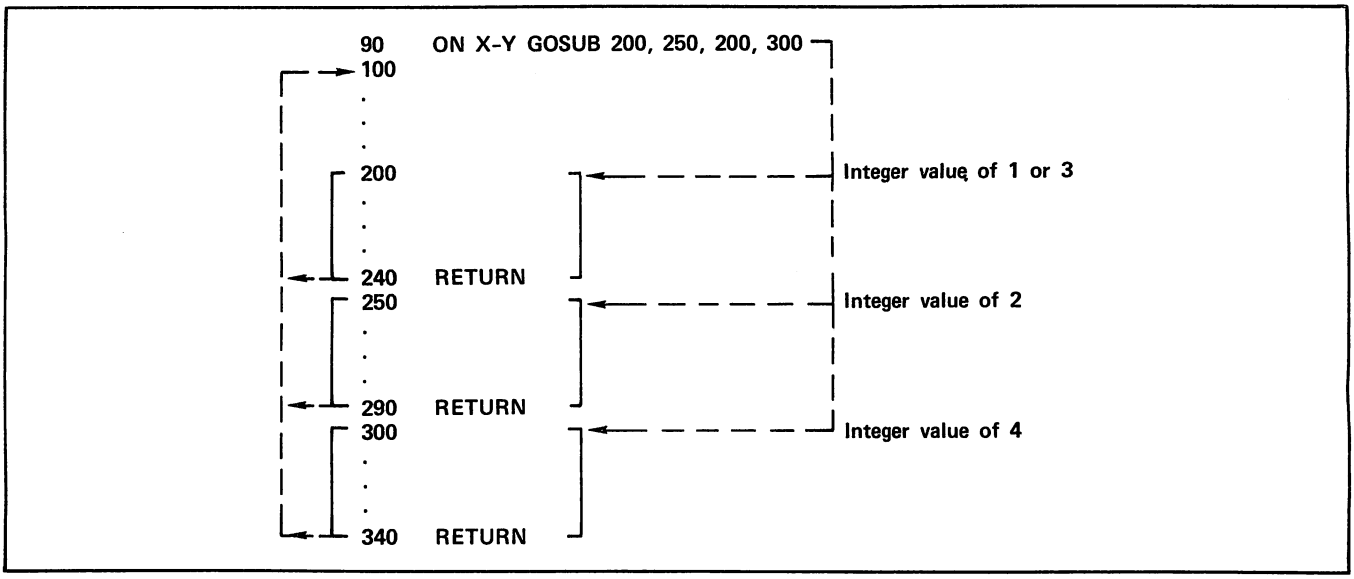


Figure 6-5. ON GOSUB Statement Example

RETURN

Figure 6-6. RETURN Statement Format

Each time a RETURN statement is executed, a line number is removed from the top of the GOSUB stack, and control is transferred to the next line following that line number. See GOSUB statement for a description of GOSUB stack. The diagnostic RETURN BEFORE GOSUB is issued if there is no return line number on the GOSUB stack (if there is no remaining GOSUB or ON GOSUB from which to return). For examples of the RETURN statement, see the GOSUB and ON GOSUB statements in this section.

# EXTERNAL SUBPROGRAMS

Transfer of control to an external subprogram is accomplished through use of the CALL statement. The following describes the CALL statement format and programming ideas for preparing external subprograms for use with a BASIC program.

## CALL STATEMENT

The CALL statement permits you to execute a separately compiled subprogram in a non-BASIC language, such as FORTRAN. When the CALL statement is executed, parameters are evaluated and passed to the subprogram, then the subprogram is executed. When the subprogram finishes, control returns to the main program at the statement immediately following the CALL statement. Values are returned as new values of the parameters that are changed by the subprogram. The format of the CALL statement appears in figure 6-7.

1. CALL subnm
2. CALL subnm ($e_1$,$e_2$, . . . ,$e_{20}$)

subnm — Indicates subprogram name.

e — Indicates any numeric or string expression.

Figure 6-7. CALL Statement Format

As shown in figure 6-7, the CALL statement can contain parameters (up to 20 parameters are allowed). If parameters are specified with the CALL statement, the parameter values are passed to the subprogram in the order that the parameters are listed in the CALL statement. The following is an example of two CALL statements, one with parameters and the other without parameters:

```
CALL SUBTEST(A,B,C,D)
CALL A1234B
```

The following rules apply to the CALL statement:

The subprogram name, subnm, can be 1 to 7 characters. The first character must be alphabetic; any alphabetic characters used must be uppercase. The last character must not be a colon. Characters +, -, *, /, comma, circumflex, and blank cannot be used.

No more than 20 parameters can be passed to the subprogram.

The CALL statement can be executed only if the B option (relocatable binary code) is used when compiling the BASIC program. See the BASIC Control Statement, section 12. If an attempt is made to execute a CALL in compile-to-memory

mode (no B option), the diagnostic UNSATISFIED EXTERNAL REFERENCE is issued. The compiled program must be loaded, along with the subprogram that has been called, before the program can be executed. See the CYBER Loader reference manual.

The CALL statement cannot appear in a BASIC source program that is the target of a CHAIN statement. See description of CHAIN statement in this section for more information.

The CALL statement, without parameters, cannot be the object of THEN in the IF...THEN...ELSE statement.

The CALL statement can be used to call subprograms written in any language that conforms to the FORTRAN calling sequence conventions. For an example of using the CALL statement to call a FORTRAN subprogram, see figure 6-8.

```
BASIC Program on File F681:

100 OPTION BASE 1       'BASE 1 NEEDED SO BASE IS SAME AS FORTRAN SUBPROGRAM
110 DIM A(2,3)
120 FOR I=1 TO 2
130       FOR J=1 TO 3
140             A(I,J)=I*J
150       NEXT J
160 NEXT I
165 PRINT "THESE ARE THE BASIC ELEMENTS"
170 FOR I=1 TO 2
185       PRINT A(I,1),A(I,2),A(I,3)
186 NEXT I
190 CALL FSUB (A(1,1))  <-------------------------------------- Call to FORTRAN subprogram.
191 PRINT "THESE ARE THE BASIC ELEMENTS CHANGED WITH THE FORTRAN SUBPROGRAM"
192 FOR I=1 TO 2
194       PRINT A(I,1),A(I,2),A(I,3)
195 NEXT I
210 STOP
220 END

FORTRAN Subprogram on File F682:

C SUBROUTINE FSUB  <-------------------------------------- FORTRAN subprogram.
C NOTE THAT ORDER OF SUBSCRIPTS MUST BE REVERSED FROM BASIC
      SUBROUTINE FSUB(A)
      DIMENSION A(3,2)
      DO 200 I=1,3
         DO 100 J=1,2
            A(I,J)=A(I,J)+10
100      CONTINUE
200   CONTINUE
      RETURN
      END

Control Statements:

/x,basic(i=f681,l=0,b=f68b)  <---------------------------- Compile BASIC program.
     .011 CP SECONDS COMPILATION TIME
/ftn5(i=f682,l=0,b=f68b)  <------------------------------- Compile FORTRAN subprogram.
    0.009 CP SECONDS COMPILATION TIME
/f68b  <-------------------------------------------------- Execute BASIC main program
                                                           and FORTRAN subprogram.

Output:

THESE ARE THE BASIC ELEMENTS
 1             2             3
 2             4             6
THESE ARE THE BASIC ELEMENTS CHANGED WITH THE FORTRAN SUBPROGRAM
 11            12            13
 12            14            16
```

Figure 6-8. BASIC Program Call to FORTRAN Subprogram

## WRITING EXTERNAL SUBPROGRAMS

The following rules apply when writing external subprograms:

All numbers processed by BASIC are stored as normalized, single-precision, floating-point values. Therefore, all numeric values passed to external routines are in normalized, single-precision, floating-point form (FORTRAN type REAL), and all numeric values returned by external routines must be stored in the same form.

All strings processed by BASIC are stored as 6-bit display code or 6- and 12-bit escape code characters packed into words with the first character being the leftmost character in the first word of the string. The 6-bit display codes are used when BASIC is executing in normal mode; the 6- and 12-bit escape codes are used when BASIC is executing in ASCII mode. The string is always passed in the 6/12 character set on both NOS and NOS/BE. Strings passed to external programs are zero-byte delimited; the last word of the string contains zeros in the last 12 bits and in all other character positions not containing valid characters. The length in 6-bit characters is indicated in the parameter block. The length in characters can be obtained by scanning the string looking for the zero-byte delimiter or by passing the length as determined by BASIC's LEN function. (The LEN function returns the count of logical, not physical, characters). In ASCII mode, two 6-bit characters sometimes constitute one logical escape code character.

External routines can change characters in a string, but they must not change the length of the string. BASIC maintains its own length indicator; therefore, externally shortening a string by moving the zero-byte delimiter is not noticed, but externally lengthening a string destroys adjacent data, just as storing a double-precision value over a single-precision value destroys adjacent data. To change the length of a string, pass an indicator back to BASIC and let BASIC change the length with a string assignment statement.

External subprograms should not perform input or output operations on files used by BASIC.

Avoid the use of subprogram names that are identical to entry points in any of the BASIC execution on-time routines; the use of such names can cause unpredictable results and program termination.

No provision is made for passing an entire array to an external subprogram. However, if the first element of a numeric array, such as A(0,0), is passed, its address can be interpreted as the address of the array because all elements of a numeric array are stored contiguously. This technique does not apply for string arrays because string array elements are not stored contiguously.

When passing BASIC numeric arrays to FORTRAN, provision must be made for the fact that BASIC normally begins array subscripting with element zero (0) and stores elements of multi-dimensional arrays in row order (A(0,0), A(0,1), A(0,2), and so forth), while FORTRAN begins array subscripting with element 1 and stores elements of multi-dimensional arrays in column order (A(1,1), A(2,1), A(3,1), and so forth). This difference causes a 2-dimensional array to appear to be transposed when it is passed between FORTRAN and BASIC. Note that in BASIC, OPTION BASE can be used to set the array subscript origin to 1 rather than 0.

# PROGRAM CHAINING

The CHAIN statement allows control to exit from a BASIC program by terminating the current program and initiating execution of another BASIC program.

## CHAIN STATEMENT

The CHAIN statement terminates the current program and initiates execution of another program. The new program can be either a BASIC program in source form or a BASIC or non-BASIC program in precompiled binary form. In either case, it is retrieved from either a local or a permanent file. Figure 6-9 illustrates the formats for the CHAIN statement.

1. CHAIN se
2. CHAIN #ne

| | |
|---|---|
| se | Indicates string expression; the name of the chained-to file; optionally includes permanent file and mode (ASCII/normal) information. |
| ne | Numeric expression; file ordinal of the chained-to file. |

Figure 6-9. CHAIN Statement Format

When the file ordinal format is used, no options can be specified. Chaining is done to the local file specified by the file ordinal, and, if the file contains a BASIC source program, the compiler is invoked with the same mode (ASCII or normal) as the chaining program. If the file is binary, the mode is determined by the binary program itself. A file ordinal expressed as a numeric expression is evaluated and rounded to an integer value.

When the string expression format is used, a filename must be specified and, optionally, a permanent file user number and/or password must be specified. Also, a mode indicator (ASCII or normal) can be specified. The optional values can be specified positionally or with keywords, as shown in figure 6-10.

```
file,user,pswd,mode
or
file,UN=user,PW=pswd,MODE=mode
or
file,ID=user,PW=pswd,MODE=mode
```

| | |
|---|---|
| file | Required file name, 1-7 characters beginning with a letter. |
| user | Indicates optional user number (ID) for permanent file access, 1-7 characters. |
| pswd | Indicates optional password for permanent file access, 1-7 characters. |
| mode | Optional mode, must be ASCII or NORMAL or leftmost substrings of these such as A,AS,ASC. |

Figure 6-10. Keywords for Optional Values

Letters used in file names and optional values must be uppercase. When keywords are used, they can be specified in any order. Missing positional parameters must be indicated by commas. For example:

```
FILEX,,,MODE
```

Keyword and positional parameters can be mixed, but any keywords must follow all positional values. For example:

```
FILEX,USER1,MODE=NORM
```

## CHAIN PROCESSING

Execution of the CHAIN statement proceeds in the following manner: for the string expression format, the string is decoded into parameters. The diagnostic ILLEGAL CHAIN PARAMETER is issued for such errors as an illegal file name, a file that is already assigned or connected to the terminal, and/or an incorrect mode value, such as ASKI. If the mode is not specified, the mode of the current chaining program is used. If the user number is not specified, the current user number (user name) is used under NOS, and the ID of PUBLIC is used under NOS/BE. For the file ordinal format, no parameters need be decoded. If a local file of the specified name exists, it is used. If one does not exist, access is attempted under NOS with GET or ATTACH (if GET is unsuccessful), and under NOS/BE with ATTACH. In both systems, attached files are attached with read-only permission. If the named file cannot be obtained, the diagnostic CHAIN FILE NOT FOUND is issued.

When the file has been found, all files in the current (chaining) program are closed as if STOP or END had been executed. The file containing the current program is returned, unless it is named INPUT or is the file being chained to, and the chained-to file is rewound. Also, for NOS in interactive mode, the new file is made the primary file, if possible (if it is not a direct access permanent file). Under NOS/BE and NOS batch mode, the new file is not treated in a special manner.

The new file is examined to determine if it contains source or binary data and the BASIC compiler or the CYBER Loader is called as appropriate. The file is rewound before and after this check. Both the compiler and the loader destroy the current program so that no return is possible.

Restrictions:

- DATA, or the values of variables and arrays, cannot be passed between programs. All information must be on files.
- The chained-to program cannot contain CALL statements if the chained-to program is in source form. However, CALL statements are permitted in precompiled binary programs. When the program is in source form, the BASIC compiler is called to compile and execute the program in one step without using the loader; therefore, any called subroutine is not available and results in an execution-time diagnostic.
- The chained-to file cannot be connected or assigned to the terminal.
- If the BASIC compiler to be used resides in a local file rather than in the system library, the filename must be BASIC.

Figure 6-11 shows how the CHAIN statement can be used in a control program to determine which of a given number of computer game programs to execute. Each game program exists as a separate file with a user number (for NOS) or an ID (for NOS/BE) of LIBRARY. The game and the control program can CHAIN back to the control program when it finishes, if the control program is on the file CONTROL and the last executed statement is as follows:

```
510 CHAIN "CONTROL,LIBRARY"
```

```
100 PRINT "ENTER NAME OF THE GAME"
110 INPUT A$
120 IF A$="POKER" GOTO 170
130 IF A$="ROULETTE" GOTO 190
140 IF A$="STARTREK" GOTO 210
150 PRINT "I DON'T HAVE THAT GAME.  TRY AGAIN"
160 GOTO 110
170 PRINT "CALLING POKER"
180 CHAIN "POKER,LIBRARY"
190 PRINT "CALLING ROULETTE"
200 CHAIN "ROULETT,LIBRARY"
210 PRINT "CALLING STARTREK"
220 CHAIN "STARTRK,LIBRARY"
230 END
```

Figure 6-11. CHAIN Processing Example

This section explains file usage in BASIC and describes the statements and functions related to input and output. Included are the statements and functions to input and print display format; read and write binary data; construct and read internal data tables; aid random access; manipulate files; and produce special output formats. Table 7-1 summarizes the input and output statements.

## BASIC FILES AND FILE I/O STATEMENTS

Data can be contained within the BASIC program internally in a data block or externally in a file. A file is a named collection of data that a BASIC program can reference and manipulate. A

TABLE 7-1. I/O STATEMENTS AND FUNCTIONS

| Statement | Effect | Usage |
|---|---|---|
| FILE | Associates a binary or display format file with a file ordinal. | FILE #1 = "TXT1"<br>#2 = "TXT2" |
| CLOSE | Disassociates a binary or display format file from a program. | CLOSE #1 |
| RESTORE | Resets data table pointer to the first data value, or restores file pointer to the beginning of the file. | RESTORE |
| NODATA | Branches to specified line number on end-of-data (internal data table) or on end-of-file (binary or display format file). | NODATA 150 |
| IF END | Branches to specified line number on end-of-file. | IF END #2 THEN 50 |
| IF MORE | Branches to specified line number when no end-of-file is indicated. | IF MORE #2 THEN 100 |
| APPEND | Positions file so that data can be added to the end of the file. | APPEND #2 |
| WRITE | Writes data to a binary file. | WRITE #1,A,B,C |
| READ | Reads data from a binary file or from an internal data table. | READ A,B,C |
| SET | Points to a specific location within a binary file so that the next READ or WRITE statement can reference the desired word. | SET #1,N1 |
| LOC(ne) | Returns the current word position of binary file ne. | LOC(2001) |
| LOF(ne) | Returns the length in words of binary file ne. | LOF(200) |
| INPUT | Reads data from a display format file or from the terminal during program execution. | INPUT X,Y |
| DELIMIT | Specifies the characters to be used as input item separators. | DELIMIT #1,(;) |
| PRINT | Prints data in display format (as specified in the PRINT line) in a file or at a terminal. | PRINT "VALUE",Y |
| TAB | Causes the print mechanism to tab to a specified column. | TAB(5) |
| PRINT USING | Prints data in a file or at a terminal as specified in an image statement or in a string expression. | PRINT USING 80,A<br>PRINT 3 USING "SET",A |
| Image Format | Establishes output format for the PRINT USING statement. | :THE TOTAL IS ##.# |
| MARGIN | Defines the right-hand margin for printed output. | MARGIN 136 |
| SETDIGITS | Specifies the number of significant digits to be output by subsequent PRINT statements. | SETDIGITS 5 |
| DATA | Creates a table of data values internal to a program. | DATA "A",1,2,3 |

logical file name (lfn) consists of 1 to 7 alphanumeric characters. The first character must always be a letter. Any letters that are used must be uppercase. Files used with BASIC are normally located on mass storage; exceptions are those files connected or assigned to the terminal. Terminal files accept and display data directly at a terminal. If a BASIC program implicitly or explicitly references the name of a nonexistent file, the operating system (NOS or NOS/BE) automatically creates an empty file by that name. Direct access to tape files is not supported.

Files named INPUT and OUTPUT have special meaning in a BASIC program. In interactive mode, if these files are connected or assigned to the terminal, data written on file OUTPUT is automatically printed at the terminal, and data read from file INPUT must be entered at the terminal. These files are always connected to the terminal in interactive mode under NOS. Sometimes the files INPUT and OUTPUT must be explicitly connected to the terminal by using the CONNECT command under NOS/BE. (See section 11.) In batch mode, data written on file OUTPUT is automatically printed at the end of the job and data read from INPUT must be included in the file that contains job control statements.

BASIC programs can read and write files in two formats: binary format (a file created by WRITE) and display format (files created by PRINT or an external method). It is more efficient and more accurate to manipulate data in binary format because no translation is needed before processing, but it is sometimes inconvenient to use binary data because it cannot be printed at the terminal or printer. Conversely, it is less efficient and sometimes less accurate to manipulate data in display format because translation into binary is necessary before the data can be used by BASIC, but it is usually more convenient to use display format data because it can be printed at the terminal or printer. Thus, all data entered on cards or at the terminal, and all printed data, is formatted in display format. In general, binary data files are written only if the data is to be read later by a BASIC program and a printed copy is not needed.

The BASIC statements described in this section are used for binary input and output, display input and output, and input and output for internal data tables. Some of the statements, such as RESTORE, apply to all of these categories; while others, such as INPUT and PRINT, apply to only one category. Table 7-2 identifies each I/O statement that is applicable to each category. The statements listed in this table are grouped according to their respective functions.

## FILE ACCESS METHODS

BASIC offers both a sequential and random access method of reading and writing files. Sequential access sometimes involves a systematic search throughout the file from beginning to end until the desired information is found. Random access allows immediate location of the information (direct retrieval). Random access, for example, can be used to advantage on a file with students' grades, provided the grades are ordered in student number order. In this case, the student number indicates the relative position of the student's record within the file; for example, student number 12645 would indicate that the student record occupied the 2645th position in the file (assuming that student numbering begins at 10001). If the student number associated with the desired record is known, retrieval of this student's grade can be almost instantaneous by using random access techniques. Display format files can only be accessed sequentially; however, both access methods apply to binary format files. Table 7-3 summarizes the differences between these two access methods.

TABLE 7-2. I/O STATEMENTS AND RELATED TYPE OF I/O

| Type of Statements | Display I/O | Binary I/O | | I/O for Internal Data Tables |
|---|---|---|---|---|
| | Sequential Access | Sequential Access | Random Access | Sequential Access |
| File Access and CLOSE statement | FILE<br>CLOSE | FILE<br>CLOSE | FILE<br>CLOSE | |
| Input | INPUT<br>DELIMIT | READ | READ | READ |
| Output | PRINT<br>PRINT USING<br>Image<br>MARGIN<br>SETDIGITS | WRITE | WRITE | DATA |
| File Control | RESTORE<br>NO DATA<br>IF END<br>IF MORE<br>APPEND | RESTORE<br>NODATA<br>IF END<br>IF MORE<br>APPEND | SET<br>RESTORE<br>NODATA<br>IF END<br>IF MORE<br>APPEND | RESTORE<br>NODATA |

TABLE 7-3. SEQUENTIAL ACCESS VERSUS RANDOM ACCESS

| Sequential Access Method | Random Access Method |
|---|---|
| Can be used with display and binary format files. | Can be used with binary format files only. |
| Data must be processed sequentially from beginning to end or until the desired information is found. | Data need not be processed sequentially. File can be positioned to any word in the file. |
| File can be created with WRITE or PRINT statement, and the READ or INPUT statement is used to systematically examine each data item until the desired information is found. | File must be created sequentially with the WRITE statement and read with the READ statement. The SET statement and two functions, LOC and LOF, supply the data to permit positioning directly to the desired word location and retrieve or write the word stored at that location. |
| File must be either in READ or WRITE mode, and whichever mode is used, a RESTORE Statement must be used before the other mode can be used. | READ and WRITE operations can be intermixed without an intervening RESTORE statement. |

## PERMANENT FILE ACCESS

BASIC processes local (temporary) files. A local file can be one of the following:

A permanent file made local with the ATTACH command

A file created by the user or by an executing program during the job terminal session

A copy of a permanent file

An attached permanent file is a permanent file that has been made directly accessible to the BASIC program. If changes are made to an attached direct access file, those changes are permanent. An indirectly accessed permanent file is a copy of a permanent file. If the job or terminal session ends before the modified copy is saved, all changes made to the file during the terminal session are lost. Similarly, if a file created by the user or by an executing BASIC program is not explicitly made permanent with operating system commands before the job or terminal session ends, the file is lost. Sections 10 and 11, and appendix D illustrate operating system commands to make temporary local files permanent, and permanent files local. Refer to the NOS or NOS/BE reference manual for a complete explanation.

## FILE STATEMENT

The FILE statement is used to associate a number (the file ordinal) with a file name. All file I/O statements require the use of a file ordinal. Figure 7-1 shows the format for the FILE statement. In figure 7-1, the file named by lfn is associated with the file ordinal specified by n.

FILE $\#n_1 = lfn_1, \#n_2 = lfn_2, \ldots, \#n_m = lfn_m$

$n_{1-m}$ Indicates the file ordinal; any numeric constant, variable or expression with a value between 1 and 131071.

$lfn_{1-m}$ Indicates the file name (a string constant or variable with seven or fewer alphanumeric characters), the first character must be a letter and the alphabetic characters used must be uppercase; string expressions can also be used for the file name.

Figure 7-1. FILE Statement Format

All file buffers, one for each name or number pair declared in all FILE statements, are allocated as the program is compiled. A maximum of 13 such buffers is allowed. Names and numbers are associated with each other and assigned to an available file buffer when a FILE statement is executed. Names and numbers must be unique, not currently in use, and buffer space must be available or a fatal error results. Therefore, a FILE statement can be executed only once, unless the file names and numbers that the FILE statement uses are different for each execution and unassigned buffers are available, or unless the names, numbers, and buffers that the FILE statement uses are first released by using a CLOSE statement.

When the FILE statement references a previously non-existing file name, an empty file is created; data can later be added to the empty file by using statements like PRINT, WRITE, and APPEND. When the FILE statement references a previously existing file (local), the FILE statement does not position the file. Therefore, if unsure of the present position of the file, RESTORE the file before using it.

In example 1 of figure 7-2, the ordinal 99 is assigned to the file OUTPUT, so that all data placed on file 99 is printed on the terminal or the printer (OUTPUT is a special file name). In example 2, files OLDM and NEWM are assigned ordinals, 1 and 11, respectively. In example 3, a file (name determined during execution of the program) is assigned ordinal 48. In example 4, both file name and ordinal are determined during execution. If the variable X is not an integer, it is rounded to an integer.

File ordinal zero has a special reserved meaning. Although using it in the FILE statement has no effect, it refers to the default input file in input-related statements and to the default output file in output-related statements. (See section 12, Batch Operations, J and K parameters.) These files are connected to the terminal when running interactively; for example, ordinal zero refers to the terminal when it is running interactively.

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

Figure 7-2. FILE Statement Examples

## CLOSE STATEMENT

The CLOSE statement is used to disassociate a file from the BASIC program. The associated ordinal and file buffer space become free for reassignment to another file by executing another FILE statement. The file is retained as a local file, and can be referenced again following an appropriate FILE statement. Explicitly stating the CLOSE statement rewinds the disassociated file (positions the file at the beginning). Figure 7-3 illustrates the format for the CLOSE statement.

```
CLOSE #ne

ne      Indicates file ordinal expressed as a numeric
        constant, variable, or expression.
```

Figure 7-3. CLOSE Statement Format

In figure 7-4, the first statement makes file DTFIL1 available as file ordinal 1. The second statement detaches DTFIL1 from the BASIC program and frees file ordinal 1. The third statement makes DTFIL2 available as file ordinal 1. The fourth line makes file DTFIL1 available once more, this time to be referenced by file ordinal 2. The fifth and sixth statements detach DTFIL1 and DTFIL2, respectively, from the BASIC program, and free their associated file ordinals and buffers.

```
100 FILE #1="DTFIL1"
110 CLOSE #1
120 FILE #1="DTFIL2"
130 FILE #2="DTFIL1"
140 CLOSE #1
150 CLOSE #2
160 END
```

Figure 7-4. CLOSE Statement Example

# FILE CONTROL STATEMENTS

The file control statements RESTORE, NODATA, IF END, IF MORE, and APPEND are used to manipulate files and internal data tables in various ways. For example, these file control statements can be used to check the position of the file pointer (NODATA, IF END, and IF MORE) and to move the file pointer to the beginning of a file (RESTORE) and to the end of an existing file (APPEND).

All of the file control statements can be used with binary and display format files; the RESTORE and NODATA statements can also be used with internal data tables created by the DATA statement.

## RESTORE STATEMENT

A file or internal data table has a pointer associated with it that indicates the position of the file or table. 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 data item is to be written. The RESTORE statement positions this pointer to the beginning of the file or internal data table. A file is not automatically rewound at the end of program execution. A file can be rewound with a RESTORE or CLOSE statement. Once a file has been restored, it can be written into or read. After execution of a RESTORE statement, a file is in sequential mode. If data is written to the file without using a SET statement, information that is on the file might be destroyed. The file can be in either binary or display format. Be careful not to write over any data that is to be saved. Figure 7-5 shows the formats for the RESTORE statement.

```
1.  RESTORE
2.  RESTORE #ne

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

Figure 7-5. RESTORE Statement Format

In figure 7-5, if format 1 of the RESTORE statement is used, the statement refers to the internal data block created by the DATA statement. If format 2 is used, the numeric expression ne must evaluate to an existing file ordinal; for example, the number associated with the file name through the FILE statement. The expression ne is rounded to an integer. An example of the RESTORE statement is shown in figure 7-6. The DATA, READ, and PRINT statements are described later in this section.

```
10 DATA 1,2,3
20 READ A,B,C
30 RESTORE
40 READ D
50 PRINT A;B;C;D
60 END

produces:

 1  2  3  1
```

Figure 7-6. RESTORE Statement Example

## NODATA STATEMENT

The NODATA statement can be used with files and internal data tables to test the location of the file position pointer. If the pointer is at the end-of-data, control branches to the statement at the line number specified in the NODATA statement. Thus, NODATA can be used to determine if all the data in a file or internal data table has been read. Figure 7-7 shows the format for the NODATA statement.

1. NODATA ln

2. NODATA #ne, ln

| | |
|---|---|
| ln | Indicates line number. |
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |

Figure 7-7. NODATA Statement Format

In figure 7-7, if format 1 is used, the NODATA statement refers to an internal data block created by the DATA statement. If format 2 is used, the NODATA statement refers to a binary or display format file with the ordinal that matches the ordinal specified as ne. The ordinal ne is rounded to an integer.

Files that have just been sequentially written have no data available for reading. Thus, when NODATA references a file that has just been sequentially written (and not yet repositioned with RESTORE), an end-of-file condition is indicated, and control immediately transfers to the statement at line number ln. Conversely, when NODATA references an internal data table that contains at least one item and has not yet been read, transfer of control is not executed.

The NODATA statement is typically used for end-of-information processing, as in figure 7-8. The first NODATA statement in line 110 detects an end-of-information on the internal data block and directs control to statement 150, where a message is printed. Note that since file #1 has just been written on the first time through, the NODATA statement in line 160 detects an end-of-information condition on file #1.

```
090 FILE #1="NODAT1"
100 DATA 1,2,3,4
110 NODATA 150
120 READ A
130 PRINT A
135 WRITE #1,A
140 GOTO 110
150 PRINT "END OF DATA BLOCK"
160 NODATA #1,180
170 STOP
180 PRINT "END OF FILE #1"
190 END

produces:

 1
 2
 3
 4
END OF DATA BLOCK
END OF FILE #1
```

Figure 7-8. End-of-Information Processing

## IF END STATEMENT

The IF END statement is logically equivalent to the NODATA statement, except IF END cannot refer to the internal data block created by DATA statements. The numeric expression ne in figure 7-9, is evaluated and rounded to an integer. The status of the file that has this integer as its file ordinal is then interrogated in a manner similar to that of the NODATA statement. Control is transferred to the statement line number ln if the pointer is found to be at the end of data. The IF END statement is customarily used for end-of-information processing, as illustrated in figure 7-10.

Do not use the second format because it might not be supported in future versions of BASIC. See Future System Migration Guidelines, appendix E.

1. IF END #ne THEN ln

2. IF END #ne GOTO ln

| | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |
| ln | Indicates line number. |

Figure 7-9. IF END Statement Format

```
100 FILE #1="IFEND"
110 IF END #1 GOTO 160
120 INPUT #1,A
130 PRINT ,A
140 LET S=S+A
150 GOTO 110
160 PRINT ,"---"
170 PRINT "TOTAL:",S
180 END


File IFEND contains the values 10, 10, 20,
20, 30, 30, 40, 40.  Program output:


              10
              10
              20
              20
              30
              30
              40
              40
             ---
TOTAL         200
```

Figure 7-10. IF END Statement Example

## IF MORE STATEMENT

The IF MORE statement is the logical converse of the NODATA statement and the IF END statement. Similar to the IF END statement, the IF MORE statement cannot refer to the internal data block. The arithmetic expression ne is evaluated and rounded

to an integer. The status of the file that has the integer as its file ordinal is interrogated. Control is transferred to the statement with the line number specified in ln, only if the pointer is found not to be at the end of data; for example, if there is data available for reading. Figure 7-11 illustrates the format for the IF MORE statement. Do not use the second format because it might not be supported in future versions of BASIC. See Future System Migration Guidelines, appendix E.

1. IF MORE #ne THEN ln

2. IF MORE #ne GOTO ln

| | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |
| ln | Indicates line number. |

Figure 7-11. IF MORE Statement Format

Figure 7-12 duplicates the example given for the IF END statement; however, it uses the IF MORE statement to allow for end-of-information processing. Note that using IF MORE allows the program to be shortened by one line.

```
100 FILE #1="IFEND"
120 INPUT #1,A
130 LET S=S+A
140 PRINT ,A
150 IF MORE #1 GOTO 120
160 PRINT ,"---"
170 PRINT "TOTAL:",S
180 END


File IFEND contains the values 10, 10, 20,
20, 30, 30, 40, 40.  Program output:


              10
              10
              20
              20
              30
              30
              40
              40
             ---
TOTAL         200
```

Figure 7-12. IF MORE Statement Example

## APPEND STATEMENT

The APPEND statement enables data to be added to the end of an existing binary or display format file (APPEND cannot be used with internal data tables created with the DATA statement). The format for the APPEND statement appears in figure 7-13.

APPEND #ne

| | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |

Figure 7-13. APPEND Statement Format

Executing an APPEND statement causes the file pointer associated with the file ordinal specified as ne to be positioned after the last data item on the file. The expression ne is rounded to an integer. The file mode (binary or display) is set depending on the mode of the last input or output operation for the file. If there is no preceding input or output operation, or if the last operation was a RESTORE, the mode is determined by the next output operation on the file. After APPEND, an output operation on the file, such as WRITE or PRINT, causes the information to be added to the file. Once the file has been positioned, any amount of data can be written. It is not necessary to execute an APPEND before each output statement. Any attempt to INPUT or READ from a file after an APPEND for that file and without an intervening RESTORE or SET causes the execution time diagnostic ILLEGAL INPUT ON FILE.

Figure 7-14 shows one method of adding information to the end of a file. In this case, the file used is a file of student grades. The first example program attempts to read through the file, and write at the end. An execution diagnostic results as shown. The second program has an APPEND statement inserted before the WRITE statement. This positions the pointer after the end of data and allows additions to be made. The program then repositions, reads, and prints the contents of the file showing that the new value was added to the end.

# BINARY I/O STATEMENTS AND FUNCTIONS

The following paragraphs describe the BASIC statements and functions used to read and write binary format files. The binary I/O statements are WRITE for creating binary files; READ for reading binary files; and SET for pointing to a specific word location within a binary file so the next READ or WRITE statement can reference the desired word. The binary I/O functions are LOC and LOF. LOC and LOF functions aid in the random access procedure and are described in the following text.

As stated at the beginning of this section, binary files cannot be directly connected to the terminal or output on a printer. Binary files must be disk files and can be referenced by using either the sequential or random access method. The READ and WRITE statements apply to both methods of accessing binary data, and the SET statement applies only when you want to randomly acccess binary data. Two BASIC built-in functions, LOC and LOF, aid in the random access procedure.

```
Incorrect APPEND Statement Example Program:


100 FILE #1="CREATED"
110 READ #1,A
120 IF MORE #1 THEN 110
130 LET A=99
140 WRITE #1,A
150 END


Output from Incorrect Program:


 ILLEGAL OUTPUT ON FILE  AT  140
BASIC EXECUTION ERROR


Corrected APPEND Statement Example Program:


100 FILE #1="CREATED"
110 APPEND #1
120 LET A=99
130 WRITE #1,A
140 RESTORE #1
150 READ #1,A
160 PRINT A;
170 IF MORE #1 THEN 150
180 END


File CREATED initially contains the values 96,
97, 98.  After program execution it contains:


 96  97  98  99
```

Figure 7-14. APPEND Statement Example

## WRITE STATEMENT

The WRITE statement is used to write a contiguous block of data to a binary file. The data is written into the file starting at the current position of the file pointer. No delimiters or end-of-line characters are written. Figure 7-15 shows the format for the WRITE statement.

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

| | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |
| e | Indicates expression, variable, or constant (numeric or string). |

Figure 7-15. WRITE Statement Format

When the WRITE statement is executed, the binary value of the expressions (e in figure 7-15) is written on the file indicated by the ordinal (ne in figure 7-15). The ordinal ne is rounded to an integer.

Files written by a WRITE statement can be read only by a READ statement in the same program or in another BASIC program. Note that for sequential access files, a simple WRITE operation causes the file pointer to be positioned at the current end-of-file; any attempt to READ from the file without an intervening RESTORE or set causes an execution time diagnostic ILLEGAL INPUT ON FILE. Figure 7-16 illustrates use of the WRITE statement. The binary values of 1 and 10 are written on file #1. The file is restored and the data is read, then it is printed.

```
100 FILE #1="OLDM"
110 LET A=1
120 LET B=10
130 WRITE #1,A,B
140 RESTORE #1
150 READ #1,D,E
160 PRINT D,E
170 END


produces:


 1             10
```

Figure 7-16. WRITE Statement Example

## READ STATEMENT

Binary files created by the WRITE statement are read by the READ statement. Figure 7-17 shows the format for the READ statement. See I/O For Internal Data Blocks (described later in this section) for alternate formats of the READ statement for reading internal data tables created by the DATA statement.

READ #ne,$v_1$,$v_2$, . . . ,$v_n$

| | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable or expression. |
| v | Indicates variable identifier (numeric or string). |

Figure 7-17. READ Statement Format

In figure 7-17, binary data from the file with ordinal ne is read and assigned to each of the variables $v_1$, $v_2$, . . . , $v_n$, respectively. The ordinal ne is rounded to an integer. Numeric data items should be assigned to numeric variables and string data items should be assigned to string variables; otherwise, unpredictable results can occur (no diagnostic is issued).

As each binary data item is read from the file and assigned to a variable in the READ statement list, the file pointer is advanced to the next data item. If an executing READ statement attempts to read beyond the end-of-file, the execution time diagnostic END OF DATA ON FILE is issued. Check for an end-of-file condition by using the IF END, IF MORE, or NODATA statement. (See File Control Statements in this section.)

Output files that have been sequentially written must be restored (via the RESTORE or the SET statements) to be read; otherwise, the execution time diagnostic ILLEGAL INPUT ON FILE is issued. (The RESTORE statement is described previously in this section under File Control Statements.)

Figure 7-18 illustrates the READ statement. The program creates a file named MYFILE, then writes a series of values to the file (one string value followed by 20 integer values). The program then reads the file information previously created, and displays the file contents at the terminal.

## SET STATEMENT

The SET statement positions a file so that the next READ or WRITE statement executed on that file references the desired word. Figure 7-19 shows the format of the SET statement. When the SET statement is executed, $ne_1$ and $ne_2$ are evaluated and rounded to an integer, then the file associated with the ordinal $ne_1$ is positioned at the word $ne_2$. The user is responsible for computing the file position to be set and should remember the following:

Numeric variables occupy just one word each on a binary file.

String variables occupy n words, where n is the integral of the (number of 6-bit characters in the string +9)/10+1; for example, a string variable of length 34 6-bit characters occupies INT(34+9)/10)+1=5 words. The 12-bit escape code ASCII characters count as two 6-bit characters.

If the logical blocks of information on the file to be referenced in random mode are all fixed length, the starting word position of any particular block can be readily computed. The starting word position of the nth block is ((n-1)* block length + 1). However, if variable length logical blocks are used, it is necessary to produce a table of relative logical block addresses when the file is being created and to record this table in some fixed area, such as the start of the file or in a separate file, if the table size is large and variable.

**SET #$ne_1$, $ne_2$**

| | |
|---|---|
| $ne_1$ | Indicates numeric constant, variable, or expression that evaluates to a file ordinal associated with a file name. |
| $ne_2$ | Indicates numeric constant, variable, or expression. |

Figure 7-19. SET Statement Format

In addition to the SET statement, BASIC provides the LOC and LOF functions to help position directly to the desired word location so the word stored at that location can be retrieved or written. Any attempt to position to a word location beyond the end of the file results in a diagnostic message RANDOM ACTION BEYOND EOF. The LOC function returns the current word position on the file where the next READ or WRITE operation is to start, and the LOF function returns the length of the specified binary file. The first word in the file is word 1.

Figure 7-20 shows how the SET statement can be used to randomly access a binary file containing 12 student grades organized in student number order. The student number indicates the relative position of the student's grade record within the file, and the student numbers range from 10001 to 10012. For example, student number 10001 indicates that the corresponding student grade record occupies the first position in the file, and student number 10012 indicates that its student grade record occupies the twelfth position in the file. The program prompts for a student number; once the number is entered, the corresponding student grade is displayed at the terminal.

```
100 FILE #1="MYFILE"
110 RESTORE #1            'ENSURES FILE IS AT BEGINNING
120 LET A$="WRITE A FILE OF SEQUENTIAL NUMBERS FROM 1 TO 20"
130 WRITE #1,A$
140 FOR I=1 TO 20
150 WRITE #1,I
160 NEXT I
170 RESTORE #1
180 READ #1,A$
190 PRINT A$
200 IF END #1 THEN 999
210 READ #1,A
220 PRINT A;
230 IF MORE #1 THEN 210
999 END

produces:

WRITE A FILE OF SEQUENTIAL NUMBERS FROM 1 TO 20
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
```

Figure 7-18. READ Statement Example

SET Statement Example Program:

```
100 FILE #1="STUDENT"
110 PRINT "ENTER A STUDENT NUMBER"
115 PRINT "ENTER 999 TO TERMINATE JOB"
120 INPUT N
125 IF N=999 THEN 999
130 LET N1=N-10000
140 SET #1,N1
150 READ #1,G1
160 PRINT "STUDENT NUMBER ";N;" GRADE ";G1
170 GOTO 120
999 END
```

File STUDENT contains the values 95, 89, 80, 84, 94, 78, 88, 68, 96, 79, 92, 90.
Program output:

```
ENTER A STUDENT NUMBER
ENTER 999 TO TERMINATE JOB
? 10001
STUDENT NUMBER  10001  GRADE  95
? 10002
STUDENT NUMBER  10002  GRADE  89
? 10003
STUDENT NUMBER  10003  GRADE  80
? 999
```

Figure 7-20. SET Statement Example

## LOC FUNCTION

The LOC function returns the current word position of a binary file. That is, LOC returns the position at which the next READ or WRITE operation on the file is to begin. The format for the LOC function appears in figure 7-21.

| LOC(ne) | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable or expression. |

Figure 7-21. LOC Function Format

The value returned by LOC is updated after each item is read or written by the READ or WRITE statement. The value is incremented by 1 for numeric values and by n for string variables, where n is a function of the length of the string. See discussion under SET statement. If RESTORE is used on the file, LOC yields the value of 1. The LOC function can also be used with the LOF function to detect an end-of-file condition.

## LOF FUNCTION

The LOF function returns the length in words of a binary file. Figure 7-22 shows the format for the LOF function.

| LOF(ne) | |
|---|---|
| ne | Indicates file ordinal expressed as a numeric constant, variable, or expression. |

Figure 7-22. LOF Function Format

When the value returned by the LOC function equals the value returned by the LOF function, the file is positioned at the last word in the file. When LOC=LOF+1, all data has been used, and an end-of-file condition is indicated. An example of both the LOC and LOF function is shown in figure 7-23.

LOC and LOF Function Example Program:

```
100 FILE #1="STUDENT"
105 RESTORE #1
110 LET M=10000
120 READ #1,G
130 LET M=M+1
140 PRINT M,G
150 IF MORE #1 THEN 120
200 RESTORE #1
210 LET S=0
220 READ #1,G1
230 IF LOC(1)=LOF(1)+1 THEN 260
240 LET S=S+G1
250 GOTO 220
260 LET A=S/LOF(1)
270 PRINT "CLASS AVERAGE ";A
300 END
```

File STUDENT contains the values 95, 89, 80, 84, 94, 78, 88, 68, 96, 79, 92, 90.
Program output:

```
 10001          95
 10002          89
 10003          80
 10004          84
 10005          94
 10006          78
 10007          88
 10008          68
 10009          96
 10010          79
 10011          92
 10012          90
CLASS AVERAGE   78.5833
```

Figure 7-23. Example of LOC and LOF Functions

# DISPLAY FORMAT I/O STATEMENTS AND FUNCTIONS

The statements INPUT, DELIMIT, PRINT, PRINT USING, image, MARGIN, and SETDIGITS are used for display format I/O. The INPUT statement is used to read display format input. The various forms of the PRINT statement, and the MARGIN and SETDIGITS statements, are used to create display format output and to control the output format.

## INPUT STATEMENT

The INPUT statement permits display format data to be read from a file or from the terminal during program execution. The two forms of the INPUT statement are shown in figure 7-24.

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

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

$v_1 - v_n$ Indicates string or numeric variables.

ne Indicates file ordinal expressed as a numeric constant, variable, or expression.

Figure 7-24. INPUT Statement Format

### Terminal Input

When a BASIC program is run interactively from a terminal, the INPUT statement without a file ordinal (format 1 in figure 7-24) 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. 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. No assignment of values in a reply takes place until the reply is validated with respect to the type of data, the number of input items, and the allowable range for each item.

Unless DELIMIT is in effect, numeric constants must be separated by commas or blanks, and string constants must be separated by commas. (See DELIMIT Statement in this section.)

A carriage return marks the end of the input reply (the end of the data to be entered). If an insufficient number of data items is entered, the diagnostic NOT ENOUGH DATA, REENTER OR TYPE IN MORE is issued. When this message appears at the terminal, either reenter the entire input line or enter a non-blank delimiter followed by the additional data items. Starting a subsequent input line with a delimiter indicates that it is a continuation of the first input line. One of the diagnostic messages TOO MUCH DATA, RETYPE INPUT or ILLEGAL DATA, or RETYPE INPUT is issued if too much data or unacceptable data is entered. In these cases, retype the entire data list.

Rules for entering data from a terminal are as follows:

- Numeric items must be delimited by commas or blanks and string items must be delimited by commas, unless a DELIMIT statement (described later) is in effect.
- A carriage return marks the end of the input reply.
- If an insufficient number of data items is entered, BASIC issues a request to reenter or type in more data.
- If too much data or unacceptable data is entered, BASIC requests that the data be reentered.
- All leading and trailing blanks (blanks between the last nonblank character of the data list and the carriage return) are eliminated from the input line.
- Redundant delimiters preceding or following data items are ignored.
- If a line ends with a delimiter, the input reply is assumed to be continued on the following line and another input prompt is issued.

Figure 7-25 illustrates a program example of using the INPUT statement to enter data from the terminal during program execution. The example shows what happens if insufficient data is supplied in response to an INPUT request. The input reply line is completely reentered (the 2 is repeated). Alternatives are to add data to the previous input reply (,2) or to continue the input reply line on the next line (2,).

### File Input

The second format in figure 7-24 reads from the file specified by the file ordinal (ne). It is essentially the same as input from a terminal except that TOO MUCH DATA and NOT ENOUGH DATA conditions are not applicable.

Rules for data input from a file are as follows:

- A display format file contains line images that can be printed at a terminal. However, the INPUT statement regards the file as one block of contiguous values with no special regard for the end of lines other than as item delimiters.
- Unless a DELIMIT statement specifies other types of delimiters, numeric items must be delimited by commas or blanks and string items must be delimited by commas.
- End-of-line (EOL) is a special indicator that marks the end of each line image on the file. It does not mark the end of the input reply from a file as a carriage return does for a terminal line. EOL simply acts as a delimiter in the same way as does a comma.
- When entering data at a terminal, an input reply has to have the same number of items that is requested by the program. Too many or too few items cause an error. When entering data from a file, if more data is requested than is available in the current line, data is automatically taken from the following line(s). If more data exists in the current line than is needed, the unused data is simply retained for the next INPUT. No diagnostic is issued. For example, the remainder of a partially-used line is not skipped.

The program terminates with the diagnostic END OF DATA ON FILE if it attempts to INPUT more data than exists in the file.

The program terminates with the diagnostic ILLEGAL DATA ON FILE if invalid numbers or strings are input. However, the file is positioned to the following delimiter to facilitate recovery by using ON ERROR. See Error and Interrupt Processing, section 4.

Redundant leading and trailing delimiters are ignored.

The program terminates with ILLEGAL INPUT ON FILE if an INPUT statement attempts to read an output file whose file pointer has not been moved to the beginning of the file (via the RESTORE statement).

If the file ordinal referenced in the second format of figure 7-24 is associated with the default input file (J parameter on BASIC control statement), the statement operates according to the format 1 rules; for example, if the file is connected or assigned to a terminal, input is prompted, and too much or too little data is diagnosed as an error. File ordinal 0 is automatically associated with the default input file (no FILE statement is required). Also, if the default input file is not connected to a terminal, the format 1 statements operate according to format 2 rules.

Example 1:

```
100 LET A$="ABCDEF"
110 PRINT "ENTER A NUMBER AND A STRING"
120 INPUT X,A$(4:5)            'TWO VALUES MUST BE SUPPLIED
130 REM NOTE INPUT INTO SUBSTRING OF A$ IN PROGRAM OUTPUT
140 PRINT "X=";X,"A$=";A$
150 END
```

produces:

```
ENTER A NUMBER AND A STRING
? 2
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  120
? 2,48
X= 0           A$=ABC48F
```

Example 2:

```
100 LET A$="ABCDEFGHI"
110 PRINT "ORIGINAL A$=";A$
120 PRINT "ENTER 5 STRING VALUES."
130 INPUT A$(4:5),B$,C$,D$,E$
140 PRINT "NOTE INPUT INTO SUBSTRING OF A$"
150 PRINT "NEW A$=";A$
160 END
```

produces:

```
ORIGINAL A$=ABCDEFGHI
ENTER 5 STRING VALUES.
? zz,trailing comma means more data on next line,
? 3rd value
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  130
? ,start with comma to add more (4th value)
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  130
? 11111,22,333,44,complete reentered reply
NOTE INPUT INTO SUBSTRING OF A$
NEW A$=ABC11111FGHI
```

Figure 7-25. INPUT Statement Example

## DELIMIT STATEMENT

DELIMIT specifies the character or characters to be used as input item separators. Figure 7-26 illustrates the formats for the DELIMIT statement. The characters specified override the default separators, comma and blank. Any characters, or abbreviations for characters (appendix A), can be specified as separators. If the abbreviation CR is specified, the entire line is accepted as one item. All preceding and trailing blanks are returned when CR is a delimiter. The operating system can add or delete trailing blanks on data lines entered through the terminal or card reader.

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

| | |
|---|---|
| ch | Indicates any character or abbreviation. |
| ne | Indicates numeric constant or variable (used as the file ordinal). |

Figure 7-26. DELIMIT Statement Format

Zero, one, two, or three characters can 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 default input file; in interactive mode, this file is the terminal. (See section 12, Batch Operations, the J parameter.)

Examples:

```
10 DELIMIT (CR)

55 DELIMIT #1,(Δ), (;)

110 DELIMIT

155 DELIMIT #1
```

In the first DELIMIT statement (line number 10), if the program is run interactively from the terminal, the input requests read all information typed up to the carriage return (end-of-line if processed in batch mode) into a single variable. It should be noted that this form usually is used to read data into a string variable. In the second example (line number 55), a blank and semicolon (;) are interpreted as delimiters (numeric and string) when entering input from a file with an ordinal of 1. The succeeding lines 110 and 155 restore the default delimiters (blank and comma).

The following descriptions compare DELIMIT not in effect and DELIMIT in effect.

### DELIMIT Not in Effect (Normal Case)

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

When input is from a terminal and carriage return is encountered before the input list is satisfied, the message NOT ENOUGH DATA, REENTER OR TYPE IN MORE is issued.

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

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

When input is from a file and data exists on the input line after the input list is satisfied, this excess data item is retained for the next INPUT request. No diagnostic is issued.

If a delimiter is encountered after the input list is satisfied, the delimiter is ignored.

Comma is the delimiter for all input items (numbers, quoted strings, and unquoted strings). Blank is a delimiter for numbers, but not for strings. When input is from a terminal, the input reply can be continued onto the next line by terminating the current line with a comma delimiter. Leading and trailing blanks are ignored.

### DELIMIT in Effect

Default delimiters are turned off except carriage return 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, leading and trailing blanks, but not delimiters, are valid string characters.

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

Carriage return (end-of-line on files) is always a delimiter and need not be explicitly defined in a DELIMIT statement. The TOO MUCH DATA and NOT ENOUGH DATA conditions are handled as in the normal case.

If CR is explicitly specified as a delimiter and insufficient data is supplied, the NOT ENOUGH DATA message is suppressed and the additional data items must be entered. It is not possible to reenter data. Input prompts continue to be issued until the entire list has been satisfied.

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

Blanks are ignored if the item being input is a number or if blank has not been specified as a delimiter.

Any specified delimiter, other than CR or blank, causes the input reply to be continued on the next line in interactive mode if the delimiter follows the last item on the line.

## PRINT STATEMENT

The PRINT statement writes display format data on a terminal or a file. The formats for the PRINT statement are shown in figure 7-27.

1. PRINT $e_1$ d $e_2$ d . . . $e_n$ d

2. PRINT #ne, $e_1$ d $e_2$ d . . . $e_n$ d

e Indicates constant, variable, or expression (numeric or string).

d Indicates delimiter (comma or semicolon); final delimiter is optional.

ne Indicates file ordinal expressed as a numeric constant, variable, or expression.

Figure 7-27. PRINT Statement Format

When the PRINT statement is executed, the value of each expression (e) is printed according to standard format. Spacing between the printed values is controlled by delimiters (d) in the print list. If no expressions are specified, a blank line is printed.

The PRINT statement of format 1 in figure 7-27 writes on a default file named OUTPUT. When running the program interactively, file OUTPUT is the terminal; when running in batch mode, file OUTPUT is the printer. The default file name can be changed by using the K option of the BASIC control statement. (See Batch Operations, section 12.) The PRINT statement of format 2 in figure 7-27 writes on a file whose ordinal is ne.

Format 1 prefixes a carriage control character at the beginning of each line. Except for the first printed line, this character is always a blank. The carriage control character is not normally affixed to lines output if the second PRINT format is used, unless the file ordinal referenced is that of the default print file OUTPUT or default file specified by the K option of the batch BASIC control statement.

To assure that the data from the format 2 PRINT statements print in proper order when the files are connected or assigned to the terminal, RESTORE or CLOSE the file before executing any other I/O statement that references a connected file.

When printing data on a file that is to be read later with the INPUT statement, ensure that items on the file are separated by delimiters. When only numbers are printed, default delimiters (blanks) are automatically included in the output. When printing a file containing strings, each string must be printed on a separate line, or explicitly specified print delimiter characters must be printed between data items. In addition, if the string includes leading or trailing blanks, quote marks must be printed around the string. Figure 7-28 illustrates an example of the PRINT statement.

```
10 LET A$="STRING"
20 LET B$="STRING2"
30 LET X=1
40 FILE #7="FILEX"
50 REM PRINT ON FILE
60 PRINT #7,A$;",";B$
70 PRINT #7,X,X+X,SIN(X)
80 REM PRINT ON TERMINAL
90 PRINT A$;",";B$
100 PRINT X,X+X,SIN(X)
110 END


produces:


STRING,STRING2
 1             2             .841471
```

Figure 7-28. PRINT Statement Example

## DEFAULT PRINT 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 are explained below.

### Numeric Formats

Numeric values are formatted in one of the three standard formats shown in table 7-4. A program example using the numeric formats is shown in figure 7-29. The following items refer to the output formats in table 7-4.

The n represents a numeric digit.

The s represents a minus sign if the value is negative and a blank if the value is positive.

Each format is terminated by one trailing blank.

Leading zeros are suppressed.

Trailing zeros after a decimal are suppressed in F format, but not in E format.

The final digit in both the first and second formats of table 7-4 is obtained by rounding to the sixth place from the first nonzero digit, for example:

123 456 789.453 is rounded to 123 457 000.000
and
.001234567 is rounded to .001234570

### String Formats

String values are formatted as a contiguous group of characters. For example, only the characters in the original string value are displayed; no quotation marks or blanks are added.

```
100 LET A1=0
110 LET B1=-124
120 LET C1=123456
130 LET D1=1234567
140 LET E1=123456.789
150 LET F1=-.00192
160 LET G1=1234567890
165 LET H1=1234567.8
170 LET J1=.07623488
180 LET K1=-.0000192
190 PRINT "INTERNAL VALUE"
200 PRINT "0","-124","123456","1234567","123456.789"
210 PRINT
220 PRINT "OUTPUT FORMAT"
230 PRINT
240 PRINT A1,B1,C1,D1,E1
250 PRINT
260 PRINT "INTERNAL VALUE"
270 PRINT "-.00192","1234567890","1234567.8",".07623488","-.0000192"
280 PRINT
290 PRINT "OUTPUT FORMAT"
300 PRINT F1,G1,H1,J1,K1
310 END

produces:

INTERNAL VALUE
0              -124           123456         1234567        123456.789

OUTPUT FORMAT

 0             -124            123456         1.23457E+6     123457.

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

OUTPUT FORMAT
-.00192         1.23457E+9     1.23457E+6     7.62349E-2     -1.92000E-5
```

Figure 7-29. Program Example of Numeric Formats

TABLE 7-4. STANDARD NUMERIC OUTPUT FORMATS

| Internal Value | Output Format Used |
|---|---|
| Exact integers less than seven digits. | snnnnnn (I format) |
| Nonintegers that after rounding can be represented as accurately in decimal notation as in exponential (E format) notation. | snnnnnnn (where one n represents a decimal point) (F format) |
| All other numbers. | sn.nnnnnE+nnn (E format) |

String constants are printed exactly as they appear in the PRINT statement, without the quotation marks. Examples of string formats using the PRINT statement are illustrated in figure 7-30.

## PRINT ZONING

The print line is divided into zones of 15 spaces each. Unless the MARGIN statement (described later) is used to specify some other value, the default margin (line length) is 75; there are five print zones in a line. 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. If a print zone is exactly filled by a print item, a comma separator causes the print mechanism to skip over the following print zone.

```
10 LET X=2
20 LET Y=2
30 LET Z=2
40 PRINT "ANSWER","X AND Z ARE ";Z,"X*Y*Z=";X*Y*Z
50 PRINT "ANSWER","X AND Z ARE "Z;"X*Y*Z=";X*Y*Z
60 END


produces:


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

Figure 7-30. String Formats Using the PRINT Statement

A semicolon used as a separator has no spacing effect (the print line zoning effect is inhibited). Numbers are printed, preceded by a blank or a minus sign, and followed by another blank, so two positive numbers are separated by two blanks. (See figure 7-31.)

When a semicolon is used to separate strings, the strings are printed consecutively without any preceding or intervening blanks, as shown in figure 7-32.

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 can occupy more than one zone.

Successive commas can be used to skip zones. Each comma causes a skip to the beginning of the next print zone. Successive semicolons have no spacing effect.

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.

If the formatted print item does not fit entirely on the current line, it is printed as the first item of the next line. If an item does not fit on an empty line, it is broken at the margin and continued on the next line. (See figure 7-33.)

## TAB Function

The TAB function causes the printing mechanism to tab to a specified column. Printing can commence in the specified column. The semicolon should be used as a separator when the TAB function is used because the semicolon has no spacing effect. Figure 7-34 illustrates the format for the TAB function. The TAB function is legal only in the PRINT statement; it should not be used in MAT PRINT or PRINT USING statements. If the argument is less than the current position, the print mechanism is positioned to the specified column of the next print line. If the argument is greater than the current line margin, it is divided by the line margin, and the remainder is used as the argument. If the argument is less than one, a warning diagnostic is issued, and the value one is substituted. Examples of the TAB function appear in figure 7-35.

```
10 LET A1=123
20 LET B2=256
30 PRINT "12345678901234567890"
40 PRINT A1;B2
50 PRINT -A1;-B2
60 PRINT -A1;14.3
70 END


produces:


12345678901234567890
 123  256
-123 -256
-123  14.3
```

Figure 7-31. Use of Semicolon With Numeric Data

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


produces:


THIS ISAN EXAMPLE
THIS IS        AN EXAMPLE
```

Figure 7-32. Use of Semicolon With String Data

## PRINT USING STATEMENT

The PRINT USING statement writes display format data on a terminal or file. The formats for the PRINT USING statement are shown in figure 7-36.

When the PRINT USING statement is executed, the value of each expression (e) is printed in the format specified by an image. The image is defined in the image statement (1n) or in the string expression (se).

```
Example 1:


10 FOR I=1 TO 10
15 PRINT I
20 NEXT I
30 END


produces:


 1
 2
 3
 4
 5
 6
 7
 8
 9
 10


Example 2:


10 FOR I=1 TO 10
15 PRINT I,
20 NEXT I
30 END


produces:


 1             2             3             4             5
 6             7             8             9             10


Example 3:


10 FOR I=1 TO 10
15 PRINT I;
20 NEXT I
30 END


produces:


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

Figure 7-33. Print Zoning Examples

**TAB(ne)**

| | |
|---|---|
| ne | Indicates constant, variable, or expression indicating the print position number. |

Figure 7-34. TAB Function Format

Format 1 (figure 7-36) prefixes a carriage control character at the beginning of each line. Except for the first printed line, this character is always a blank. The carriage control character is not normally prefixed to lines output by format 2, shown in figure 7-36, unless the file ordinal referenced is that of the default print file (for example, OUTPUT or the file specified by the K

option of the BASIC control statement). To assure that the data prints in proper order when the files are connected or assigned to the terminal under NOS by using format 2 PRINT USING statements, RESTORE or CLOSE the file before executing any other I/O statement that references a connected file.

Example 1:

```
20 PRINT TAB(10);"1";TAB(20);"2";TAB(30);"3"
30 PRINT "12345678901234567890123456789O"
40 END
```

produces:

```
         1         2         3
12345678901234567890123456789O
```

Example 2:

```
100 LET I1=12345678
110 LET I2=123456789
120 LET I3=12345678901
130 LET D1=123.4
140 LET D2=123.456
150 LET D3=123.4567
160 PRINT I1,TAB(25);D1
165 PRINT I2,TAB(25);D2
170 PRINT I3,TAB(25);D3
180 END
```

produces:

```
 1.23457E+7              123.4
 1.23457E+8              123.456
 1.23457E+10             123.457
```

Figure 7-35. TAB Function Examples

1. PRINT USING ln, $e_1$ d $e_2$ d . . . $e_n$ d

   PRINT USING se, $e_1$ d $e_2$ d . . . $e_n$ d

2. PRINT #ne USING ln, $e_1$ d $e_2$ d . . . $e_n$ d

   PRINT #ne USING se, $e_1$ d $e_2$ d . . . $e_n$ d

| | |
|---|---|
| e | Indicates constant, variable, or expression (numeric or string). |
| d | Indicates delimiter (comma or semicolon); final delimiter is optional. |
| ne | Indicates numeric constant, variable, or expression that evaluates to a file ordinal associated with a file name. |
| ln | Indicates line number of image statement (described later in this section). |
| se | Indicates string constant, variable, or expression describing the image required to format the output (described later). |

Figure 7-36. PRINT USING Statement Formats

When printing data on a file that is to be read later with the INPUT statement, ensure that items on the file are separated by delimiters. When only numbers are printed, default delimiters (blanks) are automatically included in the output. When printing a file containing strings, each string must be printed on a separate line or explicitly specified input delimiters must be included between data items. In addition to maintain leading and trailing blanks in the string, you must print the strings within quotation mark characters.

## IMAGE

The image for a PRINT USING statement describes the output format for the value to be printed. It consists of format fields for each value in the print list and optional separating literals. Each character in the image corresponds to one character in the printed output. When the format field 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 as well as the extra characters to be printed with the value (for example, decimal point, dollar sign, or asterisks).

As described under the PRINT USING statement, an image can be written as a separate statement or as a string expression. An image string is constructed as shown in figure 7-37. An image statement is not executable, so it has no effect on the results of the program if it is encountered during a normal execution sequence. The format of an image statement is as shown in figure 7-38.

$L_0F_1L_1F_2L_2 \ldots F_nL_n$

| | |
|---|---|
| L (literals) | Characters are to be printed exactly as they appear. |
| F (fields) | Specifications that picture formats for printing numeric and/or string values. |

Figure 7-37. The Image for a PRINT USING Statement

```
:image
```

Figure 7-38. Image Statement Format

Literals can contain any character or combination of characters that do not constitute a format field (F). Format fields are constructed from specification characters #, $, *, <, >, ^, +, -, comma, (, and ).

A separate image statement is referenced by its line number (from within the PRINT USING statement) as shown in example 1, figure 7-39. For separate image statements, a trailing literal cannot be terminated by a blank. For example:

```
:#LITERAL ΔΔΔ
```

is compiled as:

```
:#LITERAL
```

```
Example 1:


100 LET T1=544
200 PRINT USING 300,T1
300 :TOTAL OF ORDERS #####
400 END


produces:


TOTAL OF ORDERS   544


Example 2:


100 LET T1=544
200 PRINT USING "TOTAL OF ORDERS #####",T1
300 END


produces:


TOTAL OF ORDERS   544


Example 3:


100 LET T1=544
110 LET A$="TOTAL OF ORDERS #####"
120 PRINT USING A$,T1
130 END


produces:


TOTAL OF ORDERS   544
```

Figure 7-39. Image With PRINT USING Statement

If the image is a string, it can be included as a string constant within the PRINT statement (example 2 in figure 7-39) or it can be specified as a string expression referenced in the PRINT statement (example 3).

When a value is printed according to a field of an image, the literal following the field (if any) is also printed. Only that part of the image that is required by the print list of the associated PRINT USING statement is used.

Delimiters between items in the print list of the PRINT USING statement do not have the same meaning as they do in the PRINT statement because there is no print zoning. The final delimiter, if present, indicates that subsequent printing continues on the same line until the line is filled. Whether or not this delimiter is a comma or a semicolon, the next item printed starts in the next available space. If absent, subsequent printing begins on the next line.

If the number of values to be output in a PRINT USING statement is greater than the number of format specifications in the image, the format specifications are reused until all the variables have been output. For this case, the delimiter after the last item printed before repetition directs line action. If this delimiter is a comma, a new line is started when the image is repeated. If the delimiter is a semicolon, printing continues on the same line.

Figure 7-40 illustrates the use of the final delimiter. The result is the same if the final delimiter is a semicolon instead of a comma. Figure 7-41 shows how delimiters can affect the output when the image is reused.

```
10 LET N=5
25 PRINT USING 30,N,
30 :$TOTALS PAGE ### DATE
40 PRINT DAT$
50 END


produces:


$TOTALS PAGE   5 DATE 81/06/23.
```

Figure 7-40. Delimiters in Image

```
100 LET N=5
101 LET M=10
102 PRINT USING "##",N;M
103 PRINT USING "##",N,M
104 END


produces:


 510
 5
10
```

Figure 7-41. Delimiters in Image Reused

With respect to the print margin (the maximum line length), the output generated by PRINT USING is treated as one string. That is, if all of the output formatted by PRINT USING does not fit on the current line, it is broken at the margin and continued on the next line.

## Format Fields

There are three kinds of format fields available for use in images: numeric, string, and neuter. Numeric fields can be further subdivided into integer, fixed-point and floating-point fields. Sign control and special editing options are available for numeric fields. The numeric, string, and neuter fields are described in table 7-5. Table 7-6 describes the special options. Examples of these fields types and options are shown in figures 7-42 and 7-43.

TABLE 7-5. TYPES OF FIELDS

| Type of Field | | Image Representation | Output Format |
|---|---|---|---|
| Numeric | Integer | Indicated by any number of pound signs (#). | Value right-justified. Rounded to integer. |
| | Fixed-Point | Indicated by any number of pound signs (#) with a single leading, embedded, or trailing decimal point. | Decimal point positioned as specified, decimal portion rounded to fit up to 14 digits of accuracy. |
| | Floating-Point | Indicated by a fixed-point specification followed by at least two, and usually five, circumflexes (^). | Decimal point shifted to fit format specification and exponent adjusted accordingly. Value is rounded as for fixed-point. |
| String | | Indicated by the < or > followed by any number of pound signs (#). | Value left-justified and right-truncated (default) or right-justified and left-truncated according to leading character.<br><br>If the leading character is <, the value is left-justified in the field and right-truncated, if necessary.<br><br>If the leading character is >, the value is right-justified in the field and left-truncated, if necessary. |
| Neuter | | Indicated by any number of pound signs (#). | Numeric value right-justified and truncated to integer. String value left-justified and right-truncated. |

TABLE 7-6. SIGN AND EDIT OPTIONS

| Option Character | | Image Representation | Output Format |
|---|---|---|---|
| Sign | Blank or no sign | No sign control specified. | If a value is negative, a minus immediately precedes the leftmost significant digit, for example, minus is treated as high-order significant digit. If a value is positive, sign is not printed. |
| | + | Plus sign specified as first character in | Plus is printed for positive values; minus for negative values. |
| | - | Minus sign specified as first character in the field. | Blank is printed for positive values; minus for negative values. |
| | ( ) | Parentheses enclosing field. | Negative values enclosed in floating parentheses. Positive values enclosed in blanks. |
| | DB/CR | DB or CR specified as last two characters of the field. | Negative value followed by DB or CR. Positive value followed by two blanks. |
| Comma Insertion | | Commas can be inserted between any characters of a numeric field that can be replaced by digits when the field is used; for example, between pound signs (#), dollar signs ($), or asterisks (*). | Commas will be printed where they occur in the numeric image, provided they are surrounded by digits. |
| Floating $ | | Leading pound signs (#) in a numeric specification replaced by dollar signs ($). | A dollar sign is printed in the rightmost dollar sign position in the image that was not replaced by a digit. Unused dollar signs are replaced by blanks. |
| Check Protect | | Leading pound signs (#) or pound signs (#) following dollar signs in a numeric specification replaced by asterisks (*). | An asterisk is printed in every asterisk position in the image that was not replaced by a digit. Unused commas are replaced by asterisks. |

```
Example 1 (integer field):


10 LET A=12345.67
15 PRINT USING 20,A
20 :SUMMARY TOTAL=#######
30 END


produces:


SUMMARY TOTAL=  12346


Example 2 (fixed-point field):


10 LET A=12345.678
20 PRINT USING "TOTAL COST #####.##",A
30 END


produces:


TOTAL COST 12345.68


Example 3 (floating-point field):


10 LET A=12.345E01
20 PRINT USING "#.####^^^^",A
30 END


produces:


1.2345E+2


Example 4 (string and neuter fields):


10 LET A=12345
11 PRINT USING 12,"FRACTION =",A
12 :     <############     #####
13 END


produces:


     FRACTION =       12345
```

Figure 7-42. Format Field Types

A field begins when a combination of characters is identified. A field ends when a literal is encountered; for example, when a combination of characters appears that do not conform to the order restrictions described below or contain characters that are not allowable field characters. A field also ends when an end-of-line is encountered (or end-of-string). The fields of the image statement are described in figure 7-44.

## Order Restrictions

The following order restrictions govern the allowable combinations of specification characters in fields.

Plus and minus signs and the left parenthesis sign indicator, if present, must be positioned before all other characters of a field. Only one sign indicator is permitted per format field.

Dollar signs ($), if present, must appear following the plus or minus sign. They cannot follow a decimal point.

Any asterisks, if present, must follow the sign and/or dollar signs. Asterisks cannot follow a decimal point.

Pound signs (#) can appear anywhere after the optional sign unless dollar signs and/or asterisks are present. If they are present, the pound sign can only appear after them.

The trailing sign, right parenthesis, or DB/CR, if required, can only appear as the last characters of an image field.

Commas can appear anywhere provided they are surrounded by characters that can be replaced by digits when the field is used. Commas cannot, for example, appear next to a decimal point, a sign, or the first dollar sign.

A decimal point, if present, can appear anywhere after a leading sign or parenthesis and before a trailing sign indicator. There can be only one decimal point per field.

Figure 7-45 illustrates the result of inadvertently including a field character in a literal. Figure 7-45 shows the corrected version.

In figure 7-45, the first pound sign is identified as a separate field and is replaced by the present value of N. Since there were no further items in the print list, the literal (a blank) following the field is printed, and scanning stops.

```
Example 1 (sign specifications):


10 PRINT USING 30,11,11,11
20 PRINT USING 30,-12,-12,-12
30 :+##    -##   ###
40 END


produces:


+11     11    11
-12    -12   -12


Example 2 (comma insertion):


10 PRINT USING "###,###,###,###",1000000
20 END


produces:


      1,000,000


Example 3 (parentheses and DB/CR sign options):


500 PRINT USING 550,1000.588,1000.588
520 PRINT USING 550,-14738.10,-14738.10
550 :(###,###,###.##) OR ###,###,###.##DB YOUR CHOICE
600 END


produces:


       1,000.59  OR       1,000.59   YOUR CHOICE
     (14,738.10) OR      14,738.10DB YOUR CHOICE


Example 4 (floating dollar option):


600 PRINT USING 650,10.75,138.7,111.888
610 PRINT USING 650,-1738,-28,-29
650 :$$,$$$.##CR    $$$$$$    ($$$###.##)
660 PRINT USING "+$$$$$",-7
700 END


produces:


   $10.75        $139       $111.89
$1,738.00CR      $-28      ($ 29.00)
   -$7
```

Figure 7-43. Sign and Edit Option Examples (Sheet 1 of 2)

```
Example 5 (check protect):

600 LET F$="$***,***.##"
610 PRINT USING F$,1745.50
620 PRINT USING F$,25
700 END

produces:

$**1,745.50
$*****25.00
```

Figure 7-43. Sign and Edit Option Examples (Sheet 2 of 2)

```
1010 :TOTALS : $$,$##.##    #####$$$$#.##AMOUNT:  $$,###.##
              ①      ②      ③ ④   ⑤      ⑥       ⑦      ⑧
```

① The character T is not an allowable format field character so it indicates the start of a literal.

② The $$ begins a field. One $ by itself is not considered a numeric field.

③ Blanks are not allowable format field characters so they indicate the end of the field and beginning of a literal.

④ The # begins a new field.

⑤ The $ because of position cannot be part of the previous field, so that field ends. $$ begins a new field.

⑥ The A is not an allowable format field character, so it indicates the start of a literal.

⑦ The $$ begins a new field.

⑧ The end-of-line terminates the last field.

Figure 7-44. Fields of Image Statement Identified

```
20 LET N=5
21 PRINT USING "A IS # ## IN THE LIST", N

produces:

A IS 5
```

Figure 7-45. Field Character in Literal

```
22 PRINT USING "A IS #  ##  IN THE LIST",
   "#", N

produces:

A  IS #   5 IN THE LIST
```

Figure 7-46. Correction of Field Character Use

In figure 7-46, the string # is printed according to the neuter field # and the number is printed according to the neuter field ##.

## Special Cases

Each of the characters that comprises a field is a place holder. That is, a blank, a symbol, or a digit replaces each field character. The following section deals with rules of placement, large fields, accuracy, field overflow, and signs.

There are no restrictions on the number of pound signs allowed in a format field in an image. However, since the machine word size allows a maximum of 14 digits of accuracy, only 14 digits appear on output.

If the value to be printed in an integer or fixed-point numeric format field is greater than $10^{15}-1$, an * is printed followed by the number printed in floating-point format. (See example 1 in figure 7-47.) If printing a value according to a fixed-point format yields more than 14 digits, the least significant fractional digit positions are filled with blanks so that only 14 digits print. (See example 2 in figure 7-47.)

Example 1:

```
130 PRINT USING "####################",1.08988E20
140 END
```

produces:

```
*+1E+20
```

Example 2:

```
100 LET A=-7.82
120 PRINT USING 130,A
130 :##.###############DB
140 END
```

produces:

```
 7.8200000000000 DB
```

Example 3:

```
100 LET A=7.82
120 PRINT USING 130,A
130 :#.################^^^^^
140 END
```

produces:

```
 .78200000000000  E+001
```

Example 4:

```
100 LET A=-78
200 PRINT USING "##    .##";A,7.82
300 END
```

produces:

```
*-78    *7.82
```

Example 5:

```
400 LET A=-17.82
410 PRINT USING "$.##",A
420 END
```

produces:

```
****
```

Example 6:

```
800 LET A=12000000000.0
810 PRINT USING "#.## ^^",A
820 END
```

produces:

```
1.20E*10
```

Example 7:

```
500 LET A$="THIS IS THE TOTAL"
600 PRINT USING "<################",A$
700 PRINT USING ">################",A$
800 END
```

produces:

```
THIS IS THE TOTA
HIS IS THE TOTAL
```

Figure 7-47. Special Cases for Format Fields

If more than 14 fractional digits are requested in a floating-point format field, all digit positions beyond the 14th are replaced by blanks. The print positions of the exponent value are not affected. (See example 3 in figure 7-47.) Unless DELIMIT is used, this result is not readable by a BASIC program because the blank delimits .78200000000000 from E + 001.

If a number, including its sign, is to be printed but the number is too large for the image specification, overflow occurs. (See example 4 in figure 7-47.) When an overflow condition occurs, if the image field includes any of the special edit characters, $, *, DB, CR or (), which normally denote a monetary field, the entire printed field is filled with asterisks to indicate that an error has occurred. (See example 5 in figure 7-47.) This is a significant feature when printing a monetary value, for example, printing a check, because it prevents the printing of an unexpectedly large value. When an overflow condition occurs, but the image field does not contain any of the special edit characters, one asterisk is printed and the image is expanded to accommodate the size of the value to be printed, shifting all other fields to the right.

Because of the nature of floating-point representation, only the exponent portion of a floating-point number can overflow. In example 6 in figure 7-47, an asterisk is printed following the E, and the actual exponent is printed, shifting all other fields to the right; the result is in the format shown in the PRINT USING statement at line number 810.

A string that is too large to be printed in the image specification is truncated on the right or left, depending on whether the format specifies left-justification or right-justification. (See example 7 in figure 7-47.)

## MARGIN STATEMENT

This statement defines the right-hand margin for printed output. It overrides the default margin of 75. The formats for the MARGIN statement are shown in figure 7-48. The MARGIN statement permits the building of long lines for output on wide-carriage terminals or other devices. It also allows the program to control maximum record lengths written on files. When the MARGIN statement is used without specifying a file, it applies to standard output, such as the terminal or the printer. Output can be different than expected since it can be affected by IAF, INTERCOM, or the type of terminal used. Commands are available to change these results. See the NOS Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the INTERCOM Version 5 reference manual.

1. MARGIN $ne_2$
2. MARGIN #$ne_1$, $ne_2$

| | |
|---|---|
| $ne_1$ | Indicates file ordinal expressed as a numeric constant, variable, or expression. |
| $ne_2$ | Indicates numeric constant, variable, or expression. |

Figure 7-48. MARGIN Statement Formats

In figure 7-48, the expression $ne_2$ is evaluated and truncated to an integer. It must be in the range 0 through 131070, and its value affects all PRINT statements to the associated file or the terminal until another MARGIN statement is executed. If the number of characters in the item to be printed is longer than the defined margin, it is broken into pieces so that as many lines as required are used.

A margin value of 0 indicates that no upper limit applies (the margin is effectively set to infinity). This allows a continuous stream of characters with no line terminators to be generated. Figure 7-49 shows examples of the MARGIN statement.

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, rounded, and used as the right margin value for PRINT statements.

```
200 MARGIN #6, 136
310 MARGIN I*J/K
```

Figure 7-49. MARGIN Statement Example

Figure 7-50 shows that MARGIN controls the number of logical characters in the line, as opposed to physical 6-bit characters; for example, in ASCII mode 12-bit escape code characters count as one character and a line can be 150 6-bit characters long even though the margin is 75. This is normally only of concern when passing BASIC files to other language programs.

```
10 MARGIN 17
20 PRINT RPT$("a",26),1.75,88
30 END


produces:


aaaaaaaaaaaaaaaaa
aaaaaaaaa
 1.75
 88
```

Figure 7-50. Program Example Using MARGIN Statement

## SETDIGITS STATEMENT

The SETDIGITS statement can be used to specify the number of significant digits to be output in subsequent PRINT statements when the default formatting is used. Numeric constants that are equal to or greater than seven digits are formatted and printed as exponential constants. This statement allows data printed up to 14 significant digits to be obtained. The value assigned by SETDIGITS is truncated to an integer in the range 1 to 14. Numbers are printed within the defined significance until the end of the program or another SETDIGITS statement is encountered. Figure 7-51 illustrates the format for the SETDIGITS statement. Note that any relevant sign or exponent is still printed even if a SETDIGITS value of 1 is in effect. A program example of the SETDIGITS statement is shown in figure 7-52.

SETDIGITS ne

| | |
|---|---|
| ne | Indicates numeric constant, variable, or expression. |

Figure 7-51. SETDIGITS Statement Format

# INTERNAL DATA TABLE I/O

I/O for internal data tables (blocks of data internal to a BASIC program) uses the DATA statement to create and the READ statement to access the tables.

```
100 LET A=55.45454545
110 PRINT "A=55.45454545 AND IS NORMALLY OUTPUT AS";A
120 PRINT "SETDIGITS","VALUE OUTPUT"
130 FOR N=1 TO 10
140 SETDIGITS N
150 PRINT N,A
160 NEXT N
170 END


produces:


A=55.45454545 AND IS NORMALLY OUTPUT AS 55.4545
SETDIGITS      VALUE OUTPUT
 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
```

Figure 7-52. SETDIGITS Statement Example

## DATA STATEMENT

The DATA statement constructs an internal data table containing the values appearing in the DATA statement line; this data can then be accessed by the READ statement. Figure 7-53 shows the format for the DATA statement.

DATA $c_1, c_2, \ldots, c_n$

c Indicates numeric or string constant.

Figure 7-53. DATA Statement Format

The data values $c_1, c_2, \ldots, c_n$ are entered in the data table in the same order that they appear in the DATA statement line. The number of values per DATA statement line is restricted only by the length of the line. Any number of DATA statements can be used anywhere in the program to construct the data table; the BASIC compiler considers the statements to be contiguous statements and automatically places the data in sequential order in one internal data block before the program executes.

Both quoted and unquoted strings are allowed in a DATA statement line. Leading or trailing blanks in unquoted strings are ignored. An unquoted string can begin with plus, minus, digit, letter, or period, and can contain these characters as well as blanks. Unquoted strings cannot begin with a comma or a question mark. Other characters are allowed, but they should not be used because they might not be supported in future versions of BASIC. All characters in quoted strings are considered to be significant, including any leading or trailing blanks.

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

Figure 7-54 shows two examples of using the DATA statement to construct internal data tables. Example 2 also illustrates the diagnostic for not enough data. Both examples also demonstrate the READ statement.

```
Example 1:


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


produces:


 1             2             3


Example 2:


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


produces:


 END OF DATA  AT  110
BASIC EXECUTION ERROR
```

Figure 7-54. DATA Statement Examples

## READ STATEMENT

The READ statement (READ without a file ordinal) is used to read data values contained in the internal data table. The internal data table is a table containing data values that has been built into a program by using DATA statements. The format for the READ statement appears in figure 7-55.

READ $v_1, v_2, \ldots, v_n$

v Indicates variable identifier (numeric or string).

NOTE

An alternate form of the READ statement is provided for reading binary files created by the WRITE statement. See Binary I/O Statements in this section.

Figure 7-55. READ Statement Format

When a READ statement is executed, data values contained in the data table are placed sequentially into the variables $v_1, v_2, \ldots, v_n$. The read position pointer is advanced one data item for each value read.

The variables in a READ list must correspond in type to data items being read from a data table. For example, numeric variables must correspond to numeric data; otherwise, program execution terminates, displaying the diagnostic BAD DATA IN READ. If the ON ERROR statement is used to trap this situation, the diagnostic BAD DATA IN READ is not returned, and a subsequent READ statement accesses the next data item (the one following the bad data). Note that unquoted strings that look like numbers can be read either as strings or numbers.

If a READ statement attempts to read more data than is available, the diagnostic END OF DATA is given, and program execution terminates. Check for end-of-data by using the NODATA statement (described in this section under File Control Statements). The IF END or IF MORE statements cannot be used to check for end-of-data in an internal data table; these two statements apply only to files. After issuing a READ, use RESTORE to move the data pointer to the beginning of the data table.

In figure 7-56, the DATA statements at lines 10 and 20 establish values for the data table. The READ statement at line 30 reads the first two data values (10 and 15). The READ statements at lines 40 and 50 read the remaining data values. The substring reference F$(1:4) in the READ statement at line 50 indicates that the complete data value is to be read into a substring of F$. The character string THREE replaces characters 1 through 4 of F$. The value of F$ after execution of this READ statement is THREE5678; the PRINT statement at line 80 outputs this value.

```
10 DATA 10,15,17
20 DATA "ONE","TWO","THREE","FOUR"
25 LET F$="12345678"
30 READ A,B
40 READ C,D$
50 READ E$,F$(1:4),G$
60 PRINT A,B
70 PRINT C,D$
80 PRINT E$,F$,G$
90 END

produces:

 10            15
 17           ONE
TWO           THREE5678      FOUR
```

Figure 7-56. READ Statement Example

Matrices are arrays containing a collection of data. Matrices are widely used in all applications of programming. BASIC provides a series of functions and statements designed to simplify the use of matrices so that even the relatively inexperienced programmer can work easily with them.

The first part of this section provides an overview of what matrices are and how storage is allocated for them. The remainder of this section describes the matrix arithmetic operations, matrix functions, and the matrix I/O statements.

## MATRIX DEFINITION AND DECLARATION

A matrix is a collection of data, numeric or string, that is structured to enable referencing of specific elements of the matrix, and manipulating of data within the matrix as a unit.

A one-dimensional array consists of elements of data in a linear arrangement. A one-dimensional array is treated as a column vector (n rows by one column array) in BASIC. To obtain a row vector, declare a 2-dimensional array with one row and n columns.

A two-dimensional m by n array is a rectangular array containing m rows and n columns of elements of data. A specific element of the array can be referenced (for example, A(2,3)), or the entire array as a unit in a MAT statement can be referenced (for example, MAT A=B). If an array that has not been previously defined is referenced as a unit, it is assumed to be a 2-dimensional array.

A three-dimensional m by n by p array is an arrangement of p number of m by n matrices side by side. A specific array element, can be referenced, for example, A(1,3,5); but a three-dimensional array as a whole cannot be referenced. Also, none of the matrix statements or matrix functions can manipulate three-dimensional arrays.

The matrix statements and functions are restricted to the use of 1- and 2-dimensional arrays. Operations that use 3-dimensional arrays must be provided by user-written routines.

The number of array elements specified and used within a BASIC program is limited only by the amount of available memory, except for arrays used by the INV function; this function can work with arrays no larger than 100 by 100 elements. In all cases, numeric array elements that receive no values are assigned zero values by default, and string arrays are assigned null strings by default. It should be noted, however, that it is poor programming practice to assume that variables have a value before the program explicitly assigns values, particularly since, on many systems, this practice is not allowed. See the Future System Migration Guidelines, appendix E.

## ARRAY BOUNDARIES

The lower boundary (or origin) of an array is normally 0, but can be changed to 1 by using the OPTION statement (described in section 3). This means that array subscripting can start either with 0 (the default) or 1. For example, DIM A(2,4) defines a two-dimensional array having 3 rows and 5 columns, or 15 elements, when the base is 0. The elements of such an array are presented in figure 8-1.

When the OPTION statement is used to change the lower boundary of arrays to 1, DIM A(2,4) defines a two-dimensional array having 2 rows and 4 columns, or a total of 8 elements. The elements of such an array are shown in figure 8-2.

Therefore, when working with matrices, it is important to remember that array subscripting starts with element 0, unless the OPTION statement is used to change the lower boundary to 1; in which case, array subscripting starts with element 1.

| | column 0 | column 1 | column 2 | column 3 | column 4 |
|---|---|---|---|---|---|
| row 0 | (0,0) | (0,1) | (0,2) | (0,3) | (0,4) |
| row 1 | (1,0) | (1,1) | (1,2) | (1,3) | (1,4) |
| row 2 | (2,0) | (2,1) | (2,2) | (2,3) | (2,4) |

Figure 8-1. Array A(2,4) With OPTION BASE 0

| | column 1 | column 2 | column 3 | column 4 |
|---|---|---|---|---|
| row 1 | (1,1) | (1,2) | (1,3) | (1,4) |
| row 2 | (2,1) | (2,2) | (2,3) | (2,4) |

Figure 8-2. Array (2,4) With OPTION BASE 1

## ARRAY DECLARATION

An array declaration states that an array with a specified name and dimension be allocated to a BASIC program. Arrays can be defined explicitly in a DIM statement (described in section 3) or implicitly through usage. Arrays require a DIM statement when a subscript value greater than 10 is needed. Using the DIM statement to dimension an array whose upper subscript limit is less than 10 is good programming practice and saves space. An array not previously defined by the DIM statement is implicitly declared to have one dimension (10) when an element is referenced by an array variable with one subscript; two dimensions (10,10) when an element is reference by an array variable with two subscripts; and three dimensions (10,10,10) when an element is referenced by an array variable with three subscripts.

The following example illustrates use of the DIM statement.

```
DIM X1(50),X2(12,12),X$(6,6)
```

This statement reserves space for:

X1 A one-dimensional numeric array with 51 elements

X2 A two-dimensional numeric array with 13 by 13, or 169, elements

X$ A two-dimensional string array with 7 by 7, or 49, elements

Refer to section 3 for a description of the DIM statement and additional examples of usage.

## REDIMENSIONING

Matrices can be redimensioned to change the number of elements in an array as long as the number of elements is not greater than the total number of elements originally allocated. For example, a 3 by 5 array can be redimensioned to 3 by 3 or 5 by 3, but not to 3 by 6 or 8 by 2. However, the number of dimensions cannot be changed. For example, a vector cannot be redimensioned to be a 2-dimensional matrix, or a 2-dimensional matrix cannot be redimensioned to a vector.

Figure 8-3 shows the formats for the redimensioning specifiers. Format 1 is used to redimension a vector. Format 2 is used to redimension a 2-dimensional array. In both cases, the numeric expressions are rounded to integer values.

1. (ne)
2. ($ne_1$,$ne_2$)

ne Indicates numeric constant, variable, or expression.

Figure 8-3. Formats for Redimensioning Specifiers

Matrices can be redimensioned either implicitly or explicitly. To implicitly redimension a matrix, assign a new value by using MAT assignment or MAT arithmetic operations (described later in this section). To explicitly redimension a matrix, append a redimension specifier to matrix names in MAT READ or MAT INPUT statements or include redimension specifiers in MAT functions (described later in this section). Figure 8-4 is an example of redimensioning by using MAT READ. The DIM statement initially dimensions array A as 3 by 2. When the program is executed, MAT READ at line number 130 redimensions array A to 2 by 3. The MAT READ statement at line number 150 redimensions the matrix to 1 by 5.

```
100 OPTION BASE 1
110 DIM A(3,2)
120 DATA 1,2,3,4,5,6,1,2,3,4,5,6
130 MAT READ A(2,3)
140 MAT PRINT A;
150 MAT READ A(1,5)
160 MAT PRINT A;
170 END

produces:

 1  2  3
 4  5  6

 1  2  3  4  5
```

Figure 8-4. Redimensioning Example Using MAT READ

# MATRIX ARITHMETIC

Although it is possible to construct programs to perform matrix arithmetic operations with the ordinary statements of the language, BASIC also provides a set of statements explicitly for matrix operations. Table 8-1 identifies these statements and their effects. The matrix arithmetic statements are described in the following text in the same order as they are presented in table 8-1.

## MATRIX ASSIGNMENT

Matrix assignment is performed by replacing the elements of the matrix on the left side of the equals sign, with the corresponding elements of the matrix on the right side of the equals sign. Figure 8-5 shows the format for the matrix assignment statement.

MAT m=m

m Indicates numeric matrix identifiers.

Figure 8-5. Matrix Assignment Statement Format

TABLE 8-1. MATRIX ARITHMETIC STATEMENTS

| Statement | Example | Effect |
|---|---|---|
| Matrix assignment | MAT M1 = M2 | Assigns the elements of M2 to M1. |
| Matrix addition | MAT M1 = M2 + M3 | Replaces the elements of M1 with the sum of the elements of M2 and M3. |
| Matrix subtraction | MAT M1 = M2 - M3 | Replaces the elements of M1 with the differences of the corresponding elements of M2 and M3. |
| Matrix multiplication | MAT M1 = M2 * M3 | Replaces each element of M1 according to the usual rules of matrix algebra. |
| Matrix scalar multiplication | MAT M1 = (5) * M2 | Multiplies each element of M2 by 5 and places the product into M1. |

To perform matrix assignment, m on the left side of the equals sign must contain the same number of dimensions as m on the right of the equals sign, and the total number of elements of m on the left side of the equals sign must be equal to or greater than the total number of elements of m on the right of the equals sign. If this is not the case, program execution is terminated and the diagnostic MATRIX DIMENSION ERROR is displayed.

Figure 8-6 illustrates matrix assignment. The matrix READ statement reads fours into all elements of B, the matrix assignment statement at line 40 assigns matrix B to matrix A, and the statement at line 50 prints the values of matrix A.

```
100 OPTION BASE 1
110 DIM A(2,2),B(2,2)
120 DATA 4,4,4,4
130 MAT READ B
140 MAT A=B
150 MAT PRINT A;
160 END

produces:

 4  4

 4  4
```

Figure 8-6. Matrix Assignment Example

## MATRIX ADDITION

Matrix addition is performed by replacing the elements of the matrix on the left side of the equals sign with the sum of the corresponding elements of the two matrices on the right side of the equals sign. Matrix addition is only valid for numeric matrices. Figure 8-7 shows the format for matrix addition.

MAT $m_1=m_2+m_3$

m Indicates numeric array identifier.

Figure 8-7. Matrix Addition Format

When performing matrix addition, corresponding array dimensions on the right side of the equals sign must be identical. Matrix $m_1$ must have the same number of dimensions as $m_2$ and $m_3$, and the total number of elements in $m_1$ must be greater than or equal to the total number of elements in $m_2$ or $m_3$. If the matrices are not conformable, program execution is terminated and a diagnostic is issued. Also note that use of an operand as a result of a matrix addition operation is legal; for example, MAT $m_1=m_2+m_1$ is permitted. In figure 8-8, lines 30 through 60 produce results identical to the results of line 20.

```
05 OPTION BASE 1
10 DIM A(10,10),B(10,10),C(10,10)
20 MAT A=B+C
30 FOR I=1 TO 10
40 FOR J=1 TO 10
50 LET A(I,J)=B(I,J)+C(I,J)
60 NEXT J
70 NEXT I
99 END
```

Figure 8-8. Matrix Addition Example

## MATRIX SUBTRACTION

Matrix subtraction is performed by replacing the elements of the matrix on the left side of the equals sign with the difference of the corresponding elements of the two matrices on the right side of the equals sign. Figure 8-9 shows the format for matrix subtraction.

```
MAT m1=m2-m3
```

m Indicates numeric array identifier.

Figure 8-9. Matrix Subtraction Format

When performing matrix subtraction, matrices $m_2$ and $m_3$ must have identical dimensions; $m_1$ must have the same number of dimensions as $m_2$ and $m_3$; and the total number of elements in $m_1$ must be greater than or equal to the number of elements in $m_2$ or $m_3$. If these requirements are not true, program execution terminates and a diagnostic is displayed. Use of an operand as a result of a matrix subtraction is legal; for example, MAT $m_1=m_1-m_2$ is permitted.

Figure 8-10 illustrates matrix subtraction. The instruction at line 50 causes each element of matrix C to be replaced by the difference of the corresponding elements of matrices B and A. Line 60 prints the values for all three matrices (A, B and C).

```
 10 OPTION BASE 1
 20 DIM A(1,3),B(1,3),C(1,3)
 30 DATA 1,1,1,5,5,5
 40 MAT READ A,B
 50 MAT C=B-A
 60 MAT PRINT B,A,C
 70 END

 produces:

  5            5            5

  1            1            1

  4            4            4
```

Figure 8-10. Matrix Subtraction Example

## MATRIX MULTIPLICATION

Matrix multiplication is performed according to the usual rules of matrix algebra. Figure 8-11 shows the format for matrix multiplication.

When performing matrix multiplication, the multiplier and multiplicand matrices, $m_2$ and $m_3$, must have conformable dimensions in the normal mathematical sense; otherwise, program execution is terminated and the diagnostic MATRIX DIMENSION ERROR is displayed. That is, the column dimension of $m_2$ must be the same as the row dimension of $m_3$. The result matrix, $m_1$, can be the same matrix as $m_2$ or $m_3$.

```
MAT m1=m2*m3
```

m Indicates numeric array identifier.

Figure 8-11. Matrix Multiplication Format

Figure 8-12 illustrates matrix multiplication. In the example, line 40 computes the products of matrices B and C and places the result into matrix A. Line 50 prints matrices A, B, and C. The equation used to compute the element A(1,1) is as follows:

```
B(1,1) * C(1,1) + B(1,2) * C(2,1) = 27
(1 * 7) + (2 * 10) = 27
```

```
 05 OPTION BASE 1
 10 DIM A(3,3),B(3,2),C(2,3)
 20 DATA 1,2,3,4,5,6,7,8,9,10,11,12
 30 MAT READ B,C
 40 MAT A=B*C
 50 MAT PRINT B,C,A
 60 END

 produces:

  1            2

  3            4

  5            6

  7            8            9

  10           11           12

  27           30           33

  61           68           75

  95           106          117
```

Figure 8-12. Matrix Multiplication Example

## MATRIX SCALAR MULTIPLICATION

Matrix scalar multiplication is performed by replacing the elements of the matrix on the left side of the equals sign with the corresponding elements of the matrix on the right side of the equals sign multiplied by a given value. Figure 8-13 shows the format.

```
MAT m1=(ne)*m2
```

m Indicates numeric array identifier.

ne Indicates numeric constant, variable or expression.

Figure 8-13. Scalar Multiplication Format

Figure 8-14 illustrates scalar multiplication. Line 50 multiplies each element of matrix A by 5 and places the result back into matrix A.

```
10 DIM A(1,1)
20 DATA 4,4,4,4
30 MAT READ A
40 MAT PRINT A
50 MAT A=(5)*A
60 MAT PRINT A
70 END

produces:

 4            4
 4            4

 20           20
 20           20
```

Figure 8-14. Scalar Multiplication Example

## MATRIX FUNCTIONS

The matrix functions CON, IDN, INV, TRN, ZER, and DET are used to manipulate matrices in specific ways. All of the matrix functions, except DET, must be used in matrix statements. DET (a numeric function) can be used anywhere that a mathematical function can be used. Table 8-2 summarizes the matrix functions.

## MATRIX CON FUNCTION

This function causes all elements of the specified matrix to be initialized to one. The formats for the matrix CON function appear in figure 8-15.

1. MAT m=CON
2. MAT m=CON(ne)
3. MAT m=CON($ne_1$,$ne_2$)

m — Indicates numeric matrix identifier.

ne — Indicates a numeric constant, variable, or expression that evaluates to specific dimensions.

Figure 8-15. Matrix CON Function Format

Format 1 of the matrix CON function returns a matrix of ones with the same dimension as m; format 2 returns a (ne by 1) vector; and format 3 returns a ($ne_1$ by $ne_2$) matrix of ones.

Formats 2 and 3 of the matrix CON statement can be used to redimension the specified matrix m. However, if redimensioning exceeds the number of matrix elements or number of dimensions declared implicitly in m or by the DIM statement, program execution is terminated and the diagnostic MATRIX DIMENSION ERROR is displayed.

TABLE 8-2. MATRIX FUNCTIONS

| Function | Effect | Usage |
|---|---|---|
| Matrix CON function | Returns a matrix of ones. | MAT A = CON |
| Matrix IDN function | Returns an identity matrix (a square matrix with ones along the principle diagonal and zeros throughout the remaining matrix). | MAT A = IDN |
| Matrix ZER function | Returns a matrix of zeros. | MAT C = ZER |
| Matrix INV function | Returns the inverse matrix of of the argument matrix (to the right of the equals sign). | MAT A = INV(B) |
| Matrix TRN function | Returns the transpose of the argument matrix (to the right of the equals sign). | MAT X = TRN(Y) |
| Matrix DET function | Returns the determinant of the argument matrix (to the right of the equals sign) or the last inverted matrix. | LET A = DET(B) |

Figure 8-16 shows an example of the matrix CON function. Line 10 declares storage for a 3 by 3 matrix A whose subscript origin is 0. Line 20 declares that the data table contains nine data values. These data values are read into matrix A by the MAT READ statement at line 30. Line 50, the matrix CON function, replaces the current matrix values with all ones. Both matrix values are shown in the printed output.

```
 10 DIM A(2,2)
 20 DATA 1,2,3,4,5,6,7,8,9
 30 MAT READ A
 40 MAT PRINT A
 50 MAT A=CON
 60 MAT PRINT A
 70 END

 produces:

 1             2             3
 4             5             6
 7             8             9

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

Figure 8-16. Matrix CON Function Example

## MATRIX IDN FUNCTION

This function causes the specified matrix to be initialized to an identity matrix (a matrix consisting of ones along the main diagonal and zeros elsewhere). The format for the matrix IDN statement appears in figure 8-17.

1. MAT m=IDN
2. MAT m=IDN(ne)
3. MAT m=IDN($ne_1$,$ne_2$)

| | |
|---|---|
| m | Indicates numeric matrix identifier. |
| ne | Indicates a numeric constant, variable or expression that evaluates to desired dimensions. |

Figure 8-17. Matrix IDN Function Format

Format 1 returns an identity matrix with the same dimensions as m. Format 2 returns a vector; but, since the identity matrix must be square, format 2 is valid only if ne=1. Format 3 retains an $ne_1$ by $ne_2$ identity matrix; $ne_1$ must equal $ne_2$.

Formats 2 and 3 of the IDN function can be used to redimension the identity matrix m. However, if redimensioning exceeds the number of array elements and dimensions (originally allocated for m either implicitly or in the DIM statement), program execution terminates and the diagnostic MATRIX DIMENSION ERROR is displayed. Also, the IDN function cannot be used to initialize an array that is not square (one with a different number of rows and columns). An attempt to initialize a non-square array with the IDN function terminates program execution and the diagnostic MATRIX DIMENSION ERROR is displayed.

Figure 8-18 shows an example of the matrix IDN function. Line 10 declares storage for a 4 by 3 matrix A. Line 20 declares that the data table is to contain 12 values. These values are read into matrix A by the MAT READ statement at line 30. Line 50, the matrix IDN function, redimension matrix A to a 3 by 3 matrix and causes it to assume the value of an identity matrix. Both the original and identity matrix values are shown in the printed output.

```
 10 DIM A(3,2)
 20 DATA 2,4,6,8,10,12,14,16,18,20,22,24
 30 MAT READ A
 40 MAT PRINT A
 50 MAT A=IDN(2,2)
 60 MAT PRINT A
 70 END

 produces:

 2             4             6
 8             10            12
 14            16            18
 20            22            24

 1             0             0
 0             1             0
 0             0             1
```

Figure 8-18. Matrix IDN Function Example

## MATRIX ZER FUNCTION

This function causes the specified matrix to be initialized to zeros. The formats for the matrix ZER function are presented in figure 8-19.

1. MAT m=ZER
2. MAT m=ZER(ne)
3. MAT m=ZER($ne_1$,$ne_2$)

| | |
|---|---|
| m | Indicates numeric matrix identifier. |
| ne | Indicates a numeric constant, variable or expression that evaluates to desired dimensions. |

Figure 8-19. Matrix ZER Function Format

Format 1 of the ZER function returns a matrix of zeros that have the same dimensions as m. Format 2 returns a ne by 1 column vector of zeros. Format 3 returns a $ne_1$ by $ne_2$ matrix of zeros.

Formats 2 and 3 of the ZER function can redimension the matrix m; however, if redimensioning exceeds the number of array elements or dimensions originally declared for m either implicitly or in the DIM statement, program execution terminates and the diagnostic MATRIX DIMENSION ERROR is displayed.

In the example shown in figure 8-20, the MAT ZER function redimensions matrix A from a 2 by 3 matrix to a 2 by 2 matrix and causes all elements of the matrix to assume the value of zero. Both conditions of matrix A are displayed in the printout. Note that the subscript origin is 0.

```
10 DIM A(1,2)
20 DATA 1,2,3,4,5,6
30 MAT READ A
40 MAT PRINT A
50 MAT A=ZER(1,1)
60 MAT PRINT A
70 END

produces:

 1            2            3
 4            5            6

 0            0
 0            0
```

Figure 8-20. Matrix ZER Function Example

## MATRIX INV FUNCTION

This function calculates the inverse of a matrix. Figure 8-21 illustrates its format.

```
MAT m=INV(m)

m     Indicates numeric matrix identifier.
```

Figure 8-21. Matrix INV Function Format

Matrix m in figure 8-21 is automatically redimensioned to receive the result, if required; however, it must have been allocated originally with at least as many elements and dimensions as matrix m of the INV function. Otherwise, program execution terminates and the diagnostic MATRIX DIMENSION ERROR is displayed. Also, matrix m must be a square matrix and cannot have more than 100 by 100 elements.

If the matrix m is singular or near singular, INV returns meaningless results, and no diagnostic is issued. The DET function (described later) must be used to ensure that the results of INV are valid; the results are valid only when the value of DET is not zero.

Figure 8-22 shows an example of the matrix INV function. Line 10 declares storage for matrices A and B whose subscript origins are 0. Line 20 declares the data table of four values. Line 30 reads the data table values into matrix A. Line 50, the matrix INV function, causes matrix B to be replaced by the inverse of matrix A. The printed output displays matrix A, the inverse of A, and the inverse times the original matrix.

```
10 DIM A(1,1),B(1,1)
20 DATA 5,6,7,8
30 MAT READ A
40 MAT PRINT A
50 MAT B=INV(A)
60 IF DET=0 THEN 90  'SKIP TO END IF MAT A IS SINGULAR
70 MAT PRINT B
80 MAT A=B*A
90 MAT PRINT A
99 END

produces:

 5            6
 7            8

-4            3
 3.5         -2.5

 1            0
 0            1
```

Figure 8-22. Matrix INV Function Example

## MATRIX TRN FUNCTION

This function causes matrix m to be transposed. The matrix TRN function format is presented in figure 8-23.

```
MAT m=TRN(m)

m     Indicates numeric matrix identifier.
```

Figure 8-23. Matrix TRN Function Format

Matrix m is redimensioned to receive the result, if necessary; however, it must have been originally allocated with at least as many dimensions as matrix m (to the right of the equals sign). Otherwise, program execution terminates and the diagnostic MATRIX DIMENSION ERROR is displayed. If desired, m, on both sides of the equals sign, can be the same matrix.

Figure 8-24 shows an example of the matrix TRN function. Line 10 declares storage for matrices A and B (both are 2-dimensional matrices). Line 20 declares that the data table contains six values; line 30 reads the values into matrix A. Line 50, the matrix TRN function, causes each element of matrix B to be replaced by the transposed form of matrix A. Note that matrix B is automatically redimensioned.

```
10 DIM A(1,2),B(3,3)
20 DATA 1,2,3,4,5,6
30 MAT READ A
40 MAT PRINT A
50 MAT B=TRN(A)
60 MAT PRINT B
70 END

produces:

 1            2            3
 4            5            6

 1            4
 2            5
 3            6
```

Figure 8-24. Matrix TRN Function Example

## MATRIX DET FUNCTION

The DET function returns the determinant of a matrix. DET can determine if the most recently inverted matrix was singular. (See INV function.) The format for the DET function appears in figure 8-25.

```
1. DET

2. DET(m)

     m     Indicates numeric matrix identifier.
```

Figure 8-25. Matrix DET Function Format

When format 1 of the DET function is used, DET returns the determinant to the matrix most recently inverted by the INV function. References to DET when INV has not been invoked causes the program to terminate, and the diagnostic DET USED BEFORE INV is displayed. Repeated references to DET, without corresponding references to INV, yield the same result each time. When format 2 of DET is used, the function returns the determinant of the matrix. An example of the two formats is shown in figure 8-26.

```
10 OPTION BASE 1
20 DIM A(2,2),B(2,2)
30 DATA 1,2,3,4,10,12,14,16
40 MAT READ A,B
50 MAT A=INV(A)
60 MAT PRINT A,B
70 PRINT "DETERMINANT OF A IS ";DET
80 PRINT "DETERMINANT OF B IS ";DET(B)
90 END

produces:

-2            1

 1.5         -.5

 10           12

 14           16

DETERMINANT OF A IS -2
DETERMINANT OF B IS -8
```

Figure 8-26. Matrix DET Function Example

# MATRIX I/O

Matrix I/O statements are used to input or output complete matrices. In general, the action of the matrix output statements can be simulated by using a group of output statements, each group of statements relating to only one row of the matrix. Matrix input statements can be simulated by using one regular INPUT that lists all of the elements of the matrix in the statement input list. Table 8-3 identifies the matrix I/O statements.

TABLE 8-3. MATRIX I/O STATEMENTS

| Statement | Effect | Usage |
|---|---|---|
| MAT WRITE | Writes matrix values to a file. | MAT WRITE #2,A,B$,C |
| MAT READ | Fills matrices with data from an internal data block or binary file. | MAT READ #2,A$,B,C |
| MAT INPUT | Fills matrices with data input from the terminal or from a file. | MAT INPUT D$,E,F, |
| MAT PRINT | Outputs matrix elements to the terminal or a file. | MAT PRINT A,B$,C |
| MAT PRINT USING | Outputs matrix elements to the terminal or a file per the format specified by the image statement. | MAT PRINT USING 100,B |

## MAT WRITE STATEMENT

This statement is similar to the WRITE statement (described in section 7). It causes data from specified matrices to be written into a binary file without individually referencing each member. Elements are written in row order (A(0,0), A(0,1), A(0,2) and so forth). Files created by the MAT WRITE statement are in binary form and can only be read by the READ or MAT READ statement. Figure 8-27 shows the format for the MAT WRITE statement.

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

$m_1$,$m_2$, . . . Indicates numeric or string matrix identifiers.

ne Indicates a file ordinal expressed as a numeric constant, variable, or expression.

Figure 8-27. MAT WRITE Statement Format

When the MAT WRITE statement is used to write strings on a file, one word of binary zeros follows the string data to indicate the end of the string. Incomplete words are padded with binary zeros.

Figure 8-28 illustrates the MAT WRITE statement. Line 10 assigns the file ITEMFIL to ordinal 1, and line 20 declares space for a 2 by 4 matrix A whose subscript origin is 0. The MAT WRITE statement at line 50 causes matrix A to be written on the file ITEMFIL, and line 60 closes the file ITEMFIL. Refer to figure 8-30 in the MAT READ statement description (next topic) for a further illustration of this example.

```
10 FILE #1="ITEMFIL"
20 DIM A(1,3)
30 DATA 1,1.1,1.2,1.3,1.4,1.5,1.6,1.7
40 MAT READ A
50 MAT WRITE #1,A
60 CLOSE #1
70 END
```

Figure 8-28. MAT WRITE Statement Example

## MAT READ STATEMENT

This statement is similar to the READ statement (described in section 7). It allows data to be read into the specified matrices without individually referencing each member. Elements are read by row from an internal data table created by the DATA statement, or from a binary file created by the WRITE or MAT WRITE statement. Figure 8-29 shows the format for the MAT READ statement.

MAT READ $m_1$,$m_2$, . . .

MAT READ #ne,$m_1$,$m_2$, . . .

$m_1$,$m_2$, . . . Indicates numeric or string matrix identifiers. Subscripts denoting new dimensions can follow the matrix identifier.

ne Indicates numeric constant, variable, or expression.

Figure 8-29. MAT READ Statement Format

If an executing MAT READ statement causes the internal data table (format 1) or binary file (format 2) to be exhausted before the specified matrix is filled, program execution is terminated and the diagnostic END OF DATA or END OF DATA ON FILE is displayed.

Any matrix in the MAT READ list can be redimensioned by specifying a redimension indicator with the array name. (See Redimensioning in this section. For example, MAT READ B(3,4) causes values to be read into matrix B and matrix B to be redimensioned to a 3 by 4 matrix (assuming array base is one). Redimensioning cannot require a greater number of total elements or dimensions than originally allocated implicitly or in the DIM statement.

Figure 8-30, an example of the MAT READ statement, is an extension of the example shown in figure 8-28 for the MAT WRITE statement. Line 10 assigns the file name ITEMFIL to ordinal 1. Line 15 restores the file pointer to the beginning of the file, and line 20 declares a 2 by 4 matrix A and a 2 by 3 matrix B. Note that the subscript origins are 0. The MAT READ statement at line 30 reads into matrix A the values originally written to the file in the MAT WRITE example. Line 40 prints matrix A. Line 50 restores the file pointer to the beginning of the file. The MAT READ statement at line 60 reads into matrix B six of the eight values originally written to the file in the MAT WRITE example. Line 70 prints matrix B, illustrating which values were read into B. Line 80 closes the file ITEMFIL.

```
 10 FILE #1="ITEMFIL"
 15 RESTORE #1
 20 DIM A(1,3),B(1,2)
 30 MAT READ #1,A
 40 MAT PRINT A
 50 RESTORE #1
 60 MAT READ #1,B
 70 MAT PRINT B
 80 CLOSE #1
 90 END

 produces:

 1             1.1           1.2           1.3

 1.4           1.5           1.6           1.7

 1             1.1           1.2

 1.3           1.4           1.5
```

Figure 8-30. MAT READ Statement Example

## MAT INPUT STATEMENT

The MAT INPUT statement is similar to the INPUT statement (described in section 7). It permits matrices to be read in row order from a display format file or matrices to be entered from the terminal during program execution. Figure 8-31 shows the format for the MAT INPUT statement.

MAT INPUT $m_1,m_2, \ldots$

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

| | |
|---|---|
| $m_1,m_2, \ldots$ | Indicates numeric or string matrix identifiers. Subscripts denoting new dimensions can follow the matrix identifier. |
| ne | Indicates numeric constant, variable, or expression. |

Figure 8-31. MAT INPUT Statement Format

When format 1 of MAT INPUT is used and the program is running interactively, a question mark is displayed at the terminal prompting that the entire matrix be entered in row order (A(0,0, A(0,1), A(0,2), and so forth). Each matrix element must be separated by a delimiter (see DELIMIT statement in section 7). If all the matrix elements do not fit on one input line or the entire matrix is not entered in one input response, the matrix elements can be entered one row at a time on separate input lines by terminating each partial input reply with a non-blank delimiter. If more data values are needed to fill the matrix than are supplied and the reply is not terminated by using a delimiter, the system prints NOT ENOUGH DATA, REENTER OR TYPE IN MORE, followed by another question mark. In these instances, either reenter the complete matrix or extend the previous reply by entering a non-blank delimiter followed by the additional matrix elements. Figure 8-32 illustrates two ways of filling a 2 by 2 matrix of four data elements.

Note that, particularly for string elements, the input line can be extended beyond the width entered on the terminal. The line feed LF key or ↓ on CDC 713 terminals allows line continuation; this key is depressed in place of the carriage return key. The total line length is limited to 150 characters. It is not a delimiter and if a delimiter is required between entries on the two lines, insert the proper delimiter.

If an attempt is made to overfill a matrix with too many data elements, the system requests (with the message TOO MUCH DATA, RETYPE INPUT) that the entire matrix be reentered.

If format 2 of the MAT INPUT statement is used, the matrices are read into the program from the display format file specified by ne. The matrix elements are read in row order (A(0,0), A(0,1), A(0,2), and so on). Data on the file must be display format and must have delimiters between values.

A matrix can be redimensioned before the items are input by specifying a redimension indicator. (See Redimensioning in this section.) For example, MAT INPUT A$(2,3) redimensions the matrix A$ to a 2 by 3 string matrix (assuming array base is one). Redimensioning cannot require more than the total number of array elements or dimensions than originally allocated to the array either implicitly or in the DIM statement.

```
100 OPTION BASE 1
110 DIM A(2,2)
120 MAT INPUT A
130 MAT PRINT A
140 END

produces:

? 2,2,2,2
 2             2

 2             2


produces the following when filled in two input responses:

? 2,2
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  120
? ,2,2
 2             2

 2             2
```

Figure 8-32. MAT INPUT Statement Example

## MAT PRINT STATEMENT

The MAT PRINT statement causes matrices to be printed in row order by using the same formatting rules as for the standard PRINT or PRINT #ne statements. The formats for the MAT PRINT statement appear in figure 8-33.

1. MAT PRINT $m_1$ d $m_2$ . . . d
2. MAT PRINT #ne, $m_1$ d $m_2$ . . . d

| | |
|---|---|
| $m_1, m_2, \ldots$ | Indicates numeric or string matrix identifier. |
| d | Indicates delimiter (comma or semicolon); the final delimiter is optional. |
| ne | Indicates numeric constant, variable, or expression. |

Figure 8-33. MAT PRINT Statement Formats

As shown in figure 8-33, the MAT PRINT statement formats each element of specified matrix m to the default specified output format (shown in PRINT formats, section 7), directs the output to the terminal for printing or to the file specified by ne, and positions the print mechanism according to the delimiter, d. The delimiter is a terminator character that specifies the print zoning control for each element of the matrix. See section 7, under Print Zoning. A blank line is automatically generated after each row. An additional blank line is printed to separate two arrays in the list.

Each element of the row is checked before printing to ensure that it fits on the print line. If it does not fit, the complete element is placed on the next line. If it cannot fit on an empty line, it is broken at the margin and written on as many lines as necessary. Matrices are printed in row order. The print results can be affected by the terminal, IAF, or INTERCOM. (See MARGIN statement, section 7.) See the MAT READ and MAT INPUT statements for an example of the MAT PRINT statement.

## MAT PRINT USING STATEMENT

This statement is similar to the PRINT USING statement (described in section 7). The MAT PRINT USING statement causes the matrices to be printed according to the image contained in the string expression se or the statement at line ln. (See figure 8-34.) Each row, as it is printed, starts at the beginning of the image and always prints on a new line. Repetition of the image, however, is possible within the row. The delimiter appearing after the matrix identifier determines the delimiter to be used between each element of the matrix, but not between rows. As in the PRINT USING statement (section 7), the delimiters do not control print zoning, but do control whether or not a print item repeating the image should start a new line (,) or continue on the same line (;).

A blank line is not printed after each row or after the last line of the array. A blank line is printed to separate two arrays in the list. If all the elements of a row do not fit on one line, a break is made after the first line is full even if this is in the middle of an element. Figure 8-35 shows three program examples of the MAT PRINT USING statement.

1. MAT PRINT USING ln, $m_1$ d $m_2$ d . . . d

   MAT PRINT USING ne, $m_1$ d $m_2$ d . . . d

2. MAT PRINT #ne, USING ln, $m_1$ d $m_2$ d . . . d

   MAT PRINT #ne USING ne, $m_1$ d $m_2$ d . . . d

| | |
|---|---|
| $m_1$,$m_2$, . . . | Indicates numeric or string matrix identifiers. |
| d | Indicates delimiter (comma or semicolon); the final delimiter is optional. |
| ne | Indicates numeric constant, variable, or expression. |
| ln | Indicates line number. |
| ne | Indicates string expression. |

Figure 8-34. MAT PRINT USING Statement Formats

Example 1:

```
090 OPTION BASE 1
100 FILE #1="DATA1"
110 DIM A(2,2),B(1,4)
120 DATA 1,2,3,4
130 MAT READ A
140 MAT PRINT USING "ROW ## ## ",A
150 MAT WRITE #1,A
160 RESTORE #1
170 MAT READ #1,B
180 MAT PRINT USING 190,B
190 : ##
200 END
```

produces:

```
ROW  1  2
ROW  3  4

 1
 2
 3
 4
```

Example 2:

```
090 OPTION BASE 1
100 DIM A(3,3)
110 FOR I=1 TO 3
120 FOR J=1 TO 3
130 A(I,J)=I*J
140 NEXT J
150 NEXT I
160 MAT PRINT USING " ## ",A;
170 MAT PRINT USING " ## ",A
180 END
```

Figure 8-35. MAT PRINT USING Statement Example (Sheet 1 of 2)

```
produces:


 1   2   3
 2   4   6
 3   6   9

 1
 2
 3
 2
 4
 6
 3
 6
 9


Example 3:


080 OPTION BASE 1
090 MARGIN 70
100 DIM A$(2,2)
120 MAT INPUT A$
125 MAT PRINT A$;
130 MARGIN 30
140 MAT PRINT A$;
150 MARGIN 70
160 MAT PRINT USING 170,A$;
170 : ##########################################


produces:


? "commutative rings and integral domains"," elementary properties of domains"
 NOT ENOUGH DATA, REENTER OR TYPE IN MORE  AT  120
? ,"well-ordered principle"," finite induction and laws of exponents"
COMMUTATIVE RINGS AND INTEGRAL DOMAINS
 ELEMENTARY PROPERTIES OF DOMAINS

WELL-ORDERED PRINCIPLE FINITE INDUCTION AND LAWS OF EXPONENTS


COMMUTATIVE RINGS AND INTEGRAL
 DOMAINS
 ELEMENTARY PROPERTIES OF DOMA
INS

WELL-ORDERED PRINCIPLE
 FINITE INDUCTION AND LAWS OF
EXPONENTS


 COMMUTATIVE RINGS AND INTEGRAL DOMAINS    ELEMENTARY PROPERTIES OF DO
MAINS
 WELL-ORDERED PRINCIPLE                    FINITE INDUCTION AND LAWS O
F EXPONENTS
```

Figure 8-35. MAT PRINT USING Statement Example (Sheet 2 of 2)

Often newly written programs do not operate correctly on the first attempt to execute them. They either stop with a run-time diagnostic (such as SUBSCRIPT ERROR AT 230) or run to completion but produce incorrect results. The process of isolating and removing errors or bugs in a program is referred to as debugging.

BASIC itself provides some tools to help debug a program (PRINT statements and tracing). In addition, a companion product, CYBER Interactive Debug (CID), provides a powerful interactive debug facility. The choice of particular tools to use is determined by personal preference, mode of operation (batch or interactive), and availability. CID is the most powerful and easy-to-use tool; CID is not available at all sites.

## BASIC DEBUG FEATURES

Three debugging aids or techniques that are available in BASIC are: inserting PRINT statements, conditional tracing of program flow, and unconditional tracing.

### INSERTING PRINT STATEMENTS

A common debugging technique is to temporarily insert PRINT statements in a program. Output from these PRINT statements can indicate program control flow (what order statements are executed) and values of selected program variables at specific points in the program. However, these PRINT statements must be removed once the program is debugged and making any changes to the program (even relatively minor changes, such as removing temporary PRINT statements) provides the opportunity for introducing new errors. Refer to section 7 for the format and examples of the PRINT statement.

### CONDITIONAL TRACE STATEMENT

BASIC provides a special REM statement to control tracing of program flow. This statement is a remark that has no effect on the program unless the DL (Debug Lines) option of the DB parameter is selected when the program is compiled. (See DB parameter in section 12.) When the DL option is selected, this statement starts and stops the program trace mechanism. This trace mechanism prints a sequential list of the BASIC statements executed in the order the statements were executed. This information can be used to determine the actual flow of control through the program. The formats of this conditional trace statement are shown in figure 9-1.

```
1. REM TRACE,ALL
2. REM TRACE,PART
3. REM TRACE,NONE
```

Figure 9-1. REM TRACE Statement Formats

The first format is REM TRACE,ALL. This format causes dynamic tracing (all executable statements are traced regardless of where they occur in the source code). The second format is REM TRACE,PART. This format causes static tracing (only those statements between REM TRACE,PART and the next REM TRACE,NONE are traced). The third format is REM TRACE,NONE. This format suppresses tracing; it is the default setting for the REM TRACE statement.

The following two points further explain the use of these statements:

- REM TRACE,ALL causes the message * AT nnn to be output for every statement executed until a REM TRACE,NONE statement is executed (nnn is the line number).

- REM TRACE,PART causes the message * AT nnn to be output only for those statements lying physically between the REM TRACE,PART statement and the next sequential REM TRACE,NONE statement. Execution of statements outside this physical range (for example, those executed with GOTO, GOSUB and function references) are not traced. The GOTO, GOSUB, and function reference statements do not cause tracing to be turned off; tracing resumes when control returns to statements within the physical range.

Trace output is written on the file specified by the K parameter on the BASIC control statement. If the K parameter is not specified on the BASIC control statement, trace output is written on the file OUTPUT.

Although the REM TRACE statement does not provide information about program variable values (an advantage of using the PRINT statement), the REM TRACE statement has the advantage of not needing to be removed from the program once the tracing is complete.

Figure 9-2 shows an example of REM TRACE,ALL. Execution of the statements in this example produces a sequence list of all the executable statements in the order they were executed. Figure 9-3 shows an example using all three REM TRACE statements. REM TRACE,PART does not trace the subroutine. REM TRACE,ALL does trace the subroutine.

```
150 REM TRACE,ALL
160 FOR X=1 TO 10
170 LET X=X+1
180 NEXT X
190 END


produces:


 * AT  160
 * AT  170
 * AT  180
 * AT  170
 * AT  180
 * AT  170
 * AT  180
 * AT  170
 * AT  180
 * AT  170
 * AT  190
```

Figure 9-2. REM TRACE,ALL Example

```
100 PRINT "REM TRACE,PART BEING EXECUTED"
110 REM TRACE,PART
120 GOSUB 200
130 PRINT "REM TRACE,NONE BEING EXECUTED"
140 REM TRACE,NONE
160 GOSUB 200
165 PRINT "REM TRACE,ALL BEING EXECUTED"
170 REM TRACE,ALL
190 GOSUB 200
195 STOP
200 PRINT "SUBROUTINE 200"
210 RETURN
240 END


produces:


REM TRACE,PART BEING EXECUTED
 * AT  120
SUBROUTINE 200
 * AT  130
REM TRACE,NONE BEING EXECUTED
SUBROUTINE 200
REM TRACE,ALL BEING EXECUTED
 * AT  190
 * AT  200
SUBROUTINE 200
 * AT  210
 * AT  195
```

Figure 9-3. REM TRACE Statement Example

## UNCONDITIONAL TRACE PARAMETER

In addition to the conditional trace statements that can be included within the program, BASIC provides a parameter on the BASIC control statement that can be used to force program flow tracing to be printed no matter what statements are included in the program.

If the TR (trace) option of the DB parameter is selected on the BASIC control statement when the program is compiled, the message * AT nnn is output for each executable line encountered during program execution. The nnn represents the line number of the executed statement.

This debug feature does not require program modifications, not even insertion of REM TRACE statements. However, only a full trace of the entire program can be obtained.

# CYBER INTERACTIVE DEBUG

The CYBER Interactive Debug (CID) facility is a companion to BASIC. CID permits external monitoring and controlling of the execution of the program from an interactive terminal without making any changes to the program. The CID commands and features are only available if CID is installed in the system. CID commands and features can be used when the BASIC program is compiled in debug mode. The use and features of CID are described below. For further information, see the CYBER Interactive Debug reference manual.

The CID facility allows the following to be done:

Suspend program execution when control reaches a predefined point (called a breakpoint).

Suspend program execution when a particular event, such as a store into a specific variable or program termination, occurs (called a trap).

Display and/or change the values of program variables while program execution is suspended.

Restart program execution at the point of interruption or at some other point in the program.

## ENTERING AND EXITING THE CID ENVIRONMENT

To execute a BASIC program under CID control, you must compile and execute the program in debug mode. Debug mode is turned on by using the system control statement DEBUG or DEBUG(ON). The DEBUG control statement must be entered before compiling and executing the program. When debug mode is on, the operating system and terminal can be used in the normal manner.

When compiling the BASIC program, you can specify the parameter DB in the BASIC control statement. See section 12 for the options that can be selected using the DB parameter. DB=0 must not be specified; if DB=0 is specified, the program will not be compiled for use with CID even if debug mode is turned on.

A CID session is terminated by using the CID command QUIT. The QUIT command returns control to the operating system; however, the debug environment remains in effect until the system control statement DEBUG(OFF) is issued. DEBUG(OFF) should be used if subsequent debugging is not needed.

## EXECUTING UNDER CID CONTROL

A debug session is a sequence of interactions between the programmer and CID that occurs while the object program is executing in debug mode. The session begins with the execution of the object program. Under the NOS/BE EDITOR or under the NOS BASIC subsystem, the session can be initiated with the RUN command. Under the NOS BATCH subsystem, the session can be initiated using the BASIC control statement X,BASIC,I=lfn; this control statement compiles and executes the BASIC program on file lfn.

When the debug session begins, control transfers to an entry point in CID and the following message is issued:

```
CYBER INTERACTIVE DEBUG
?
```

The question mark prompts for input of a CID command. In response, enter a CID command and press the transmission key (RETURN on most terminals). CID processes the command and generates any appropriate output, such as a message or another prompt.

# REFERENCING BASIC LINE NUMBERS AND VARIABLES

The following paragraphs explain the formats for referencing variables and line numbers in CID commands.

## VARIABLES

Program variables are referenced in CID commands in the same format as they are in BASIC statements. Simple and subscripted variables, full arrays, and substring addressing can be referenced. Variables referenced in CID commands must exist in the program; and execution must be suspended inside a function before formal parameters are known to CID.

See figure 9-4 for examples.

```
A
A1$
X$(2:4)
X(1)
```

Figure 9-4. Variables Examples

## LINE NUMBERS

Line numbers for CID commands are referenced with a special format not similar to BASIC. This format is shown in figure 9-5.

Example:

```
L.310
```

| | |
|---|---|
| L.n | |
| n | Indicates line number. |

Figure 9-5. Line Number Referencing Format

An exception to this format is the GOTO command, which references line numbers in the same format as it does in BASIC statements. (See the GOTO command described in this section.)

Example:

```
GOTO 310
```

# RESUMING PROGRAM EXECUTION

The CID commands GO, GOTO, and STEP can be used to resume the execution of a program. These commands are explained in the following paragraphs.

## GO COMMAND

The GO command resumes program execution from the point at which program execution was suspended. The format is shown in figure 9-6.

```
GO
```

Figure 9-6. GO Command Format

When the GO command is entered, the program resumes execution from the last point of suspension and executes until it reaches a breakpoint or trap. The GO command cannot be used after an END trap because execution is complete and cannot continue any further.

## GOTO COMMAND

The GOTO command resumes execution of the program at a specified line number. This command has the same format as the BASIC statement GOTO. The format is shown in figure 9-7.

| | |
|---|---|
| GOTO n | |
| n | Indicates line number. |

Figure 9-7. GOTO Command for CID Format

Example:

```
GOTO 50
```

The GOTO command causes program execution to continue at the specified statement line number 50. Execution proceeds until the program reaches a breakpoint or trap.

## STEP COMMAND

The STEP command executes a specified number of lines. Execution begins where it was previously interrupted. The format is shown in figure 9-8.

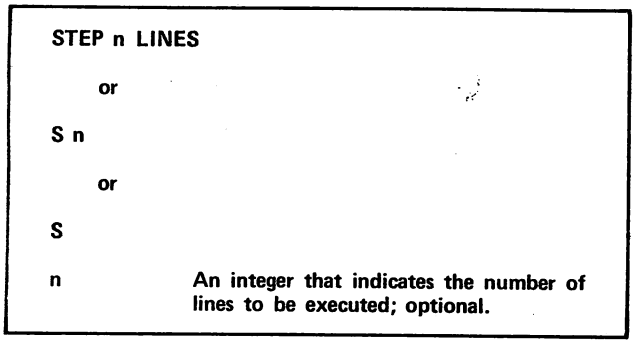


Figure 9-8. STEP Command Format

If the n parameter is not specified, the value used for the previous STEP command is used; if there is no previous STEP command, the value 1 is used.

A message is issued after the number of lines specified in the STEP command are executed. Figure 9-9 shows the format of the step message.

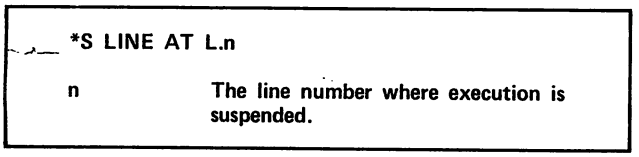


Figure 9-9. Step Message Format

If a breakpoint or trap is reached before the specified number of lines are executed, the breakpoint or trap overrides the STEP command and terminates the STEP operation.

Example:

```
STEP,7
```

# SETTING AND CLEARING BREAKPOINTS AND TRAPS

The following commands allow specific breakpoints and traps to be set or cleared in the BASIC program. A breakpoint is a point within a program at which CID takes control. When program execution reaches a breakpoint, execution is suspended, a message is issued, and CID requests input of commands by displaying a question mark. Any number of commands can be entered once CID gains control. A breakpoint is set by using the SET BREAKPOINT command and cleared by using the CLEAR BREAKPOINT command.

Traps are set to cause program suspension on the occurrence of a particular event. Traps are set by using the SET TRAP command and are cleared by using the CLEAR TRAP command.

## SET BREAKPOINT COMMAND

The SET BREAKPOINT command sets a breakpoint at a specific program line number. The format is shown in figure 9-10.

See figure 9-11 for examples.

A breakpoint remains set until it is explicitly cleared. Figure 9-12 shows the format of the breakpoint message displayed when the program reaches a breakpoint during execution.

Example:

```
*B #1, AT L.110
```

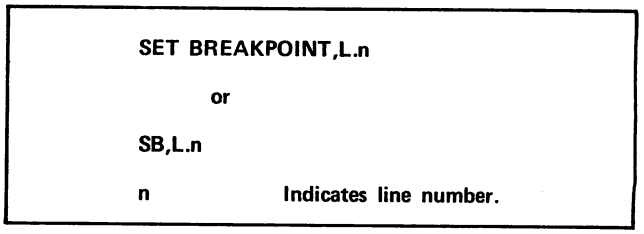


Figure 9-10. SET BREAKPOINT Command Format

```
SET BREAKPOINT,L.120
SB,L.150
```

Figure 9-11. SET BREAKPOINT Examples

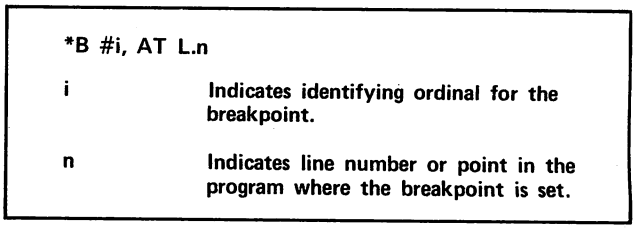


Figure 9-12. Breakpoint Message Format

## CLEAR BREAKPOINT COMMAND

The CLEAR BREAKPOINT command clears a breakpoint that exists at a specific line number. The format is shown in figure 9-13.

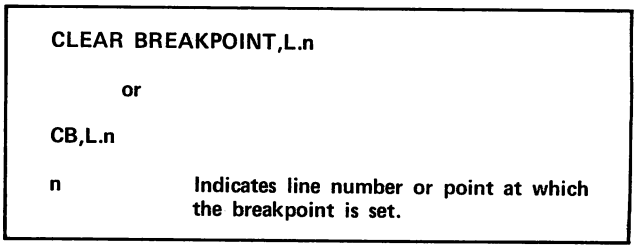


Figure 9-13. CLEAR BREAKPOINT Command Format

See figure 9-14 for examples.

The last example clears all of the breakpoints. Breakpoints can also be cleared by referring to the identification number, such as CB,#1. See the CYBER Interactive Debug reference manual for more details.

```
CLEAR BREAKPOINT,L.120
CB,L.150
CB,*
```

Figure 9-14. CLEAR BREAKPOINT Examples

## SET TRAP COMMAND

The SET TRAP command is used to set a trap of a specified type for a specified range of applicability. The format is shown in figure 9-15.

When a trap event occurs, program execution is suspended, a message is issued, and CID requests commands. Any number of commands can be entered once CID is in control. Figure 9-16 shows the format of the trap message.

There are many forms of the keyword type used in the trap commands. See the CYBER Interactive Debug reference manual for a complete list. Two important types are STORE and LINE.

STORE traps can catch stores into any variable or range of variables; however, BASIC string pointers are sometimes manipulated without affecting the string to which they point, so extraneous STORE traps can occur for string variables. The source statement should be inspected to verify that the named string variable is actually being referenced. The first example in figure 9-17 shows an example of the STORE trap.

LINE traps cause a trap prior to execution of each statement in the specified range. For instance, in the second example of figure 9-17, a trap occurs at any statement from line 100 to line 200 so this type of trap enables program flow to be traced in a specific area.

## CLEAR TRAP COMMAND

The CLEAR TRAP command clears specific traps. The format is shown in figure 9-18.

See figure 9-19 for examples.

Traps can also be cleared by referring to the identification number, such as CT,#1 or by clearing all traps at once, CT,*. See the CYBER Interactive Debug reference manual for more details.

SET TRAP,type,scope

or

ST,type,scope

| | |
|---|---|
| type | Indicates keyword describing the condition that causes the trap. |
| scope | Indicates range of applicability. |

Figure 9-15. SET TRAP Command Format

*T #i type AT n

| | |
|---|---|
| i | Indicates identification ordinal for this trap. |
| type | Describes briefly the condition that caused the trap. |
| n | Indicates line in the program where execution was suspended. If IN, rather than AT, is specified, then execution is suspended inside, not before the indicated line. |

Figure 9-16. Trap Message Format

| | |
|---|---|
| SET TRAP,STORE,A(4,7) | Trap occurs after any STORE in variable A(4,7). |
| ST,LINE,L.100...L.200 | Trap occurs before any line in the range of 100 to 200 is executed. |

Figure 9-17. SET TRAP Command Examples

CLEAR TRAP,type,scope

or

CT,type,scope

| | |
|---|---|
| type,scope | Type and scope must be the same as those used when setting the trap. |

Figure 9-18. CLEAR TRAP Command Format

```
CLEAR TRAP,LINE,L.100...L.200
CT,STORE,A(4,7)
```

Figure 9-19. CLEAR TRAP Examples

## DEFAULT TRAPS

Three traps are on by default and cannot be cleared: END, ABORT, and INTERRUPT. The END trap occurs whenever the BASIC program reaches normal completion, which occurs when a STOP, END, or CHAIN statement is executed. Program execution can be resumed at a specific line with the GOTO command; the GO command cannot be used because the program has completed and cannot continue from these traps.

The END trap for the CHAIN statement causes execution to end at the point just before the next chained-to program is executed. Compilation and execution of the chained-to program is not automatic; enter the QUIT command in order to terminate the present CID session, then enter any necessary control statements in order to compile and execute the chained-to program.

The ABORT trap occurs whenever the BASIC program terminates because of an error. If the program executes an ON ERROR statement before the trap occurs, a GO command causes execution to resume at the ON ERROR line. If the program does not execute an ON ERROR, execution cannot be resumed with a GO command; however, it can be resumed with the GOTO command. Note that the ABORT trap does not occur for interactive input errors and the normal BASIC recovery options still apply.

The INTERRUPT trap occurs whenever the BASIC program is interrupted from the terminal. If ON ATTENTION is in effect, that is, if the program executes an ON ATTENTION statement before the interrupt is trapped, GO causes the program to begin executing at the ON ATTENTION line. If ON ATTENTION is not in effect, GO causes execution to resume at the point where it was interrupted. The GOTO command can be used to cause execution to restart at a particular line.

# DISPLAYING PROGRAM VALUES

Three of the commands that allow program values in the BASIC program to be displayed are PRINT, MAT PRINT, and LIST,VALUES. The first two commands are similar to BASIC statements.

## PRINT COMMAND FOR CID

The PRINT command is similar to the BASIC PRINT statement. It prints values of program variables or computed expressions. The format is shown in figure 9-20.

See figure 9-21 for examples.

Variables used in the output list must exist in the program. Multiple semicolons must be used to separate the PRINT command from the next command on the same line (also true for the MAT PRINT command). Expressions cannot contain references to functions or to the exponentiation operator. CID does not allow partial print lines. The trailing comma or semicolon is ignored in CID. Images, PRINT USING statements, and file ordinals cannot be used in CID.

PRINT output-list

| | |
|---|---|
| output-list | List of any number of restricted arithmetic or string expressions; separated by commas or semicolons. |

Figure 9-20. PRINT Command for CID Format

```
PRINT "THE VALUE OF B=";B
PRINT A,A*A,A+135,7,B(17,J)
PRINT C$(1)(2:3)
```

Figure 9-21. PRINT Command for CID Examples

## MAT PRINT COMMAND FOR CID

The MAT PRINT command is similar to the BASIC MAT PRINT statement. It prints complete 1-, 2-, or 3-dimensional arrays. (The BASIC MAT PRINT statement prints only 1- or 2-dimensional arrays.) The format is shown in figure 9-22.

See figure 9-23 for examples.

Arrays listed in the array list must exist in the BASIC program. Elements of the array are printed in row order with spacing between items controlled by the comma or semicolons (as with the PRINT command). A blank line is output after each row and an extra blank line is output between matrices. The MAT PRINT command is separated from the next command on the same line by using two semicolons (as with the PRINT command).

MAT PRINT array-list

| | |
|---|---|
| array-list | List of one or more of 1-, 2- or 3-dimensional arrays; separated by commas or semicolons. |

Figure 9-22. MAT PRINT Command for CID Format

```
MAT PRINT A,B
MAT PRINT X1$
```

Figure 9-23. MAT PRINT Command for CID Examples

## LIST VALUES COMMAND

The LIST VALUES command lists values of all variables within the program. The format is shown in figure 9-24.

The names and values of all variables, including arrays, are listed in alphabetical order. Formal arguments of user-defined functions are listed only if program execution was suspended inside and while executing the function DEF.

```
LIST VALUES

or

LV
```

Figure 9-24. LIST VALUES Command

# CHANGING AND TESTING PROGRAM VALUES

Two commands that can be used to change and test program values are the LET command and the IF command. These commands are similar to the BASIC statements.

## LET COMMAND FOR CID

The LET command is similar to the BASIC LET statement. It assigns values to program variables. The command can be used with simple and subscripted variables and substrings. The format is shown in figure 9-25.

See figure 9-26 for examples.

The variables referenced must exist in the BASIC program being debugged. Multiple assignments, references to functions, and use of the exponentiation operator are not allowed; all other arithmetic operators (+, -, *, and /) and the string concatenation operator can be used in the expressions.

## IF COMMAND FOR CID

The IF command is similar to the BASIC IF statement. It controls the selection of CID commands based on a comparison of program variables or computed values. See the CYBER Interactive Debug reference manual for further uses of the IF command in debug mode. The format is shown in figure 9-27.

The following is an example of the IF command:

```
IF A<=B THEN PRINT A
```

# OTHER COMMANDS AND FEATURES

There are many other CID features and commands. The following is a list of some features and commands not explained in this manual:

Sets of CID commands can be predefined to execute automatically when a breakpoint or trap occurs.

Breakpoints can be defined to occur every nth time through a loop.

A debug session can be suspended so that operating system commands can be entered. The debug session can then be resumed.

Sequences of commands can be saved on and read from files.

Other commands include HELP, LIST,BREAKPOINT and LIST,TRAP.

CID can be used interactively in ASCII mode only under NOS. CID can be used in normal mode under NOS and NOS/BE. See the CYBER Interactive Debug reference manual for further information regarding CID commands and features.

```
LET nv=ne

nv          Indicates numeric variable.

ne          Indicates restricted arithmetic expression.

    or

LET sv=se

sv          Indicates string variable.

se          Indicates restricted string expression.
```

Figure 9-25. LET Command for CID Format

```
LET A=A+45
LET B$(3,2)=A$ + "ABC"
LET D$(3:6)="DEFG"
LET F$(1)(4:6)=""
```

Figure 9-26. LET Command for CID Examples

```
IF re THEN db

re          Indicates any BASIC relational expression; variables must exist in BASIC program being debugged.

db          Indicates any CID or BASIC debugging command.
```

Figure 9-27. IF Command for CID Format

LIST VALUES

or

LV

Figure 9-24. LIST VALUES Command

# CHANGING AND TESTING PROGRAM VALUES

Two commands that can be used to change and test program values are the LET command and the IF command. These commands are similar to the BASIC statements.

## LET COMMAND FOR CID

The LET command is similar to the BASIC LET statement. It assigns values to program variables. The command can be used with simple and subscripted variables and substrings. The format is shown in figure 9-25.

See figure 9-26 for examples.

The variables referenced must exist in the BASIC program being debugged. Multiple assignments, references to functions, and use of the exponentiation operator are not allowed; all other arithmetic operators (+, -, *, and /) and the string concatenation operator can be used in the expressions.

## IF COMMAND FOR CID

The IF command is similar to the BASIC IF statement. It controls the selection of CID commands based on a comparison of program variables or computed values. See the CYBER Interactive Debug reference manual for further uses of the IF command in debug mode. The format is shown in figure 9-27.

The following is an example of the IF command:

```
IF A<=B THEN PRINT A
```

# OTHER COMMANDS AND FEATURES

There are many other CID features and commands. The following is a list of some features and commands not explained in this manual:

Sets of CID commands can be predefined to execute automatically when a breakpoint or trap occurs.

Breakpoints can be defined to occur every nth time through a loop.

A debug session can be suspended so that operating system commands can be entered. The debug session can then be resumed.

Sequences of commands can be saved on and read from files.

Other commands include HELP, LIST,BREAKPOINT and LIST,TRAP.

CID can be used interactively in ASCII mode only under NOS. CID can be used in normal mode under NOS and NOS/BE. See the CYBER Interactive Debug reference manual for further information regarding CID commands and features.

LET nv=ne

| | |
|---|---|
| nv | Indicates numeric variable. |
| ne | Indicates restricted arithmetic expression. |

or

LET sv=se

| | |
|---|---|
| sv | Indicates string variable. |
| se | Indicates restricted string expression. |

Figure 9-25. LET Command for CID Format

```
LET A=A+45
LET B$(3,2)=A$ + "ABC"
LET D$(3:6)="DEFG"
LET F$(1)(4:6)=""
```

Figure 9-26. LET Command for CID Examples

IF re THEN db

| | |
|---|---|
| re | Indicates any BASIC relational expression; variables must exist in BASIC program being debugged. |
| db | Indicates any CID or BASIC debugging command. |

Figure 9-27. IF Command for CID Format

# TERMINAL OPERATION UNDER NOS 10

NOS is the Network Operating System for CDC's CYBER 170, CYBER 70, and 6000 Series computer systems. NOS provides BASIC users with both a batch and an interactive processing environment. BASIC can be accessed from a remote time-sharing terminal, such as a Teletype Model 33 or Model 35 teletypewriter (TTY), or a CDC Model 713 CRT terminal.

When accessing NOS from a remote terminal, BASIC programs can be entered and executed by using either the BASIC subsystem or BATCH subsystem when under the NOS Interactive Facility (IAF). Also, data files created under NOS can be built at the terminal through use of TEXT mode, Text Editor, or XEDIT.

This section illustrates the use of the BASIC and BATCH subsystems, a method of creating data files at a terminal, and some tips on using the line editor when writing programs at a TTY or CRT terminal.

For a detailed description of 713 CRT and Model 33 or Model 35 TTY terminal usage, see the Networks Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the NOS Time-Sharing reference manual. For a detailed description of Text Editor or XEDIT usage, see the Text Editor reference manual or the XEDIT reference manual.

## ENTERING A PROGRAM

The process for interactively entering a program into a file is shown in the examples that follow. To correct an existing syntax, semantic, or logic error, enter the line number that contains the error, type in the corrected line, and press the carriage return key. To delete a line, enter the line number and press the carriage return key. To correct an error while typing a line, backspace n characters by pressing the backspace key or by holding down the control key and pressing H once for each incorrect character; then type the correct information. For additional control key information, refer to the NOS Network Products Interactive Facility reference manual, the NOS Time-Sharing User's reference manual, and the operator's guide for a specific terminal.

## BASIC SUBSYSTEM

When in the BASIC subsystem under IAF (or TELEX), a BASIC program can be written at a TTY or CRT terminal, and the program can be edited or executed interactively. The program in figure 10-1 was created and run at a terminal under IAF and the BASIC subsystem. Responses entered are in lowercase; the carriage return key is pressed after typing in each response.

For a detailed description of the NOS commands used in figure 10-1 and other available commands, see the Networks Interactive Facility reference manual or the NOS Time-Sharing User's reference manual. In figure 10-1, the program is saved as a file named EX4. The program in this file is now stored as an indirect-access permanent file that can later be accessed by use of the OLD command. (See figure 10-2. Responses entered are in lowercase.)

Use the REPLACE command to store the changed program; this replaces the old program with the corrected program. For example, the following command stores an updated program in file EX4:

```
REPLACE,EX4
```

The updated file EX4 is lost if the session is logged off before storing the corrected version.

## BATCH SUBSYSTEM

The batch subsystem provides batch control statement capability from the terminal. It enables control statements to be typed at the terminal; otherwise, control statements must be entered through a card reader at the central site or must be entered from a remote batch terminal that calls the statements from a procedure file or includes the statements in a submitted job.

BASIC can be run interactively in the batch subsystem. The BASIC control statement in the form X,BASIC is issued to call the BASIC compiler. All options of the BASIC control statement described in the section on Batch Operations are available when BASIC is run interactively in the batch subsystem.

The program contained in file SUM (figure 10-3) was written during a previous terminal session, while in the BASIC subsystem, and saved with the SAVE command. The BATCH command requests the batch subsystem, and the X,BASIC(I=SUM) command requests the BASIC compiler to compile and execute the program found in file SUM.

## USING DATA FILES

To create a data file under NOS 1 or NOS 2, specify the name of the new file and enter the TEXT command. The TEXT command permits the file to be created without sequence numbers. If, after the file is created, corrections, additions, or deletions are required, enter EDIT or XEDIT and use Text Editor or XEDIT commands. (For a complete description of Text Editor or XEDIT commands, see the Text Editor reference manual or the XEDIT reference manual.)

```
/basic                                  Request BASIC subsystem.
OLD, NEW, OR LIB FILE: new,ex4          Create new file named EX4.

 READY.

10 print "type a number"
20 input x
25 let f=1
30 for i=1 to x                         Enter BASIC statements.
40 let f=f*i
50 print "factorial ";x,"is ";f
60 goto 10
70 end
list                                    List file EX4.

10 PRINT "TYPE A NUMBER"
20 INPUT X
25 LET F=1
30 FOR I=1 TO X
40 LET F=F*I
50 PRINT "FACTORIAL ";X,"IS ";F
60 GOTO 10
70 END

 READY.
run                                     Compile and execute program.

 FOR WITHOUT NEXT  AT  30               Program contains an error.
BASIC COMPILATION ERRORS

 RUN COMPLETE.
45 next i                               Correct error by entering more BASIC statements.
24 if x=0 then 70
list                                    List file EX4.

10 PRINT "TYPE A NUMBER"
20 INPUT X
24 IF X=0 THEN 70
25 LET F=1
30 FOR I=1 TO X
40 LET F=F*I
45 NEXT I
50 PRINT "FACTORIAL ";X,"IS ";F
60 GOTO 10
70 END

 READY.
run                                     Compile and execute program again.

TYPE A NUMBER
? 3                                     Input 3 to executing program.
FACTORIAL  3   IS  6
TYPE A NUMBER
? 0                                     Input 0 to executing program.
 RUN COMPLETE.

save,ex4                                Make file EX4 permanent.

 READY.
```

Figure 10-1. BASIC Subsystem Under NOS

```
old,ex4 <------------------------------ Make file EX4 accessible.

 READY.
list <------------------------------ List program.

10 PRINT "TYPE A NUMBER"
20 INPUT X
24 IF X=0 THEN 70
25 LET F=1
30 FOR I=1 TO X
40 LET F=F*I
45 NEXT I
50 PRINT "FACTORIAL ";X,"IS ";F
60 GOTO 10
70 END

 READY.
run <------------------------------ Compile and execute program.

TYPE A NUMBER
? 6
FACTORIAL  6   IS  720
TYPE A NUMBER
? 0
 RUN COMPLETE.
bye <------------------------------ Log off.
```

Figure 10-2. OLD Command Under NOS

```
batch <------------------------------ Enter the BATCH subsystem.
RFL,0.
/get,sum
/list,f=sum
10 INPUT N
20 PRINT TAB(2);"INTEGER","SUM"
30 LET S=0
40 FOR I=1 TO N
50 LET S=S+I
60 PRINT TAB(5);I,S
70 NEXT I
80 END
 EOI ENCOUNTERED.
/rewind,sum
REWIND,SUM.
/x,basic(i=sum) <------------------------------ Compile and execute program on file SUM.
? 10
 INTEGER        SUM
     1           1
     2           3
     3           6
     4           10
     5           15
     6           21
     7           28
     8           36
     9           45
     10          55
```

Figure 10-3. Program Executed Interactively Under BATCH Subsystem

Under NOS 2, data files also can be created in a two step process. First of all, the data, with line numbers, is entered into a file. The line numbers allow the use of the in-line editing commands. When all the data is correct, the in-line edit command WRITEN will copy all the data from the line numbered file, excluding the line numbers, to the file specified in the command. (See appendix I for further explanation of the in-line editing commands.)

In figure 10-4, the first data file is created using the two step process. The data is initially entered into file TCLIENT using the AUTO mode. (In AUTO mode, line numbers are supplied by the system.) Using the line numbers as a reference, the data is corrected. When the data has been corrected, the WRITEN command is used to write the data, without line numbers, to file CLIENT. File CLIENT is then made permanent using the SAVE command.

The second data file is created using the TEXT command and inserting the data line by line. Each line ends by pressing the carriage return key. TEXT Mode is terminated by using the user break 1 or 2; see the Network Products Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), NOS Time-Sharing User's reference manual, and the operator's guide for a specific terminal. After terminating TEXT mode, file UPDATE is made permanent under its respective name by the using the SAVE command.

Later, local copies of the files are made by using the GET command in order to make them accessible to the BASIC program. For additional file handling information, see appendix D.

## RENUMBERING BASIC LINES

In the BASIC subsystem, the RESEQ command resequences BASIC programs and automatically updates all line references. The format of the RESEQ command is shown in figure 10-5, and an example of this command is illustrated in figure 10-6.

For additional information, see the NOS Time-Sharing User's reference manual, the Network Products Interactive Facility reference manual (NOS 1 sites), and Volume 3 of the NOS 2 reference set (NOS 2 sites).

```
new,tclient <-------------------------------- Establish a new file named TCLIENT.

 READY.
ascii <-------------------------------------- Activate ASCII mode.

 READY.
auto <--------------------------------------- Activate automatic line number
                                              generation.
00100 J.Brown,1422 East St,charge no 1111,510.00
00110 R.Apple,3434 Chery St,charge no 2211,222.22 <-- Enter data into file TCLIENT.
00120 N.Redi,7896 Algo Ave,charge no 1660,133.98
00130 <-------------------------------------- Enter user break to stop line number
 *TERMINATED*                                 generation.

alter,110,/Chery/Cherry/ <------------------- Correct date.

00110 R.Apple,3434 Cherry St,charge no 2211,222.22

 READY.                                       Write all data in file TCLIENT
writen,client <------------------------------ (excluding line numbers) to file
                                              CLIENT.
 READY.

save,client <-------------------------------- Make file CLIENT permanent.
 READY.

new,update <--------------------------------- Establish new file UPDATE.

 READY.
text

ENTER TEXT MODE.

10.56
20.53 <-------------------------------------- Enter data into file UPDATE.
13.78

 <------------------------------------------- Enter user break to exit text mode.
 EXIT TEXT MODE.
```

Figure 10-4. Using Data Files Under NOS (Sheet 1 of 2)

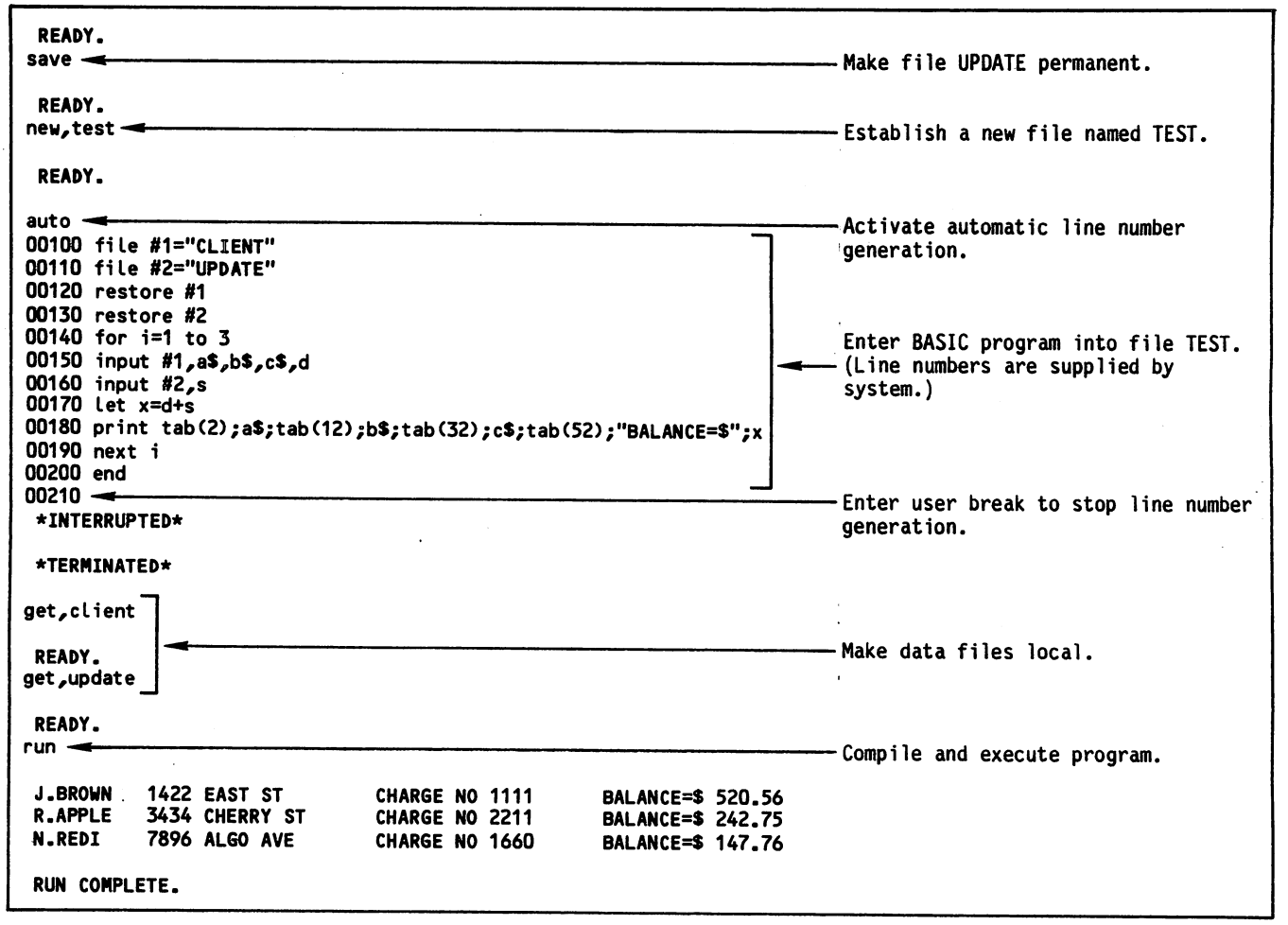


Figure 10-4. Using Data Files Under NOS (Sheet 2 of 2)

RESEQ,nn,ii

| | |
|---|---|
| nn | Indicates new line number of the first statement in the file; maximum size is five digits; if omitted, the default is 00100. |
| ii | Indicates increment to be added to nn; default value is 10. |

Figure 10-5. RESEQ Command Format

```
/basic <------------------------------ Enter BASIC subsystem.
OLD, NEW, OR LIB FILE: old,ex4 ------- Make file EX4 accessible.

 READY.
list <------------------------------- List BASIC program (file EX4).

10 PRINT "TYPE A NUMBER"
20 INPUT X
24 IF X=0 THEN 70
25 LET F=1
30 FOR I=1 TO X
40 LET F=F*I
45 NEXT I
50 PRINT "FACTORIAL ";X,"IS ";F
60 GOTO 10
70 END

 READY.
reseq <------------------------------ Resequence file EX4.

 READY.
list <------------------------------- List resequenced file.

00100 PRINT "TYPE A NUMBER"
00110 INPUT X
00120 IF X=0 THEN 00190
00130 LET F=1
00140 FOR I=1 TO X
00150 LET F=F*I
00160 NEXT I
00170 PRINT "FACTORIAL ";X,"IS ";F
00180 GOTO 00100
00190 END

 READY.
```

Figure 10-6. RESEQ Command Example

The Network Operating System/Batch Environment (NOS/BE) permits multiple-user access to CDC's CYBER 170, CYBER 70, and 6000 Series computers. From a remote terminal, the INTERCOM commands and directives can be used to enter and execute BASIC programs interactively, to create and submit BASIC programs for batch execution, and to create data files to be accessed by BASIC programs. The remote terminal can be any teletypewriter (TTY) or CRT supported by NOS/BE.

This section describes and illustrates the creation of BASIC programs for interactive processing; a method of creating data files to be accessed by a BASIC program; and the utility for renumbering BASIC programs. For a complete description of Text Editor commands, and remote terminals supported by NOS/BE, see the INTERCOM Version 5 reference manual. Creation and submission of BASIC programs for batch processing is described in the section on Batch Operations.

## ENTERING A PROGRAM

When creating a BASIC program that is to be run interactively or submitted for batch execution, first enter text edit mode. Text edit mode can be entered at any time after the login sequence is completed by typing EDITOR and the carriage return key after the system prompt:

```
COMMAND-
```

The system editor responds with two consecutive periods, indicating text edit mode is in effect. After the EDITOR command, enter the following command (after the periods):

```
..FORMAT,BASIC (CR)
```

This command establishes a special BASIC program environment. The maximum line length is established at 150 characters. BASIC line numbers serve as EDITOR sequence numbers, and EDIT with SEQUENCE, CREATE, ADD, or RESEQ becomes illegal. Once specified, the BASIC format environment remains in effect for the duration of the terminal session or until the one of the following is specified: a FORMAT without parameters, or a FORMAT with a COMPASS, FORTRAN or COBOL parameter (such as FORMAT,COBOL).

Once the FORMAT command is accepted (apparent by two periods displayed on the next line following the command), enter program text in one of the following two forms: line number (one space) text (for the BASIC format) or line number = text (in other formats). If an error is made while typing a line, back space n characters by pressing the backspace key or by typing CONTROL H, n times, and enter the correct information, or erase the entire line by pressing CONTROL X. The CONTROL key must be held down while the H or X key is pressed. To correct an existing line, reenter the line number and type the correct information. To delete an existing line, type DELETE, line number. If an entered line exceeds 150 key strokes, it is truncated and a message is displayed at the terminal.

## INTERACTIVE BASIC TERMINAL SESSION

A BASIC program can be entered, edited, and executed interactively from a CRT or TTY terminal. Figure 11-1 was created and run at a TTY terminal. Responses entered are in lowercase. Press the carriage return key, (CR) , after typing in each response.

## USING THE BASIC COMMAND INTERACTIVELY

Basic can be run interactively using the full capability of the BASIC control statement, described in section 13, by performing the following steps:

1. Create the BASIC program under EDITOR.
2. Save the program by entering:

   ```
   SAVE,lfn
   ```

   For a program created in BASIC format

   ```
   SAVE,lfn,NOSEQ
   ```

   For a program created in other than BASIC format
3. To leave EDITOR, type in END.
4. Connect required files to terminal by entering:

   ```
   CONNECT,lfn1,lfn2,...
   ```

   Normally the J and K files on the BASIC command (default INPUT and OUTPUT) should by connected.
5. Compile and execute the program by entering:

   ```
   BASIC(I=lfn,...)
   ```

An example of these command parameters is shown in figure 11-2. BASIC command parameters are described in the section on Batch Operations.

```
CONTROL DATA INTERCOM 5.1
DATE  06/25/80
TIME  14.30.50

PLEASE LOGIN
login

ENTER USER NAME-xxx

xxxxxxxx ENTER PASSWORD-


06/25/81   LOGGED IN AT  15.35.15.
           WITH USER-ID 4K
           EQUIP/PORT 63/073


LOGIN       CREATED 06/25/81  TODAY IS 06/25/81
```

← Login procedure.

```
COMMAND- editor
..format,basic
```

← Calls EDITOR.
← Requests BASIC format.

```
..10 rem this program computes interest payments
20 print "enter total amount of loan"
21 input a
22 if a<=0 then 200
26 print "enter interest percentage"
30 input j
35 let j=j/100
40 print "enter total number of years"
45 input n
50 print "enter number of payments per year"
55 input m
60 let n=n*m
65 let i=j/m
70 let b=1+i
75 let r=a*i/(1-1/b**n)
79 amount per payment =$$$$#.##
80 :total interest =$$$$$#.##
81 print using 79,r
82 print using 80,r*n-a
88 print
94 :interest     app to prin $  balance $
95 print using 94
100 let l=a*i
110 let p=r-l
120 let a=a-p
130 print using 135, l,p,a
135 :###.##       ###.##        #####.##
140 if a>=r then 100
150 print using 135, a*i,r-a*i
155 print
160 print using "last payment =$$$#.##",a*i+a
170 goto 20
200 end
run,basic
```

← Enters BASIC program line-by-line.
← Compiles and executes program.

```
ILLEGAL STATEMENT AT   79
NON-IMAGE REFERENCED  AT  81
 BASIC COMPILATION ERRORS
```

← BASIC issues diagnostic.

```
..79 :amount per payment =$$$$#.##
run,basic
```

← Corrects error.
← Compiles and executes program again.

Figure 11-1. Interactive BASIC Terminal Session (Sheet 1 of 2)

```
ENTER TOTAL AMOUNT OF LOAN
?5000
ENTER INTEREST PERCENTAGE
?11
ENTER TOTAL NUMBER OF YEARS
?2
ENTER NUMBER OF PAYMENTS PER YEAR
?6
AMOUNT PER PAYMENT = $467.97
TOTAL INTEREST =  $615.66

INTEREST     APP TO PRIN $  BALANCE $
 91.67       376.31         4623.69
 84.77       383.20         4240.49
 77.74       390.23         3850.26
 70.59       397.38         3452.88
 63.30       404.67         3048.21
 55.88       412.09         2636.12
 48.33       419.64         2216.48
 40.64       427.34         1789.14
 32.80       435.17         1353.97
 24.82       443.15          910.82
 16.70       451.27          459.55
  8.43       459.55

LAST PAYMENT =$467.97
ENTER TOTAL AMOUNT OF LOAN
?0
```

Program asks for input and generates output.

```
..save,basprog
..store,basprog
..list,all,sup
```

Saves edit file in local file BASPROG.
Makes BASPROG a permanent file.
Lists edit file without sequence numbers.

```
    10 REM THIS PROGRAM COMPUTES INTEREST PAYMENTS
    20 PRINT "ENTER TOTAL AMOUNT OF LOAN"
    21 INPUT A
    22 IF A<=0 THEN 200
    26 PRINT "ENTER INTEREST PERCENTAGE"
    30 INPUT J
    35 LET J=J/100
    40 PRINT "ENTER TOTAL NUMBER OF YEARS"
    45 INPUT N
    50 PRINT "ENTER NUMBER OF PAYMENTS PER YEAR"
    55 INPUT M
    60 LET N=N*M
    65 LET I=J/M
    70 LET B=1+I
    75 LET R=A*I/(1-1/B**N)
    79 :AMOUNT PER PAYMENT =$$$$#.##
    80 :TOTAL INTEREST =$$$$$#.##
    81 PRINT USING 79,R
    82 PRINT USING 80,R*N-A
    88 PRINT
    94 :INTEREST     APP TO PRIN $  BALANCE $
    95 PRINT USING 94
   100 LET L=A*I
   110 LET P=R-L
   120 LET A=A-P
   130 PRINT USING 135, L,P,A
   135 :###.##       ###.##        #####.##
   140 IF A>=R THEN 100
   150 PRINT USING 135, A*I,R-A*I
   155 PRINT
   160 PRINT USING "LAST PAYMENT =$$$#.##",A*I+A
   170 GOTO 20
   200 END
..
```

Figure 11-1. Interactive BASIC Terminal Session (Sheet 2 of 2)

```
COMMAND- editor
..format,basic
..100 print "sample program"
save,ex1
..end
COMMAND- connect,input,output
COMMAND- basic(i=ex1,l)

    EX1            BASIC 3.5 81027      06/25/81  16.08.25.      PAGE     1

         100 PRINT "SAMPLE PROGRAM"


SAMPLE PROGRAM
```

Figure 11-2. BASIC Command Parameters Under NOS/BE

## USING DATA FILES

Data files to be used by a BASIC program can be created under EDITOR. To create data files acceptable to the BASIC program, select a format; the format must be a format other than BASIC. In the BASIC format, line numbers are part of the text and cannot be removed.

Data is entered one line at a time in line number=text format. After the entire file is created, save the file (file becomes local file) without sequence numbers by using the SAVE,lfn,NOSEQ command. EDITOR line numbers are stripped when the SAVE command with no sequence number option (NOSEQ) is selected. To edit a file that was saved without sequence numbers, enter the EDIT,lfn,SEQ command. The SEQ parameter causes an EDITOR line number to be appended to each line of text. An example of using data files under NOS/BE is illustrated in figure 11-3.

```
COMMAND- editor
..format,fortran                          <- Chooses a format other than BASIC for data.
..create                                  <- Creates and places data in edit file; the line numbers are supplied by EDITOR. The equals sign terminates input.
   100=j.brown,1422 east st,charge no 1111,500.00
   110=r.apple,3434 cherry st,charge no 2211,222.22
   120=h.redi,7896 algo ave,charge no 1660,133.98
   130==
..save,client,noseq                       <- Saves edit file as local file CLIENT without sequence numbers.
..create
   100=10
   110=20
   120=30
   130==
..save,update,noseq
..delete,all                              <- Deletes contents of edit file.
..format,basic                            <- Chooses BASIC format for program.

..5 file #2="update"                      <- Creates a BASIC program.
10 file #1="client"
20 restore #1
30 restore #2
40 for i=1 to 3
50 input #1,a$,b$,c$,d
60 input #2,s
70 let x=d+s
80 print tab(2);a$;tab(12);b$;tab(32);c$;tab(52);"balance=$";x
90 next i
100 end
run,basic                                 <- Executes the program.

 J.BROWN   1422 EAST ST       CHARGE NO 1111     BALANCE=$ 510      <- Program output.
 R.APPLE   3434 CHERRY ST     CHARGE NO 2211     BALANCE=$ 242.22
 H.REDI    7896 ALGO AVE      CHARGE NO 1660     BALANCE=$ 163.98

..save,test                               <- Saves the program as local file TEST.
..end                                     <- Exits EDITOR.
COMMAND-
```

Figure 11-3. Using Data Files Under NOS/BE

# RENUMBERING BASIC LINES

The BRESEQ command provides a means of resequencing the line numbers in a BASIC local file. Line number references in the BASIC program are automatically updated. The format for the BRESEQ command is shown in figure 11-4. When only one parameter is specified, it is assumed to be the starting line number for the new file, and the default increment value (10) is used.

BRESEQ,lfn,start,incr

| | |
|---|---|
| lfn | Indicates filename of the local file to be resequenced. |
| start | Indicates new line number to be assigned to the first line in the file. |
| incr | Indicates increment to be added to nn; default value is 10. |

Figure 11-4. BRESEQ Command Format

The BASIC file must exist as a local file and cannot be the local name for an attached permanent file. To resequence a permanent file, copy the file and assign a unique filename. This can be accomplished by the use of the COPY command or by loading the file into an EDIT file and then using the SAVE command.

The BRESEQ command affects only the specified local file and not the edit file. If further modifications are to be performed, the resequenced file must be reloaded into the EDITOR edit file by using the following directive:

```
EDIT,filename
```

An example of the BRESEQ command and reloading of the resequenced file is shown in figure 11-5.

```
COMMAND- editor

..format,basic
..5 print "type a positive number"
10 input a
50 if a<0 then 80
60 print using 71, a
71 :+### is positive
75 stop
80 print a;" is negative, try again"
100 goto 10



save,ex

..breseq(ex,10,10)

..edit,ex

..list,all,sup


00010 PRINT "TYPE A POSITIVE NUMBER"
00020 INPUT A
00030 IF A<0 THEN 00070
00040 PRINT USING 00050, A
00050 :+### IS POSITIVE
00060 STOP
00070 PRINT A;" IS NEGATIVE, TRY AGAIN"
00080 GOTO 00020
..
```

Figure 11-5. BRESEQ Command Example

A batch job includes a user-written program, associated data, and control statements organized as separate logical records. A batch job can be input through a card reader at the central site, input from a remote batch terminal, invoked from a procedure, or, if the batch job is stored on a file or created during an interactive terminal session, it can be entered into the batch queue from the interactive terminal.

This section describes the general structure of a batch job, the BASIC control statement parameters, and the procedure for creating and submitting a batch job under NOS or NOS/BE. Figure 12-1 shows the control statements for a batch job under NOS and figure 12-2 shows the control statements under NOS/BE. The BASIC statement can be used to compile and execute your program, or you can use the B option on the BASIC control statement to place the object code on a file. Figures 12-1 and 12-2 place the object code on the file LGO, then load and execute the file LGO.

| | |
|---|---|
| Job statement | Specifies job name, and optionally, the memory and time requirements, priority, and other information. |
| USER and CHARGE statements | Specifies accounting information for NOS. CHARGE might be optional at your site. |
| BASIC statement | Calls the BASIC compiler. If the B option is specified, the object code is written on the specified file; otherwise, it is written into memory and executed immediately. |
| LGO. | Leads and executes the binary file LGO. If B = LGO is not specified on the BASIC control statement, omit this statement. |
| 7/8/9 | Indicates end-of-record. |

Figure 12-1. Job Structure Under NOS

## DECK STRUCTURE

Compile-to-memory enables you to compile and execute a BASIC program without loading a binary file. Thus, you need only specify the BASIC control statement in order to compile and execute the program, and you need not and must not specify the B option.

| | |
|---|---|
| Job statement | Specifies job name, and optionally, the memory and time requirements, priority, and other information. |
| ACCOUNT statement | Specifies accounting information for NOS/BE. |
| BASIC statement | Calls BASIC compiler. If the B option is specified, the object code is written on the specified file; otherwise, it is written into memory and executed immediately. |
| LGO. | Leads and executes the binary file LGO. If B = LGO is not specified on the BASIC control statement, omit this statement. |
| 7/8/9 | Indicates end-of-record. |

Figure 12-2. Job Structure Under NOS/BE

Both compile-to-memory (no B option on control statement) and compiling to a binary file (using the B option) are allowed on NOS and NOS/BE. An example of a compile-to-memory job deck for use under NOS is shown in figure 12-3; an example for use under NOS/BE is shown in figure 12-4. An example of compiling a BASIC program to a binary file and then loading and executing that file under NOS is shown in figure 12-5; an example for use under NOS/BE is shown in figure 12-6.

Information on entering a job from an interactive terminal can be found in the Network Products Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), the XEDIT reference manual for NOS, and in the INTERCOM Version 5 reference manual for NOS/BE.

A 6/7/8/9 statement specifies end-of-information (end-of-deck). A complete description of the BASIC control statement follows. Refer to the NOS or NOS/BE reference manual for a detailed description of these and other control statements.

## BASIC CONTROL STATEMENT

Programs submitted for batch processing must include a BASIC control statement. This control statement calls the compiler and is formatted as follows:

BASIC($p_1$,...,$p_n$)

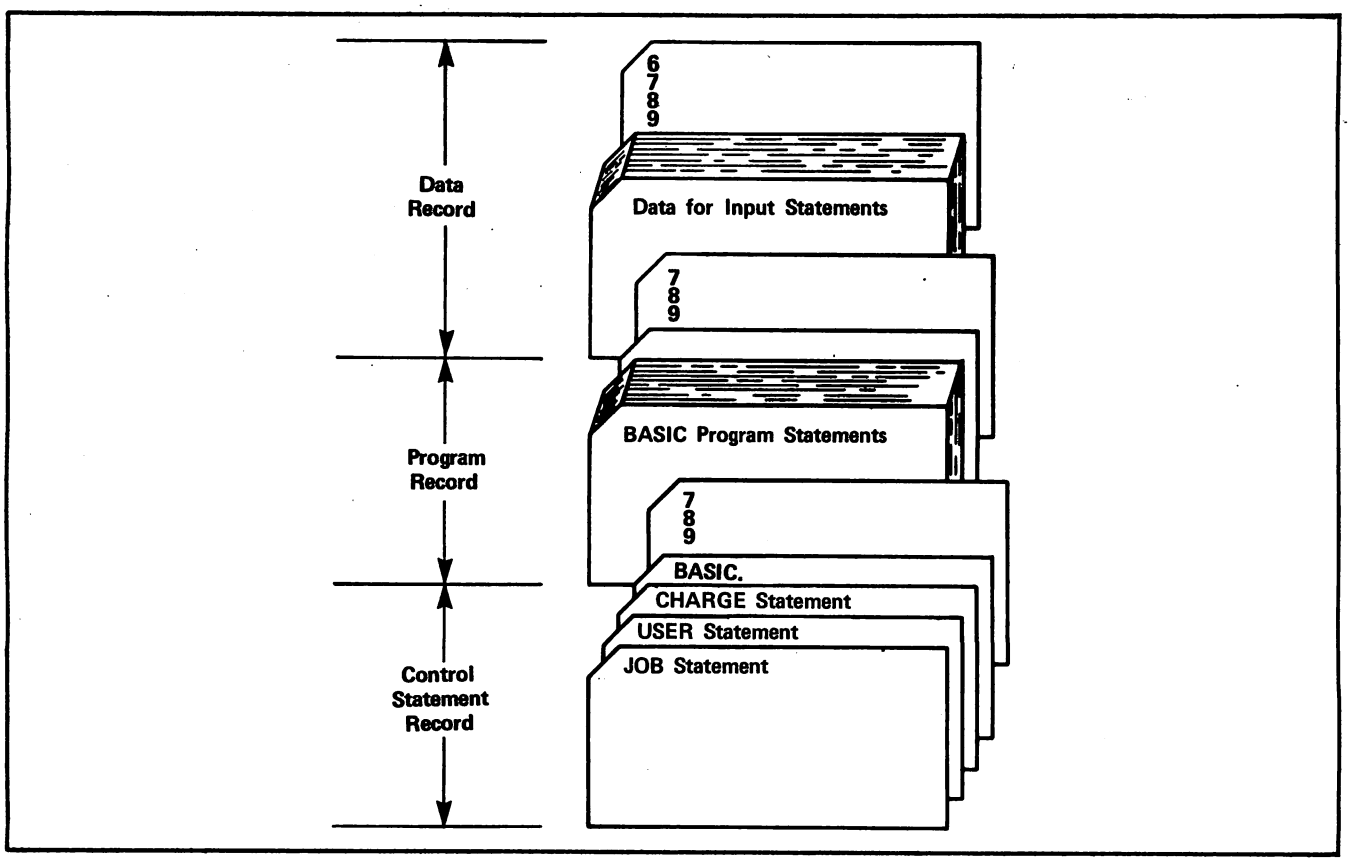


Figure 12-3. BASIC Compile and Execute Job Under NOS

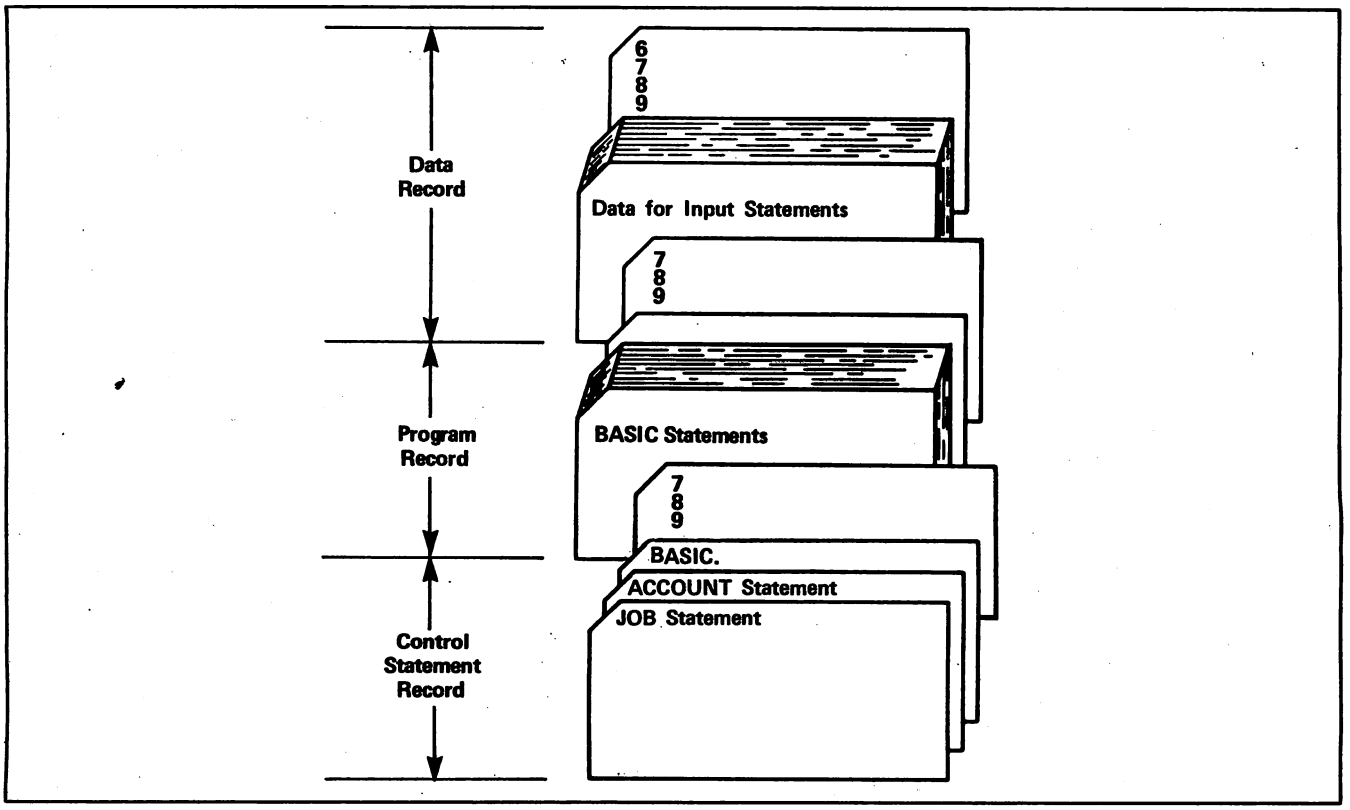


Figure 12-4. BASIC Compile and Execute Job Under NOS/BE

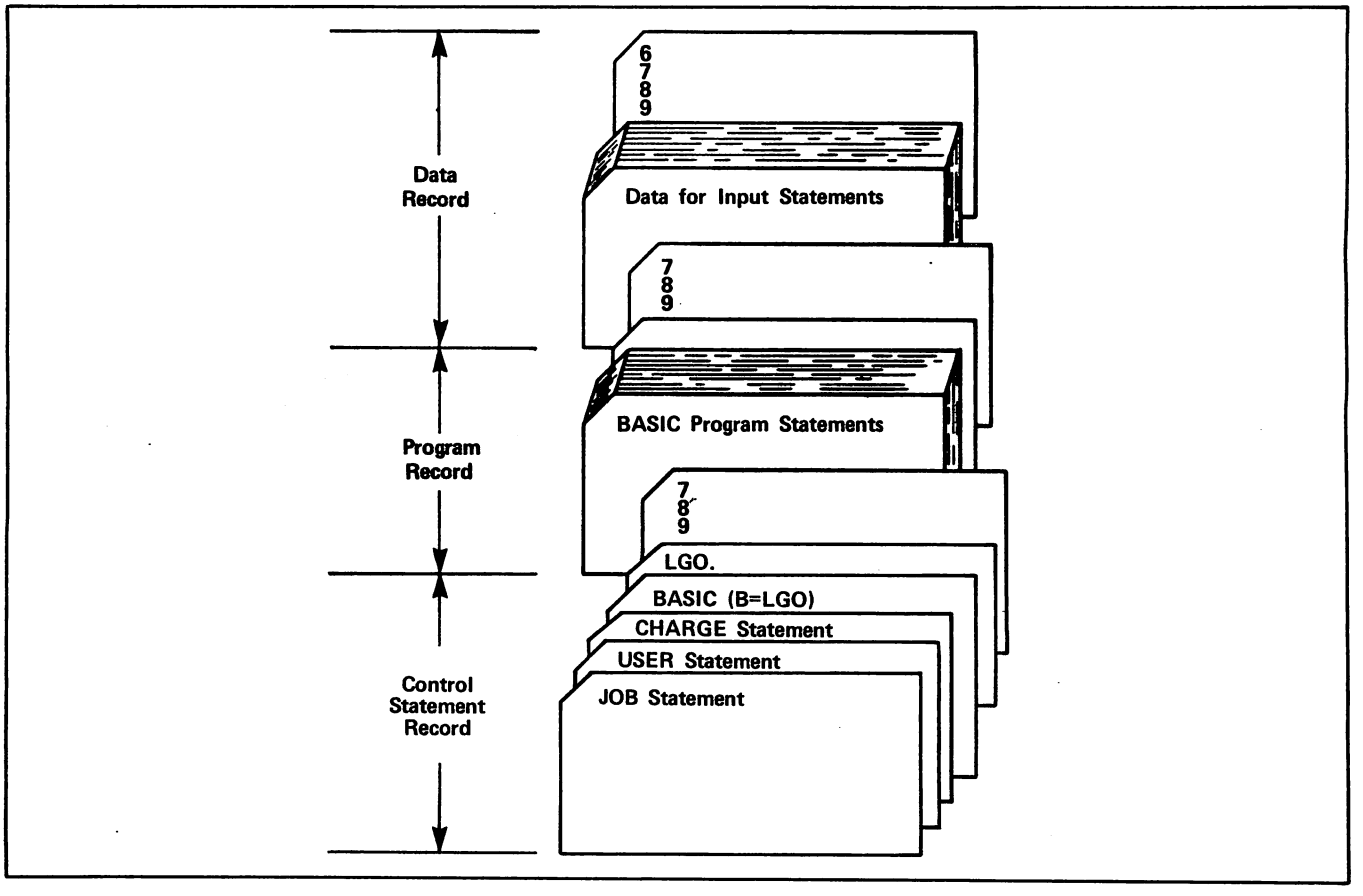


Figure 12-5. BASIC Compile to Binary File, Load, and Execute Job Under NOS

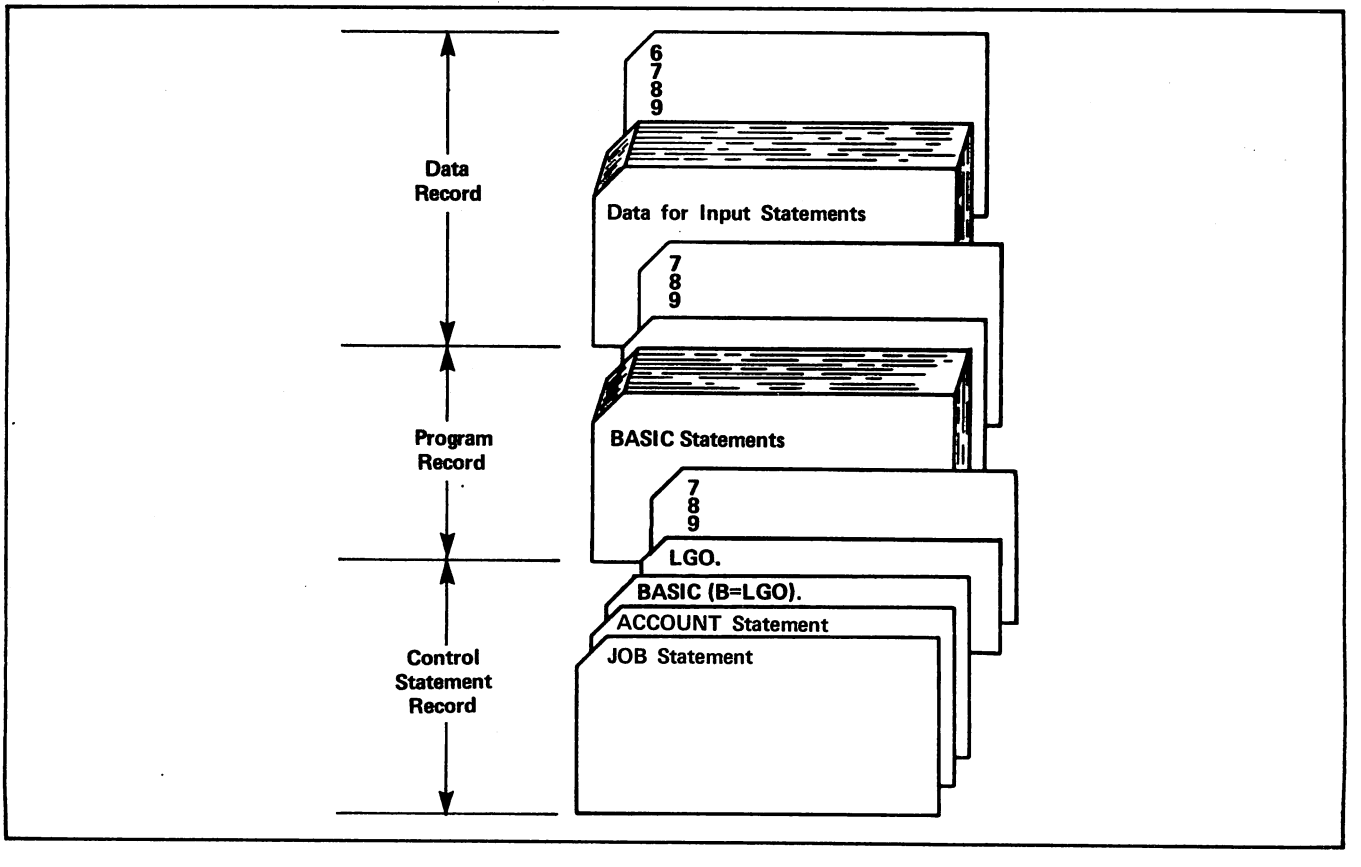


Figure 12-6. BASIC Compile to Binary File, Load, and Execute Job Under NOS/BE

The simplest form of the BASIC control statement is:

```
BASIC.
```

This control statement specifies that the BASIC program on file INPUT is to be compiled and executed. A source listing is produced on file OUTPUT unless the control statement was issued from a terminal. A relocatable binary file is not produced. The parameters ($p_1,...,p_n$) associated with this control statement permit the selection of the following parameter types:

Compiler listable output options

Compiler input options

Compiler binary options

Program execution options

Tables 12-1 through 12-4 list available control statement parameters under the appropriate category, and describe their use. Some control statement parameters can have multiple values associated with them. Multiple values are separated by slashes and are cumulative.

The following examples illustrate some combinations of control statement parameters and the following paragraphs discuss possible options.

Compile and Execute

```
BASIC(B=SAM,GO)
```

The above control statement compiles the program found on file INPUT (I parameter default), places the compiler binary output on file SAM (B=SAM), and loads and executes the compiled program (GO). Execution-time output is written on file OUTPUT (K parameter default). Compile-time errors prevent execution and, when detected, are written to file OUTPUT (E and EL parameter default). A source list is created on file OUTPUT (L parameter default and LO parameter default) unless it is assigned to a terminal. When under NOS, source listing is not written when the program is in interactive mode (default L value is zero) because file OUTPUT is automatically associated with the terminal.

ASCII Compile and Execute

```
BASIC(AS,I=PROG3)
```

The AS parameter specifies that the source code found in file PROG3 is encoded in ASCII characters and that data produced by the BASIC program is in the ASCII character set. The program is compiled-to-memory and executed immediately (B and GO parameter defaults). A source listing is produced on file OUTPUT unless it is assigned to a terminal (L parameter default). On NOS, the source code of the program must be in ASCII 6/12 characters. On NOS/BE, the source can be in either display code (6-bit) characters or in ASCII 8/12 characters.

Compile, Execute, and List

```
BASIC(I=SOURCE,L=LIST)
```

This control statement compiles-to-memory and executes; compiler input (source) is on file SOURCE. Listable compiler output is written on file LIST. Source listing is specified by the LO parameter default. Error diagnostics are written on file LIST (default of E parameter). Source code and data do not contain ASCII characters.

Compile and Execute with Listing Options and Controls

```
BASIC(I=TESTP,EL=F,LO=0,PD=8,PS=20)
```

The compiler input (source code) is on file TESTP (I=TESTP) and the program is compiled and executed (default B and GO parameters). Compile-time errors are written on file OUTPUT. However, warning diagnostics are suppressed (EL=F). Also, a source and object listing is written on file OUTPUT (default L parameter and LO=0). Print density of file OUTPUT is set to 8 (PD=8). ROUTE (DISPOSE on NOS/BE) the file OUTPUT to a device that can print 8 lines per inch. Page size for the printed output file is set at 20 lines per page (PS=20).

## REM LIST STATEMENT

The REM LIST statement controls the source code listing produced by the BASIC compiler. The format of the REM LIST statement is shown in figure 12-7.

```
1. REM LIST,NONE
2. REM LIST,ALL
```

Figure 12-7. REM LIST Statement Format

REM LIST,NONE inhibits all further source code listing until a REM LIST,ALL statement is executed, which causes listing of the source code to resume. Both statements are printed by BASIC to permit you to see where the source code listing was suppressed.

TABLE 12-1. COMPILER LISTABLE OUTPUT PARAMETERS

| Parameter | Parameter Format | Description | Remarks |
|---|---|---|---|
| Compile-Time List File (L) | omitted | For batch jobs, default list file is OUTPUT. For interactive jobs, default is no compiler listable output file (same as L=0). | If the program is in ASCII, the listing file must be sent to the ASCII printer, not to the normal 64-character printer. |
| | L | Listable compiler output on file OUTPUT. | |
| | L=lfn | Listable compiler output on file lfn. | |
| | L=0 (zero) | No compile list file. | |
| Listing Options (LO) | omitted or LO or LO=S | Produces a source listing on a file specified by the L parameter. | This parameter can have multiple values associated with it. Values are separated by slashes and are cumulative. |
| | LO=O (letter O) | Produces a source listing and an object listing on L file. | S is on by default. |
| | LO=0/O (zero/ letter) | Produces an object listing on L file. | The number zero (0) turns off all previously specified values. In this case, it turns off the default S value. The letter O turns on object listing. |
| | LO=0 (zero) | Turns off all list options. | |
| Burstable Listing Control (BL) | omitted | Page ejects between portions of the compiler output (source listing and object listing) are suppressed; listing is not burstable. If the compile-to-memory and execute-in-one-step option is selected (no B option), both the page between the compiler output eject and the first line of execution output is suppressed; four blank lines are listed instead. | The installation can change the meaning of BL omitted to that specified by the BL option that follows. |
| | BL | Includes page ejects between compiler output portions and between the compiler output and the first line of execution output. | Does not apply if the L file is the terminal. |
| Print Density Control (PD) | omitted | Specifies the print density (lines-per-inch) on the files specified by the L and K parameters as the installation default (usually 6). | |
| | PD=6 | For L and K files, sets print density to 6. Automatically resets after all output is written to the files. | Only effective on an output device whose density can be changed. |
| | PD=8 or PD | For L and K files, sets the print density to 8 and automatically resets the density to the default value after output is written to the files. | |

TABLE 12-1. COMPILER LISTABLE OUTPUT PARAMETERS (Contd)

| Parameter | Parameter Format | Description | Remarks |
|---|---|---|---|
| Page Size Control (PS) | omitted | If PD is not specified, uses installation default page size (number of printable lines per page excluding upper and lower margins for the file).<br><br>If PD specifies a nondefault print density, PS is calculated as follows:<br><br>PS=PD*(default PS)/(default PD). | |
| | PS=n (where n is $4 \leq n \leq 32768$) | Establishes the L file page size as n printable lines per page. This parameter has no effect on execution output. Lines are not counted at execution time. | Four is the smallest possible page size because each page must include a 3-line header and at least one additional line. |
| Compile-Time Error File (E) | omitted | Compile-time error diagnostics written on the file specified by the L parameter. If there is no L file (L=0), diagnostics are written to file OUTPUT. | If the program is in ASCII, the listing file must be sent to the ASCII printer, not to the normal 64-character printer. |
| | E | Compiler error diagnostics are written on file ERRS. | |
| | E=lfn | Error diagnostics are written on file lfn. | Diagnostics are listed only once even when the file specified by the E parameter is the same as that specified by the L parameter. |
| Error Level Control (EL) | omitted or EL=W | Writes warning diagnostics and fatal compiler diagnostics on the file specified by the E parameter. | |
| | EL=F | Fatal error diagnostics, but no warning diagnostics, are written on the E file. | |

TABLE 12-2. COMPILER INPUT PARAMETERS

| Parameters | Parameter Format | Description | Remarks |
|---|---|---|---|
| ASCII Character Set (AS) | omitted or AS=0 (zero) | Source program and data files contain only normal (non-ASCII display code) characters. (See appendix A.) | |
| | AS | Source program and data are encoded in the extended ASCII character set. (See appendix A.) Program runs in ASCII mode. | Under NOS/BE, a normal character set source program is also acceptable. |
| Compile-Time Input (I) | omitted | Compiler input (BASIC source program) is on file INPUT. | Normally the program ID (name) contained in optionally generated relocatable binary decks is the name of the source file specified in the I parameter. The only exception occurs when the I file is the system file, INPUT or COMPILE; in which case, the program name in the binary deck is BASICXX. |
| | I | Compiler input (BASIC source program) is on file COMPILE. | |
| | I=lfn | Compiler input (BASIC source program) is on file lfn. | |

TABLE 12-3. COMPILER BINARY OUTPUT PARAMETERS

| Parameter | Parameter Formats | Description | Remarks |
|---|---|---|---|
| Binary File (B) | omitted or B=0 (zero) | Compile-to-memory. Does not produce a relocatable binary. | Automatic execution is controlled by the GO parameter. See table 12-4. If in CID mode (DEBUG (ON) has been executed, or DB=ID has been specified), a relocatable binary is written onto the reserved system file ZZZZZDC. |
| | B | Binary of compiled program written on file BIN. | |
| | B=lfn | Binary of compiled program written on file lfn. | |
| Debug (DB) | omitted | Trace feature, force binary generation, and program execution are not activated. CID feature is not activated unless an explicit DEBUG or DEBUG(ON) command has been previously issued. | Binary generation and automatic program execution are inhibited by compilation errors, and REM TRACE statements are only comments. Generation of CID information is controlled solely by an explicit DEBUG command. |
| | DB=0† (zero) | CID and trace features are not activated. | Same as if omitted except that CID information is inhibited even if an explicit DEBUG command has been issued. |
| | DB | Same as DB=B/DL. | The default list of parameters. |
| | DB=0/B† (zero) | Forces binary generation and/or program execution regardless of compilation errors. | See B and GO parameters. A program containing compilation errors executes normally until a statement that caused the compilation error is encountered. |
| | DB=0/DL† (zero) | Activates program tracing as controlled by REM TRACE debug lines. | See section 9. |
| | DB=0/ID† (zero) | Activates generation of CID information. | Causes generation of CID tables and special code. |
| | DB=0/TR† (zero) | Traces all statements regardless of REM TRACE debug lines. | Turns on the trace feature. |
| | DB=TR | Same as DB=B/DL/TR. | TR parameter is added to the default list of parameters. |
| | DB=ID | Same as DB=B/DL/ID. | ID parameter is added to the default list of parameters. |

†The zero turns off all previously specified values. For example, DB=0/TR turns off default values B and DL and turns on TR.

TABLE 12-4. PROGRAM EXECUTION PARAMETERS

| Parameter | Parameter Formats | Description | Remarks |
|---|---|---|---|
| ASCII Mode (AS) | omitted or AS=0 (zero) | Program runs in normal mode. Data files are presumed to be in normal, not ASCII mode (display code, not ASCII, characters). | |
| | AS | Program runs in ASCII mode. All character data is interpreted as ASCII, not display code. See appendix A. | |
| Execution Control (GO) | omitted | Compiled BASIC program executes without loading provided it was compiled-to-memory (i.e., no B parameter specified) and there were no compilation errors. When the B option (table 12-3) is specified, the compiled program does not execute. | Program can be executed despite compilation errors. See DB parameter table 12-3. |
| | GO | Compiled BASIC program executes provided there were no compilation errors. | See DB parameter in table 12-3. |
| | GO=0 (zero) | Inhibits execution. Neither the compile-to-memory version nor the relocatable binary version of the BASIC program executes. | |
| Execution-Time Input File (J) | omitted or J | Default input file for compiled BASIC program (file read when INPUT statement is executed) is INPUT. | |
| | J=lfn | Default input file for compiled BASIC program is lfn. | |
| | J=0 (zero) | No default run-time input file. | Use of the INPUT statement aborts the executing BASIC program. |
| Execution-Time Print File (K) | omitted or K | Default output file for compiled BASIC program (file used for PRINT statement and run-time error diagnostics) is OUTPUT. | J and K options control the default input and output files of the compiled program because BASIC does not provide a means of controlling file assignment for the simple form of the PRINT and INPUT statements; also, the normal mode of operation, compile-to-memory and execute-in-one-step option, prohibits file assignments from being manipulated by intervening loader control statements. When loading and executing a program from relocatable binaries, parameters can be used to change the names of the J and K files, such as when relocatable binaries have been written on file LGO:<br><br>LGO, FILEIN, FILEOUT.<br><br>This causes the program to be loaded and executed with INPUT data from FILEIN and output PRINT data on FILEOUT. |
| | K=lfn | Default output file for the compiled BASIC program is lfn. | |

TABLE 12-4. PROGRAM EXECUTION PARAMETERS (Contd)

| Parameter | Parameter Formats | Description | Remarks |
|---|---|---|---|
| Debug and Trace (DB) | | Activates trace feature and forces execution regardless of compilation errors. See table 12-3. | |
| Print Density Control (PD) | | Controls density (lines per inch) of printed output. See table 12-1. | |

The REM LIST statement is effective only when the source list option (L parameter or L and LO parameters) is activated via the BASIC control statement. If the source list option is off when a REM LIST statement is encountered, REM LIST is treated like a REM statement (a comment). Figure 12-8 illustrates the REM LIST statement.

## BATCH PROCESSING FROM A TERMINAL

BASIC programs can be created at a terminal and submitted for batch processing. This is accomplished by setting up the program in a Text Editor file that includes control statements.

### NOS

Figure 12-9 shows an example of a terminal session where a job is created and submitted for batch processing.

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 (converts /EOR and /EOF found in this deck to end-of-record and end-of-file, respectively).

See the Network Products Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the NOS Time-Sharing User's reference manual (Reformatting Submit File) for remaining directive descriptions. The options of the BASIC control statement are available to the interactive user when using the batch subsystem.

### NOS/BE

To send a batch job to NOS/BE from a remote terminal, first enter EDITOR, as described in Terminal Operation under NOS/BE. You can then issue the CREATE command to construct the program statements to be processed. When using CREATE, a FORMAT command need not be specified, but if one is, the format cannot be BASIC.

```
For the program:


10 LET J=10
20 REM LIST,NONE
30 PRINT J
40 LET S=J*.07
50 REM LIST,ALL
60 PRINT S
70 END


the compiler-generated source listing is:


1      DONE              BASIC 3.5 81208     81/08/13. 16.27.26.   PAGE  1
0
          10 LET J=10
          20 REM LIST,NONE
          50 REM LIST,ALL
          60 PRINT S
          70 END
```

Figure 12-8. REM LIST Statement Example

```
/batch
RFL,0.
/new,guide
/100 /job
110 bin014g.
120 user,xxxxxxx,xxx.
130 charge,xxxx,xxxxxxx.
150 basic.
151 dayfile,prog.
152 replace,prog.
153 exit.
154 dayfile,prog.
155 replace,prog.
160 /eor
170 /noseq
180 let a=304
190 let b=403
200 let t=a*b
210 print t
220 end
250 /eof
submit,guide,b
 10.01.57. SUBMIT COMPLETE.  JOBNAME IS ACLIBCP
/enquire,jn=bcp
 ACLIBCP  IN INPUT QUEUE.
```

Figure 12-9. Batch Processing From a Terminal Under NOS

The job must include the NOS/BE control statements, along with the BASIC program. Each control statement and BASIC statement is entered on a separate line. A line with *EOR indicates the place in the deck where an end-of-record mark is to be inserted; when the EDITOR command SAVE is issued, the *EOR is transformed into an actual end-of-record mark. A typical deck setup is shown in figure 12-10.

When entering BASIC statements (under a format other than BASIC), the EDITOR sequence numbers are distinct from the BASIC line numbers and must be specified separately. In figure 12-10, 610 is the EDITOR sequence statement number generated by the system, and 100 is the BASIC line number input from the terminal. Once this is accomplished, the file can be modified by using EDITOR commands and can be saved by using the SAVE,lfn,NOSEQ form of the SAVE command.

To submit a batch job created under EDITOR, save the edit file without sequence numbers, then submit the saved file to the batch input queue by using the BATCH or ROUTE command. The following are two types of processes for submitting a job into file TESTJOB for batch execution; results are automatically printed at the central site.

```
    SAVE,TESTJOB,NOSEQ
    BATCH,TESTJOB,INPUT
or
    SAVE,TESTJOB,NOSEQ,
    ROUTE,TESTJOB,DC=IN.
```

Optionally, the job can be submitted for batch processing with the results directed to the submitting terminal for inspection. If acceptable, the job can be printed at the central site. Figure 12-11 shows an example of printing a batch job.

Refer to the INTERCOM Version 5 reference manual for additional details and examples concerning these commands.

```
COMMAND-editor
..create <------------------------------ Creates the file.
100=job statement.<---------------------- Control statements (lines 100-600).
    .
    .
    .
500=basic.
    .
    .
    .
600=*eor
610=100 input x<------------------------- BASIC statements (lines 610-700).
620=110 if x=0 then 190
    .
    .
    .
680=170 print "factorial";x;"is";f
690=180 goto 110
700=190 end
710=*eor <------------------------------- End of BASIC source record; optional if no succeeding information.
= <-------------------------------------- End CREATE mode.
..s,testjob,ns<-------------------------- Saves job in file named TESTJOB with no sequence line numbers.
```

Figure 12-10. Batch Processing From a Terminal Under NOS/BE

| | |
|---|---|
| BATCH,TESTJOB,INPUT,HERE | Submits the job. |
| Allow time for batch job to complete. | |
| FILES | Lists file names so you can identify remote output file lfn created by the job. |
| BATCH,lfn†,LOCAL | Makes remote output file local to terminal. |
| PAGE,lfn†,L | Prepares to display contents of file lfn. The L is optional to display ASCII coded file. |
| When prompted with READY . . . enter 1 to see the first page and + to see each additional page. Enter E or END to exit from the PAGE mode. | |
| BATCH,lfn,PRINT,id | Submits the file to the batch print queue with user identification id. |

†The remote output file name consists of the first five characters from the job statement (job name) and two characters generated by the system.

Figure 12-11. Printing a Batch Job

Each computer has its own character set, which includes a collection of graphics (letters, digits, and special symbols) that the computer recognizes. Associated with each graphic of a character set is a number called a code. The code represents the character within the computer. Computers differ in their codes as well as their graphic sets, so in order to permit intercomputer communication, the American Standard Code for Information Interchange (ASCII) has been established. The ASCII character set includes all letters (uppercase and lowercase), digits, and many special symbols. The characters used in BASIC are taken from the ASCII character set. Table A-1 lists the full ASCII character set.

Any one of several character sets can be used on CDC CYBER and the 6000 Series computers. These character sets include CDC 63- and 64-character sets, ASCII 63- and 64-character sets, and ASCII 128-character set. Differences in character sets occur in either graphics (CDC or ASCII), or the number of characters (63 or 64 and normal or 128-ASCII). Graphic differences are a function of the terminal or printer that is used; some devices use CDC symbols; others use ASCII symbols. These differences do not affect the BASIC program provided the programmer realizes that all BASIC characters are defined in terms of ASCII codes. When using CDC character set devices, the programmer must use CDC symbols equivalent (same internal code) to ASCII symbols required by BASIC. Table A-2 lists ASCII and CDC character sets so that equivalent symbols can be easily determined; for example, the ASCII # is equivalent to the CDC ≡ .

Differences in internal representations of characters can have an effect on programs and program results. Differences in program results can only occur when a program is developed on a 63-character set system, then run on a 64-character set system, or vice versa, and the program uses the OPTION COLLATE NATIVE statement and the normal (not ASCII) character set. Normally, however, there is no problem because computer systems operate either with a 63- or 64-character set system and do not switch between character sets. Even if one of the character set disparities exists, there is a problem only if the program uses string data that contain the characters % (percent) and : (colon). The character % is not available in a 63-character set system; the character : has a numeric code 0 in a 64-character system, but a numeric code 51 ($63_8$) in a 63-character set system. As a result, for a program in normal mode that uses the OPTION COLLATE NATIVE statement, the relation ":BCD"<"ABCD" is true in a 64-character set system, but false in a 63-character system.

## CHARACTER USAGE RESTRICTIONS

If operating with the 64-character set in normal mode, it is advisable to restrict the use of the colon, as follows:

Never use the colon at the end of a string.

Do not use multiple colons (:::) because they could be interpreted as the end of the string or end of the line.

Do not use a colon at the end of a line or on a line by itself.

Do not use :A, :B, :D, :E, or :H at the end of a terminal output (PRINT) line because it is interpreted by the operating system as a terminal command. Other colon and letter combinations, :L, :F, :G, and :I, could be misinterpreted depending on the carriage control at the beginning of a terminal PRINT line.

## NOS ASCII 128-CHARACTER SET

NOS enables the BASIC programmer at an ASCII code terminal to make use of the ASCII 128-character set. (See table A-1.) This character set, which includes lowercase letters, special symbols, and device control characters, is only available when the user's terminal and program are in ASCII mode. A terminal is switched into ASCII mode by entering the ASCII command. The terminal is returned to normal mode by entering the NORMAL command. (Refer to the Network Products Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS 2 reference set (NOS 2 sites), or the NOS Time-Sharing User's reference manual). Under the BASIC subsystem, the BASIC compiler and the BASIC program automatically operate in ASCII mode when the terminal is in ASCII mode. In order for the compiler and program to handle the ASCII 128-character set in batch mode, the AS parameter must be explicitly specified.

In order to provide 128 characters, some characters must be represented as 12-bit instead of 6-bit characters. The 6-bit characters are distinguished from the 12-bit characters by using the 6-bit codes $74_8$ and $76_8$ as escape codes to indicate that the next six bits are actually part of this 12-bit character. This coding method is referred to as the 6/12 or extended display code. When operating in ASCII mode, the BASIC compiler assumes that all data files contain 128-ASCII characters, so display code $74_8$ and $76_8$ are interpreted as escape code characters; they are never characters by themselves.

## NOS/BE ASCII 128-CHARACTER SET

NOS/BE provides an ASCII 128-character set, as listed in table A-1. This character set includes the symbols, uppercase and lowercase letters, and control characters that are available when the terminal operates in ASCII mode.

In NOS/BE, the terminal is switched to ASCII mode only when directed from within the user program. There is no ASCII command available under NOS/BE. To specify ASCII mode, include the AS parameter option in the BASIC control statement. Once the program switches the terminal to ASCII mode, the ASCII mode remains in effect until the program terminates.

In order to provide 128 characters, each character is represented by 12 bits (the eight rightmost bits in a 12-bit byte). If a BASIC program is run in ASCII mode, all associated data files must be in 8/12 ASCII code. NOS/BE does not use the 6/12 display code. BASIC converts all 8/12 characters to 6/12 characters so that only 6/12 characters are available internally. On output, BASIC converts the 6/12 characters back to 8/12 characters.

TABLE A-1. EXTENDED CHARACTER SETS

| BASIC Character | BASIC Character Abbreviation | BASIC Decimal Code† | Display Code (6/12-Bit Octal) | ASCII Code (7-Bit Octal) | ASCII Code (7-Bit Hexadecimal) | BASIC Character | BASIC Character Abbreviation | BASIC Decimal Code† | Display Code (6/12-Bit Octal) | ASCII Code (7-Bit Octal) | ASCII Code (7-Bit Hexadecimal) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 00†† | null | | c | LCC | 99 | 7603 | 143 | 63 |
| A | UCA | 65 | 01 | 101 | 41 | d | LCD | 100 | 7604 | 144 | 64 |
| B | UCB | 66 | 02 | 102 | 42 | e | LCE | 101 | 7605 | 145 | 65 |
| C | UCC | 67 | 03 | 103 | 43 | f | LCF | 102 | 7606 | 146 | 66 |
| D | UCD | 68 | 04 | 104 | 44 | g | LCG | 103 | 7607 | 147 | 67 |
| E | UCE | 69 | 05 | 105 | 45 | h | LCH | 104 | 7610 | 150 | 68 |
| F | UCF | 70 | 06 | 106 | 46 | i | LCI | 105 | 7611 | 151 | 69 |
| G | UCG | 71 | 07 | 107 | 47 | j | LCJ | 106 | 7612 | 152 | 6A |
| H | UCH | 72 | 10 | 110 | 48 | k | LCK | 107 | 7613 | 153 | 6B |
| I | UCI | 73 | 11 | 111 | 49 | l | LCL | 108 | 7614 | 154 | 6C |
| J | UCJ | 74 | 12 | 112 | 4A | m | LCM | 109 | 7615 | 155 | 6D |
| K | UCK | 75 | 13 | 113 | 4B | n | LCN | 110 | 7616 | 156 | 6E |
| L | UCL | 76 | 14 | 114 | 4C | o | LCO | 111 | 7617 | 157 | 6F |
| M | UCM | 77 | 15 | 115 | 4D | p | LCP | 112 | 7620 | 160 | 70 |
| N | UCN | 78 | 16 | 116 | 4E | q | LCQ | 113 | 7621 | 161 | 71 |
| O | UCO | 79 | 17 | 117 | 4F | r | LCR | 114 | 7622 | 162 | 72 |
| P | UCP | 80 | 20 | 120 | 50 | s | LCS | 115 | 7623 | 163 | 73 |
| Q | UCQ | 81 | 21 | 121 | 51 | t | LCT | 116 | 7624 | 164 | 74 |
| R | UCR | 82 | 22 | 122 | 52 | u | LCU | 117 | 7625 | 165 | 75 |
| S | UCS | 83 | 23 | 123 | 53 | v | LCV | 118 | 7626 | 166 | 76 |
| T | UCT | 84 | 24 | 124 | 54 | w | LCW | 119 | 7627 | 167 | 77 |
| U | UCU | 85 | 25 | 125 | 55 | x | LCX | 120 | 7630 | 170 | 78 |
| V | UCV | 86 | 26 | 126 | 56 | y | LCY | 121 | 7631 | 171 | 79 |
| W | UCW | 87 | 27 | 127 | 57 | z | LCZ | 122 | 7632 | 172 | 7A |
| X | UCX | 88 | 30 | 130 | 58 | { | LBR | 123 | 7633 | 173 | 7B |
| Y | UCY | 89 | 31 | 131 | 59 | : | VLN | 124 | 7634 | 174 | 7C |
| Z | UCZ | 90 | 32 | 132 | 5A | } | RBR | 125 | 7635 | 175 | 7D |
| 0 | | 48 | 33 | 060 | 30 | ~ | TIL | 126 | 7636 | 176 | 7E |
| 1 | | 49 | 34 | 061 | 31 | DEL | DEL | 127 | 7637 | 177 | 7F |
| 2 | | 50 | 35 | 062 | 32 | NUL††† | | 0 | 7640 | 000 | 00 |
| 3 | | 51 | 36 | 063 | 33 | SOH | | 1 | 7641 | 001 | 01 |
| 4 | | 52 | 37 | 064 | 34 | STX | | 2 | 7642 | 002 | 02 |
| 5 | | 53 | 40 | 065 | 35 | ETX | | 3 | 7643 | 003 | 03 |
| 6 | | 54 | 41 | 066 | 36 | EOT | | 4 | 7644 | 004 | 04 |
| 7 | | 55 | 42 | 067 | 37 | ENQ | | 5 | 7645 | 005 | 05 |
| 8 | | 56 | 43 | 070 | 38 | ACK | | 6 | 7646 | 006 | 06 |
| 9 | | 57 | 44 | 071 | 39 | BEL | | 7 | 7647 | 007 | 07 |
| + | | 43 | 45 | 053 | 2B | BS | | 8 | 7650 | 010 | 08 |
| - | | 45 | 46 | 055 | 2D | HT | | 9 | 7651 | 011 | 09 |
| * | | 42 | 47 | 052 | 2A | LF | | 10 | 7652 | 012 | 0A |
| / | | 47 | 50 | 057 | 2F | VT | | 11 | 7653 | 013 | 0B |
| ( | | 40 | 51 | 050 | 28 | FF | | 12 | 7654 | 014 | 0C |
| ) | | 41 | 52 | 051 | 29 | CR | | 13 | 7655 | 015 | 0D |
| $ | | 36 | 53 | 044 | 24 | SO | | 14 | 7656 | 016 | 0E |
| = | | 61 | 54 | 075 | 3D | SI | | 15 | 7657 | 017 | 0F |
| SP (space) | | 32 | 55 | 040 | 20 | DLE | | 16 | 7660 | 020 | 10 |
| , | | 44 | 56 | 054 | 2C | DC1 | | 17 | 7661 | 021 | 11 |
| . | | 46 | 57 | 056 | 2E | DC2 | | 18 | 7662 | 022 | 12 |
| # | | 35 | 60 | 043 | 23 | DC3 | | 19 | 7663 | 023 | 13 |
| [ | | 91 | 61 | 133 | 5B | DC4 | | 20 | 7664 | 024 | 14 |
| ] | | 93 | 62 | 135 | 5D | NAK | | 21 | 7665 | 025 | 15 |
| %§§ | | 37 | 63 | 045 | 25 | SYN | | 22 | 7666 | 026 | 16 |
| " (quote) | QUO | 34 | 64 | 042 | 22 | ETB | | 23 | 7667 | 027 | 17 |
| _ (underline)§ | UND | 95 | 65 | 137 | 5F | CAN | | 24 | 7670 | 030 | 18 |
| ! | | 33 | 66 | 041 | 21 | EM | | 25 | 7671 | 031 | 19 |
| & | | 38 | 67 | 046 | 26 | SUB | | 26 | 7672 | 032 | 1A |
| ' (apostrophe) | | 39 | 70 | 047 | 27 | ESC | | 27 | 7673 | 033 | 1B |
| ? | | 63 | 71 | 077 | 3F | FS | | 28 | 7674 | 034 | 1C |
| < | | 60 | 72 | 074 | 3C | GS | | 29 | 7675 | 035 | 1D |
| > | | 62 | 73 | 076 | 3E | RS | | 30 | 7676 | 036 | 1E |
| | | | 74 (escape code) | | | US | | 31 | 7677 | 037 | 1F |
| | | | | | | | | | 7400 | - | - |
| \ | | 92 | 75 | 134 | 5C | @ | | 64 | 7401 | 100 | 40 |
| | | | 76 (escape code) | | | ^(circumflex) | | 94 | 7402 | 136 | 5E |
| | | | | | | | | | 7403 | - | - |
| ; (semicolon) | | 59 | 77 | 073 | 3B | :§§ | | 58 | 7404 | 072 | 3A |
| | | | 7600 | - | - | | | | 7405 | - | - |
| a | LCA | 97 | 7601 | 141 | 61 | | | | 7406 | - | - |
| b | LCB | 98 | 7602 | 142 | 62 | \ | GRA | 96 | 7407 | 140 | 60 |

†These codes are the decimal equivalent of the 7-bit octal ASCII codes. They are returned by the ORD function, used by the CHR$ function and used for comparing strings when the standard collating sequence is in effect (regardless of the character set used) and when the native collating sequence is in effect and the ASCII character set is being used. (See AS parameter or BASIC control statement.)

††Twelve zero bits at the end of a 60-bit word are an end-of-line or end-of-record mark rather than two colons. Colons at the end of lines or strings are considered part of the end-of-line or end-of-string marker. In the 63-character set, this display code represents a null character.

†††Those characters which are not included in the NOS/BE 95-character set are shaded.

§On TTY models having no underline, the backarrow (←) takes its place.

§§ In a 63-character set the internal octal representation for colon (:) is $63_8$, and the internal octal representation for percent (%) is $7404_8$. (The characters reverse posiitons.)

TABLE A-2. CDC AND ASCII 63- AND 64-CHARACTER SETS

| BASIC | | | Display Code (Octal) | CDC | | | ASCII | | |
|---|---|---|---|---|---|---|---|---|---|
| Character | Character Abbreviation† | Decimal Code †† | | Graphic | Hollerith Punch (026) | External BCD Code | Graphic Subset | Punch (029) | Code (Octal) |
| : (colon)††† | | 58 | 00§ | : (colon)††† | 8-2 | 00 | : (colon)††† | 8-2 | 072 |
| A | UCA | 65 | 01 | A | 12-1 | 61 | A | 12-1 | 101 |
| B | UCB | 66 | 02 | B | 12-2 | 62 | B | 12-2 | 102 |
| C | UCC | 67 | 03 | C | 12-3 | 63 | C | 12-3 | 103 |
| D | UCD | 68 | 04 | D | 12-4 | 64 | D | 12-4 | 104 |
| E | UCE | 69 | 05 | E | 12-5 | 65 | E | 12-5 | 105 |
| F | UCF | 70 | 06 | F | 12-6 | 66 | F | 12-6 | 106 |
| G | UCG | 71 | 07 | G | 12-7 | 67 | G | 12-7 | 107 |
| H | UCH | 72 | 10 | H | 12-8 | 70 | H | 12-8 | 110 |
| I | UCI | 73 | 11 | I | 12-9 | 71 | I | 12-9 | 111 |
| J | UCJ | 74 | 12 | J | 11-1 | 41 | J | 11-1 | 112 |
| K | UCK | 75 | 13 | K | 11-2 | 42 | K | 11-2 | 113 |
| L | UCL | 76 | 14 | L | 11-3 | 43 | L | 11-3 | 114 |
| M | UCM | 77 | 15 | M | 11-4 | 44 | M | 11-4 | 115 |
| N | UCN | 78 | 16 | N | 11-5 | 45 | N | 11-5 | 116 |
| O | UCO | 79 | 17 | O | 11-6 | 46 | O | 11-6 | 117 |
| P | UCP | 80 | 20 | P | 11-7 | 47 | P | 11-7 | 120 |
| Q | UCQ | 81 | 21 | Q | 11-8 | 50 | Q | 11-8 | 121 |
| R | UCR | 82 | 22 | R | 11-9 | 51 | R | 11-9 | 122 |
| S | UCS | 83 | 23 | S | 0-2 | 22 | S | 0-2 | 123 |
| T | UCT | 84 | 24 | T | 0-3 | 23 | T | 0-3 | 124 |
| U | UCU | 85 | 25 | U | 0-4 | 24 | U | 0-4 | 125 |
| V | UCV | 86 | 26 | V | 0-5 | 25 | V | 0-5 | 126 |
| W | UCW | 87 | 27 | W | 0-6 | 26 | W | 0-6 | 127 |
| X | UCX | 88 | 30 | X | 0-7 | 27 | X | 0-7 | 130 |
| Y | UCY | 89 | 31 | Y | 0-8 | 30 | Y | 0-8 | 131 |
| Z | UCZ | 90 | 32 | Z | 0-9 | 31 | Z | 0-9 | 132 |
| 0 | | 48 | 33 | 0 | 0 | 12 | 0 | 0 | 060 |
| 1 | | 49 | 34 | 1 | 1 | 01 | 1 | 1 | 061 |
| 2 | | 50 | 35 | 2 | 2 | 02 | 2 | 2 | 062 |
| 3 | | 51 | 36 | 3 | 3 | 03 | 3 | 3 | 063 |
| 4 | | 52 | 37 | 4 | 4 | 04 | 4 | 4 | 064 |
| 5 | | 53 | 40 | 5 | 5 | 05 | 5 | 5 | 065 |
| 6 | | 54 | 41 | 6 | 6 | 06 | 6 | 6 | 066 |
| 7 | | 55 | 42 | 7 | 7 | 07 | 7 | 7 | 067 |
| 8 | | 56 | 43 | 8 | 8 | 10 | 8 | 8 | 070 |
| 9 | | 57 | 44 | 9 | 9 | 11 | 9 | 9 | 071 |
| + | | 43 | 45 | + | 12 | 60 | + | 12-8-6 | 053 |
| - | | 45 | 46 | - | 11 | 40 | - | 11 | 055 |
| * | | 42 | 47 | * | 11-8-4 | 54 | * | 11-8-4 | 052 |
| / | | 47 | 50 | / | 0-1 | 21 | / | 0-1 | 057 |
| ( | | 40 | 51 | ( | 0-8-4 | 34 | ( | 12-8-5 | 050 |
| ) | | 41 | 52 | ) | 12-8-4 | 74 | ) | 11-8-5 | 051 |
| $ | | 36 | 53 | $ | 11-8-3 | 53 | $ | 11-8-3 | 044 |
| = | | 61 | 54 | = | 8-3 | 13 | = | 8-6 | 075 |
| SP (space) | | 32 | 55 | blank | no punch | 20 | blank | no punch | 040 |
| , (comma) | | 44 | 56 | , (comma) | 0-8-3 | 33 | , (comma) | 0-8-3 | 054 |
| . (period) | | 46 | 57 | . (period) | 12-8-3 | 73 | . (period) | 12-8-3 | 056 |
| # | | 35 | 60 | ≡ | 0-8-6 | 36 | # | 8-3 | 043 |
| [ | | 91 | 61 | [ | 8-7 | 17 | [ | 12-8-2 | 133 |
| ] | | 93 | 62 | ] | 0-8-2 | 32 | ] | 11-8-2 | 135 |
| %††† | | 37 | 63††† | %††† | 8-6 | 16 | %††† | 0-8-4 | 045 |
| " (quote) | QUO | 34 | 64 | ≠ | 8-4 | 14 | " (quote) | 8-7 | 042 |
| __ (underline) | UND | 95 | 65 | ↱ | 0-8-5 | 35 | __ (underline) | 0-8-5 | 137 |
| ! | | 33 | 66 | ∨ | 11-0 | 52 | ! | 12-8-7 | 041 |
| & | | 38 | 67 | ∧ | 0-8-7 | 37 | & | 12 | 046 |
| ' (apostrophe) | | 39 | 70 | ↑ | 11-8-5 | 55 | ' (apostrophe) | 8-5 | 047 |
| ? | | 63 | 71 | ↓ | 11-8-6 | 56 | ? | 0-8-7 | 077 |
| < | | 60 | 72 | < | 12-0 | 72 | < | 12-8-4 | 074 |
| > | | 62 | 73 | > | 11-8-7 | 57 | > | 0-8-6 | 076 |
| @ | | 64 | 74 | ≤ | 8-5 | 15 | @ | 8-4 | 100 |
| \ | | 92 | 75 | ≥ | 12-8-5 | 75 | \ | 0-8-2 | 134 |
| ^(circumflex) | | 94 | 76 | ¬ | 12-8-6 | 76 | ^(circumflex) | 11-8-7 | 136 |
| ; (semicolon) | | 59 | 77 | ; (semicolon) | 12-8-7 | 77 | ; (semicolon) | 11-8-6 | 073 |

†The BASIC character abbreviation can be used only with the ORD function.

††These decimal codes are the values returned by the ORD function, used by the CHR$ function, and used for string comparison when the native collating sequence is in effect and the normal (not ASCII) character set is in use.

†††In installations using a 63 character set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch); the % character and related card code; do not exist and translations yield a blank ($55_8$).

§ Twelve zero bits at the end of a 60-bit word in a zero-byte record are an end-of-line or end-of-record mark rather than two colons.

BASIC produces three categories of diagnostic messages: dayfile messages, compile-time diagnostics, and execution-time diagnostics. These messages and diagnostics are listed in tables B-1 through B-4.

## DAYFILE MESSAGES

When a job is operating interactively, dayfile messages are displayed at the terminal. In contrast, dayfile messages for a batch job are appended to the output file for the job. Special control statements are required to access the dayfile of a job submitted by using the NOS command SUBMIT. (See section 12, Batch Operations.)

Dayfile messages are listed in table B-1. BASIC automatically increases its memory field length as required up to the maximum allowed; therefore, this maximum is the field length referred to in the dayfile messages.

## COMPILE-TIME DIAGNOSTICS

While compiling or translating a program into object code, BASIC checks the source code for such things as incorrect syntax, improper use of statements, and missing or illegal arguments. If any of these checks fail, the program (in most cases) compiles unsuccessfully and an error message, indicating the nature of the problem, is returned to the terminal from where the program originated. The messages that can be produced during program compilation are listed in table B-2. These messages are printed in the following format:

```
message AT line-number
```

With the following exceptions, all compile-time diagnostics listed in table B-2 inhibit program execution. The messages OBSOLETE FORM, LINES TRUNCATED AT 150 CHARACTERS, WARNING - FUNCTION REDEFINITION, and WARNING - FUNCTION REFERENCE BEFORE DEFINITION are warning types of diagnostics that do not inhibit program execution. The program that contains compilation errors can be forced to execute by specifying the DB=B parameter in the BASIC control statement.

## EXECUTION-TIME DIAGNOSTICS

BASIC allows two modes of execution-time error processing. During normal error processing, control is returned to the operating system. If the program has executed an ON ERROR statement, the program retains control. The program can then inspect the error number by use of the ESM function.

Errors 100, 106, and 115 can be recovered from only once. Should these errors occur a second time during the same execution period, the BASIC program aborts without transferring control to the ON ERROR address.

Execution-time diagnostics are listed in alphabetical order in table B-3. These messages are printed in the following format:

```
message AT line-number
```

For ease of reference, diagnostics are listed by error numbers in table B-4.

TABLE B-1. DAYFILE MESSAGES

| Message | Significance | Action |
|---|---|---|
| BAD CONTROL CARD ARGUMENT-parm | The specified control statement parameter or the parameter value is invalid. | Correct the parameter. |
| BASIC COMPILATION ERRORS | Indicates that errors occurred during compilation. | Correct the errors. |
| BASIC EXECUTION ERROR | An error has terminated program execution. | Correct the error. |
| INPUT FILE EMPTY OR MISPOSITIONED | Input file is empty or positioned at end-of-information. | Rewind the input file. |
| FIELD LENGTH TOO SHORT FOR BASIC | The maximum field length is too short to allow compilation. | Increase field length. |
| FL TOO SMALL FOR EXECUTION | The program compiled correctly but there was not enough assigned memory for execution. This condition is usually caused by excessive array dimensions. This message only occurs in compile-to-memory and execute mode. | Increase field length. |

TABLE B-2. COMPILE TIME DIAGNOSTICS

| Message | Significance | Action |
|---|---|---|
| BLANK FILE STATEMENT | File ordinal or name missing in a file statement. | Correct and rerun. |
| BLANK CLOSE STATEMENT | The CLOSE statement does not specify which file to close. | Correct and rerun. |
| DEF WITHIN DEF | A DEF statement occurs before the current multiple-line function definition is terminated by FNEND. | Move the DEF statement outside of the multiple-line function. |
| DELIMITER OVERFLOW | More than three characters are specified in the DELIMIT statement. | Specify three or fewer characters. |
| DUPLICATE LINE NO | The same line number was used twice. | Change one of the line numbers. |
| END NOT LAST | An END statement is placed prior to the last statement. | Remove the END statement and replace it with a STOP statement if necessary. |
| FL TOO SMALL FOR COMPILATION | The maximum field length allowed is too small to allow compilation. The more compilation options requested, the more memory required. The B option requires more memory than the L. | Increase field length. |
| FNEND MISSING | A multiple-line function is not terminated by FNEND before the end of the program. | Supply an FNEND statement. |
| FOR NESTED TOO DEEP | FOR statements are nested more than ten deep. | Rewrite so that no more than ten FOR statements are nested. |
| FOR WITHOUT NEXT | A FOR statement has no balancing NEXT statement. | Supply a NEXT statement. |
| ILLEGAL ARGUMENT IN ASC | The argument in an ASC function is not a character or a defined abbreviation for a character. | Replace the argument with a valid one. |
| ILLEGAL BOUND | An array bound declared in a DIM statement is < 0 or > 131070. If OPTION BASE 1 was specified, the array bound cannot be = 0. | Replace the array bound with a valid one. |
| ILLEGAL CHARACTER | BASIC encountered an unrecognizable character. | Replace the character with a valid one. |
| ILLEGAL COMPARISON | A numeric quantity was compared to a string in an IF statement. | Replace the comparison with a valid one. |
| ILLEGAL EXTERNAL NAME | A name in a CALL statement does not begin with a letter, or it is longer than seven characters. | Correct the name. |
| ILLEGAL FILE NAME | The specified name is not allowed as a file name. | Replace the file name with a valid one. |
| ILLEGAL FILE NUMBER | The number in a FILE statement is < 0 or > $(2^{18}-1)$. | Replace the file number with a valid one. |
| ILLEGAL FN NAME | The user function name is not in the form FNx or FNx$. | Correct the function name. |
| ILLEGAL LINE NO | A line number is > 99999. | Replace the line number with one ≤ 99999. |
| ILLEGAL LINE REF | Referenced line number is incorrectly written or > 99999. | Correct the line number reference. |

TABLE B-2. COMPILE TIME DIAGNOSTICS (Contd)

| Message | Significance | Action |
|---|---|---|
| ILLEGAL MARGIN | The margin specified in a MARGIN statement is < 0 or > 131070. | Specify the margin with a valid value. |
| ILLEGAL NUMBER | A numeric constant is incorrectly written. | Write the constant correctly. |
| ILLEGAL OPERAND | A string is used in an arithmetic expression. | Write the expression correctly. |
| ILLEGAL REDIMENSIONS | An array specified in a DIM statement has been dimensioned in a previous DIM statement, or a statement attempts to change the number of dimensions (subscripts) of an array. | Delete the duplicate DIM statements or use the proper number of subscripts. |
| ILLEGAL STATEMENT | A statement does not begin with a recognizable word or is written incorrectly. | Rewrite the statement. |
| ILLEGAL STATEMENT WITHIN IF | The statement is not allowed as an object of THEN or ELSE in an IF THEN ELSE statement. The object of THEN or ELSE must be executable. | Replace the invalid statement with a valid one. |
| ILLEGAL STRING | A string constant is incorrectly written. | Rewrite the string correctly. |
| ILLEGAL USE OF LEFT PAREN | An attempt was made to use an argument with a system function when none was required. | Remove the argument. |
| ILLEGAL USING | USING clause is not allowed where it is written or it is not allowed at all. | Correct the placement of the USING clause. |
| INVALID BASE STATEMENT | OPTION BASE statement appears after the DIM statement or array reference. | Place the OPTION BASE statement before the DIM statement or array reference. |
| INVALID BASE VALUE | Base value is not 0 or 1. | Correct the value in the OPTION BASE statement. |
| INVALID CHANGE | CHANGE statement arguments are other than string-expression TO one-dim-array or one-dim-array TO string-expression. | Replace this statement. It is no longer supported. |
| LINES TRUNCATED AT 150 CHARACTERS | Some lines are greater than 150 characters. Although lines were truncated, program compilation continued. | Shorten the lines. |
| LINES OUT OF ORDER | Line numbers are not in ascending order. | Renumber lines in ascending order. |
| MISSING LINE NO | A statement was written without a line number. | Rewrite the statement with a line number. |
| NEXT WITHOUT FOR | A NEXT statement has no balancing FOR statement. | Supply a FOR statement. |
| NON IMAGE REFERENCED | The line number referenced in the USING clause is not an image statement. | Change the line number to one that references an image statement. |
| NOT ENOUGH ARGUMENTS | The number of arguments in a function reference is less than the number expected by the function. | Reference the function with with the proper number of arguments. |

TABLE B-2. COMPILE TIME DIAGNOSTICS (Contd)

| Message | Significance | Action |
|---|---|---|
| OBSOLETE FORM | The statement form used is no longer supported; compilation continues. | Use proper statement form. |
| PARAMETER LIST CONFLICT | Too many or too few parameters for the function reference; a string is used where the function expects a number; or a number is used where the function expects a string. | Replace the invalid parameter list with a valid one. |
| READ WITHOUT DATA | The program contains a READ statement but no DATA statement. | Include DATA statements. |
| RECURSIVE FN | A user function calls itself. This is not allowed. | Eliminate the recursion. |
| REDEFINITION OF COLLATE | The program contains more than one OPTION COLLATE statement. | Remove the excessive OPTION COLLATE statement(s). |
| SET VALUE ILLEGAL | The value in the SET statement is specified as a string or is not specified at all. | Replace the invalid value with a valid one. |
| STATEMENT TOO COMPLEX | The statement or the expression is too long or complex. | Simplify the expression or break the statement into two or more simpler statements. |
| TOO MANY ARGUMENTS | The number of arguments in a function reference is greater than the number expected by the function. The number of arguments in a CALL statement is greater than 20. | Replace the argument list with one containing the proper number of arguments. |
| TOO MANY FILES | More than 13 FILE statements are in the program. | Use fewer FILE statements. |
| TOO MANY FORMALS | The DEF statement contains more than 20 formal parameters. | Rewrite the DEF statement with 20 or fewer parameters. |
| TRANSFER INTO DEF | The statement refers to a line that is part of a multiple-line function definition. | Change the statement reference. |
| TRANSFER OUT OF DEF | A statement within a multiple-line function definition refers to a line number not contained in the DEF... FNEND block. | Change the statement reference. |
| UNDEFINED IN REF | The user function referenced is undefined. | Refer to a defined function. |
| UNDEFINED LINE REF | The line number referenced does not exist. Several statements can reference the same nonexistent line; only the first reference is diagnosed. | Refer to a defined line number. |
| WARNING-DIM AFTER REFERENCE | The DIM statement for an array appears after the first reference to the array. | Move the DIM statement for the array before the first reference to the array. |
| WARNING - FUNCTION REDEFINITION | A user-defined function was redefined within the program; compilation continues. | Remove the affected function reference. |
| WARNING - FUNCTION REFERENCE BEFORE DEFINITION | A user-defined function was referenced before it was defined with DEF; compilation continues. | Move the function definition ahead of the function reference. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS

| Message | Error Number | Significance | Action |
|---|---|---|---|
| ARGUMENT IS POLE IN COT | 148 | The argument for the COT function is a multiple of $\pi$; therefore, the results are undefined. | Make sure the argument is not a multiple of $\pi$. |
| ARGUMENT IS POLE IN TAN | 153 | The argument for the TAN function is a multiple of $\pi/2$; therefore, the results are undefined. | Make sure the argument is not a multiple of $\pi/2$. |
| ARGUMENT NEGATIVE IN LOG | 154 | The argument for the LOG function is negative. | Make sure the argument is positive. |
| ARGUMENT NEGATIVE IN SQR | 160 | The argument for the SQR function is negative. | Make sure the argument is positive. |
| ARGUMENT TOO LARGE IN COS | 152 | The argument for the COS function must be less than 2.21069E14. | Make sure the argument is less than 2.21069E14. |
| ARGUMENT TOO LARGE IN COT | 149 | The argument for the COT function must be less than 2.21069E14. | Make sure the argument is less than 2.21069E14. |
| ARGUMENT TOO LARGE IN EXP | 156 | The argument for the EXP function must be less than 2.21069E14. | Make sure the argument is less than 2.21069E14. |
| ARGUMENT TOO LARGE IN SIN | 150 | The argument for the SIN function must be less than 2.21069E14. | Make sure the argument is less than 2.21069E14. |
| ARGUMENT TOO LARGE IN TAN | 151 | The argument for the TAN function must be less than 2.21069E14. | Make sure the argument is less than 2.21069E14. |
| ARGUMENT IS ZERO IN LOG | 155 | The argument for the LOG function is zero. | Make sure the argument is nonzero. |
| ARRAY TOO SMALL IN CHANGE | 163 | Array in the CHANGE statement is not large enough to hold the string length plus one word for each character of the string. | Replace this statement; it is no longer supported. |
| AUTO RECALL STATUS MISSING | 116 | Internal error. | Follow site procedures for reporting and resolving system problems. |
| BAD DATA IN READ | 126 | A string was read when a number was expected, or vice versa. | Correct the DATA statement. |
| BAD FORMAT FIELD | 127 | The current data conversion field in the image is for string data only, but the item to be printed is a number, or vice versa. | Correct the print image. |
| BAD TAB ARG - 1 USED | 197 | A TAB function was issued that contained a bad argument. A tab of 1 (col 1) was assigned. Execution continues. | Change the TAB setting, or take no action. |
| CHAIN FILE NOT FOUND | 144 | The file referenced in CHAIN does not exist as a local or permanent file. | Check the spelling of the file name. |
| COMPILATION ERROR | 119 | The statement caused a compilation error; therefore, it cannot be executed. This error occurs only if the DB=B option is specified. | Correct the statement. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS (Contd)

| Message | Error Number | Significance | Action |
|---|---|---|---|
| CPU ERROR EXIT 00 | 107 | An illegal instruction was executed. Could result from an error in a FORTRAN or COMPASS subroutine. | Correct the subroutine. If there are no errors in the subroutine, follow site-defined procedures for reporting software errors or operational problems. |
| CPU ERROR EXIT 01 | 108 | Address is out-of-range. Can result from an error in a FORTRAN or COMPASS subroutine. | Correct the subroutine. If there are no errors in the subroutine, follow site-defined procedures for reporting software errors or operational problems. |
| CPU ERROR EXIT 03 | 110 | Address is out-of-range, or infinite operand. | Correct the subroutine. If there are no errors in the subroutine, follow site-defined procedures for reporting software errors or operational problems. |
| CPU ERROR EXIT 05 | 112 | Indefinite operand or address is out-of-range. Could result after division of zero by zero if an ON ERROR was used to continue execution. Could result from an error in a FORTRAN or COMPASS subroutine that modified the parameters passed. | Correct the calculation that generated the faulty number or change ON ERROR code to correct the faulty variable before using it again, or correct the subroutine. |
| CPU ERROR EXIT 06 | 113 | Indefinite or infinite operand. Could result after division of zero by zero if an ON ERROR was used to continue execution. Could result from an error in a FORTRAN or COMPASS subroutine that modified the parameters passed. | Correct the calculation that generated the faulty number or change ON ERROR code to correct the faulty variable before using it again, or correct the subroutine. |
| CPU ERROR EXIT 07 | 114 | Address is out-of-range, or indefinite operand. Could result after division of zero by zero if an ON ERROR was used to continue execution. Could result from an error in a FORTRAN or COMPASS subroutine that modified the parameters passed. | Correct the calculation that generated the faulty number or change ON ERROR code to correct the faulty variable before using it again, or correct the subroutine. |
| DET USED BEFORE INV | 162 | DET without a parameter was called before a square numeric matrix was inverted by INV. | Before issuing DET, invert a matrix (with INV), or supply a parameter to DET. |
| DIVISION BY ZERO | 125 | An attempt was made to divide by zero. | Make sure no division by zero occurs. |
| ECS OR CY 170 PARITY ERROR | 101 | A hardware error occurred. | Follow site procedures for reporting and resolving system problems. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS (Contd)

| Message | Error Number | Significance | Action |
|---|---|---|---|
| END OF DATA | 120 | A READ statement was executed after the internal data block was exhausted. | Check for end-of-data, or supply more data. |
| END OF DATA ON FILE | 136 | A READ# or INPUT# statement was executed after file data was exhausted. | Check for end-of-data, or supply more data. |
| ERROR IN CHANGE | 164 | The length as specified in the first element of the array that is being changed to a string is greater than 131070, less than 0, or an element is not a valid character code. | Replace this statement; it is no longer supported. |
| FILE ALREADY OPEN | 143 | The file name specified in the FILE statement has been assigned a file number in a previous FILE statement and is still in use. | Close the file before attempting to open it again. |
| FILE CLOSED/UNDEFINED | 141 | The file number referenced does not correspond to an active file. | Check the file number or activate the file with a FILE statement. |
| FILE NUMBER ALREADY IN USE | 142 | The file number specified in the FILE statement is already assigned to an open, active file. | Specify an unused file number. |
| GOSUB NESTED TOO DEEP | 123 | More than 40 GOSUB statements are nested. | Nest 40 or fewer GOSUB statements. |
| HUNG IN AUTO RECALL | 117 | Internal system error. | Follow site procedures for reporting and resolving system problems. |
| ILLEGAL ACTION ON BINARY FILE | 175 | A DELIMIT, MARGIN, OR SETDIGITS was attempted on a binary file. | Do not attempt a DELIMIT, MARGIN, or SETDIGITS on a binary file. |
| ILLEGAL ACTION ON CODED FILE | 171 | A SET statement or LOC or LOF function was attempted on a coded file. | Do not attempt a SET, LOC or LOF on a coded file. |
| ILLEGAL CHAIN PARAMETER | 145 | A parameter in the CHAIN statement is incorrectly formed, or the referenced file is assigned or connected to the terminal. | Form the parameter correctly. |
| ILLEGAL CHARACTER | 165 | A string in a string comparison or a string that is referenced in a CHANGE statement contains an invalid character; usually caused by processing non-ASCII data in ASCII mode, or vice versa. | Eliminate the invalid character or change the mode. |
| ILLEGAL CHR$ ARGUMENT | 196 | Argument does not correspond to an ordinal in the collating sequence. | Correct the argument. |
| ILLEGAL DATA ON FILE | 135 | An illegal number or string was encountered when INPUT from a file was attempted; usually caused by reading a string when a number was expected. | Check data on the file. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS (Contd)

| Message | Error Number | Significance | Action |
|---|---|---|---|
| ILLEGAL DATA, RETYPE INPUT | 133 | An improperly formed number or string was entered; usually caused by entering a string when a number was expected. | Reenter the entire line. |
| ILLEGAL FILE NAME | 139 | The file name is not allowed as a NOS file name. | Choose another name. |
| ILLEGAL FILE NUMBER | 138 | The file number referenced is less than zero or is greater than 131071. | Use a file number within the proper range. |
| ILLEGAL INPUT ON FILE | 137 | The input operation, READ or INPUT, is not valid for the current mode of the file (READ on a coded file, INPUT on a binary file, READ or INPUT on an output file). | Use the RESTORE statement to permit change of mode. |
| ILLEGAL LABEL | 170 | The label referenced in a JUMP statement or NXL function does not exist, is greater than 99999, or is the label of a REM statement. | Correct the label. |
| ILLEGAL LPAD$ ARGUMENT | 192 | The LPAD$ numeric argument is negative, indefinite, or infinite. | Correct the argument. |
| ILLEGAL MARGIN | 131 | Margin specified is outside the allowable range of 0 through 131070. | Specify the margin within the range of 0 through 131070. |
| ILLEGAL ORD ARGUMENT | 194 | The value of the ORD argument is neither a valid character nor a valid character mnemonic for characters in the collating sequence. | Correct the argument. |
| ILLEGAL OUTPUT ON FILE | 130 | The output operation, PRINT or WRITE, is not valid for the current mode of the file (WRITE on a coded file, PRINT on a binary file, PRINT or WRITE on an input file). An attempt to WRITE or PRINT on a read-only permanent file causes this error. | Restore the file to change mode. |
| ILLEGAL RPAD$ ARGUMENT | 193 | The RPAD$ numeric argument is negative, indefinite, or infinite. | Correct the argument. |
| ILLEGAL RPT$ PARAMETER | 191 | The RPT$ parameter is negative, indefinite, or infinite. | Correct the parameter. |
| ILLEGAL SET VALUE | 172 | The SET value is negative, indefinite, or infinite. | Correct the parameter. |
| ILLEGAL SUBSTR PARAMETER | 169 | Parameters specified in the SUBSTR function are outside the legal range as determined by the actual string length. | Specify parameters within the allowable range. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS (Contd)

| Message | Error Number | Significance | Action |
|---|---|---|---|
| INDEFINITE OPERAND | 111 | An indefinite floating-point value was used in a calculation. Could result after division of zero by zero if an ON ERROR was used to continue execution. Could result from an error in a FORTRAN or COMPASS subroutine that modified the parameters passed. | Correct the calculation that generated the faulty number; change ON ERROR code to correct the faulty variable before using it again; or correct the subroutine. |
| INFINITE OPERAND | 109 | An invalid floating-point number was used in a calculation. Could result from division by zero if ON ERROR was used to continue. Could result from an error in a FORTRAN or COMPASS subroutine that modified the parameters passed. | Correct the calculation that generated the faulty number; change ON ERROR code to correct the faulty variable before using it again; or correct the faulty subroutine. |
| INPUT WITHIN INPUT | 195 | INPUT statement includes a function reference that attempts to execute another INPUT statement. No diagnostic is returned if the second reference INPUT is in another file. | Eliminate one of the INPUT statements. |
| I/O TIME LIMIT | 106 | Time limit exceeded. | Increase the time limit. |
| MASS STORAGE LIMIT | 118 | Mass storage limit exceeded. | Increase the mass storage limit. |
| MATRIX DIMENSION ERROR | 161 | Dimension inconsistency in one of the MAT statements or the dimension is greater than 100 times 100 in the INV function. | Correct the dimensioning error. |
| MEMORY OVERFLOW | 166 | Field length exceeded. | More field length needed. |
| NEGATIVE NUMBER TO POWER | 158 | An attempt was made to raise a negative number to a noninteger exponent. | Correct the error. |
| NO FILE SPACE. ADD ANOTHER FILE STMT | 140 | All declared fill buffers are used. | Add another FILE statement or CLOSE a file. |
| NO FORMAT FIELD SPECIFIED | 128 | The print image does not contain a data conversion field but the print list specifies that data is to be printed. | Rewrite the print image to include a data conversion field. |
| NONNUMERIC STRING | 167 | The string in the VAL function is nonnumeric. | Make the string numeric. |
| NOT ENOUGH DATA, REENTER OR TYPE IN MORE | 134 | Not enough data was entered in response to an input request. | Either reenter the entire input line or enter a delimiter followed by the additional data items. |
| XXX NOT IN PPLIB | 103 | System software malfunction. | Follow site procedures for reporting and resolving system problems. |

TABLE B-3. EXECUTION TIME DIAGNOSTICS (Contd)

| Message | Error Number | Significance | Action |
|---|---|---|---|
| ON EXPRESSION OUT OF RANGE | 122 | The expression in the ON statement is negative, zero, or exceeds the count of line numbers. | Make sure the expression is valid. |
| OPERATOR DROP OR KILL | 105 | The operator dropped or killed the program. | None. |
| OPERATOR RERUN | 115 | The operator reran the program. | None. |
| POWER TOO LARGE | 159 | The exponent in an expression is such that an overflow occurs. | Use a smaller exponent. |
| PPU ABORT | 102 | A PPU abort occurred. The program was terminated by an operating system-detected error. | Follow site procedures for reporting and resolving system problems. |
| PP CALL ERROR | 104 | Internal system error. | Follow site-defined procedures for reporting software errors or operational problems. |
| RANDOM ACTION BEYOND EOF | 174 | The SET value is greater than LOF or a WRITE operation on a random file attempted to extend the file length. | Correct the error. |
| RANDOM FILE EMPTY | 173 | A SET was attempted on an empty file. | Correct the error. |
| RETURN BEFORE GOSUB | 124 | A RETURN statement was encountered with no GOSUB in effect. | Add a GOSUB or remove the RETURN. |
| STRING OVERFLOW | 168 | An attempt was made to create a string that contains more than 131070 (6-bit) characters. | Use two or more strings that are shorter than the limit. |
| SUBSCRIPT ERROR | 121 | An attempt was made to reference an element outside the bounds of an array. | Use a correct subscript value or specify a larger array with a DIM statement. |
| TAPE FILE IS NOT ALLOWED | 147 | An attempt was made to use a tape file. | Use mass storage for the file. Copy an existing tape file to mass storage before using with BASIC. |
| TIME LIMIT EXCEEDED | 100 | The program time limit was exceeded. | Increase the time limit. Check the program for a nonending loop. |
| TOO MUCH DATA, RETYPE INPUT | 132 | Too many data items were entered in response to an input request. All items entered on the last type-in are ignored. | Reenter the entire input line. The exact number of items requested should be entered. |
| UNSATISFIED EXTERNAL REFERENCE | 129 | An attempt was made to execute a CALL statement in compile-to-memory mode. | Use the B and GO options on the BASIC control statement. |
| ZERO TO A NEGATIVE POWER | 157 | Exponent in an expression is negative when the mantissa is zero. | Correct the error. |

TABLE B-4. EXECUTION TIME DIAGNOSTICS BY ERROR NUMBER

| Error Number | Message |
|---|---|
| 100 | TIME LIMIT EXCEEDED |
| 101 | ECS OR CY 170 PARITY ERROR |
| 102 | PPU ABORT |
| 103 | xx NOT IN PPLIB |
| 104 | PP CALL ERROR |
| 105 | OPERATOR DROP OR KILL |
| 106 | I/O TIME LIMIT |
| 107 | CPU ERROR EXIT 00 |
| 108 | CPU ERROR EXIT 01 |
| 109 | INFINITE OPERAND |
| 110 | CPU ERROR EXIT 03 |
| 111 | INDEFINITE OPERAND |
| 112 | CPU ERROR EXIT 05 |
| 113 | CPU ERROR EXIT 06 |
| 114 | CPU ERROR EXIT 07 |
| 115 | OPERATOR RERUN |
| 116 | AUTO RECALL STATUS MISSING |
| 117 | HUNG IN AUTO RECALL |
| 118 | MASS STORAGE LIMIT |
| 119 | COMPILATION ERROR |
| 120 | END OF DATA |
| 121 | SUBSCRIPT ERROR |
| 122 | ON EXPRESSION OUT OF RANGE |
| 123 | GOSUB NESTED TOO DEEP |
| 124 | RETURN BEFORE GOSUB |
| 125 | DIVISION BY ZERO |
| 126 | BAD DATA IN READ |
| 127 | BAD FORMAT FIELD |
| 128 | NO FORMAT FIELD SPECIFIED |
| 129 | UNSATISFIED EXTERNAL REFERENCE |
| 130 | ILLEGAL OUTPUT ON FILE |
| 131 | ILLEGAL MARGIN |
| 132 | TOO MUCH DATA, RETYPE INPUT |
| 133 | ILLEGAL DATA, RETYPE INPUT |
| 134 | NOT ENOUGH DATA, REENTER OR TYPE IN MORE |
| 135 | ILLEGAL DATA ON FILE |
| 136 | END OF DATA ON FILE |
| 137 | ILLEGAL INPUT ON FILE |
| 138 | ILLEGAL FILE NUMBER |
| 139 | ILLEGAL FILE NAME |
| 140 | NO FILE SPACE. ADD ANOTHER FILE STMT |
| 141 | FILE CLOSED/UNDEFINED |
| 142 | FILE NUMBER ALREADY IN USE |
| 143 | FILE ALREADY OPEN |
| 144 | CHAIN FILE NOT FOUND |
| 145 | ILLEGAL CHAIN PARAMETER |
| 147 | TAPE FILE IS NOT ALLOWED |
| 148 | ARGUMENT IS POLE IN COT |
| 149 | ARGUMENT TOO LARGE IN COT |
| 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 EXP |
| 157 | ZERO TO A NEGATIVE POWER |
| 158 | NEGATIVE NUMBER TO POWER |
| 159 | POWER TOO LARGE |
| 160 | ARGUMENT NEGATIVE IN SQUARE ROOT |
| 161 | MATRIX DIMENSION ERROR |

TABLE B-4. EXECUTION TIME DIAGNOSTICS BY ERROR NUMBER (Contd)

| Error Number | Message | Error Number | Message |
|---|---|---|---|
| 162 | DET USED BEFORE INV | 173 | RANDOM FILE EMPTY |
| 163 | ARRAY TOO SMALL IN CHANGE | 174 | RANDOM ACTION BEYOND EOF |
| 164 | ERROR IN CHANGE | 175 | ILLEGAL ACTION ON BINARY FILE |
| 165 | ILLEGAL CHARACTER | 191 | ILLEGAL RPT$ PARAMETER |
| 166 | MEMORY OVERFLOW | 192 | ILLEGAL LPAD$ ARGUMENT |
| 167 | NONNUMERIC STRING | 193 | ILLEGAL RPAD$ ARGUMENT |
| 168 | STRING OVERFLOW | 194 | ILLEGAL ORD ARGUMENT |
| 169 | ILLEGAL SUBSTR PARAMETER | 195 | INPUT WITHIN INPUT |
| 170 | ILLEGAL LABEL | 196 | ILLEGAL CHR$ ARGUMENT |
| 171 | ILLEGAL ACTION ON CODED FILE | 197 | BAD TAB ARG - 1 USED |
| 172 | ILLEGAL SET VALUE | | |

# GLOSSARY C

Abort -
The procedure to terminate a program or job when a specified condition exists.

Alphanumeric -
The letters, digits, and special characters in the computer character sets defined in appendix A, tables A-1 and A-2.

ASCII -
American National Standard Code for Information Interchange, used as the ASCII 128-character set with either 6- or 12-bit characters.

BASIC -
Beginner's all-purpose symbolic instruction code, an elementary programming language.

Batch Processing -
A processing method that accumulates and processes together a number of related input items.

Bound Specifier -
An integer used to define the largest subscript for an array.

Breakpoint -
A designated location in a program where, if reached during program execution, a break or suspension in execution occurs.

Character Set -
The numbers, letters, and symbols having meaning in a given device or coding system.

Compile -
The procedure that translates a program from a high-level programming language, such as BASIC, into machine instructions called object code.

Concatenate -
The procedure of uniting or linking a series of characters; chaining.

Constant -
A value assumed to be fixed or invariable in a given operation or calculation.

CYBER Interactive Debug (CID) -
The facility that externally monitors and controls execution of a program, usually from an interactive terminal.

Debug -
The procedure to trace, detect, and eliminate mistakes in a program or in any software.

Direct Access File -
The permanent file, itself, that is made local.

Display Code -
An internal code set that is used by CDC CYBER 70, CYBER 170, and 6000 Series computers to represent alphanumeric and special characters. (Refer to tables A-1 and A-2 in appendix A.)

End-of-File (EOF) -
A boundary within a sequential file; the end of a file.

End-of-Information (EOI) -
The definition of the actual end of a named file.

End-of-Line (EOL) -
A special indicator that marks the end of each line or card image. EOLs are automatically written on coded files created by BASIC.

End-of-Record (EOR) -
A special indicator that marks the end of a logical record.

File -
A collection of data with an associated name.

Function -
A procedure that returns a value; invoked by a function reference in an expression.

Indirect Access File -
A separate local copy of the permanent file (used under NOS).

Input/Output (I/O) -
The equipment used to process data with a computer or the data processed and produced by the computer.

Interactive -
A two-way exchange of information; alternating input/output dialog; contrast with batch processing.

Interrupt -
The procedure to stop a running program in such a way that it can be resumed at a later time. The interrupt key depends on the terminal and system that is being used.

Local File -
Any file assigned to a job; this includes all temporary files (indirect access permanent files), all direct access permanent files, and all files that are not permanent.

Login -
The procedure to initially establish a terminal session.

Logoff -
The procedure used to end a terminal session.

Null String -
A data string that has a length of zero.

On-Line -
The condition when equipment communicates with the host computer.

Parameter Variable -
A variable that is given a specific value for a particular purpose or process.

Permanent File -
A file that remains in the operating system permanent file system after the user logs off.

Record -
A collection of related items of data treated as a unit. A complete set of such records can form a file.

Statement -
Each line of a program that begins with a line number.

String -
A sequence of contiguous characters or bits treated as a unit.

String Variable -
A variable that holds string values.

Subscripted Variable -
A representation for one value in an array of values; consists of numeric and string variables.

Substring -
A character string that is part of another string.

Temporary File -
A file that is released from the NOS system when the user logs off. It is a local file of an indirect access permanent file.

Time-Sharing -
The allocation of available computer time among all users, such that each user has equivalent access to system resources.

Trap (noun) -
The established mechanism for detecting a specified condition and causing a transfer of control. In CID, the location to which control is transferred is in CID itself.

Trap (verb) -
The automatic transfer of control to a predefined location upon the detection of some specified condition.

Variable -
An established identifer that represents a value or values that can change during program execution.

A file is a collection of information with an associated name. A BASIC program is an example of a file. A BASIC program frequently reads in another file containing data. All or part of the output from a program can be stored in a file instead of being printed at the terminal. This file can then be listed on a teletypewriter or on a high-speed printer, or simply used as data for another program.

NOS recognizes two types of files, local and permanent. A local file is any file assigned to a job; this includes all temporary and all attached direct access files. Before any file can be used, it must be made local. A permanent file is one that remains in the NOS permanent file system after the system is logged off. There can be both a local and a permanent copy of the same file. After the system is logged off, the permanent copy is retained and the local copy is released.

There are two types of permanent files, indirect access and direct access. An indirect access file is used indirectly; it is always a separate local copy of the permanent file that is used. With a direct access file, the permanent file (not a copy) is made local. (See figure D-1.) An indirect access file is created by using the NOS system commands: REPLACE and SAVE; a local copy is made available to the user by either the OLD, GET, or LIB commands; the local copy 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 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 local file. The local file established by a NEW, OLD, or LIB command, under the BASIC subsystem, is always primary. The NEW command creates a primary file; the OLD and LIB commands obtain a primary file from an indirect access file. There can be only one primary file and usually this file 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 local file. To refer to a local file with a NOS command, the file name must be specified, as in: LIST,F=DAT, or SAVE,DAT. In SAVE,DAT, file DAT is retained as a permanent file; DAT can be a primary or local file. When the current primary file is released by entry of the OLD, NEW, or LIB commands, all primary and local files are released unless the ND (no drop) is included in the command.

## NOS FILE CONTROL COMMANDS

The following subsections include brief descriptions of some NOS file manipulation commands. Specific information can be obtained pertaining to permanent files by using the CATLIST command described in the Network Products Interactve Facility reference manual (NOS 1 sites), Volume 3 of the NOS version 2 reference set (NOS 2 sites), or in the NOS Time-Sharing User's reference manual.

If the following commands are entered in batch mode, they should end with a period. The following commands are divided into those that access direct access permanent files and those that access indirect access permanent files.

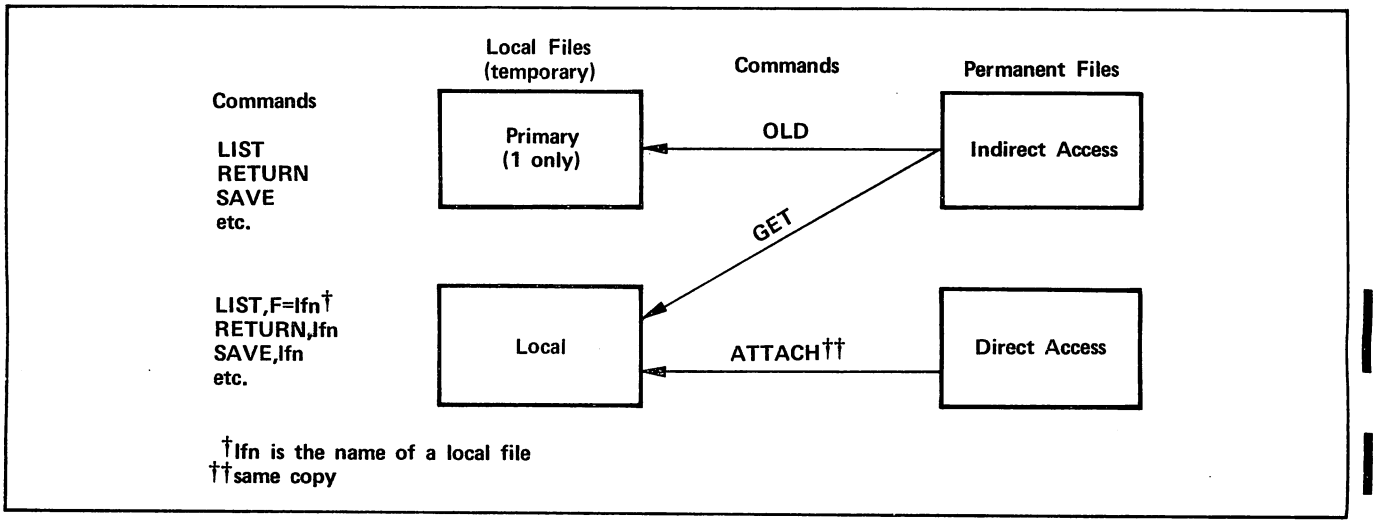


Figure D-1. NOS Files

## DIRECT ACCESS PERMANENT FILES

Figure D-2 illustrates the formats for the commands DEFINE and ATTACH, which are used to access direct access permanent files under NOS. The DEFINE command creates an empty permanent file pfn with a local file name. The ATTACH command makes a permanent file pfn a local file. For a description of the parameters not explained for ATTACH, see the DEFINE command.

## INDIRECT ACCESS PERMANENT

Figure D-3 illustrates the formats for the commands that access the indirect access permanent files under NOS. For a description of the statement parameters shown in these formats, see the DEFINE and CHANGE commands (figures D-2 and D-3).

The SAVE command creates an indirect access permanent file, permits a copy of the specified local file to be retained on the permanent file system, and specifies the subsystem to be associated with the file.

The GET command retrieves a copy of a specified indirect access file for use as a local file. To reference the local file by a name other than the pfn, the lfn parameter is used. The current primary file remains primary unless the file name specified by lfn is that of the current primary file. In that case, the contents of the primary file are replaced by a copy of pfn, which becomes the new primary file.

The OLD command 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 was saved while a subsystem, other than the null subsystem, was active.

The LIB command requests a copy of specified indirect access permanent files from the catalog of a special user library; this file becomes a primary file.

The REPLACE command permits the contents of an indirect access permanent file to be replaced with the contents of a local file. If pfn does not exist, a new permanent file is created.

The CHANGE command allows attributes of permanent files to be changed without further operation of the file; this is valid only for the originator of the file.

- DEFINE,lfn=pfn/CT=n,M=m,NA.

| | |
|---|---|
| lfn | If DEFINE is to be used to create an empty direct access permanent file, lfn (local file name) is specified only 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 | This is the permanent file name. If pfn is omitted, the system assumes lfn = pfn. |
| CT | Permanent File Category where n is one of the following (n can be abbreviated by concatenating the underlined letters):<br>PRIVATE = private<br>SPRIV = semi-private<br>PUBLIC = public |
| M | File or User Permission where m is one of the following (m can be abbreviated by concatenating the underlined letters):<br>WRITE = write permission<br>MODIFY = modify permission<br>READMD = read in modify mode<br>READAP = read in append mode<br>EXECUTE = execute file permission |
| NA | If a resource is unavailable, NOS suspends requests until a resource is free. |

- ATTACH,lfn=pfn/M=m,NA.

| | |
|---|---|
| lfn=pfn | This is used when desirable to reference an attached file by other than its permanent file name. If a current temporary file is referenced as lfn, the contents of that file are lost when the permanent file is attached. |
| M=m | This indicates modify permission. If omitted, the system assumes read permission only. |
| NA | This allows waiting for the direct access file to become available. If the file is currently being accessed, the job is suspended. IAF uses a user break, such as CTL P, to terminate the request. Enter STOP to terminate the request under the NOS Time-Sharing system. |

Figure D-2. Direct Access Permanent File Commands

- SAVE,lfn=pfn/CT=n,M=m,ss=subsyst,NA.
- GET,lfn=pfn/NA.
- OLD,lfn=pfn.
- LIB,lfn=pfn.
- REPLACE,lfn=pfn/NA.
- CHANGE,nfn=ofn/CT=n,M=m,ss=subsyst,NA.

| | |
|---|---|
| nfn | This is the new permanent file name to be assigned. |
| ofn | This is the current permanent file name. |
| CT and M | These are to be specified only if they are to be changed. For a description of the command parameters, see DEFINE command. |

Figure D-3. Indirect Access Permanent File Commands

## EXAMPLE OF FILE CONTROL COMMANDS

Figure D-4 illustrates a series of programs that use the system commands to create, reference, list, and purge files with a time-sharing terminal. The example is divided into three main columns. The leftmost column contains a transcript of the text entered and received at the terminal. The center column represents the area of temporary files. The center column 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 temporary files and shows when temporary files enter the working area and how long they remain. The rightmost column represents permanent files. It shows when a copy of a temporary file is made into a permanent file and how long that permanent file exists.

Temporary files are created with the NEW command or a copy of a file that already exists in the system. All temporary files are released when they are logged off the system. Local files include temporary and direct access files assigned to a job.

Duration of a file is indicated by a solid vertical line. An arrow point signals destination and termination. The copying of a file from lfn to pfn, or the reverse, is indicated by a broken horizontal line.

For a complete explanation of system commands, consult the Network Products Interactive Facility reference manual (NOS 1 sites), Volume 3 of the NOS version 2 reference set (NOS 2 sites), or the NOS Time-Sharing User's reference manual.

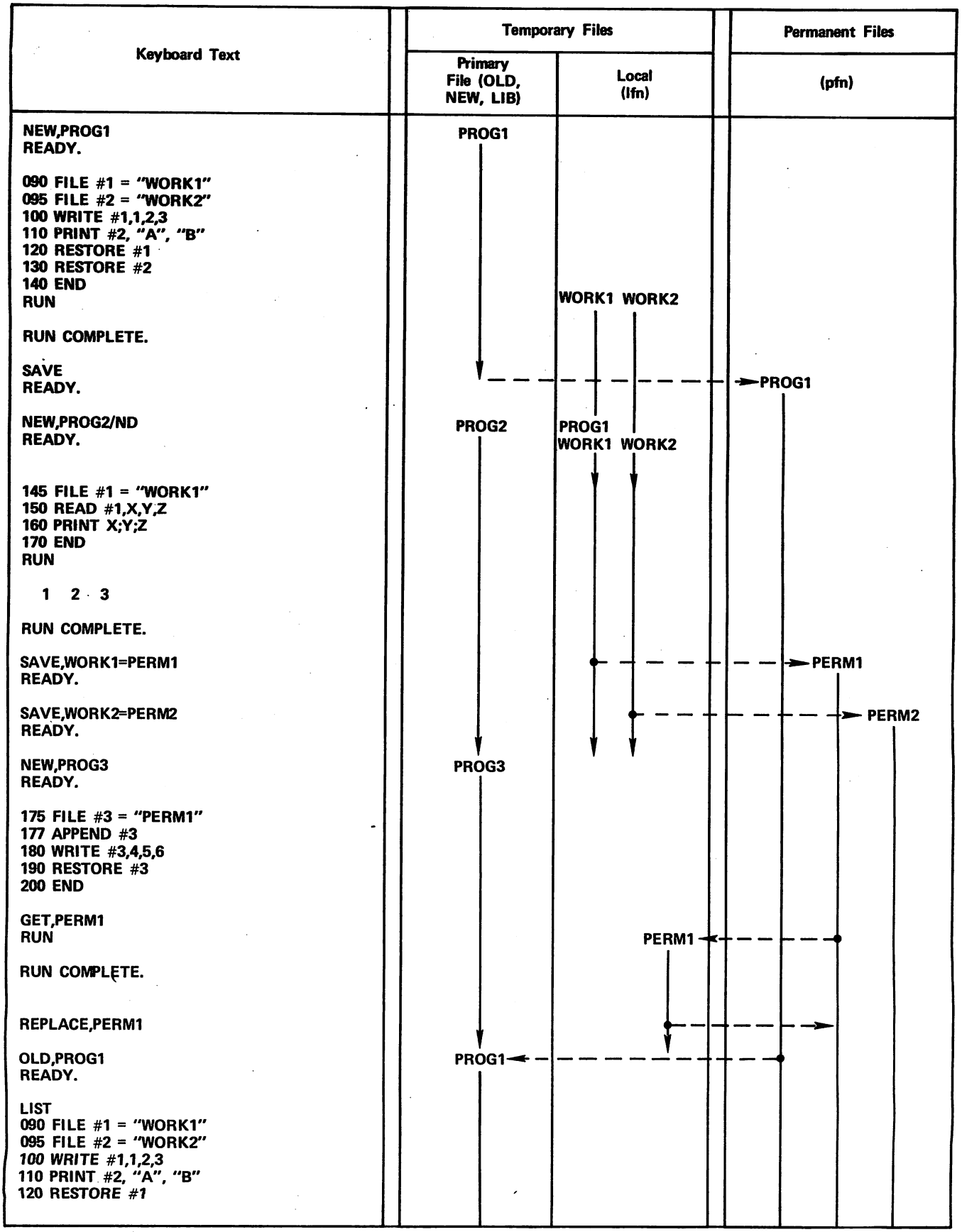


Figure D-4. File Control Commands (Sheet 1 of 2)

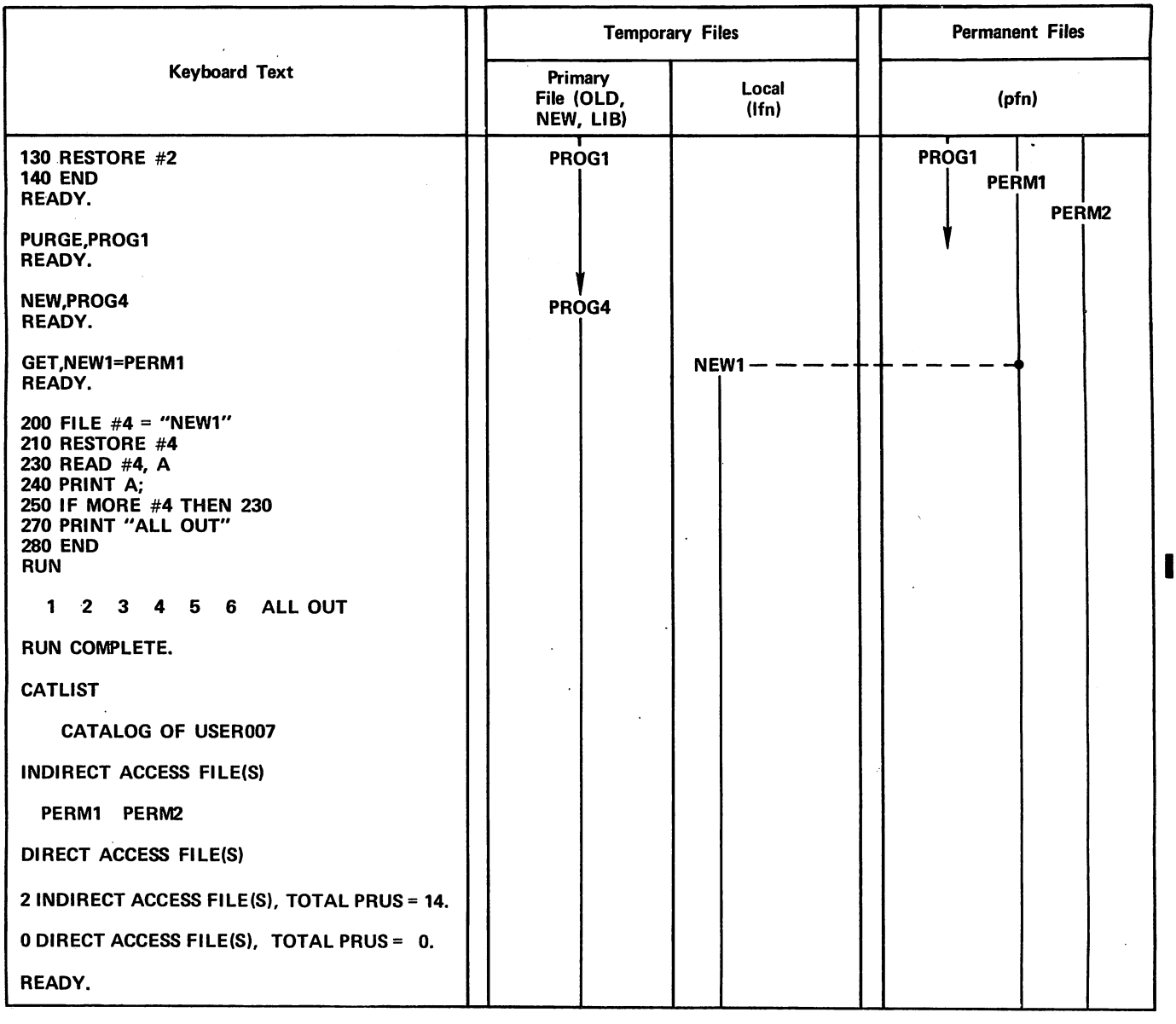


Figure D-4. File Control Commands (Sheet 2 of 2)

# FUTURE SYSTEM MIGRATION GUIDELINES



This appendix contains programming practices recommended by CDC for users of the software described in this manual. When possible, application programs based on this software should be designed and coded in conformance with these recommendations.

Two forms of guidelines are given. The general guidelines minimize application program dependence on the specific characteristics of a hardware system. The feature use guidelines ensure the easiest migration of an application program to future hardware or software systems.

## GENERAL GUIDELINES

Good programming techniques always include the following practices to avoid hardware dependency:

Avoid programming with hardcoded constants. Manipulation of data should never depend on the occurrence of a type of data in a fixed multiple such as 6, 10, or 60.

Do not manipulate data based on the binary representation of that data. Characters should be manipulated as characters, rather than as octal display-coded values or as 6-bit binary digits. Numbers should be manipulated as numeric data of a known type, rather than as binary patterns within a central memory word.

Do not identify or classify information based on the location of a specific value within a specific set of central memory word bits.

Avoid using COMPASS in application programs. COMPASS and other machine-dependent languages can complicate migration to future hardware or software systems. Migration is restricted by continued use of COMPASS for stand-alone programs, by COMPASS subroutines embedded in programs using higher-level languages, and by COMPASS owncode routines used with CDC standard products. COMPASS should only be used to create part or all of an application program when the function cannot be performed in a higher-level language or when execution efficiency is more important than any other consideration.

## FEATURE USE GUIDELINES

The recommendations in the remainder of this appendix ensure the easiest migration of an application program for use on future hardware or software systems. These recommendations are based on known or anticipated changes in the hardware or software system, or comply with proposed new industry standards or proposed changes to existing industry standards.

### ASC Function

Do not use the ASC function. Use the equivalent ORD function instead.

### ANSI Form

If both an ANSI form and a non-ANSI form exist, use the ANSI form. Non-ANSI forms might not be supported in future versions of BASIC.

### Blanks

Do not embed blanks within line numbers, keywords, variable names, and any other elements of the language.

### CHANGE Statement

Do not use the CHANGE statement. Use string functions or substring notation to manipulate characters. Do not manipulate the numeric codes for characters.

### Characters in Unquoted Strings

Use only the characters plus, minus, period, blank, digit, and letter in unquoted strings. Future versions of BASIC might only allow these characters; if other characters are needed, use quoted strings.

### CLK$ and DAT$ Functions

Do not dismantle values returned by the CLK$ and DAT$ functions; use the result as a whole. The order of fields in the result might be different in a future version of BASIC.

### Collating Sequence

Do not rely on the display code collating sequence (native collating sequence in normal mode, non-ASCII character set in use). The display code collation order might not be supported in future systems.

### DEF Function

Do not redefine a user-defined function within a program. In the future, redefining a function might not be possible.

### END Statement

Use the END statement in all programs. Future versions of BASIC might require the use of this statement.

### Exponentiation

Use the circumflex character (∧) rather than two asterisks (**) for exponentiation.

### File Numbers

Do not use file numbers greater than 255. Larger values might not be supported in future versions of BASIC.

### FOR...NEXT Loops

Do not transfer control into a FOR...NEXT loop. Results are unpredictable and future versions of BASIC might not allow it.

### Function Names Used as Variables

Do not use a function name as a variable within a function definition (do not place the name on the right side of an equals sign). This usage might not be permitted or might generate code with a different meaning in future versions of BASIC.

### IF..GOTO Statement

Avoid using this statement. Use IF...THEN instead. IF...GOTO might not be supported in future versions of BASIC.

### Keywords and Language Elements

Do not run keywords and variable names together. A statement such as PRINTT might not be supported in the future versions of BASIC.

### Multiple Assignments

Do not use multiple assignments. The form of such assignments might change in future versions of BASIC.

### Obsolete Forms

Avoid using any statement or function that causes the compile-time diagnostic OBSOLETE FORM. The BASE statement, the CHANGE statement, and the SUBSTR$ function are examples of obsolete forms that should be avoided.

### ON ne THEN Statement

Avoid using ON ne THEN $ln_1,ln_2,...,ln_n$ because this form might not be supported in future versions of BASIC. ON ne GOTO $ln_1,ln_2,...,ln_n$ should be used instead.

### Presetting Variables

Do not assume that variables will be preset to zero or null. Future versions of BASIC might not automatically preset variables.

### Referencing Functions

Define functions before referencing them. Future versions of BASIC might require the function definition to appear before the first reference to the function.

### RND Function

Use the RND function without a parameter. The parametric form might not be supported in future versions of BASIC.

### Simple and Subscripted Variable Names

Do not use the same name for array variables as for scalar variables. The use of the same name for both types of variables is not supported by standard BASIC and might not be supported in future versions.

### SUBSTR Function

Do not use the SUBSTR function. Use the equivalent substring notation instead.

# DIFFERENCES BETWEEN BASIC 3.5 AND BASIC 3.4 F

BASIC 3.5, the subject of this reference manual, is a version of BASIC 3.4 that was updated to conform to the American National Standard (ANSI) for Minimal BASIC. Due to syntax and semantic changes to the product, BASIC 3.5 is not 100 percent upward compatible with BASIC 3.4. Therefore, some BASIC 3.4 programs operate differently when compiled under BASIC 3.5. The following text identifies these differences and, where possible, provides suggestions for modifying the program to compensate for the affected change. Differences between 3.4 and 3.5 that are extensions (does not effect existing 3.4 programs) are not listed. BASIC 3.4 binaries continue to operate the same, except in those cases noted below.

## ARRAY BOUNDARIES

Unless otherwise instructed, in BASIC 3.5 the lower boundary (origin) of all arrays in a program is zero; in BASIC 3.4 the lower boundary is one. Therefore, arrays in BASIC 3.5 normally have one more element along each dimension than the arrays in BASIC 3.4. The OPTION statement using BASE n (was BASE statement in BASIC 3.4) is provided to set the lower boundary of an array to zero or one. Thus, if array subscripts are to begin with element 1 rather than element 0, use OPTION BASE 1 to change the origin to 1. (See OPTION statement described in section 3.)

## ROUNDING VERSUS TRUNCATION OF NUMERIC VALUES

BASIC 3.5 rounds all index, subscript, or pointer values that require integer values (for example, subscripts, TAB arguments, substring indexes, and ON statement indexes; BASIC 3.4 truncates these values to integer values. To truncate numeric quantities in a BASIC 3.5 program, use the INT function to force the truncation.

## TRAILING BLANKS IN UNQUOTED STRINGS OF DATA STATEMENTS AND INPUT REPLIES

BASIC 3.5 ignores all trailing blanks in unquoted strings of DATA statements and INPUT replies that use standard delimiters. BASIC 3.4 returns all trailing blanks of unquoted strings, unless the trailing blanks are at the end of a line (a string not followed by a delimiter), then BASIC 3.4 ignores the blanks. If trailing blanks are important to a program, enclose all unquoted strings with trailing blanks within quotation marks (for example: STRING1, STRING2,"THEN END Δ").

## INPUT VALIDATION

BASIC 3.5 validates all interactive responses to an INPUT request as to data type, number of data items input, and range of data values, before assigning any of them to the program. BASIC 3.4 validates and assigns INPUT responses one at a time. No programming changes can compensate for this difference.

## NOT ENOUGH DATA

When insufficient data is entered in response to an INPUT request, BASIC 3.5 permits either the entire INPUT response or only the additional items required to satisfy the request to be reentered. To add data, begin the next response with a comma. BASIC 3.4 only allows the additional data required to be entered to complete the INPUT request. No programming changes can compensate for this difference. This BASIC 3.5 response also applies to BASIC 3.4 binaries run under the BASIC 3.5 library.

## NUMERIC DATA READ AS CHARACTER STRING DATA

In BASIC 3.5, unquoted strings in DATA statements that look like numbers can be read either as numbers or as strings. In BASIC 3.4, this type of string can only be read as numbers.

## OUTPUT FORMATTING

BASIC 3.5 prints all integers greater than or equal to 1E7 in E Format (d.dddddd E+nn) if no other format is specified. In BASIC 3.4, integer values up to 1E9 are printed in integer format.

## PRINT ZONES

If a print zone is exactly filled in BASIC 3.5, the comma separator causes the print mechanism to skip over the next print zone causing spaces to be output; in BASIC 3.4, the next print zone is not skipped if the current print zone is exactly filled. In those cases where output must conform to BASIC 3.4 output, replacing the comma separator with a semicolon causes the print mechanism to be positioned at the first character of the next print zone.

## TAB POSITION

In BASIC 3.5, TAB(n) causes the print mechanism to be positioned so that the next character prints in column n. In BASIC 3.4, TAB(n) positions the print mechanism so that the next character prints in column n+1. If positioning is critical, add 1 to all TAB arguments in the *equivalent* BASIC 3.5 program.

## NEGATIVE TAB ARGUMENT VALUES

When BASIC 3.5 encounters a negative TAB value (TAB(n) where n<0), it resets the TAB value to 1 and issues an execution time warning diagnostic (error message number 197). BASIC 3.4 ignores negative TAB values. Change negative TAB values to positive TAB values in a BASIC 3.5 program to compensate for this difference.

## BACKWARD TABBING

In BASIC 3.5, TAB(n) positions the print mechanism to positon n on the next line, if n is less than the current print position; BASIC 3.4 ignores backward tabbing.

## COLLATING SEQUENCE

ASCII is the standard collating sequence used by BASIC 3.5 for string comparison operations and for computing values of the CHR$ and ORD functions regardless of the character set being used. In BASIC 3.4, the collating sequence depends upon the character set being used. It is display code if a normal, non-ASCII character set is being used; it is ASCII if an extended ASCII character set is being used. In BASIC 3.5, the OPTION statement using COLLATE can be used to select the collating sequence native to the character set currently being used by the program.

## FOR...NEXT LOOP CONTROL VARIABLE

In BASIC 3.5, the value of the loop control variable, upon normal exit from a FOR block via its NEXT statement, is the first value not used; in BASIC 3.4, it is the last value used. That is, in BASIC 3.5 the control variable value is the last value used plus one additional STEP value (+1 when no STEP value is specified), and in BASIC 3.4, the control variable value is the last value used upon exit from a loop.

## INPUTTING ARRAY DATA

BASIC 3.5 allows an entire array being read by a MAT INPUT statement to appear on one INPUT line in row order. A delimiter following the last item on the line indicates that the response is continued on the next line. BASIC 3.4 allows only one row of the array in each input reply line. In BASIC 3.5, if only one row of the array is entered, the diagnostic NOT ENOUGH DATA is received. The data for the complete matrix can be reentered or the remaining data can be entered to complete the matrix by beginning the response with a comma. This BASIC 3.5 feature also applies to BASIC 3.4 binaries run under the BASIC 3.5 library.

## REFERENCING DET BEFORE INV

Referencing the DET function before a matrix has been inverted via the INV function is considered a fatal error by BASIC 3.5. BASIC 3.4 simply returns a value of zero if no matrix has been inverted.

## REDIMENSIONING RESULT MATRICES

If required, BASIC 3.5 automatically redimensions a result matrix to accommodate the result; BASIC 3.4 generates a fatal error if the result matrix does not conform to the previously specified dimensions. No programming change can compensate for this difference. Redimensioning also applies to BASIC 3.4 binaries run under the BASIC 3.5 library.

## INVERTING A SINGULAR MATRIX

BASIC 3.5 does not diagnose as fatal error an attempt to invert a singular matrix; BASIC 3.4 does diagnose this as a fatal error. The DET (determinant) function must be used in BASIC 3.5 programs to determine if the matrix was singular or nearly singular; when DET returns a zero, it indicates that the matrix is singular.

## INVALID USE OF THE CHR$ FUNCTION

If the argument given to the CHR$ function is not the ordinal of any character in the selected collating sequence, BASIC 3.5 generates a fatal error and BASIC 3.4 returns a null string and no diagnostic. Use the ON ERROR mechanism to simulate 3.4 under 3.5.

## PRINT USING INTEGER FORMAT

In BASIC 3.5, values are rounded to an integer when printing according to an integer PRINT USING image field. In BASIC 3.4, these values are truncated. To force truncation under BASIC 3.5, use the INT function in the PRINT list.

# IMPLEMENTATION-DEFINED FEATURES G

BASIC, Version 3.5, is a revision of BASIC, Version 3.4. BASIC 3.5 conforms to the American National Standard for Minimal BASIC as specified in document ANSI X3.60-1978 published by the American National Standard institute. The ANSI publication identifies some features as implementation-defined. These features and their definitions for BASIC 3.5 are shown in table G-1.

TABLE G-1. IMPLEMENTATION-DEFINED FEATURES

| Item | BASIC 3.5 Definition/Comment |
|---|---|
| Initial value of numeric and string variables | Numeric variables are preset to zero; string variables are preset to null. However, your program should not depend on this initialization. See Future System Migration Guidelines, appendix G. |
| End-of-line (End of source line) | Indicated with carriage return when entering source lines at a terminal, with end-of-card when entering statements on cards. Trailing blanks are ignored by the BASIC compiler or removed by the operating system. Internally, end-of-line is denoted by a zero-byte terminator. |
| End-of-input reply | Same as end-of-line. |
| Precision of numeric constants | Approximately equal to 13+ decimal digits. Not all standards of BASIC support 13 digits of precision because only six digits are required. |
| Range of numeric constants | Range can be from 3.13152E-294 to 1.26501E+322. However, the standard only requires a range of 1E-38 to 1E+39. |
| Length of string constant | Length is limited only by line length. Since line length is longer for BASIC 3.5 than required by the ANSI standard, string constants can be longer than required by the ANSI standard. |
| Length of line | Length can be 150 characters; the ANSI standard requires only 72 characters. |
| Length of string associated with a string variable | Length can be 131,070 6-bit characters; the ANSI standard requires only 18 characters. |
| Precision of numeric value associated with a numeric variable | Same as for precision of numeric constant. |
| Range of numeric value associated with a numeric variable | Same as for range of numeric constant. |
| End-of-print line | Internally, it is a zero-byte terminator; last two or more 6-bit characters of a word are zero. |
| Print significance-width (d) | Width is six digits, the minimum required by the ANSI standard. The d controls the number of digits printed when the default format is used. |
| Print extra-width (e) | Width is three digits. The minimum required by the ANSI standard is two, but BASIC 3.5 uses three to accommodate the large exponents available on CYBERs. |
| Length of print zone | Length is 15 characters. The minimum required by the ANSI standard is d+e+6=15. |
| Margin | Margin is 75 characters. |
| Input-prompt | Prompt is "?", the same as recommended by the standard. |

The following sample programs illustrate some common features of BASIC. They are not presented as models for programming or mathematical techniques in problem solving.

The program in figure H-1 illustrates the use of the DEF and GOSUB statements to calculate the value PI by evaluation of a series.

The program in figure H-2 illustrates the use of a FOR...NEXT loop to calculate a table of factorials.

```
00100 DEF FNA(D)=(1/D)
00110 DEF FNB(D)=(D-FNA(B))
00120 DEF FNC(D)=(D+FNA(B))
00130 PRINT "CALCULATE A VALUE FOR PI"
00140 PRINT
00150 LET Z=100000
00160 PRINT "NUMBER OF ITERATIONS";Z
00170 PRINT
00180 LET A=1
00190 LET B=3
00200 FOR I=1 TO Z
00210 LET A=FNB(A)
00220 GOSUB 00280
00230 LET A=FNC(A)
00240 GOSUB 00280
00250 NEXT I
00260 PRINT "PI=";4*A
00270 STOP
00280 LET B=B+2
00290 RETURN
00300 END


produces:


CALCULATE A VALUE FOR PI

NUMBER OF ITERATIONS 100000


PI= 3.1416
```

Figure H-1. Using DEF and GOSUB Statements

The program in figure H-3 illustrates the sorting of a list of names (string variables) into alphabetic order.

The program in figure H-4 illustrates the inversion of a Hilbert Matrix (n times n) by using BASIC matrix operations.

The interactive terminal session shown in figure H-5 illustrates the CYBER Interactive Debug (CID) facility under NOS.

```
00100 LET A=1
00110 LET Z=20
00120 FOR I=1 TO Z
00130 LET A=A*I
00140 PRINT "FACTORIAL";I,A
00150 NEXT I
00160 END


produces:


FACTORIAL 1      1
FACTORIAL 2      2
FACTORIAL 3      6
FACTORIAL 4      24
FACTORIAL 5      120
FACTORIAL 6      720
FACTORIAL 7      5040
FACTORIAL 8      40320
FACTORIAL 9      362880
FACTORIAL 10     3.62880E+6
FACTORIAL 11     3.99168E+7
FACTORIAL 12     4.79002E+8
FACTORIAL 13     6.22702E+9
FACTORIAL 14     8.71783E+10
FACTORIAL 15     1.30767E+12
FACTORIAL 16     2.09228E+13
FACTORIAL 17     3.55687E+14
FACTORIAL 18     6.40237E+15
FACTORIAL 19     1.21645E+17
FACTORIAL 20     2.43290E+18
```

Figure H-2. Using FOR...NEXT Loop

```
00100 PRINT "UNSORTED LIST"
00110 READ N
00120 FOR I=1 TO N
00130 READ A$(I)
00140 PRINT A$(I)
00150 NEXT I
00160 FOR I=1 TO N-1
00170 FOR J=I+1 TO N
00180 IF A$(I)<A$(J) THEN 00220
00190 LET T$=A$(I)
00200 LET A$(I)=A$(J)
00210 LET A$(J)=T$
00220 NEXT J
00230 NEXT I
00240 PRINT
00250 PRINT "SORTED LIST"
00260 FOR I=1 TO N
00270 PRINT A$(I)
00280 NEXT I
00290 STOP
00300 DATA 8
00310 DATA MARY,JOHN,SUE,JOE,JACK,BILL,TED,ANN
00320 END
```

produces:

```
UNSORTED LIST
MARY
JOHN
SUE
JOE
JACK
BILL
TED
ANN

SORTED LIST
ANN
BILL
JACK
JOE
JOHN
MARY
SUE
TED
```

Figure H-3. Sorting String Variables

```
00100 DIM A(20,20),B(20,20)
00110 READ N
00120 MAT A=CON(N,N)
00130 MAT B=CON(N,N)
00140 FOR I=1 TO N
00150 FOR J=1 TO N
00160 LET A(I,J)=1/(I+J-1)
00170 NEXT J
00180 NEXT I
00190 MAT B=INV(A)
00200 MAT PRINT B;
00210 DATA 4
00220 END
```

produces:

```
-6.66667E-2 -.266667  4. -12.  9.33333

-.266667  14.9333 -104.  192. -102.667

 4. -104.  960. -1980.  1120.

-12.  192. -1980.  4320. -2520.

 9.33333 -102.667  1120. -2520.  1493.33
```

Figure H-4. Using Matrix Operations

```
/basic
OLD, NEW, OR LIB FILE: old,db1

 READY.
debug(on) <-------------------------------- Enters CID facility command while in the BASIC subsystem.

 READY.
list

100 LET A=2.1
110 LET B=A*A
120 LET C$="SUBSTRING ADDRESSING"
130 PRINT A,B
140 PRINT C$
150 END

 READY.
run <-------------------------------------- Compiles and executes the BASIC program.

 CYBER INTERACTIVE DEBUG
? sb l.110 <------------------------------- Sets breakpoint at 110.
? go <------------------------------------- Initiates execution.
 *B #1, AT L.110 <------------------------- Program reaches breakpoint.
? print a,b <------------------------------ Displays values of variables A and B.
  2.1            0 <--------------------- B=0 since line 110 has not yet executed.
? cb l.110 <------------------------------- Clears breakpoint at line 110.
? st line l.110...l.120 <------------------ Sets line traps.
? goto 100 <------------------------------- Resumes execution at line 100.
 *T #1, LINE AT L.110 <-------------------- LINE trap detected at line 110.
? let a=2.3 <------------------------------ Assigns 2.3 to variable A.
? go <------------------------------------- Resumes execution.
 *T #1, LINE AT L.120
? let b=30 <------------------------------- Assigns 30 to variable B.
? print a,b
  2.3            30
? ct * <----------------------------------- Clears all traps.
? goto 100 <------------------------------- Resumes execution at line 100.
 2.1            4.41
SUBSTRING ADDRESSING
 *T #17, END IN L.150 <-------------------- Default trap occurs at program termination.
? print a,c$
  2.1            SUBSTRING ADDRESSING
? let a=20
? let c$=c$(1:9) <------------------------- Replaces C$ with substring of C$.
? print a,c$
  20             SUBSTRING
? lv <------------------------------------- Lists all program values for program DB1.
 P.DB1
 A =  20,   B =  4.41,   C$ = "SUBSTRING"
? goto 100 <------------------------------- Resumes execution at line 100.
 2.1            4.41
SUBSTRING ADDRESSING
 *T #17, END IN L.150
? quit <----------------------------------- Terminates this CID session.

SRU      10.236 UNTS.

 RUN COMPLETE.
debug(off) <------------------------------- Exits CID environment.

 READY.
```

Figure H-5. Using CID Under NOS

# IN-LINE EDITING COMMANDS I

This appendix briefly covers the in-line editing (IEDIT) commands available when the user is in BASIC subsystem under NOS 2/IAF. IEDIT is an extension of the in-line editing functions available under NOS 1/IAF. For a detailed presentation of these commands, see Volume 3 of the NOS Version 2 reference set.

In the BASIC subsystem, IEDIT allows the user to perform some fundamental editing functions on the user's files without explicitly entering or exiting the editor. IAF recognizes IEDIT commands and generates calls to the in-line editor. This allows the user to intermix IEDIT commands with operating system commands.

The file to be edited is called the edit file. To use IEDIT commands, the edit file must be the primary file and a line-numbered file. The edit file is positioned to beginning-of-information (BOI) before and after each IEDIT command. The edit file will be altered whenever an IEDIT command which changes the content of the file is successfully executed.

IEDIT commands consist of a command name followed by parameters. The command name must be separated from the parameters by a non-blank separator; also, parameters must be separated from each other by a non-blank separator. In this appendix, a comma is always used as the separator.

## PARAMETERS

IEDIT command parameters must be specified in the order defined by each command format. The MOVE, DUP, and READ command parameters are position dependent; therefore, embedded parameters which are omitted must be explicitly indicated by two successive separators. For all other commands, omitted parameters need not be explicitly indicated by two successive separators.

The parameters used by the IEDIT commands are the lines parameter, the string parameter, and the file parameter. The formats of these parameters are discussed in the following paragraphs.

### LINES PARAMETER

The lines parameter specifies a noncontiguous set of lines in the edit file. The general format of the lines parameter is as follows:

```
m,n,p..q,r,s..t,u..v,w
```

where m, n, p, q, r, s, t, u, v, and w are line numbers. If a line in the edit file is referenced more than once in a lines parameter, a syntax error occurs.

### STRING PARAMETER

The string parameter consists of a sequence of characters (possibly the null string) with a string delimiter at the beginning and end of the sequence. The general format of the string parameter is as follows:

```
/string/
```

The string delimiter cannot occur within the string. A valid string delimiter is any character except a digit, a comma, an asterisk, a colon, or a space. In this appendix, a slash is used as the string delimiter.

### FILE PARAMETER

The file parameter specifies a file name consisting of one to seven characters where each character is a letter or digit. Either uppercase or lowercase letters can be used.

## COMMANDS

The following paragraphs present the formats and functions of the IEDIT commands.

### ALTER COMMAND

The ALTER command allows the user to change the specified string of characters in the edit file. The format of the ALTER command is as follows:

```
ALTER,lines,/string1/string2/
```

lines — A line and/or a range of lines in the edit file. This parameter is optional. If omitted, all lines in the file are considered when the command is executed.

string1 — The character string to be replaced. This parameter is required; however, it can be the null string.

string2 — The character string to replace string1. This parameter is required; however, it can be the null string.

## DELETE COMMAND

The DELETE command deletes the specified line or lines containing the specified character string from the edit file. The format of the DELETE command is as follows:

```
DELETE,lines,/string/
```

lines — A line and/or a range of lines to be deleted. This parameter is optional. If omitted, all lines containing the specified string are deleted.

string — A string of characters that must be contained in all lines that are deleted. This parameter is optional. If omitted, all specified lines are deleted.

Note that although both the lines parameter and the string parameter are optional, at least one must be specified.

## DUP COMMAND

The DUP command allows the user to duplicate lines in the edit file. The duplicated lines can be inserted anywhere in the edit file. The DUP command copies the specified lines; it does not move or delete the original lines. The format of the DUP command is as follows:

```
DUP,lines,n,z
```

lines — A line and/or range of lines to be duplicated. This parameter is required.

n — The line number after which the duplicated lines are to be inserted. This parameter is optional. If omitted, the default line number is the number of the last line in the edit file.

z — The line number increment for the duplicated lines. This parameter is optional. If omitted, the default value is 1.

## LIST COMMAND

The LIST command displays lines in the edit file. The user can also use the LIST command to display lines in the edit file which contain the specified character string. The format of the LIST command is as follows:

```
LIST,lines,/string/
```

lines — The line and/or range of lines to be displayed. This parameter is optional.

string — The character string that must be contained in all lines which are displayed. This parameter is optional.

If both the lines parameter and the string parameter are omitted, the entire edit file is displayed.

## MOVE COMMAND

The MOVE command moves lines to another place in the edit file. The format of the MOVE command is as follows:

```
MOVE,lines,n,z
```

lines — The line and/or range of lines to be moved. This parameter is required.

n — The line number after which the lines being moved are to be inserted. This parameter is optional. If omitted, the default line number is the number of the last line in the edit file.

z — The line number increment for the lines being moved. This parameter is optional. If omitted, the default value is 1.

## READ COMMAND

The READ command adds the contents of the specified file to the edit file. The user can control where the lines being added to the edit file will be inserted. The format of the READ command is as follows:

```
READ,file,n,z
```

file — The name of the file which is being added to the edit file. This parameter is required.

n — The line number after which the lines being added should be inserted. This parameter is optional. If omitted, the default line number is the last line in the edit file.

z — The line number increment for the lines being added. This parameter is optional. If omitted, the default value is 1.

## WRITE COMMAND

The WRITE command writes lines from the edit file to the specified file. The line numbers associated with the lines in the edit file are included in the data written to the specified file. The format of the WRITE command is as follows:

```
WRITE,file,lines,/string/
```

file — The name of the file to which the specified lines are written. This parameter is required.

lines — The line and/or range of lines to written to the specified file. This parameter is optional.

string — The string of characters which must be contained in all lines written to the specified file. This parameter is optional.

If both the lines parameter and the string parameter are omitted, the entire edit file, including line numbers, is written to the specified file.

## WRITEN COMMAND

The WRITEN command writes lines from the edit file to the specified file. The line numbers associated with the lines in the edit file are not included in the data written to the specified file. The format of the WRITEN command is as follows:

```
WRITEN,file,lines,/string/
```

file — The file name to which the lines are to be written. This parameter is required.

lines — The line and/or range of lines to be written. This parameter is optional.

string — The character string which must be contained in all lines written to the file. This parameter is optional.

If both the lines parameter and the string parameter are omitted, the entire edit file, excluding line numbers, is written to the specified file.

# INDEX

COMMENT SHEET

MANUAL TITLE: BASIC Version 3 Reference Manual

PUBLICATION NO.: 19983900 REVISION: H

NAME:

COMPANY:

STREET ADDRESS:

CITY: STATE: ZIP CODE:

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

___ Please reply ___ No reply necessary

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**

**Publications and Graphics Division**

215 Moffett Park Drive
Sunnyvale, California 94086

CUT ALONG LINE

FOLD

FOLD

CUT ALONG HERE

CUT ALONG HERE

# BASIC VERSION 3 SUMMARY CARD

## LANGUAGE ELEMENTS

### BASIC CHARACTER SET

| | |
|---|---|
| Alphabetic: | A thru Z |
| Numeric: | 0 thru 9 |
| Special: | + / . : @ |
| | , ( $ ? ' |
| | - ) = ; blank |

Any character available to the operating system can be used in data and string constants.

### VARIABLES

#### Numeric Variables

A numeric variable consists of a single alphabetic character or a single alphabetic character followed by a numeric character. Numeric variables are preset to zero before program execution.

#### String Variables

A string variable consists of a single alphabetic character followed by a dollar sign ($) or a single alphabetic character and a numeric character followed by a dollar sign.

#### Subscripted Variables

A numeric subscripted variable consists of a numeric variable followed by a subscript list bounded by parenthesis.

A string subscripted variable consists of a string variable followed by a subscripted list bounded by parenthesis.

## FORMAT FIELD SPECIFICATION CHARACTERS

| | |
|---|---|
| @ | Numeric character, alphabetic character, alphanumeric character |
| $ | Currency sign; floating when more than one |
| * | Check protect; leading blanks replaced by * |
| < | Left-justify string; right truncate |
| > | Right-justify string; left truncate |
| ^ | Floating point indicator |
| + | Plus printed for positive values; minus for negative |
| - | Blank printed for positive values; minus for negative |
| , | Comma insertion |
| . | Decimal point insertion |
| () | Negative values enclosed in parentheses; positive values in blanks |
| DB | DB inserted for negative value; two blanks for positive |
| CR | CR inserted for negative value; two blanks for positive |

### CONSTANTS

A numeric constant consists of an integer, decimal, or exponential number with absolute value in the range $3.13152 \times 10^{-294}$ to $1.26501 \times 10^{322}$ and is accurate to a maximum of 14 decimal digits.

A string constant consists of a string of 0 to 131070 6-bit characters or 0 to 65535 6/12-bit characters in display code.

#### Integer Constants

±n

#### Decimal Constants

±n ±.n ±n.n

#### Exponential Constants

±nE±s ±n.nE±s ±.nE±s ±n.E±s

| | |
|---|---|
| n | Decimal digits |
| E±s | Exponent |
| s | Base 10 scale factor |

#### String Constants

"string"

| | |
|---|---|
| string | String of alphanumeric characters. |

If the character " is to appear in the string, it must be specified by two consecutive " marks.

### OPERATORS

#### Arithmetic Operators

| | |
|---|---|
| Exponentiation: | ^ or ** |
| Division: | / |
| Multiplication: | * |
| Addition; unary plus: | + |
| Subtraction; unary minus: | - |

#### Relational Operators

| | |
|---|---|
| Equal to: | = |
| Not equal to: | <> or >< |
| Greater than: | > |
| Greater than or equal to: | >= or => |
| Less than: | < |
| Less than or equal to: | <= or =< |

#### Logical Operators

| | |
|---|---|
| Logical negation: | NOT |
| Logical multiplication or intersection: | AND |
| Logical addition or union: | OR |

#### String Operators

| | |
|---|---|
| String concatenation: | + |

## STATEMENTS AND FUNCTIONS

Throughout the following summary tables these notations are used. Terms in these tables that are in lowercase represent words or symbols supplied by the programmer.

| | |
|---|---|
| a | Alphabetic identifier |
| c | Numeric or string constant |
| ch | Any character or carriage return |
| d | Delimiter |
| e | Expression, constant, or variable |
| f | Format specification |
| l | Letter |
| lfn | Logical file name |
| ln | Line number |
| m | Matrix identifier (1- or 2-dimensional array) |
| na | Numeric array name |
| nc | Numeric constant |
| ne | Numeric expression, constant, or variable |
| r | Relational expression |
| sc | String constant |
| se | String expression, constant, or variable |
| snv | Simple numeric variable |
| stm | Executable statement |
| stv | String variable |
| sv | Simple variable |
| v | Variable identifier (simple, subscripted, numeric, or string) |
| x | Constant, variable, function, or numerical expression |
| [] | Enclosed elements are optional. |
| {\|} | Only one element must be selected. |
| ... | Repeat elements as needed. |

BASIC STATEMENTS

| Statement Format | Function | Section |
|---|---|---|
| APPEND #ne | Allows the user to add additional information to the end of an existing file. | 7 |
| CALL subnm (e1,e2,...,e20) | Allows the user to access external subroutines by subprogram name and pass up to 20 parameters to that subroutine. | 6 |
| CHAIN se | Executes program on file specified by se. Program can exist as binary or BASIC source. | 6 |
| CHAIN #ne | Executes program on file whose ordinal is ne. Program can exist as binary or BASIC source. | 6 |
| CLOSE #ne | Sets the named file to the beginning of information and detaches it from the BASIC program to allow reassignment by another FILE statement. | 7 |
| DATA c1,c2,...,cn | Creates a block of data internal to the BASIC program. | 7 |
| DEF FNa [(sv1,sv2,...,sv20)]=ne | Defines a new single-line numeric or string (FNa$) function to be used within the BASIC program. | 5 |
| DEF FNa$ [(sv1,sv2,...,sv20)]=se | | |
| DEF FNa [(sv1,sv2,...,sv20)] | Defines the start of a new numeric or string multiple-line function. The end of the function definition is indicated by FNEND. | 5 |
| DEF FNa$ [(sv1,sv2,...,sv20)] | | |
| DELIMIT (ch1),...,(ch3) | Defines separators between input items from terminal. | 7 |
| DELIMIT #ne,(ch1),...,(ch3) | Defines separators between input items on specified file. | 7 |
| DIM m1(nc1,...,nc3),...,mn(nc1,...,nc3) | Declares the dimensions of an array variable; nc must be integer. | 3 |
| END | Specifies end of program; must be last statement in program. | 3 |
| FILE #n1=lfn1, #n2=lfn2,...,#nn=lfnn | Defines file ordinal and equates it to a file name. | 7 |
| FNEND | Specifies the end of a multiple-line function definition. | 5 |
| FOR snv=ne1 TO ne2 [STEP ne3] | Begins a FOR...NEXT loop. | 4 |
| GOSUB ln | Transfers program control to a subroutine beginning at line number indicated. | 6 |
| GOTO ln | Interrupts the normal sequence of program execution and transfers program control to indicated line number. | 4 |
| IF END #ne {THEN\|GOTO} ln | This statement is equivalent to the NODATA statement except that this statement cannot refer to an internal data block. | 7 |
| IF r {THEN {ln\|stm}\|GOTO ln} | Transfers program control to line ln or executes statement stm if relational expression is true. Control falls through to next line if r is false. | 4 |
| IF r {THEN {ln1\|stm1}\|GOTO ln1} ELSE {ln2\|stm2} | Transfers program control to line ln1 or executes statement stm1 if relational expression r is true; transfers control to line ln2 or executes stm2 if r is false. | 4 |
| IF MORE #ne {THEN\|GOTO} ln | This statement is the logical converse of the NODATA and IF END statements. | 7 |
| (image) : character string | Specifies output formats. | 7 |
| INPUT v1,v2,...,vn | Reads data from the terminal. | 7 |
| INPUT #ne,v1,v2,...,vn | Reads coded data from specified file #ne. | 7 |
| JUMP ne | Transfers control to statement where line number=ne. | 4 |
| [LET] {v1=v2=v3=...=vn} =e | Assigns a value to a variable during program execution. | 3 |
| MARGIN ne2 | Defines a right-hand margin for output to a terminal. | 7 |
| MARGIN #ne1,ne2 | Defines a right-hand margin for output to a specified file. | 7 |
| MAT m=m | Matrix assignment. | 8 |
| MAT m1=m2+m3 | Matrix addition. | 8 |
| MAT m1=m2-m3 | Matrix subtraction. | 8 |
| MAT m1=m2*m3 | Matrix multiplication. | 8 |
| MAT m=(ne)*m | Matrix scalar multiplication by value of an expression. | 8 |
| MAT m=INV(m) | Inverts a matrix. | 8 |
| MAT m=TRN(m) | Transposes a matrix. | 8 |
| MAT m=ZER[(ne1[,ne2])] | Returns a matrix of all zeros. | 8 |
| MAT m=CON[(ne1[,ne2])] | Returns a matrix of all ones. | 8 |
| MAT m=IDN[(ne1[,ne2])] | Creates an identity matrix. | 8 |

BASIC STATEMENTS (Contd)

19983900 H

| Statement Format | Function | Section |
|---|---|---|
| MAT READ {m1,m2,...,mn \| #ne,m1,m2,...,mn} | Reads matrices from internal data block or reads matrices from file #ne in binary format. Can be used for redimensioning a matrix. (See Matrix Redimensioning.) | 8 |
| MAT PRINT {m1 d m2 d m3 d...d \| #ne,m1 d m2 d...d} | Prints matrices on a terminal or prints matrices in a coded format on specified file. | 8 |
| MAT PRINT USING {ln,m1 d m2 d...mn d \| se,m1 d m2,...mn d} | Matrices to be formatted to an image statement (ln) or an image (se) on a terminal. | 8 |
| MAT PRINT #ne USING {ln,m1 d m2 d...mn d \| se,m1 d m2 d...mn d} | Matrices to be formatted to an image statement (ln) or an image (se) on a specified file. | 8 |
| MAT INPUT {m1,m2,...,mn \| #ne,m1,m2,...,mn} | Inputs matrices from a terminal or a file #ne. | 8 |
| MAT WRITE #ne,m1,m2,...,mn | Writes matrices in binary format on a specified file. | 8 |
| NEXT snv | Terminates a FOR...NEXT loop and increments the value tested by the loop. | 4 |
| NODATA ln | Tests data pointer for increment beyond end-of-data block. Branches to ln if end-of-information is encountered. | 7 |
| NODATA #ne, ln | Transfers program control to specified line number if file is positioned at end-of-information. | 7 |
| ON ATTENTION GOTO ln | Transfers control to ln on runtime terminal interrupt. | 4 |
| ON ATTENTION THEN ln | Transfers control to ln on runtime terminal interrupt. | 4 |
| ON ATTENTION | Turns on normal terminal interrupt processing. | 4 |
| ON ERROR GOTO ln | Transfers control to ln on runtime error. | 4 |
| ON ERROR THEN ln | Transfers control to ln on runtime error. | 4 |
| ON ERROR | Turns on normal error processing. | 4 |
| ON ne GOSUB ln1,ln2,...,lnn | Permits a branch to a specific BASIC subroutine line number. The line referenced is dependent on the value of ne. | 6 |
| ON ne {GOTO\|THEN} ln1,ln2,...,lnn | Expression is evaluated and rounded to an integer value; transfers control to line number ln1 if ne=1, line number ln2 if ne=2, and so forth until lnn. | 4 |
| OPTION (BASE {0\|1} [,COLLATE {STANDARD\|NATIVE}] ; COLLATE {STANDARD\|NATIVE} [,BASE {0\|1}]) | Sets the lower boundary of all arrays being used by the program to base 0 (zero) or to base 1, and selects the collating sequence to be used for string comparison and for value computation of the CHR$ and ORD functions. Both or only one of these functions can be selected. | 3 |
| PRINT e1 d e2 d...en d | Prints data at terminal. | 7 |
| PRINT USING {ln,e1 d e2 d...en d \| se,e1 d e2 d...en d} | Output to be formatted by an image statement (ln) or an image (se) on a terminal. | 7 |
| PRINT #ne,e1 d e2 d...en d | Prints data on specified file. | 7 |
| PRINT #ne USING ln,e1 d e2 d...en d | Output to be formatted by an image statement on specified file. | 7 |
| RANDOMIZE | Overrides the predefined sequence of random numbers generated by the RND function. | 5 |
| READ v1,v2,...,vn | Accesses data created by DATA statements. | 7 |
| READ #ne,v1,v2,...,vn | Reads binary data from named file created by WRITE FILE statements. | 7 |
| REM LIST, {NONE\|ALL} | Controls optional source listing. | 12 |
| REM TRACE, {ALL\|PART\|NONE} | Controls optional trace facility. | 9 |
| REM ch1...chn | Inserts comments in program; ch1...chn represents any character string which does not trigger REM LIST or REM TRACE. | 3 |
| RESTORE | Reinitializes data pointer to the first word of the data block. | 7 |
| RESTORE #ne | Sets named file to beginning-of-information. | 7 |
| RETURN | Resumes execution at statement following most recently executed GOSUB statement. | 6 |
| SET #ne1,ne2 | Positions a file pointer to the desired relative word location which is to be referenced by the next READ or WRITE statement. | 7 |
| SETDIGITS ne | Specifies number of significant digits for output. | 7 |
| STOP | Terminates program execution at places other than the END statement. | 3 |
| WRITE #ne,e1,e2,...,en | Writes data in binary format on specified file. | 7 |

CUT ALONG HERE

CUT ALONG HERE



BASIC FUNCTIONS

| Function | Meaning | Section |
|---|---|---|
| ABS(ne) | Finds the absolute value of ne. | 5 |
| ASC(ch)<br>ASC(abr) | Returns the ASCII code in decimal of the character in its argument. | 5 |
| ASL(x) | Returns the line number of the statement at which the last terminal interrupt occurred. | 4 |
| ATN(ne) | Finds the arctangent of ne in the principal value range - π/e to + π/2. | 5 |
| CHR$(ne) | Returns the character with decimal code (ordinal position in the collating sequence) that corresponds to ne. | 5 |
| CLK(x) | Returns the time of day in hours and fractions of an hour in a 24-hour scale (x is a dummy argument). | 5 |
| CLK$ | Returns the time of day as a string. | 5 |
| COS(ne) | Finds the cosine of ne expressed in radians. | 5 |
| COT(ne) | Finds the cotangent of ne expressed in radians. | 5 |
| DAT$ | Returns the date as a string. | 5 |
| DET or DET(m) | Returns the determinant of the matrix most recently inverted by the INV function, or the determinant of matrix m. | 5 |
| ESL(x) | Returns the line number of the statement that caused the most recent program execution. | 4 |
| ESM(x) | Returns the error number associated with the most recent program execution error. | 4 |
| EXP(ne) | Finds the value of e to power of ne. | 5 |
| INT(ne) | Finds the largest integer not greater than ne.<br>Example: INT(5.95)=5 and INT(-5.95)=-6. | 5 |
| LEN(se) | Determines current length of string se. | 5 |
| LGT(ne) | Finds the base 10 logarithm of ne; ne > 0; otherwise an execution error will cause the program to terminate. | 5 |
| LOC(ne) | Returns the current word position in a file. | 7 |
| LOF(ne) | Returns the length in words of the referenced binary file (ne). | 7 |
| LOG(ne) | Finds the natural logarithm of ne where ne > 0; otherwise an execution error will cause the program to terminate. | 5 |
| LPAD$(se,ne) | Pads string se out to ne characters by adding spaces on the left of string se. | 5 |
| LTRM$(se) | Trims string se of all leading space characters. | 5 |
| LWRC$(se) | Returns a string consisting of the se string value with all uppercase letters replaced by their lowercase equivalents. | 5 |
| MAX($ne_1$,...,$ne_{20}$) | Returns the maximum element in the list. | 5 |
| MIN($ne_1$,...,$ne_{20}$) | Returns the minimum element in the list. | 5 |
| NXL(ne) | Returns the line number of the statement where the program execution is to resume. | 4 |
| ORD(se) | Returns the decimal code (ordinal position) of a character in string se in the collating sequence being used. See appendix A. | 5 |
| POS($se_1$,$se_2$,ne)<br>or<br>POS($se_1$,$se_2$) | Returns the position of string $se_2$ within string $se_1$. The position search begins with character ne. If ne is not indicated, the default is the first character. | 5 |
| RND or<br>RND(ne) | Returns pseudo-random numbers from the set of numbers uniformly distributed over the range 0 ≤ RND < 1.0. For RND, the same sequence of random numbers is returned unless the predefined sequence is overridden with the RANDOMIZE statement.<br><br>If ne > 0, a random number sequence is initialized based on the value of ne and the first number of the sequence is returned.<br><br>If ne = 0, the next number in the established sequence of random numbers is returned. If the sequence was not previously established by an ne > 0 RND reference, a standard constant is used to initiate the sequence.<br><br>If ne < 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 ne < 0 RND references return the next number in the sequence. | 5 |
| ROF(ne) or<br>ROF(ne,ne) | Finds the value of the first argument rounded to the number of decimal places specified by the second argument. Omission of ne rounds variable ne to the nearest integer. | 5 |
| RPAD$(se,ne) | Pads string se to ne characters by inserting blanks on the right of string se. | 5 |
| RPT$(se,ne) | Returns the string created by repeating the se string ne times. | 5 |
| RTRM$(se) | Trims string se of all trailing space characters. | 5 |
| SGN(ne) | Assigns a value of 1 if ne is positive; 0 if ne is 0; or -1 if ne is negative. | 5 |
| SIN(ne) | Finds the sine of ne where ne is an angle expressed in radians. | 5 |
| SQR(ne) | Finds the square root of ne where ne ≥ 0, otherwise an execution error causes the program to be terminated. | 5 |
| STR$(ne) or<br>STR$(ne,se) | Converts numeric value ne to string representation. The result is controlled by image string se, if present. | 5 |
| TAB(ne) | Moves print line to position ne. Can only be used in PRINT statement. | 7 |

BASIC FUNCTIONS (Contd)

| Function | Meaning | Section |
|---|---|---|
| TAN(ne) | Finds the tangent of ne where ne is an angle expressed in radians. | 5 |
| TIM(x) | Returns elapsed time in seconds (x is a dummy argument). | 5 |
| UPRC$(se) | Returns string se with all lowercase letters replaced by their uppercase equivalents. | 5 |
| USR$ | Returns the NOS 7-character user name (number). Under NOS/BE this function returns the string USERNUM. | 5 |
| VAL(se) | Converts string se to its numeric value. | 5 |

## BASIC CONTROL STATEMENT

BASIC.
BASIC(parameter-list)

| | | |
|---|---|---|
| AS | omitted<br>AS=0 | Source program and data files contain non-ASCII characters. |
| | AS | Source program and data files contain ASCII characters. |
| B | omitted<br>B=0 | Do not produce relocatable binary. |
| | B | Write relocatable binary on file BIN. |
| | B=lfn | Write relocatable binary on file lfn. |
| BL | omitted | Suppress page ejects on output listing. |
| | BL | Do not suppress page ejects on output listing. |
| DB | omitted<br>DB=0 | Do not activate the debug and trace features. |
| | DB | Same as DB=B/DL. |
| | DB=0/B | Force binary generation and/or program execution regardless of compilation errors. |
| | DB=0/DL | Activate program tracing as controlled by REM TRACE debug lines. |
| | DB=0/ID | Activates generation of CID information. |
| | DB=0/TR | Traces all statements regardless of REM TRACE debug lines. |
| | DB=TR | Same as DB=B/DL/TR. |
| | DB=ID | Same as DB=B/DL/ID. |
| E | omitted | Write compile-time error diagnostics on the file specified by the L parameter. If L=0 then write diagnostics to OUTPUT. |
| | E | Write compile time error diagnostics on file ERRS. |
| | E=lfn | Write compile time error diagnostics on file lfn. |
| EL | omitted<br>EL=W | Write warning diagnostics and fatal compile-time diagnostics on the file specified by the E parameter. |
| | EL=F | Write only fatal compile-time diagnostics on the file specified by the E parameter. |
| GO | omitted | If no B parameter is specified, execute compiled program without loading. If B parameter is specified, do not execute compiled program. |
| | GO | Execute compiled program, if no compilation errors. |
| | GO=0 | Do not execute compiled program. |
| I | omitted | Compile source program from file INPUT. |
| | I | Compile source program from file COMPILE. |
| | I=lfn | Compile source program from file lfn. |
| J | omitted<br>J | Read data from default file INPUT. |
| | J=lfn | Read data from default file lfn. |
| | J=0 | No default runtime data file. |
| K | omitted<br>K | Write execution-time output on default output file OUTPUT. |
| | K=lfn | Write execution-time output on default output file lfn. |
| L | omitted | If batch job, write compile-time output on default output file OUTPUT. If interactive job, suppress compile-time output. |
| | L | Write compile-time output on file OUTPUT. |
| | L=lfn | Write compile-time output on file lfn. |
| | L=0 | Suppress compile-time output. |
| LO | omitted<br>LO<br>LO=S | Write source listing on the file specified by the L parameter. |
| | LO=O | Write object and source listing on the file specified by the L parameter. |
| | LO=O/O | Write object listing only on the file specified by the L parameter. |
| | LO=0 | Turn off all list options. |
| PD | omitted | Use the installation default print density on the files specified by the L and K parameters. |
| | PD=6 | Use a print density of 6 lines/inch on the files specified by the L and K parameters. |
| | PD<br>PD=8 | Use a print density of 8 lines/inch on the files specified by the L and K parameters. |
| PS | omitted | If PD is not specified, use installation default page size for the file specified by the L parameter. If PD is specified, use PS=PD*(default PS)/(default PD). |
| | PS=n | Use a page size of n lines/page; 4≤n≤32768. |

# BASIC CONTROL STATEMENT PARAMETERS

## Compiler Listable Output

BL Burstable Listing Control

| | |
|---|---|
| omitted | Suppresses page ejects on output listing. |
| BL | Does not suppress page ejects on output listing. |

E Compile-Time Error File

| | |
|---|---|
| omitted | L parameter file, if no L file, OUTPUT. |
| E | File ERRS. |
| E=lfn | File lfn. |

EL Error Level Control

| | |
|---|---|
| omitted or EL=W | Fatal and warning to E parameter file. |
| EL=F | Fatal only to E file. |

L Compile-Time List File

| | |
|---|---|
| omitted | Default file OUTPUT. |
| L | File OUTPUT. |
| L=lfn | File lfn. |
| L=0 | None. |

LO Listing Options

| | |
|---|---|
| omitted or LO or LO=S | Source listing on L parameter file. |
| LO=0 | Object listing on L parameter file. |
| LO=0/0 | Object listing only on L file. |
| LO=0 | None. |

PD Print Density Control

| | |
|---|---|
| omitted | Default on L and K parameter files. |
| PD=6 | Density 6. |
| PD=8 or PD | Density 8. |

PS Page Size Control

| | |
|---|---|
| omitted | Default. |
| PS=n | (n is $4 \leq n \leq 32768$) |

## Compiler Input

AS ASCII Character Set

| | |
|---|---|
| omitted or AS=0 | Normal (non-ASCII). |
| AS | Encoded and run in ASCII. |

I Compile-Time Input

| | |
|---|---|
| omitted | File INPUT. |
| I | File COMPILE. |
| I=lfn | File lfn. |

## Compiler Binary Output

B Binary File

| | |
|---|---|
| omitted or B=0 | None. |
| B | File BIN. |
| B=lfn | File lfn. |

DB Debug, Trace, and Force Binary Generation and/or Program Execution

| | |
|---|---|
| omitted | None activated. |
| DB=0 | Debug and trace not activated. |
| DB | Default(DB=B/DL). |
| DB=0/B | Execute normally regardless of compilation errors. |
| DB=0/DL | Program tracing with REM Trace. |
| DB=0/ID | Interactive debug (CID tables and special code). |
| DB=0/TR | Trace all statements. |
| DB=TR | Same as DB=B/DL/TR. |
| DB=ID | Same as DB=B/DL/ID. |

## Program Execution

AS ASCII Mode

| | |
|---|---|
| omitted or AS=0 | Run in normal (non-ASCII) mode. |
| AS | Run in ASCII. |

DB Debug and Trace
(see Compiler Binary Output)

GO Execution Control

| | |
|---|---|
| omitted | Execute without loading if B not specified. |
| GO | Execute compiled program, if no compilation errors. |
| GO=0 | Inhibits execution. |

J Execution-Time Input File

| | |
|---|---|
| omitted or J | Default file INPUT. |
| J=lfn | Default file lfn. |
| J=0 | None. |

K Execution-Time Print File

| | |
|---|---|
| omitted | Default file OUTPUT. |
| K | Same as omitted. |
| K=lfn | Default file lfn. |

PD Print Density Control
(see Compiler Listable Output)

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION