SOFTWARE REFERENCE MANUAL

HDOS SYSTEM

Chapter 5

EXTENDED BENTON HARBOR BASIC

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

595-2339
Printed in the United States of America

TABLE OF CONTENTS

TAB GUIDE

INTRODUCTION

Extended BENTON HARBOR BASIC (Ex. B. H. BASIC) is a conversational programming language which is an adaptation of Dartmouth BASIC*. (BASIC is an acronym for Beginners' All Purpose Symbolic Instruction Code.) It uses simple English statements and familiar algebraic equations to perform an operation or a series of operations to solve a problem. BENTON HARBOR BASIC is an interpretive language, compact enough to run in a Heath computer with minimal memory, yet powerful enough to satisfy most problem-solving requirements. The interpretive structure of BASIC affords excellent facilities for the detection and correction of programming errors. It uses advanced techniques to perform intricate manipulations and to express problems more efficiently.

Manual Scope

This Manual is written for the user who is already familiar with the language BASIC. It also describes the extended implementation of Dartmouth BASIC and, in so doing, provides a brief summary of the language. However, this manual is not intended as an instruction Manual for the language BASIC. If you are not familiar with BASIC, we suggest that you obtain the Heathkit Continuing Education course entitled "Basic Programming," Model EC-1100, before attempting to use this Manual.

*BASIC is a registered trademark of the Trustees of Dartmouth College.

Hardware Requirements

Extended BENTON HARBOR BASIC runs on an H8/H17 or H89 Computer System with a minimum of 16K bytes of random access memory.

In order to run BASIC, you must first copy the file BASIC.ABS from your software distribution disk onto the system disk you plan to use. Use PIP or ONECOPY to accomplish this. Refer to Chapter 1, the HDOS "Operating System" Manual, for assistance.

Once the file BASIC.ABS is present, you can run BASIC by typing

```
RUN△dev:BASIC ↵
```

where "dev:" is the device name (SY∅: or SY1:) that contains the file BASIC.ABS. If you do not type a device name, HDOS assumes the file is on SY∅:. For example:

```
>RUN△BASIC ↵
EXTENDED BENTON HARBOR BASIC #110.00.00.
*
```

BASIC uses the asterisk (*) as its prompt character.

Note that the part number may be different. However, a part number will be displayed.

BASIC ARITHMETIC

Data Types

BASIC supports three different data types:

1. Numeric data.
2. Boolean data.
3. String data.

NUMERIC DATA

BASIC accepts real and integer numbers. A real number contains a decimal point. BASIC assumes a decimal point **after** integer data. Any number can be used in mathematical expression without regard to its type. Real numbers must be in the approximate range of 10^{-38} to 10^{+37}. Integer numbers must lie in the range of 0 to 65535. All numbers used in BASIC are internally represented in floating point, which allows approximately 6.9 digits of accuracy. Numbers may be either negative or positive.

In addition to integer and real numbers, BASIC recognizes a third format. This format, called exponential notation, expresses a number as a decimal number raised to a power of 10. The exponential form is

```
XXE(±)NN
```

where E represents the algebraic statement "times ten to the power of," XX represents up to a six-digit integer or real number, and NN represents an integer from 0 to 38. Thus, the number is read as "XX times 10 to the ± power of NN."

Numeric data in all three forms may be used in the immediate mode, program mode in data statements, or in response to READ and INPUT statements.

Unless otherwise specified, all the numbers including exponents are presumed to be positive.

The results of BASIC computations are printed as decimal numbers if they lie in the range of 0.1 to 999999☆. If the results do not fall in this range, the exponential format is used. BASIC automatically suppresses all leading and trailing zeros in real and integer numbers. When the output is in exponential format, it is in the form

```
(±) X.XXXXXE (±) NN
```

The following are examples of typical inputs and the corresponding output. Note the dropping of leading and trailing zeros, truncation to six places of accuracy, conversion to exponential notation when necessary, and conversion to decimal notation where permitted.

INPUT NUMBER	OUTPUT NUMBER	COMMENTS
0.1	.1	(leading zero dropped)
.0079	7.90000E-03	(<.1 converts to exponential)
0022	22	(leading zeros dropped)
22.0200	22.02	(trailing zeros dropped)
999999	999999	(format maintained)
1000000	1.00000E+06	(converted to exponential)
100000007	1.00000E+08	(truncated to 6 places)
-10.1E+2	-1010	(converted to decimal format)

BOOLEAN DATA

Boolean values are a subclass of numeric values. Values representing the positive integers from 0-65,535 (2^{16-1}) may be used as Boolean data. When using numeric data as Boolean values, the numeric data represents the equivalent 16-bit binary numbers. Fractional parts of numeric data used with Boolean operators are discarded. If the numeric value with the fractional part does not fall into the range of 0-65,535, an illegal number error is generated.

STRING DATA

Extended BASIC handles data in a character string format. Data elements of this type are made up of a string of ASCII characters up to 255 characters in length. Extended BASIC provides operators and functions to manipulate string data. Any printable ASCII character (with the exception of the quotation mark itself) may appear in an Extended BASIC string. In addition to the printable ASCII characters, the line feed and bell characters are also permitted. A string may not be typed on more than one line. A carriage return is rejected as an illegal string character.

☆NOTE: This may be changed. See "CNTRL 1," Page 5-38.

Variables

A BASIC variable is an algebraic symbol representing a number. Variable naming adheres to the Dartmouth specification. That is, variable names consist of one alphabetic character which may be followed by one digit (zero to nine). The following is a list of acceptable and unacceptable variables, and the reason why the variable is unacceptable.

ACCEPTABLE VARIABLES	UNACCEPTABLE VARIABLES	REASON FOR UNACCEPTABILITY
C	2C	A digit cannot begin a variable.
A5	AF	A second character in a variable must be a number (0-9).
D	3	A single number is not an acceptable variable.
L2	$2	The first character of a variable must be a letter (A-Z).

Subscripted variables, string variables, and subscripted string variables are permitted. See "Subscripted Variables," Page 5-12, and "String Manipulation" on Page 5-21.

A value is assigned to a variable when you indicate the value in a LET, READ, or INPUT statement. These operations are discussed in "LET" (Page 5-50), "PRINT" (5-55), and "INPUT AND LINE INPUT" (Page 5-64).

The value assigned to a variable changes each time a statement equates the variable to a new value. The RUN command sets all variables to zero (0). Therefore, it is only necessary to assign an exact value to a variable when an initial value other than zero is required.

Subscripted Variables

In addition to the variables described above, BASIC permits subscripted variables. Subscripted variables are of the form:

```
An (N1, . . . . . , N8),
```

where A is the variable letter, n is a number (optional) 0-9, and N_1 thru N_8 are the integer dimensions of the variable. Subscripted variables provide you with the ability to manipulate lists, tables, matrices, or any set of variables. Variables are allowed one to eight subscripts.

The use of subscripts permits you to create multi-dimensional arrays of numeric and string variables. It is important to note that a dimensioned variable is distinguished from a scaler value of the same name. For example, all four of the following are distinct variables:

```
A, A(N), A$, A$,(N)
```

When you are referencing a subscripted variable, each element in the subscript list may consist of an arbitrarily complex expression so long as it evaluates to a numeric value within the allowable range for the indicated dimension. Thus, the subscripted variable A(5,5), would be dimensioned as:

```
X = A(2,3)                    is legal
X = A(2↑2, VAL("4.0"))        is legal as it is equivalent to A(4,4)
X = A(2,"4.0")                is not legal as ("4.0" is a string)
```

The following are graphic illustrations of simple subscripted variables. In these particular examples, a simple variable (A) is followed by one or two integer expressions in parentheses. For example,

```
A(I)
```

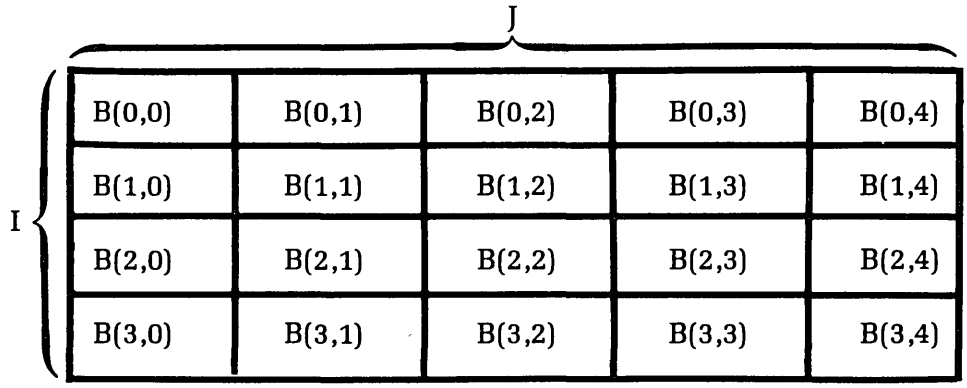
where I may assume the values of 0 to 5, allows reference to each of the six elements A(0), A(1), A(2), A(3), A(4), and A(5). A graphic representation of this 6-element, single-dimension array is shown below. Each box represents a memory location reserved for the value of the variable of the indicated name. Often, the entire array is referred to as A(.

A(0)
A(1)
A(2)
A(3)
A(4)
A(5)

NOTE: Subscripted variables begin at zero. Therefore, the previous example 0 to 5 defines six elements.

A two-dimensional array B(I, J) allows you to refer to each of the elements (B0,0), B(0,1), B(0,2),...., B(0,J),...., B(I,J).

This is graphically illustrated as follows, for B(3,4).



NOTE: A variable cannot be dimensioned twice in the same program unless you first clear it with the CLEAR statement.

BASIC does not presume any dimension. Therefore, the DIMension (DIM) statement must be used to define the maximum number of elements in any array. It is described in "DIM (DIMENSION)" on Page 5-40.

Expressions

An expression is a group of symbols to be evaluated by BASIC. Expressions are composed of numeric data, Boolean data, string data, variables, or functions. In an expression, these are alone or combined by arithmetic, relational, or Boolean operators.

The following examples show some expressions BASIC recognizes.

ARITHMETIC EXPRESSIONS	BOOLEAN EXPRESSIONS	STRING EXPRESSIONS	DESCRIPTION
1.02	255	"YES"	Data
1.02+ 16	255 OR 003	"YES" + "NO"	Combined
A < B		"YES" < "NO"	Relational

A major feature of BASIC is its extensive use of expressions in situations when many other BASICs only permit variables or numbers. This feature permits you to perform very sophisticated operations within a particular command or function. It is important to note that not all expressions can be used in all statements. The explanations describing the individual statements detail any limitations.

Arithmetic Operators

BASIC performs exponentiation, multiplication, division, addition, and subtraction. BASIC also supports two unary operators (− and NOT). The asterisk (*) is used to signifiy multiplication and the slash (/) is used to indicate division. Exponentiation is indicated by the up arrow (↑).

THE PRIORITY OF ARITHMETIC OPERATIONS

When multiple operations are to be performed in a single expression, an order of priority is observed. The following list shows the arithmetic operators in order of descending precedence. Operators appearing on the same line are of equal precedence.

−(Unary)	(negation)
↑	(exponentiation)
* /	(multiplication division)
+ −	(addition subtraction)

Parentheses are used to change the precedence of any arithmetic operations, as they are in common algebra. Parentheses receive top priority. Any expression within parentheses is evaluated before an expression without parentheses. The innermost leftmost parenthetical expression has the greatest priority.

UNARY OPERATORS

BASIC supports two unary operators: – and NOT. These operators are referred to as unary because they require only one operand. For example:

```
A = -2
C = NOT D
```

The unary operator (–) performs arithmetic negation. The NOT operator performs Boolean negation. See Page 5-19.

EXPONENTIATION

Exponentiation (↑) is used to raise numeric or variable data to a power. For example:

A = B↑2 is equivalent to A = B * B.

NOTE: The operand must not be negative. The exponent may be negative. A negative operand generates a syntax error. For greatest efficiency, B↑2 should be written as B* B and B↑3 should be written as B * B * B. All other powers should use the ↑.

MULTIPLICATION AND DIVISION

BASIC uses the asterisk (*) and the slash (/) as symbols to perform the algebraic operations of multiplication and division, respectively. Both multiplication and division require numeric data as operands.

The following examples use the multiplication and division operators:

```
*PRINT 2*6 ㊐
 12

*PRINT 2/3 ㊐
 .666667

*PRINT 6/3*2 ㊐
 4

*
```

NOTE: This last expression evaluates to 4, not 1; as * and / have equal precedence and, therefore, the leftmost operator is evaluated first.

ADDITION AND SUBTRACTION

The plus sign (+) and the minus sign (−) perform arithmetic addition and subtraction. In addition, the plus operator (+) performs string concatenation if both operands are string data. The following examples use the plus and minus operators;

```
*PRINT 3 (CR)
 3
*PRINT 3+5 (CR)
 8

*PRINT 10-3 (CR)
 7

*PRINT "HEATH" + " " + "COMPUTER" (CR)
HEATH COMPUTER
*
```

SUMMARY

In any given expression, BASIC performs arithmetic operations in the following order:

1. Parentheses have top priority. Any expression in parentheses is evaluated prior to a nonparenthetical expression.
2. Without parentheses, the order of priority is:
   a. Unary minus and NOT (equal priority).
   b. Exponentiation (proceeds from left to right).
   c. Multiplication and division (equal priority, proceeds from left to right).
   d. Addition and subtraction (equal priority, proceeds from left to right).
3. If the rules in either 1 or 2 do not clearly designate the order of priority, the evaluation of expression proceeds from left to right.

The following examples illustrate these principles. The expression 2↑3↑2 is evaluated from left to right:

1. 2↑3 = 8 (leftmost exponentiation has highest priority).
2. 8↑2 = 64 (answer).

The expression 12/6*4 is evaluated from left to right since multiplication and division are of equal priority:

1. 12/6 = 2 (division is the left-most operator).
2. 2*4 = 8 (answer).

The expression 6+4*3↑2 evaluates as:

1. 3↑2 = 9 (exponentiation has highest priority).
2. 9*4 = 36 (multiplication has second priority).
3. 36+6 = 42 (addition has lowest priority; answer).

Parentheses may be nested, (enclosed by additional **sets** of parentheses). The expression in the innermost set of parentheses is evaluated first. The next innermost left-justified is second, and so on, until all parenthetical expressions are evaluated. For example:

```
6 *((2↑3+4)/3)
```

Evaluates as:

1. 2 ↑3 = 8 (exponentiation in parentheses has highest priority).
2. 8+4 = 12 (addition in parentheses has next highest priority).
3. 12/3 = 4 (next innermost parentheses are evaluated).
4. 4*6 = 24 (multiplication outside of parentheses is lowest priority).

Parentheses prevent confusion or doubt when you are evaluating the expression. For example, the two expressions

```
D*E↑2/4+E/C*A↑2
((D*(E↑2))/4)+((E/C)*(A↑2))
```

are executed identically. However, the second is much easier to understand.

Blanks should be used in a similar manner, as BASIC ignores blanks (except when they are part of a string enclosed in quotation marks). The two statements:

```
10 LET B = 3 * 2 + 1
10 LET B=3*2+1
```

are identical. The blanks in the first statement make it easier to read.

Relational Operators

Relational operators compare two variables or expressions. They are generally used with an IF THEN statement. The result of a comparison by the relational operators is either a true or a false. A false is represented by zero, and true is represented by 65535 (2^{16}-1). NOTE: These values are chosen so when they are used as Boolean values, false is all zeros and true is all ones.

The following table lists relational operators as used in BASIC.

ALGEBRAIC SYMBOL	BASIC SYMBOL	EXAMPLE	MEANING
=	=	A=B	A is equal to B.
<	<	A<B	A is less than B.
≤	<=	A <=B	A is less than or equal to B.
>	>	A>B	A is greater than B.
≥	>=	A> =B	A is greater than or equal to B.
≠	<>	A< >B	A is not equal to B.

The symbols =<, =>, >< are not accepted and BASIC generates a syntax error if they are used.

The following examples show the results of using relational operators.

```
*PRINT 3<4 ↵      (true)
 65535

*PRINT 4<3 ↵      (false)
 0
```

EX. B.H. BASIC differs from most other BASICs in the use of the relational operator. When you are using BASIC, you may use the relational operators in any expression. When the expression is evaluated, the appropriate numeric answer (0 or 65535) will be used as the answer to that expression.

Boolean Operators

OR

The operator OR performs a Boolean OR on the two integer operands. The integer operands (which must lie in the range of 0 to 65535) are converted to 16-bit binary numbers. The Boolean (logical) 16-bit OR is applied and the result is returned to the equivalent integer representation. NOTE: As the Boolean value chosen to represent true (65535) and false (0), the OR operator implements a standard truth table OR function. For example:

```
*PRINT 132 OR 255 ®          00000000 10000100   132
 255                         00000000 11111111   255
                             00000000 11111111   255
```

and

```
*PRINT (3>2) OR (4>9) ®
 65535
```

AND

The AND operator performs a Boolean (logical) AND on the two integer operands. These integer operands must lie in the range of 0 to 65535. The integer operands are converted into 16-bit binary numbers and the logical AND is performed. The result is returned to the equivalent integer representation. NOTE: The AND operator implements a standard AND truth table on the values true (65535) AND false (0). For example:

```
*PRINT 132 AND 255 ®         00000000 10000100   132
 132                         00000000 11111111   255
*                            00000000 10000100   132
```

and

```
*PRINT (3>2) AND (9>7) ®
 65535
```

NOT

The NOT operator Boolean negation. That is, the numeric value of the variable is converted into a 16-bit Boolean data value; each **bit** is inverted, and the 16-bit binary number is restored to numeric data. For example:

```
*PRINT NOT 0 ®      0 = 00000000 00000000    and
 65535          65535 = 11111111 11111111
*
```

STRING MANIPULATION

Extended BENTON HARBOR BASIC is capable of manipulating string information. A string is a sequence of characters treated as a single unit of an expression. It can be composed of alphanumeric and other printing characters. An alphanumeric string contains letters, numbers, blanks, or any combination of these characters. A character string may not exceed 255 characters. The blank, bell, form feed, and TAB are considered to be printing characters.

String Variables

The dollar sign ($) following a variable name indicates a string variable. For example:

```
B$
     and
L6$
```

are string variables. A string variable (B$) is used in the following example.

```
*B$ = "HI": PRINT B$  ↵
HI
```

NOTE: The string variable B$ is separate and distinct from the variable B.

Any array name followed by the $ character notes that the dimensioned variable is a string. For example:

```
L$(n)        A2$(n)       (single-dimensioned string variables).
D$(m,n)      H1$(m,n)     (multiple-dimensioned string variables).
```

The numbers in parentheses indicate the location within the array. See "Subscripted Variables," Page 5-12.

The same variable can be used as a numeric variable and as a string variable in one program. For example, each of the following is a different variable:

```
B            B(n)
B$           B$(m,n)
```

The following are illegal, as they are double declarations of the same variable.

```
A$(n)        A$(n,m)
```

String arrays are defined with a dimension (DIM) statement in the same way numerical arrays are defined.

String Operators

Extended BASIC provides you with the ability to manipulate strings. The string manipulation operators are: plus (+), for concatenation, and the relational operators.

CONCATENATION

Concatenation connects one string to another without any intervening characters. This is specified by using the plus (+) symbol and only works with strings. The maximum length of a concatenated string is 255 characters. For example:

```
*PRINT "THE HEATH " + "COMPUTER" ENTER
THE HEATH COMPUTER
```

RELATIONAL OPERATORS FOR STRINGS

Relational operators, when applied to strings, indicate alphabetic sequence. The relational comparison is done on the basics of the ASCII value associated with each character, on a character-by-character basis, using the ASCII collating sequence. A null character (indicating that the string is exhausted) is considered to head the collating sequence. For example:

```
*PRINT "ABC" < "DEF" ENTER
 65536        (The relation shown is true)
*PRINT "ABC">"ABCD" ENTER
 0
              (The relation is false. "ABC" is less than "ABCD".)
```

NOTE: In any string comparison, trailing blanks are not ignored. For example:

```
*PRINT "CDE" = "CDE " ENTER
 0            (The equality is false.)
```

The following table indicates how relational operators are used with string variables in Extended BASIC.

OPERATOR	EXAMPLE	MEANING
=	A$ = B$	String A$ and B$ are alphabetically equal.
<	A$ < B$	String A$ is alphabetically less than B$
>	A$ > B$	String A$ is alphabetically greater than B$
< =	A$ < = B$	String A$ is equal to or less than B$.
> =	A$ > = B$	String A$ is equal to or greater than B$.
< >	A$ < > B$	String A$ and B$ are not alphabetically equal.

THE COMMAND MODE

Using the Command Mode for Statement Execution

You may solve a problem in BASIC by using a complete program or by use of the **command** mode. **Command** mode makes BASIC an extremely powerful calculator.

Lines of program material entered for later execution are identified by line numbers. BASIC identifies those lines entered for immediate execution by the absence of the line number. That is to say, statements that begin with line numbers are stored, and statements without line numbers are executed immediately when a carriage return is received. For example:

```
10 PRINT "THIS IS A COMPUTER" ↵
```

is not executed when it is entered at the console terminal. However, the statement:

```
*PRINT "THIS IS THE HEATH COMPUTER" ↵
```

when the RETURN key is typed, is immediately executed as:

```
THIS IS THE HEATH COMPUTER
```

The **command** mode of operation is useful in program de-bugging and performing simple calculations which do not justify the writing of a complete program.

For example, in order to facilitate program de-bugging, you may place STOP statements liberally throughout a program.

If you use STOP in this manner, an error message will be printed. This is a normal response and not a programming error on your part. Once BASIC encounters a STOP statement, the program halts. You can examine and change data values using the **command** mode. The statement

```
CONTINUE ↵
```

is used to continue execution of the program. You can also use the GOSUB and IF commands. Values assigned to variables remain intact using this technique. A SCRATCH, CLEAR, or another RUN command resets these values.

The ability to place multiple statements on a single line is an advantage in the **command** mode. For example:

```
*B = 2:PRINT B:PRINT B + 1 ↵
 2
 3
*
```

Program loops are allowed in the **command** mode. For example, a table of squares can be produced as follows:

```
*FOR A = 1 TO 10:PRINT A,A * A:NEXT A ↵
 1             1
 2             4
 3             9
 4            16
 5            25
 6            36
 7            49
 8            64
 9            81
 10           100
*
```

Some statements cannot be used in the **command** mode. The INPUT statement, for example, is not available in the **command** mode, and its use results in the USE error message. There are certain command functions in the **command** mode which make no sense when used in the **command** mode. Statements available in the **command** mode are covered in "Command Mode Statements" on Page 5-27 and "Statements Valid in the Command or Program Mode" on Page 5-33.

BASIC STATEMENTS

A program is composed of one or more lines or "statements" instructing BASIC to solve a problem. Each program line begins with a line number identifying the line and its statement. The line number indicates the desired order of statement execution. Each statement starts with an English word specifying the operation to be performed. Single statements are terminated with the return key. Multiple statements are separated by a colon (:) with the last statement terminated by a return (a non-printing character). A DATA statement cannot share a line with other statements. (See Page 5-59.)

Line Numbers

An integer number begins each line in a BASIC program. BASIC executes the program statements in numerical sequence, regardless of the input order. Statement numbers must lie in the range of 1 to 65,534. It is good programming practice to number lines in increments of 5 of 10 to allow insertion of forgotten or additional statements when de-bugging the program.

The length of a BASIC statement must not exceed one line. There is no method to continue a statement to a following line. However, multiple statements may be written on a single line. In this situation, each statement is separated by a colon. For example:

```
10 PRINT "VALUES",A,A+1 is a single line print statement, whereas
10 LET A=12: PRINT A,A+1,A+2 is a line containing two statements, LET and PRINT.
```

Virtually all statements can be used anywhere in a multiple statement line. There are, however, a few exceptions. They are noted in the discussion of each statement. NOTE: Only the first statement on a line can have a line number. Program control cannot be transferred to a statement **within** a line, but only to the beginning of a line.

Each time you type a statement with a line number, BASIC performs some simple syntactical checks before inserting the line into your program. BASIC checks to see if all of the keywords are spelled correctly, and translates them to upper case. It makes sure that all function calls are immediately followed by an open parenthesis "(".BASIC makes several other checks of the line to check for simple syntax errors. If the line is determined to be incorrect, the message

```
SYNTAX ERROR
```

will be typed and the line will not be inserted into your program. Note that this preliminary syntax check will not detect all possible errors; BASIC may accept the line when you type it and then detect an error later when you execute your program.

Statement Types

BENTON HARBOR BASIC supports three different types of statements. First, there are statements valid only in the command mode. These statements are used for loading programs, erasing memory, and other such functions directing BASIC's activities. Second, there are statements valid as both commands or within a program. Third, there are statements valid only within a program. These statements may not be used in the command mode. Most statements fall into the second category. This means they can appear within a program or be typed directly in the command mode and immediately executed.

As noted earlier, some statements valid in both modes may not be meaningful in both modes.

BASIC is designed to allow maximum versatility in its structure. Thus, almost everywhere that BASIC requires a number or a string, BASIC allows a numeric or a string **expression**. For example, you could cause the SIN of 3 to be printed by typing

```
PRINT SIN(6/2)
```

The following three sections are organized as command mode statements, command and program mode statements, and program mode statements. They can be found, respectively in: "Command Mode Statements" (Page 5-27). "Statements Valid in the Command or Program Mode" (Page 5-33), and "Program Mode Statements" (Page 6-63).

To simplify some practical descriptions in these sections and those following, the notations below are used to describe allowed expressions:

1. "iexp" indicates an integer expression, an expression lying in the range of 0 to 65535. The fractional part of any integer expression is discarded when the integer is formed.
2. "nexp" indicates a numeric expression. This may be an integer, decimal, or exponential expression with up to 6 decimal places.
3. "sexp" indicates a string expression. String expressions are limited to a maximum of 255 printing ASCII characters.
4. "linnum" indicates a line number. This must be an unsigned decimal number, or the expression LNO (iexp). See the discussion of the LNO function for more information.
5. "sep" indicates a separator. Separators such as the comma and the semi-color are used to delineate certain portions of BASIC statements.
6. "[ ]" brackets indicate optional portions of a statement, depending on the exact function desired.

7. "var" indicates a variable. This may be a numeric or string variable, depending upon the example.
8. "name" indicates a string used to identify a date, a program, or a language record.
9. "fname" indicates an HDOS file descriptor ("file name"). A file descriptor may include a device specification, and a file name and extension. The device specification and extension may be omitted, in which case BASIC will supply a default.

Command Mode Statements

The command mode statements cannot be used within a program. For example, the RUN statement cannot be used within a program to make it self-starting. Any attempt to incorporate one of these statements within a program generates a USE error message.

BUILD

This statement is used to insert or replace many program lines. The form of the BUILD statement is

```
BUILD iexp1, iexp2 ↵      where  iexp1 ≡ Starting number of build sequence.
                                 iexp2 ≡ Increment.
```

When BUILD is executed, the initial line number iexp 1 is displayed on the terminal. Any text entered after the new line number is displayed becomes the new line, replacing any pre-existing line. Once the line is completed by a carriage return, the next line number is displayed. NOTE: If a null entry is given (a carriage return typed directly after the line number is displayed), the line whose number is displayed is eliminated if it existed.

Build is illustrated in the following example. CTRL-C terminates BUILD.

```
*BUILD 100,10 ↵
 100 PRINT "LINE 100" ↵
 110 PRINT "LINE 110" ↵
 120 PRINT "LINE 120" ↵
 130 ^C                    (CTRL-C typed here)
*LIST ↵
 100 PRINT "LINE 100"
 110 PRINT "LINE 110"
 120 PRINT "LINE 120"
*
```

BASIC performs a preliminary syntax check on lines entered via BUILD. Should an error be detected, BUILD will give an error message. For example:

```
*BUILD 10,10 <CR>
 10 PRINT "LINE 10" <CR>
 20 PRANT "LINE 20" <CR>          (note the error)
SYNTAX ERROR
 20 PRINT "LINE 20" <CR>          (re-enter the line 20)
 30
```

BYE

The BYE command is used to terminate BASIC and return to HDOS command mode. BYE will not save your program, close your files, or in any other way clean up. If you want to save the program you have written, use SAVE or REPLACE before using BYE. BYE will ask you if you are sure before terminating. For example:

```
*BYE <CR>
SURE?YES <CR>
>
```

CONTINUE

CONTINUE begins or resumes the execution of a BASIC program. CONTINUE has the unique feature of not affecting any existing variable values, nor does it affect the GOSUB or FOR stack. CONTINUE is normally used to resume execution after an error in the program or after a CTRL-C stops the program. CONTINUE may be used to enter a program or a specific line (in conjunction with a GOTO). CONTINUE is unlike RUN, which resets all variables, stacks, etc.. The form of the CONTINUE statement is:

```
CONTINUE <CR>
```

In the following example, CONTINUE starts the program at a specific line number.

```
*GOTO 100 ↵
*CONTINUE ↵      (start execution at line 100)
```

CONTINUE is also useful for entering a program with a variable or variables set at particular values. For example:

```
*A = 23.5 ↵      (Program continues execution at Line 230
*GOTO 230 ↵      with variable A set to the value 23.5,
*CONTINUE ↵      regardless of previous program effects on A.)
```

DELETE

The DELETE statement is used to remove several lines from the BASIC source program. The form of the DELETE statement is

```
DELETE iexp1, iexp2 ↵
```

The lines between and including iexp1 and iexp2 are deleted.

A syntax error is flagged if "iexp1" is greater than "iexp2." Normally, DELETE is used to eliminate a number of lines of text. The SCRATCH command is used to eliminate all text. A RETURN typed directly after a line number eliminates that line. This technique is used to eliminate a single line.

LIST

This command lists the program on the console terminal for reviewing, editing, etc. The form of the list command is:

```
LIST [LINNUM1], [LINNUM2]    ↵
```

Line numbers are indicated by the optional integer expressions. If no line numbers are specified, the entire program is listed. If a single line number ("iexp1") is specified, EX. B.H. BASIC lists that single line. You can use a CTRL-O or CTRL-C to abort the listing. If both of the optional line numbers are specified, separated by a comma (,), all lines within the range of iexp1 to iexp2 are listed. You can abort a listing by using the control characters.

The following are examples of the LIST command.

```
 10 LET A=5:LET B=6
 20 PRINT A,B,A+B,
 30 LET C=A/B
 40 PRINT C
 50 END
*RUN  CR
 5    6    11     .833333

 END AT LINE 50
*LIST  CR

 10 LET A=5:LET B=6
 20 PRINT A,B,A+B,
 30 LET C=A/B
 40 PRINT C
 50 END
*LIST 20  CR

 20 PRINT A,B,A+B,
*LIST 20,40  CR

 20 PRINT A,B,A+B,
 30 LET C=A/B
 40 PRINT C
*
```

OLD

The OLD command is used to read some pre-existing program into BASIC. OLD performs a SCRATCH command, destroying the previous program before reading in the new one. The format for the OLD command is:

```
OLD "fname"  CR
```

where "fname" is the file name of the program to be loaded. If no device code is specified, BASIC assumes SY$\emptyset$:. If no extension is specified, BASIC assumes .BAS. For example:

```
*OLD "DEMO"  CR
*OLD "SY1:STARTREK.GAM"  CR
```

If you want to load a new program without disturbing your variables and their values, use the CHAIN command.

BASIC performs a preliminary syntax check on lines read in via the OLD command, just as it would for lines you type yourself on the console. Should the OLD command detect any such syntax errors in the lines being read, it will insert the characters *ERR* at the spot in the line the error was detected. This should never occur with programs which you have entered and modified with BASIC, since

BASIC will not let you type lines with such errors. However, such errors could occur is you used the text editor, EDIT, to modify or create a BASIC program.

You can detect such occurrences by listing the program and looking for the '*ERR*' symbol. Executing a line with the *ERR* symbol in it will generate a syntax error.

REPLACE

The REPLACE command is identical to the SAVE command except that REPLACE will allow you to replace a pre-existing file. See the SAVE command discussion below for more information. The syntax of the REPLACE command is:

```
REPLACE "fname" ⏎
```

The default device is SY0:, the default extension is .BAS. Note that you can use the REPLACE command to obtain a copy of the current program. For example, if you had a hard-copy terminal configured as an alternate terminal (device AT:), the command

```
REPLACE "AT:" ⏎
```

would cause BASIC to write the source for the program to the AT: terminal, thus giving you a hard-copy listing. The SAVE command cannot be used to do this since SAVE opens the file specified for read to see if it already exists. HDOS will tell the SAVE command that the file AT: does exist, and SAVE will then inform you of your "error."

RUN

A prepared program may be executed using the RUN statement. The program is executed starting at the lowest numbered statement. All variables and stacks are cleared (set to zero) before program execution starts.

The form of the RUN statement is:

```
*RUN ⏎
```

After program completion, BASIC prompts the user with an asterisk (*) in the left margin, indicating that it is ready for additional command statements. If the program should contain errors, an error message is printed that indicates the error and the line number containing the error, and program execution is terminated. Again, a prompt is given. The program must now be edited to correct the error and rerun. This process is continued until the program runs properly without producing any error messages. See "Errors" (Page 5-79) for a discussion of error messages.

Occasionally, a program contains an error that causes it to enter an unending loop. In this case, the program never terminates. The user may regain control of the program by typing CTRL-C. This aborts the program and returns control to the user. Storage is not altered in this process. CONTINUE resumes program execution. RUN clears the storage and restarts program execution.

SAVE

The SAVE command is used to save a BASIC program as an HDOS file. The file can then be listed or copied onto different devices, edited by the text editor, and reread by BASIC (via the OLD command). The SAVE command is the normal method of saving a program that you might want to use again. The format of the SAVE command is:

```
SAVE "fname"  ®
```

where "fname" is the name of the file which is to be written. If no device is specified, BASIC assumes SY∅:. If no extension is specified, BASIC assumes .BAS. NOTE: The file fname must not already exist on the specified device. BASIC will not allow you to replace a file with the SAVE command. This is done so you will not accidentally use the same name for two programs and inadvertently destroy one of them. If you wish to store an updated version of a program, you can delete the old version first via UNSAVE, or you can use the REPLACE command. For example:

```
*SAVE "SY1:INCOMETX"  ®
*LIST 10  ®
00010 PRINT "HI THRER"              (note the error)
*10 PRINT "HI THERE"  ®             (correct the error)
*SAVE "SY1:INCOMETX"  ®             (attempt to replace program)

! ERROR - FILE ALREADY EXISTS
*REPLACE "SY1:INCOMETX"  ®          (replace program)
*
```

Remember, you can only use "SY1:" if you have a multiple-drive system.

SCRATCH

SCRATCH clears all current storage areas used by BASIC. This deletes any commands, programs, data, strings, or symbols currently stored by BASIC.

SCRATCH should be used for entering a new program from the terminal keyboard to ensure that old program lines are not mixed with new program lines. It also assures a clear symbol table. The form of the SCRATCH statement is:

```
*SCRATCH  ®
```

Before destroying stored information, the user is asked "SURE?". A "Y" reply causes SCRATCH to proceed. Any other response cancels SCRATCH. For example:

```
*SCRATCH ®        (Scratch statement entered.)
 SURE? Y ®        (Are you sure, answer Y (YES,)
*                 (BASIC is ready for a new entry.)
```

Statements Valid in the Command or Program Mode

You may use the statements in this section in either the command or the program mode. A few of them have only subtle uses in one mode or the other. Because they may be used in both modes, they are listed in this section.

CHAIN

The CHAIN command is used to start the execution of another BASIC program. The format of the CHAIN command is:

```
CHAIN sexp ®          (or)
CHAIN sexp,linnum ®
```

where "sexp" is a string expression containing the file name of the program to be executed. If no device is specified, BASIC assumes SY0:. If no extension is specified, BASIC assumes .BAS.

The CHAIN command causes the current program text to be deleted, the new program to be read in, and execution to begin. If no line number is specified, execution begins at the first line of the new program. If a line number is specified, execution begins at that line number. Note that the GOSUB and FOR loop tables are cleared by the CHAIN process, but no data values (numeric and string variables and arrays) are affected by the CHAIN. However, the data pointer is reset to the top of the data statements. In addition, user-defined functions are undefined, and the random process is restarted. Open data files not affected.

You can use the CHAIN command in the command mode as a quick way to load and execute a program. For example,

```
*CHAIN "DEMO" ®
HI, I'M A BASIC DEMO PROGRAM!
          (etc.)
```

You can use the CHAIN command in the execution mode to start a different program executing, while maintaining any open files and data values. Thus, a program that is too large to fit in memory all at once can be written in several sections, with each section chaining to the next one when ready. As an example, assume we have written a payroll maintenance program that is too large to all fit in memory. This program can perform 5 different functions upon the payroll file. One of these functions might be "add an employee", another one "print monthly checks", and so forth. Because the entire program will not fit in memory at one time, we have split it into five pieces, each of which performs one of the five functions. A section of the program might look like:

```
00020 DIM A$(4)
00030 A$(0)="SY1:PAYROL1.BAS"
00040 A$(1) = "SY1:PAYROL2.BAS"
              .
              .
              .
02000 INPUT "WHAT FUNCTION (1-5)",F
02010 CHAIN A$(F-1)
```

This program inputs a number from the operator, indicating which function is to be performed, and then CHAINs to the appropriate program.

The value of A$ and the values of all other variables are preserved during the CHAIN. In this example, the individual service programs CHAIN back to the master program when done, with a statement

```
CHAIN "PAYROLL",2000 ↵
```

so the PAYROLL program does not start over at the beginning, but instead, starts at line 2000.

CLEAR

CLEAR sets the contents of all variables, arrays, string buffers, and stacks to zero. The program itself is not affected. The command is generally used before a program is rerun to insure a fresh start if the program is started with a command other than RUN. The form of the CLEAR statement is:

```
CLEAR ↵
CLEAR varname ↵
```

All variables, arrays, string buffers, etc., are cleared before a program is executed by RUN. Therefore, a clear statement is not required. However, a program terminated prior to execution (by a STOP command or an error) does not set these variables, etc., to zero. They are left with the last value assigned. If the variable name (varname) is specified, the CLEAR command clears the named variable, array, or DEF FN (user defined function).

Note that the memory space used by string variables and arrays is not freed when CLEAR varname is used. String values should be set to null (for example, A$ = "") before clearing so the string space can be recovered.

For example:

CLEAR A ↵	Clears variable A
CLEAR A$ ↵	Clears the string variable A$
CLEAR A(↵	Clears the dimensioned variable A(

If a section of the program is to be rerun after appropriate editing, the variables, arrays, dimensions, etc., should be reinitialized. You can accomplish this by using the CLEAR statement in the command mode.

CLOSE

The CLOSE statement is used to close an HDOS file. To read or write to a file, three things must be done in sequence:

1. The file must be opened (see OPEN).
2. The I/O is performed (via "INPUT #chan" or "PRINT #chan").
3. The file must be closed.

The format of the CLOSE statement is

```
CLOSE #chan1 ↵                    (or)
CLOSE #chan1, . . . ,#chann ↵
```

where "#chan" is the channel number assigned to the file when it was opened. The CLOSE command does three things:

1. If the file was OPENed for writing, the new file is entered into the disk's directory. If the file is not closed, it, and the information written to it, are lost.

2. The BASIC channel number is freed so a different file may be OPENed on that channel.
3. If there are no open channels with numbers higher than the one being closed, the buffer space in the FILE table (see the FREE command) is freed up. That is, if channels 1 and 2 are open, and you close 1, then no FILE table space is freed. When you later close channel 2, then the FILE table space for both channels 1 and 2 is freed.

If your program blows up without closing its channels, you may want to type CLEAR to discard the partially written files. If you want to save any partial files, use CLOSE in command mode to close the files.

If the channel number(s) listed in the CLOSE command have not been opened or have already been closed, they are ignored.

CNTRL (CONTROL)

Control is a multi-purpose command used to set various options and flags. The form of the CONTROL statement is:

```
CNTRL iexp1, iexp2 ↵
```

The various CNTRL options are:

	iexp1	iexp2
CNTRL	0,	nnn
CNTRL	1,	n
CNTRL	2,	n
CNTRL	3,	n
CNTRL	4,	n

CNTRL 0

The CNTRL 0, nnn command sets up a GOSUB routine to process CTRL-B characters. The line number of the routine is specified as "iexp2." When a CTRL-B is entered from the terminal program, control is passed to the specified statement (beginning at the line iexp2) via a GOSUB linkage, after the statement being executed is completed. For example:

```
00010 CNTRL 0,500
00020 FOR A=1 TO 9
00030 PRINT A,A*A,A*A*A
00040 NEXT A
00050 END
00500 PRINT "THAT TICKLES"
00510 RETURN
*RUN ®
 1            1              1
 <CTRL-B> 2       4               8
THAT TICKLES
 3            9              27
 4            16 <CTRL-B>        64
THAT TICKLES
 5            25             125
 6            36             216

<CTRL-B>THAT TICKLES
 7            49    <CTRL-B>  343
THAT TICKLES
 8            64             512
 9            81             729
END AT LINE 50
*
```

During the execution of the program containing these three statements, a CTRL-B from the keyboard momentarily interrupts execution for the program. The program completes the line in progress and then enters the subroutine at line 500 printing the string.

```
THAT TICKLES
```

It then moves to the next statement, a RETURN. This causes the program to continue with normal program execution. NOTE: The CNTRL 0, nnn must be executed before it is operational.

CNTRL 1

The CNTRL 1, n command sets the number of digits permitted before the exponential notation is used. Normal mode N = 6. For example:

```
*CNTRL 1,2 ↵        (Numbers ≥ 100 are to be in exponential format.)

*PRINT 101 ↵
1.01000E+02
```

CNTRL 2

The CNTRL 2, n command controls the H8 front panel LED display mode. The control functions are:

CNTRL 2,0 ↵	*Turn display off (Normal mode).*
CNTRL 2,1 ↵	*Turn display on without update. (For writing into a display. See the example under "The SEG Function, SEG (NARG)" on Page 5-72.")*
CNTRL 2,2 ↵	*Turn display on with update (to monitor a register or memory location).*

NOTE: The CNTRL 2,n command has no effect on an H89, since there is no front panel display on this model.

CNTRL 3

The CNTRL 3, n command controls the size of a print zone. This is normally 14. However, CNTRL 3, n can change the number of spaces in a print zone.

```
*CNTRL 3,5 ↵
*PRINT 1,2,3,4,3,2,1,0 ↵
   1     2     3     2     3     4     3     2     1     0
```

CNTRL 4

The CNTRL 4, n command is used to control the HDOS Operating System's overlay handling. Part of the HDOS system does not reside permanently in RAM, but is kept on the disk in SYØ:. When it is needed, it is read into memory temporarily. This section of HDOS is called the "overlays", and is used when files are opened and closed. The statement

```
CNTRL 4,1 ↵
```

will cause these HDOS overlays to remain in memory permanently. This will greatly speed up the execution of the RUN, SAVE, UNSAVE, OLD, REPLACE, OPEN, and CLOSE statements, at the cost of about 2.5K bytes of free RAM. Executing the statement

```
CNTRL 4,0 ↵
```

restores HDOS to its normal mode and allows BASIC to make use of that 2.5K bytes of RAM. When you first run BASIC, it starts up in the CNTRL 4,0 mode. Users with sufficient free space will find a significant speed increase by using the CNTRL 4,1 command.

NOTE: The CNTRL 4,n command cannot be executed as a program statement. If you want to "lock" the overlays in memory, do so before executing the program. Good programming practice dictates that you do a CNTRL 4,n command prior to putting the program into memory.

DIM (DIMENSION)

The DIMENSION statement explicitly defines the maximum dimensions of array variables. A single dimension array is often called a vector. The form of the DIMENSION statement is:

```
*DIM varname (iexp1, . . . . ,iexpn) ,varname2 ( . . . .) ↵
```

The expressions "iexp1" through "iexpn" are integer expressions specifying the bounds of each dimension. Dimensions are 0 to "expn." So, for example, the statement:

```
DIM A(5,5) ↵
```

reserves an array 6×6 or 36 values. If the dimensioned variable is numeric, the values are preset to zero. If the dimensioned variable is a string, all the values are preset to a null string.

You may declare several variables in one DIMENSION statement by separating them with commas. For example:

```
*DIM A6(3,2), B(5,5), C3(10,10) ↵
```

dimensions the following arrays

VARIABLE		SIZE
A6	4 by 3	12 elements
B	6 by 6	36 elements
C3	11 by 11	121 elements

You can place a DIMENSION statement anywhere in a multiple statement line and it can appear anywhere in the program. However, an array can only be dimensioned once in a program unless it is cleared. DIMENSION statements must be executed before the first reference to the array, although good programming practices place all DIMENSION statements in a group among the first statements of a program. This allows them to be easily identified and changed if alterations are required later. The following example demonstrates the use of the DIMENSION statement with subscripted variables and a two-level FOR statement.

```
*LIST ®
 10 REM DIMENSION DEMO PROGRAM
 20 DIM A(5,10)
 30 FOR B=0 TO 5
 40 LET A(B,0)=B
 50 FOR C=0 TO 10
 60 LET A(0,C)=C
 70 PRINT A(B,C);
 80 NEXT C:PRINT :NEXT B
 90 END

*RUN ®
 0  1  2  3  4  5  6  7  8  9  10
 1  0  0  0  0  0  0  0  0  0  0
 2  0  0  0  0  0  0  0  0  0  0
 3  0  0  0  0  0  0  0  0  0  0
 4  0  0  0  0  0  0  0  0  0  0
 5  0  0  0  0  0  0  0  0  0  0
 END AT LINE 90
*
```

FOR AND NEXT

FOR and NEXT statements define the beginning and end of a program loop. A program loop is a set of repeated instructions. Each time they are repeated they modify a variable in some way until a predetermined condition is reached, causing the program to exit from the loop. The FOR NEXT statement is of the form:

```
FOR var = nexp1 to nexp2 [STEP nexp3]
NEXT var
```

When BASIC encounters the FOR statement, the expressions nexp1, nexp2, and nexp3 (if present) are evaluated. The variable "var" may be a scaler numeric variable, or it may be an element of a numeric array. It is assigned a value of "nexp1." For example:

```
*FOR A=2 TO 20 STEP 2:PRINT A;:NEXT A  CR
 2  4  6  8  10  12  14  16  18  20
```

causes the program to execute as long as A is less than or equal to 20. Each time the program passes through the loop, the variable A is incremented by 2 (the STEP number). Therefore, this loop is executed a total of 10 times. When incremented to 22, program control passes to the line following the associated NEXT statement. It is important to note that the initial value used for the variable is the value assigned to the variable expression when it entered the FOR-NEXT loop. For example:

```
*A=10:FOR A=2 TO 20 STEP 2:PRINT A;:NEXT A  CR
 2  4  6  8  10  12  14  16  18  20
*
```

Prior to execution, the variable A is assigned the value 10. The program passes through the loop 10 times. A is reset to 2 and then increments from 2 to 20.

If "nexp2" $\geq$ 0, and the initial value of var $\geq$ "nexp2", the loop terminates. For example, the program:

```
*LIST  CR
 10 FOR J=2 TO 18 STEP 4
 20 J=18
 30 PRINT J;:NEXT J
 40 END

*RUN  CR
 18
 END AT LINE 40
*
```

is only executed once, since the value of J = 18 is reached on the first pass, satisfying the termination condition.

A loop created by the statement:

```
*FOR A=20 TO 2 STEP 2:PRINT A;:NEXT A  ↵
 20
*
```

is executed only once, as the initial value exceeds the terminal value. However, if this example is modified to read:

```
*FOR A=20 TO 2 STEP -2:PRINT A;:NEXT A  ↵
 20  18  16  14  12  10  8  6  4  2
*
```

the negative step allows normal operation.

In summary, for positive STEP values, the loop is executed until the variable (var) is greater than the final assigned value (nexp2). For negative STEP values, the loop is executed until the variable (var) is less than the final assigned value (nexp2).

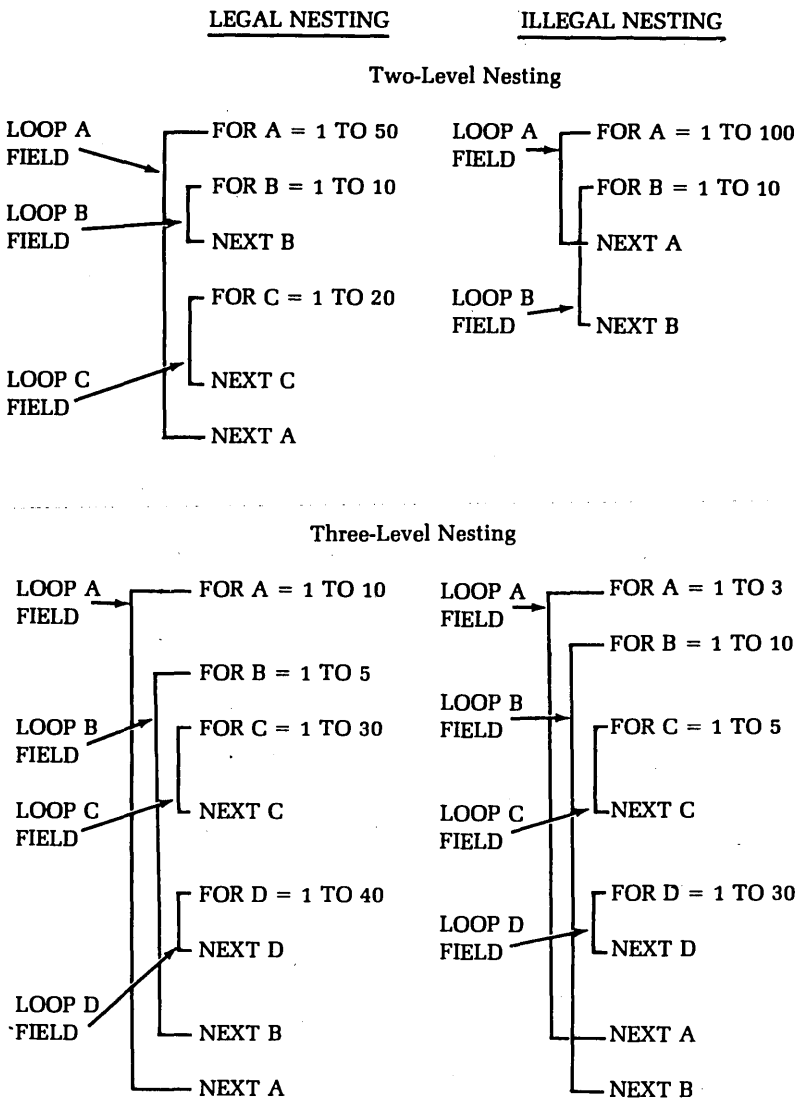
If the loop does not terminate, execution is transferred to the statement following the FOR statement. Therefore, a series of statements may be executed using the incremented value of the variable. If the loop does terminate, execution is transferred to the statement following NEXT.

The expressions in the FOR statement can be any acceptable BASIC numeric expressions.

If the STEP expression and the word STEP are omitted from the FOR statement, a step of +1 is the default value. Since +1 is an extremely common step value, the STEP portion of the statement is frequently omitted. For example:

```
*FOR A=2 TO 10:PRINT A;:NEXT A  ↵
 2  3  4  5  6  7  8  9  10
*
```

Nesting is a technique frequently used in programming. It consists of placing one or more loops completely inside another loop. The field or operating range of the loop (the lines from the FOR statement to the corresponding NEXT statement) must not cross the field of another loop. The following two examples show legal and illegal nesting of FOR NEXT loops.



Note that both columns of nesting illustrations are shown in two-level and three-level forms. However, right-hand columns are not truly nesting but a crossover of FOR and NEXT loops (fields), and therefore are illegal. Also note that each of these examples uses the implied STEP value of 1.

The depth of nesting depends upon the amount of memory space available.

It is possible to exit from a FOR NEXT loop without reaching the variable termination value. This can be done using a conditional transfer such as an IF statement within the loop. However, control can only be transferred into a loop if the loop is left during prior program execution without being completed. This ensures the assignment of values to the termination and step variables.

Both FOR and NEXT statements can appear anywhere on a multiple statement line.

The NEXT statement does not require the variable. If the variable is not given, BASIC will NEXT the innermost FOR loop.

FREE

The FREE statement displays the amount of memory used by EX. B.H. BASIC and any program material. It also displays the total amount of free space left, which is dependent on the amount of memory in the computer and the program size. This command is particularly valuable when you are gauging the size of the program's data structure and establishing limits on a DIMENSION command. The FREE command also indicates the cause of memory overflow errors. The form of the FREE statement is:

```
*FREE (CR)
```

The form of the printout is:

```
TEXT = nnnn     (Bytes used by program text.)
SYMB = nnnn     (Bytes used by variables and arrays.)
FORL = nnnn     (Bytes used by FOR loops.)
GSUB = nnnn     (Bytes used by GOSUBs.)
STRN = nnnn     (Bytes used by STRINGs.)
WORK = nnnn     (Bytes used by expression and function evaluation.)
FILE = nnnn     (Bytes used by file buffers.)
FREE = nnnn     (Total number of free bytes.)
```

For example, running the program

```
*10 GOSUB 10 ↵
```

BASIC soon returns a memory overflow error. Executing FREE shows the user a very large GOSUB table. This, and the statement provided in the error message, enables one to determine the program is in a GOSUB loop.

```
*FREE ↵
 TEXT = 9
 SYMB = 0
 FORL = 0
 GSUB = 0
 WORK = 0
 STRN = 0
 FILE = 0
 FREE = 7248
*10GOSUB 10 ↵
*RUN ↵

! ERROR - MEM OVR AT LINE 10
*FREE ↵
 TEXT = 9
 SYMB = 0
 FORL = 0
 GSUB = 7515
 WORK = 0
 STRN = 0
 FILE = 0
 FREE = 16
*
```

Note that the FILE table never requires less than 283 bytes. This table contains the disk file buffers necessary to read and write files. The 283 bytes are required for BASIC's internal buffer, which it uses for such commands as OLD, SAVE, and REPLACE.

You can compute the amount of space used by the FILE table with the formula:

bytes = N * 256

where N is the number of the highest-numbered channel that is open. Thus, when your program opens files, it should open them on the lowest numbered channels first. If you open a file on channel 3, space is reserved for the buffers for channels 1 and 2, even if they are never opened.

FREEZE

The FREEZE command is used to store BASIC, your program, and all of your program's variables on SY∅: or SY1:. The format of the command is:

```
FREEZE "fname"  ↵
```

where "fname" is the file name under which the frozen program will be stored. If no device is specified, BASIC assumes SY∅:. If no extension is specified, BASIC assumes .BAF (for BASIC Frozen). Of course, you cannot specify a device of "SY1:" unless you have a second drive installed on your computer system.

The FREEZE command allows you to suspend work temporarily; perhaps to power-down overnight or to allow some more important work to interrupt. This command is **not** intended as a general-purpose, program-save command; the SAVE and REPLACE commands are provided for normal program saving. The file created by the FREEZE command is in absolute binary format and cannot be displayed or edited. Its sole use is to be unfrozen with the UNFREEZE command.

The file is quite large because it contains all of the BASIC interpreter in addition to your program and variables. Frozen programs are non-transferrable, in that they cannot be unfrozen by a different version of BASIC than the one they were frozen with.

NOTE: All files must be closed before a program is saved via FREEZE.

GOSUB AND RETURN

A subroutine is a section of program performing some operation required one or more times during program execution. Complicated operations on a volume of data, mathematical evaluations too complex for user-defined functions, or a previously written routine are all examples of processes best performed by a subroutine.

More than one subroutine is allowed in a single program. Good programming practices dictate that subroutines should be placed one after another at the end of the program in line number sequence. A useful practice is to assign distinctive line number groups to subroutines.

For example, a main program uses line numbers through 300. The 400 block is assigned to subroutine #1 and the 500 block is assigned to subroutine #2. Thus, any errors, program modifications, etc., involving the subroutine are easily identified.

Subroutines are normally placed at the end of a program, but before data statements if there are any.

Program execution begins and continues until a GOSUB statement is encountered. The form of the GOSUB statement is:

```
*GOSUB LINNUM ↵
```

where LINNUM is the line number of the first line in the subroutine. Once GOSUB is executed, program control transfers to the first line of the subroutine and the subroutine is executed. For example:

```
60 GOSUB 500 ↵
```

in this example, control (the sequence of program execution) is transferred to line 500 in the program after line 60 is executed. The first line in the subroutine may often be a remark to identify the subroutine, or it may be any executable statement.

Once program control is transferred to a subroutine, program execution continues in the normal line-by-line manner until a RETURN statement is encountered. The RETURN statement is of the form:

```
RETURN ↵
```

RETURN causes the program control to return to the statement **following** the original GOSUB statement. A subroutine must always be terminated by a RETURN.

Before BASIC transfers control to a subroutine, the next sequential statement to be processed after the GOSUB statement is internally recorded. The RETURN statement draws on this stored information to restart normal program execution. Using this technique, BASIC always knows where to transfer control, no matter how many times subroutines are called.

Subroutines can be nested in the same manner that FOR NEXT statements can be nested. That is, one subroutine can call another subroutine, and if necessary, that subroutine may call a third subroutine, etc. If, during execution of the subroutine a RETURN is encountered, control is returned to the line following the GOSUB calling the subroutine. Therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and can have more than one RETURN. Multiple RETURN statements are often necessary when a subroutine contains conditional statements imbedded in it, which cause different subroutine completions dependent on the program data.

It is possible to transfer to the beginning or to any part of the subroutine. Multiple entry points and returns make the GOSUB statement an extremely versatile tool.

Extended BASIC permits unlimited GOSUB nesting. However, nesting uses memory and excessive nesting depth will cause an overflow.

GOTO

The GOTO statement provides unconditional transfer of program execution to another line in the program. The GOTO statement is of the form:

```
*GOTO LINNUM ㊡
```

When this statement is executed, program control transfers to the line number specified by LINNUM. For example:

```
 10 LET A=1
 20 GOTO 40
 30 LET A=2
 40 PRINT A
 50 END
*RUN ㊡
 1

 END AT LINE 50
*
```

Program lines in this example are executed in the following order:

10, 20, 40, 50

Line 30 is never executed because the GOTO statement in line 20 unconditionally transfers control to line 40. After the unconditional transfer takes place, normal sequential execution resumes at line 40.

IF THEN (IF GOTO)

The IF THEN (IF GOTO) conditionally transfers program execution from the normal consecutive order of program lines, depending on the results of a relation test. The forms of the IF statement are:

```
IF expression {THEN} LINNUM ㊡      or
              {GOTO}
IF expression THEN statement ㊡
```

The expression frequently consists of two variables combined by the relational operators described in "Relational Operators" (Page 5-86). In the first form, if the result of the expression is true, control passes to the specified line number (LINNUM). In the second form, control passes to the statement following THEN on the remainder of the line. If the result of the expression is false, control passes to the next line. The following examples show use of the IF THEN statement.

```
 10 READ A
 20 B=10
 30 IF A=B△THEN 50
 40 PRINT "A< >B",A:END
 50 PRINT "A=B",A
 60 DATA 10,5,20
 70 END

*RUN      ®
A=B      10

END AT LINE 70
*CONTINUE     ®
A< >B  5

END AT LINE 40
*CONTINUE     ®
A< >B  20

END AT LINE 40
*
```

NOTE: The expression can be an arbitrarily complex expression. For example:

```
IF (A<3) AND NOT (B>C) THEN 33
```

LET

The LET statement assigns a value to a specific variable. The form of the LET statement is:

```
LET var = nexp                    or
LET var$ = sexp
```

The variable "var" may be a numeric variable or a string variable "var$." The expression may be either an arithmetic "nexp" or a string expression "sexp." However, all items in a statement must be either numeric or string, they cannot be mixed. If they are mixed, a type conflict error is flagged. NOTE: Unlike standard BASIC, multiple assignments are not permitted. For example,

```
LET A=B=3 ®
```

causes A to be set to 65,535 (true) if B is equal to 3, or it causes A to be set to 0 (false) if B is not equal to 3. It does not cause both A and B to be set to 3.

You may omit the key word LET if you prefer. For example, the following two statements produce identical results.

```
10 LET A = 6          or
10 A = 6
```

The LET statement is often referred to as an assignment statement. In this context, the meaning of the equal (=) symbol should be understood as it is used in BASIC. In ordinary algebra, the formula Y = Y + 1 is meaningless. However, in BASIC, the equal sign denotes replacement rather than equality. Thus, the formula Y = Y + 1 is translated as add 1 to the current value of Y and store the new result at the location indicated by the variable Y.

Any values previously assigned to Y are combined with 1. An expression such as D = B + C instructs BASIC to add the values assigned to the variables B and C and store the resultant value at the location indicated by the variable D. The variable D is not evaluated in terms of previously assigned values, but only in terms of B and C. Therefore, if previous assignments gave D a different value, the prior value is lost when this statement is executed.

LOCK

The LOCK statement protects your program by preventing you from executing of the following command mode statements:

BUILD	CLEAR	SCRATCH
BYE	DELETE	UNFREEZE
CHAIN	RUN	

It also prevents the entry or deletion of program text. Variables can be changed, but not deleted. The form of the LOCK statement is:

```
*LOCK ↵
```

A lock error (LOCK) is generated if you attempt to enter a "locked out" command mode statement, such as RUN. Use the UNLOCK statement to abort the LOCK mode.

ON . . . GOSUB

The ON . . . GOSUB statement allows you to program a computed GOSUB. When you use the ON . . . GOSUB statement, use a RETURN at the end of the subroutine to return program control to the statement **following** the ON . . . GOSUB statement. The form of the ON . . . GOSUB statement is:

```
ON iexp1 GOSUB LINNUM1, . . . . . ,LINNUMn    ↵
```

When it is processing an ON . . . GOSUB statement, BASIC evaluates the expression "iexp1" and uses the result as an index to the list of statement numbers LINNUM1 through LINNUMn. If the expression "iexp1" evaluates to 1, for example, control is passed to line number "LINNUM1". If the expression "iexp1" evaluates to 3, for example, control is passed to line number "LINNUM3". If the expression "iexp1," evaluates to 0, or to an index greater than the number of statement numbers listed, control is passed to the next program statement.

ON . . . GOTO

The ON . . . GOTO statement allows you to perform a computed GOTO. The form of the ON . . . GOTO statement is:

```
ON iexp1 GOTO LINNUM1, . . . . . , LINNUMn  ↵
```

When it is processing an ON . . . GOTO statement, BASIC evaluates the expression "iexp1" and uses the result as an index to the list of statement numbers LINNUM1 thru LINNUMn. If the expression "iexp1" evaluates to 1, for example, control is passed to the line number given by the expression "LINNUM1". If the expression "iexp1" evaluates to 3, for example, control is passed to line number given by the expression "LINNUM3". If the expression "iexp1" evaluates to 0, or to an index greater than the number of statement numbers listed, control is passed to the next program statement.

OPEN

The OPEN command is used to open HDOS files so that they can be read or written from BASIC. The format of the OPEN command is:

```
OPEN sexp FOR READ  AS FILE #iexp  ↵          (or)
OPEN sexp FOR WRITE AS FILE #iexp  ↵
```

The first form is used to open files for reading via the INPUT command. The second form is used to open files for writing via the PRINT command.

"sexp" is a string value that contains the HDOS file name. If no device is specified, BASIC assumes SYØ:. Remember, you can only specify "SY1:" for a device if you have a second drive on your system. If no extension is specified, BASIC assumes .DAT. Any legal device may be used. "iexp" represents the channel number that is to be assigned to the file. BASIC has five channels, 1 through 5. This means that you can have a maximum of five files open at one time. You can close a file and then re-use its channel for some other file. After the OPEN statement, the only way to refer to the file is by its channel number; the file name is no longer needed. For example:

```
OPEN "TEMP" FOR WRITE AS FILE #3
OPEN "SY1:RALPH.WRK" FOR READ AS FILE #1
OPEN A$ FOR WRITE AS FILE #1
OPEN "TT:" FOR WRITE AS FILE #2
```

To print or output to the "alternate terminal" device:

```
00010 OPEN "AT:" FOR WRITE AS FILE #1
00020 FOR I=1 TO 10
00030 PRINT #1,I,SQR(I)
00040 NEXT I
00050 CLOSE #1
00060 STOP
00070 END
```

NOTES:

1. Although five channels are available, 1, 2, 3, 4, and 5, you should use the lowest-numbered channel available when opening a file to minimize the amount of memory space required. See the FREE command discussion (Page 5-45) for more information.

2. Although files may be opened to any legal device, including the console terminal (device TT:), you should use the regular INPUT and PRINT statements for communicating with the console. Due to the requirements of HDOS device I/O, BASIC saves up the data you write to a file via PRINT until there are 256 bytes of data, and then writes the 256 bytes all in one group. Likewise, when reading, BASIC reads-ahead a 256 byte block of data and then supplies it as needed to the INPUT #chan statements. Thus, if you write to the console via a channel opened on the device TT:, the lines will not appear on the console when you PRINT them but when BASIC has accumulated 256 bytes worth (or when the file is closed).

OUT

The OUT statement is used to output binary numbers to an output port. The form of the OUT statement is:

```
OUT iexp1, iexp2 ↵
```

The expression "iexp1" is used as the port address, and "iexp2" is the value to be placed at that port. Both iexp1 and iexp2 are decimal numbers. The low-order 8-bits generated by the decimal numbers in iexp1 or iexp2 are used. If you wish to write iexp1 and iexp2 in octal notation for ease in conversion to the actual binary values, write a subroutine or function to perform octal to decimal conversion.

PAUSE

The PAUSE statement causes BASIC to delay before executing the next statement. The form of the pause statement is:

```
PAUSE [iexp]
```

If the optional expression iexp is omitted, PAUSE suspends execution until you type a carriage return. If the expression iexp is present, PAUSE delays 2* iexp milliseconds, and then allows execution to resume. The maximum value for iexp is 30,000, allowing a maximum delay of about 60 seconds.

The PAUSE statement is particularly useful when you are viewing long outputs on a CRT display. You can insert a PAUSE at appropriate points in the program, allowing you to view the information on the CRT before the information scrolls off the screen.

POKE

> WARNING
>
> The POKE function gives an experienced BASIC user direct control of virtually all of the features of the computer. However, subtle misusage of POKE can interfere with the operating system and cause it to cease correct functioning. For this reason, Heath cannot provide consulting support for users who use the POKE function.

The POKE statement is used to place a value in a particular memory location. The form of the POKE statement is:

```
POKE Location, Value
```

The "Location" is a decimal integer in the range of 0 to 65,535. This references any individual byte of a memory location. The "Value" is also an integer expression lying in the range of 0 to 255. You can examine the contents of a memory location by using the PEEK function described on Page 5-70.

PRINT

The PRINT statement is used to output data to the console terminal or to an HDOS file. The form of the PRINT statement is:

```
PRINT [nexp1,sep1, . . [,nexpn, sepn]]              (or)
PRINT #chan, [nexp1,sep1, . . [,nexpn,sepn]]
```

The first form shown is for writing text and values to the console terminal. The second form is for writing values and text to an HDOS file. 'chan' is the channel number of a file which must have been opened for WRITE. See the discussion of the OPEN and CLOSE command for more information. Except for the destination of the data, both forms of the command are otherwise identical.

The expressions and separators contained within the brackets are optional. When used without these optional expressions and separators, the simple

```
PRINT          or
PRINT#CHAN,
```

statement outputs a blank line.

Printing Variables

The PRINT statement can be used to evaluate expressions and to simultaneously print their results, or to simply print the results of a previously evaluated expression or evaluations. Any expression contained in the PRINT statement is evaluated before the result is printed. For example:

```
 10 A=4:B=6:C=5+A
 20 PRINT
 30 PRINT A+B+C
 40 END
*RUN ®

 19

 END AT LINE 40
*
```

All numbers are printed with a preceding and following blank. You can use PRINT statements anywhere in a multiple-statement line. NOTE: The terminal performs a carriage-return line feed at the end of each PRINT statement unless you use the separators described in "Use of the , and ;" (Page 5-57). Thus, in the previous example, the first PRINT statement outputs a carriage-return line feed and the second print statement outputs the number 19 followed by a carriage-return line feed.

Printing Strings

The PRINT statement can be used to print a message (a string of characters). The string may be alone or it may be used together with the evaluation and printing of a numeric value. Characters to be printed are designated by enclosing them in quotation marks ("). For example:

```
 10 PRINT "THIS IS A HEATH COMPUTER"
*RUN ®
 THIS IS A HEATH COMPUTER

 END AT LINE 65535
*
```

The string contained in a PRINT statement may be used to document the variable being printed. For example:

```
 10 LET A=5:LET B=10
 20 PRINT "A + B",A+B
 30 END
*RUN ®
 A + B            15

 END AT LINE 30
*
```

When a character string is printed, only the characters between the quotes appear. No leading or trailing blanks are added as they are when a numeric value is printed. Leading and trailing blanks can be added within the quotation marks.

Use of the "," and ";"

The console terminal is normally initialized with 80 columns divided into five zones. (See CNTRL 3, n for exception.) Each zone, therefore, consists of 14 spaces. When an expression in the PRINT statement is followed by a comma (,) the next value to be printed appears in the next available print zone. For example:

```
 10 A=5.55555:B=2
 20 PRINT A,B,A+B,A*B,A-B,B-A
 30 END
*RUN ®
 5.55554        2              7.55554        11.1111        3.55554
 -3.55554

 END AT LINE 30
*
```

NOTE: The sixth element in the PRINT list is the first entry on a new line, as the five print zones of a 72-character line were used.

Using two commas together in a PRINT statement causes a print zone to be skipped. For example:

```
 10 A=5.55555:B=2
 20 PRINT A,B,A+B,,A*B,A-B,B-A
 30 END
*RUN ®
 5.55554        2              7.55554                       11.1111
 2.55554        -3.55554

 END AT LINE 30
*
```

If the last expression in a PRINT statement is followed by a comma, no carriage-return line feed is given when the last variable is printed. The next value printed (by a later PRINT statement) appears in the next available print zone. For example:

```
 10 LET A=1:LET B=2:LET C=3
 20 PRINT A,
 30 PRINT B
 40 PRINT C
 50 END
*RUN  ®
 1               2
 3

 END AT LINE 50
*
```

At certain times, it is desirable to use more than the designated five print zones. If such tighter packing of the numeric values is desired, a semicolon (;) is inserted in place of the comma. A semicolon does not move the next output to the next PRINT zone, but simply prints the next variable, including its leading and trailing blank. For example:

```
 10 LET A=1:LET B=2:LET C=3
 20 PRINT A;B;C
 30 PRINT A+1;B+1
 40 PRINT C+1
 50 END
*RUN
 1  2  3
 2  3
 4

 END AT LINE 50
*
```

NOTE: If either a comma or a semicolon is the final character in a PRINT statement, no final carriage-return line feed is printed.

READ AND DATA

The READ and DATA statements are used in conjunction with each other to enter data into an executing program. One statement is never used without the other. The form of the statements are:

```
READ var1, . . ., varn
DATA val1, . . ., valn
```

The READ statement assigns the values listed in the DATA statement to the specified variables var1 through varn. The items in the variable list may be simple variable names, arrays, or string variable names. Each one is separated by a comma. For example:

```
 5 DIM A (2,3)
 10 READ C,B$,A (1,2)
 20 DATA 12,THIS IS SIX,56
 30 PRINT C,B$,A (1,2)
*RUN <CR>
 12            THIS IS SIX 56

 END AT LINE 65535
*
```

Because data must be read before it can be used in the program, READ statements generally occur in the beginning of a program. You may, however, place a READ statement anywhere in a multiple-statement line. The type of value in the DATA statement must match the type of corresponding variable in the READ statement. When the DATA statement is exhausted, BASIC finds the next sequential DATA statement in the program. NOTE: BASIC does not automatically go to the next DATA statement for every READ statement. Therefore, one DATA statement may supply values for several READ statements if the DATA statement contains more expressions than the READ statement has variables.

The data values in a DATA statement must be separated by commas. If the value is to be read into a numeric variable or array, it must be a number. If the value is to be read into a string variable or array, no specific format is required. If the value is enclosed in quotes (""), the quoted characters are assigned to the string variable. If the value is not enclosed in quotes, BASIC uses the characters until a comma or the end of the line is reached. Thus, if you wish to read a comma as part of the value, you **must** enclose the value in quotes.

You may not include a quote character in the value. For example:

```
10 READ A$,B$,C$
20 PRINT A$,B$,C$
30 DATA HI THERE,"HI, THERE",YES
*RUN ↵
HI THERE          HI, THERE       YES
```

A field in DATA statement may be left null by means of two adjacent commas. This causes the associated variable to retain the old value. For example:

```
 10 A=1:B=1:C=1
 20 READ A,B,C
 30 PRINT A,B,C
 40 DATA 3,,4
 50 END
*RUN ↵
 3          1          4

 END AT LINE 50
*
```

If a DATA statement appears on a line, it must be the only statement on the line. DATA statements may not follow any other statement on the line. Other statements should not follow DATA statements.

DATA statements do not have to be executed to be used. That is, they may be the last statement in a program, and be used by a READ statement executed earlier in the program. However, if DATA statements appear in a program in such a place that they are executed (there are executable statements beyond the DATA statement), the executed DATA statement has no effect. Therefore, location of DATA statements is arbitrary as long as the values contained within the DATA statements appear in the correct order. However, good programming practice dictates all DATA statements occur near the end of the program. This makes it easy for the programmer to modify the DATA statements when necessary.

If a value contained in a DATA statement is incorrect, the illegal character error message is printed. All subsequent READ statements also cause the message. If there is no data available in the data table for the READ statement to use, the no data error message is printed.

If the number of values in the data list exceed those required by the program READ statements, they are ignored, and thus not used.

REM (REMARK)

The REMARK statement lets you insert notes, messages, and other useful information within your program in such a form that it is not executed. The contents of the REMARK statement may give such information as the name and purpose of the program, how the program may be used, how certain portions of the program work, etc.. Although the REMARK statement inserts comments into the program without affecting execution, they do use memory which may be needed in exceptionally long programs.

REMARK statements must be preceded by a line number when used in the program. They may be used anywhere in a multiple statement line. The message itself can contain any printing character on the keyboard and can include blanks. BASIC ignores anything on a line following the letters REM.

RESTORE

The RESTORE statement causes the program to reuse data starting at the first DATA statement. It resets the DATA statement pointer to the beginning of the program. The RESTORE statement is of the form:

```
RESTORE
```

For example:

```
 10 READ A,B,C
 20 PRINT A,B,C
 30 RESTORE
 40 READ D,E,F
 50 PRINT D,E,F
 60 DATA 1,2,3,4,5,6,7,8
 70 END
*RUN ㊞
 1          2          3
 1          2          3

 END AT LINE 70
*
```

This program does not utilize the last five elements of the DATA statement. The RESTORE command resets the DATA statement pointer and the READ D,E,F statement uses the first three data elements, as does the initial READ statement.

The CLEAR command includes the RESTORE function.

STEP

The STEP command permits you to step through a program a single line or a few lines at a time. The form of the step command is:

```
STEP iexp ®
```

where the integer expression iexp indicates the number of lines to be executed before stopping. Execution of the desired lines is indicated by the prompt NXT = nnnn, where nnnn is the next line number to be executed. A STEP 2 is required to execute the first program line. All future single-line executions require a STEP or STEP 1. For example:

```
 10 READ A,B,C
 20 PRINT A,B,C
 30 RESTORE
 40 READ D,E,F
 50 PRINT D,E,F
 60 DATA 1,2,3,4,5,6,7,8
 70 END

*CLEAR ®

*STEP 3 ®
 1            2            3
 NXT= 30
*STEP ®
 NXT= 40
*STEP ®
 NXT= 50
*STEP ®
 1            2            3
 NXT= 60
*STEP 2 ®

 END AT LINE 70
*
```

UNFREEZE

The UNFREEZE command is used to restore a program that has been frozen with the FREEZE command. See "FREEZE" (Page 5-47) for more information. The format of the UNFREEZE command is:

```
UNFREEZE "fname" ®
```

where "fname" is the name of the previously frozen file. If no device is specified, BASIC assumes SYØ:. If not extension is specified, BASIC assumes .BAS.

UNLOCK

The UNLOCK statement aborts the LOCK mode and restores the use of all command mode statements. The form of the UNLOCK statement is:

```
*UNLOCK ⏎
```

UNSAVE

The UNSAVE command is used to delete programs and files from the disk. The form of the UNSAVE command is:

```
UNSAVE "fname" ⏎
```

where fname is the name of the file to delete. If no device is specified, BASIC assumes SY∅:. If no extension is specified, BASIC assumes .BAS . Unless the file or the disk is write-protected, you can use UNSAVE to delete any file: a BASIC program, a data file, or anything else.

Program Mode Statements

PROGRAM MODE statements are valid only when utilized within a program. If they are entered in the command mode, an illegal use error is flagged.

DATA

The DATA statement discussed in "Read and Data" (Page 5-59) is a program only statement, although it is used in conjunction with a READ statement, which may be used in either the command or program mode.

DEF FN

The DEF FN statement defines single-line program functions created by the user. The form of the DEF FN statement is:

```
DEF FN varname (arg1 [,arg2, . . . . ,argn] ) = expr
```

The variable name (varname) must be a legal string or numeric variable name and cannot be previously dimensioned. However, it may be previously defined. The latest definition takes precedence. The argument list "(arg1 [,arg2,.....,arg3])" must be supplied to indicate a function. Note: the arguments are real, not dummy variables, and do change as evaluation proceeds.

```
10 REM DEFINE A SQUARE FUNCTION
20 DEF FN S1(I) = I * I
30 PRINT FN S1(3),I,FN S1(5),I
40 END

*RUN ®
 9            3            25           5

 END AT LINE 40
*
```

END

The END statement causes control to return to the command mode. An END statement message is typed, giving the line number of the END statement. END also causes the "next statement" pointer to be set to the beginning of the program so a CONTINUE resumes execution at the beginning of the program.

An END statement may appear anywhere in the program, as many times as desired. If a program does not contain an END statement, it "runs off the end." In this case, BASIC generates a pseudo end statement at line 65,535.

INPUT AND LINE INPUT

The INPUT and LINE input statements are used when data is to be read from the console terminal, or from an HDOS file. The form of the INPUT statement is

```
INPUT prompt;var1, . . . , varn          (or)
INPUT #chan, prompt;var1, . . . , varn
```

The #chan specification (shown in the second example) is optional, and if present specifies the channel number of the file (which must have been previously OPENed for INPUT) to read from. An INPUT statement with a file channel number specified works just like a regular INPUT statement, except that a line is read from the file rather than the console. Values are read from the line in exactly the same way as they would be from a line typed at the console. If necessary, BASIC will read more lines from the file to satisfy the INPUT statement. Any unused values on the line are discarded.

If the first element following the INPUT statement is a string, INPUT assumes it is a prompt and types the string instead of the question mark (?). If you do not want a prompt string but the first variable is a string variable, a leading semicolon is required. For example:

```
INPUT ;S3$(2)
```

tells BASIC that the data read from the console terminal is to be placed in the third element of the string array S3$. Note that a prompt is meaningless when inputting from HDOS files.

The data line input from the console or read from the HDOS file is identical in format to the DATA statement except that the DATA keyword is omitted. String values need not be enclosed in quotes unless they contain the comma (,) character. Multiple data values on the same line must be separated by commas.

As in the DATA statement, null fields (two commas in a row) cause the variable to retain its previous value. If the user response (or the line read if you are inputting from an HDOS file) does not supply sufficient data to complete the INPUT statement, another "?" prompt is issued (if you are inputting from the console) and another line is read from the console or the data file. CAUTION: If you supply too much data or there is too much on a line read from a file, it will be ignored. The next INPUT statement issues a fresh read to the terminal or file.

When there are several values to be entered via the INPUT statement, it is helpful to print a message explaining the data needed, using the prompt string. For example:

```
10 INPUT "THE TIME IS?";T
```

When this line of the program is executed, BASIC prints

```
THE TIME IS?
```

and then waits for a response.

The LINE INPUT statement is used to input one line of string data from the console terminal and assign it to a string variable. Its form is identical to the INPUT form except that the supplied line is read in its entirety into the string variable, regardless of commas (,) or quotes ("). For example:

```
LINE INPUT "YES OR NO?";A$          (or)
LINE INPUT #2,;A$
```

Note that the channel number in the second example must be followed by a comma; the following semicolon tells BASIC that A$ is the variable name, not a prompt.

LINE INPUT, unlike READ and INPUT, allows you to read a string containing a quote (") character. Note that you should **not** enclose your reply in quotes, since these will be accepted as part of the string.

STOP

The STOP statement causes BASIC to enter the command mode. The message stating the line number of the STOP is printed. The "next line" pointer is left after the STOP statement, so a CONTINUE statement causes execution to resume on the line immediately after the STOP statement. The STOP statement is of the form:

```
STOP
```

The STOP statement can occur several times throughout a single program with conditional jumps determining the actual end of the program. The following example uses the STOP statement to examine a variable during execution.

```
10 A=1:B=2:C=3
20 PRINT A,B,C
30 END

*RUN CR
 1             2             3

 END AT LINE 30
*15STOP CR

*RUN CR

 STOP AT LINE15
*PRINT A CR
 1
*15 CR                              Stop deleted

*RUN CR
 1             2             3

 END AT LINE 30
*
```

PREDEFINED FUNCTIONS

Introduction

There are 28 predefined functions in EX. B.H. BASIC. These functions perform standard mathematical operations such as square roots, logarithms, string manipulations, and special features. Each function has an abbreviated three- or four-letter name, followed by an argument in parentheses. As these functions are predefined, they may be used throughout a program when required. Predefined functions use numeric expressions (nexp), integer expressions (iexp), and string expressions (sexp).

The abbreviation (narg) is used to indicate a numeric argument, a decimal number lying in the approximate range of 10^{-38} to 10^{+37}. Certain functions do not permit the argument to assume this wide range, as indicated in the function description.

The predefined functions may be used in either the command or program mode.

Arithmetic and Special Feature Functions

THE ABSOLUTE VALUE FUNCTION, ABS (nexp)

The ABSOLUTE VALUE function gives the absolute value of the argument. The absolute value is the positive portion of the numeric expression. For example:

```
*PRINT ABS(-5.5) CR
 5.5                    or,

*PRINT ABS(SIN(3.5)) CR
 .350783
*
```

NOTE: The sine of 3.5 radians is $-.350783$.

THE ARC TANGENT FUNCTION, ATN (nexp)

The ARC TANGENT function returns the arc tangent of the argument. For example:

```
*PRINT ATN(1/1)*57.296;"DEGREES" ↵
 45.0001 DEGREES
*PRINT 4*ATN(1) ↵
 3.14159
*                                    NOTE: π = 3.14159
```

THE CHARACTER INPUT FUNCTION, CIN (chan)

The CIN function is used to read a character from any open file, or from the console terminal (if chan = 0). If the value returned is positive, then it is the next byte read from the file, or the next character read from the console (if chan = 0). If the value returned is negative, then an end-of-file has been detected on the file, or no line has yet been entered on the console (if chan = 0). For example:

```
*PRINT CIN(0) ↵
 -1
*
```

THE COSINE FUNCTION, COS (nexp)

The COSINE function returns the COSINE of the argument (nexp) expressed in radians. For example:

```
*PRINT COS(60/57.296) ↵
  .500003
*
```

THE EXPONENTIAL FUNCTION EXP (nexp)

The EXPONENTIAL function returns the value e^{nexp}. If "nexp" exceeds 88, an overflow is flagged, as the result exceeds 10^{38}. If "nexp" is less than −88, an overflow error occurs. An example of the exponential function is:

```
*PRINT EXP(1),EXP(2),EXP(COS(60/57.296)) ↵
 2.71828          7.38905          1.64873
*
```

THE INTEGER FUNCTION, INT (narg)

The INTEGER function returns the value of the greatest integer value, not greater than "narg". If the argument is a negative number, the INTEGER function returns the negative number with the same or smaller absolute value. For example:

```
*PRINT INT (38.55)
 38

*PRINT INT (-3.3)
 -3
```

THE LINE NUMBER FUNCTION, LNO (iexp)

BASIC statements that refer to the line numbers (such as GOTO, GOSUB, and so forth) do not allow the line number to be expressed as a numeric expression. The LNO function is provided to convert an integer expression into a line number. For example:

```
GOTO 20 ↵
                (and)
GOTO LNO(2*10)
```

both cause a jump to statement number twenty. You can use the LNO function anywhere a line number is required; it provides a very powerful "computed GOTO" facility. A program can compute the line number it wishes to jump to (or call, via GOSUB) by using the LNO function. Some more examples:

```
GOSUB LNO(2*Y+100)

ON I GOTO 20,30,LNO(I),LNO(I*2)

IF (A=B) THEN GOTO LNO(A)
```

THE LOGARITHM FUNCTION, LOG (nexp)

The LOGARITHM function returns the natural logarithm (LOG to the base e) of the argument. You can find the Logarithms of a number N in any other base by using the formula:

$$LOG_a\ N = LOG_eN/LOG_ea$$

where "a" represents the desired base. Most frequently, "a" is 10 when you are converting to common logarithms. For example:

```
*PRINT "A POWER RATIO OF 2 IS";10*(LOG(2)/LOG(10));"DECIBELS" ↵
 A POWER RATIO OF 2 IS 3.0103 DECIBELS
*
```

THE PAD FUNCTION, PAD (0)

The PAD function returns the value of the keypad pressed on the H8 front panel. For example:

```
*PRINT PAD(0) ®
 6                    The #6 key was pressed.
```

The PAD function uses all the front panel debounce and repeat software contained in PAM-8. (See "The Segment Function," Page 5-72, for an additional example.)

NOTE: The PAD function must be completely executed before any other function will respond. Therefore, CTRL-C, etc., will not work until you press an H8 front panel key.

The PAD function is intended for use on an H8 computer, where front panel access is necessary. On an H89 computer, there is no front panel. If a BASIC program using the PAD function is run on an H89, a zero (0) will be returned as soon as the PAD(0) is executed. CTRL-C is not disabled on the H89.

THE PEEK FUNCTION, PEEK (iexp)

The PEEK functions returns the numeric value of the byte at memory location iexp. iexp is in decimal.

THE PIN FUNCTION, PIN (iexp)

The PIN function returns the value input from port "iexp" where iexp is a decimal expression ranging from 0-255. For example:

```
*A=PIN(38) ®
```

Where "A" now contains the data that was at port #38 (46 octal).

THE POSITION FUNCTION, POS (chan)

The POSITION function returns the current terminal printhead (cursor) position. The argument "chan" specifies the I/O channel number (see the OPEN statement) you wish to interrogate. BASIC maintains a separate cursor address for each I/O channel in use. Channel 0 is always the console channel, and is always considered "open." Thus, use POS(0) to read the position of the console cursor. The value returned is a decimal number indicating the column number of the printhead (cursor) position. For example:

```
*PRINT POS(0), POS(0), POS(0); POS(0); POS(0) ®
 1             14             28   32   36
*
```

THE RANDOM FUNCTION, RND (narg)

The RANDOM number function returns the next element in a pseudo-random series. The RANDOM number generator is not truly random, and may be manipulated by controlling the argument. If narg>0, the random number generator returns the next random number in the series. If narg = 0, the random number generator returns the previously returned random number. If narg<0, the value "narg" is used as a new seed for a random number, thus starting an entire new series. Using these three inputs to the random number series, the program may continuously return the same number while de-bugging the program, determine what the series of numbers will be when the program is run, or start a series of new random numbers each time BASIC is loaded. For example:

```
 10 RUN FOR A=0 to 2
 20 PRINT RND(1)
 30 NEXT
 40 END

*RUN <CR>
 .93677
 .566681
 .53128

 END AT LINE 40
*20PRINT RND(0) <CR>

*RUN <CR>
 .332306
 .332306
 .332306

 END AT LINE 40
*20PRINT RND(-1) <CR>

*RUN <CR>
 6.25305E-02
 6.25305E-02
 6.25305E-02

 END AT LINE 40
*20PRINT RND(-5) <CR>

*RUN <CR>
 .460968
 .460968
 .460968

 END AT LINE 40
*
```

THE SEGMENT FUNCTION, SEG (narg)

The SEG function returns a numeric value which is the correct 8-bit binary number to display the digit on the H8 front panel LED's. The argument must be an integer between 0 and 9. The following program demonstrates the use of PAD, POKE, and SEG in EX. B.H. BASIC.

```
10 REM A PROGRAM TO USE THE FRONT PANEL LEDS. CNTRL 2,1 TURNS
20 REM ON THE LEDS WITHOUT UPDATE. THE KEYPAD NOW DRIVES THE
30 REM DISPLAY THRU BASIC. 8203 IS THE FIRST LED MEM LOCATION.
40 CNTRL 2,1
50 A=8203
60 FOR I=A△TO A+8
70 POKE I,SEG(PAD(0))
80 NEXT I
90 GOTO 60
*RUN ㊐
```

When the program is executed, the H8 front panel LEDs respond to the H8 keypad numeric entries. To escape from the program, you would type CTRL-C and then press a key on the H8 front panel.

The SEG function is not useful on an H89 computer. Running this sample program on your H89 will produce no results. Type CTRL-C to exit.

THE SIGN FUNCTION, SGN (narg)

The SIGN function returns the value +1 if "narg" is a positive value, 0 if "narg" is 0, and −1 if "narg" is negative. For example:

```
*PRINT SGN(5.6) ㊐
 1

*PRINT SGN(-500) ㊐
 -1

*PRINT SGN(12-12) ㊐
 0

*
```

THE SINE FUNCTION SIN (nexp)

The SIN function returns the sine of the argument (nexp) expressed in radians. For example:

```
*PRINT SIN(30/57.296) ㊐
 .499999
*
```

SQUARE ROOT FUNCTION, SQR (narg)

The SQUARE ROOT function returns the square root of "narg". The argument "narg" must be greater than or equal to 0 (for example, positive).

```
*FOR A=0 TO 5:PRINT A,SQR(A),A*A:NEXT ↵
 0          0               0
 1          1               1
 2          1.41421         4
 3          1.73205         9
 4          2               16
 5          2.23607         25

*
```

THE MAXIMUM FUNCTION, MAX (nexp1, ...,nexpn)

The MAXIMUM function returns the maximum value of all the expressions which are arguments of the function. For example:

```
 10 LET A=1
 20 PRINT MAX(COS(A),SIN(A)/COS(A))
 30 END
*RUN ↵
 1.55741
 END AT LINE 30
*
```

The expression containing the maximum value is the expression for the tangent of 1 radian, (1.55741).

THE MINIMUM FUNCTION, MIN (nexp1, . . . ,nexpn)

The MINIMUM function returns the lowest value of all expressions contained in the argument. For example:

```
*PRINT MIN(1,2,3,4,.5) ↵
 .5
*
```

THE TANGENT FUNCTION, TAN (nexp)

The TANGENT function returns the TANGENT of the argument "nexp" expressed in radians. For example:

```
*PRINT TAN (45/57.296) ↵
 .999996
*
```

THE SPACE FUNCTION, SPC (iexp)

The SPACE function spaces the printhead (cursor) iexp spaces to the right of its present position. For example:

```
*PRINT 12.14,SPC(20);600 ↵
 12                14                  600
*
```

THE TAB FUNCTION TAB (iexp)

The TAB function moves the printhead (cursor) to the iexp th column. NOTE: If the printhead is at or past the iexp th column, the function is ignored. For example:

```
*PRINT TAB(20);60,70 ↵
                    60          70
*
```

String Functions

BASIC contains various functions for processing character strings in addition to the functions used for mathematical operations. These functions allow the program to concatenate two strings, access a part of string, generate a character string corresponding to a given number, or generate a number for a given string.

THE CHARACTER FUNCTION, CHR$ (iexp)

The CHARACTER function returns a string that consists of a single character. The character generated has the ASCII code "iexp". NOTE: "iexp" is a decimal number and must be converted to octal for comparison with most ASCII character tables. See "Appendix B" on Page 5-97. For example:

```
*PRINT CHR$(65) ↵
A
*PRINT CHR$(70) ↵
F
*
```

NOTE: If iexp = 0, the generated string is null.

THE STRING FUNCTIONS, STR$ (narg)

The STRING function encodes the argument (narg) into ASCII in the same format used by the PRINT statement for numbers. These characters are returned as a string, with leading and trailing blanks. For example:

```
*PRINT STR$(100) ↵
 100                    } STR$ function
*PRINT "100" ↵
 100                    } Normal string printing
*
```

THE ASCII FUNCTIONS, ASC (sexp)

The ASCII function returns the ASCII code for the first character in the string expression (sexp). If the string is a null, the ASCII function returns a zero. The return is a decimal number and must be converted to octal for comparison to most ASCII tables. See "Appendix B" on Page 5-97. For example:

```
*PRINT ASC("ABC") ↵
 65
*PRINT CHR$(65) ↵
A
*
```

THE LEFT STRING FUNCTION, LEFT$ (sexp, iexp)

The LEFT STRING function returns the "iexp" left-most characters of the string expression (sexp). If "iexp" equals 0, the null string is returned. For example:

```
*PRINT LEFT$("HELLO, THIS IS A TEST",10) ↵
HELLO, THI
*
```

THE RIGHT STRING FUNCTION, RIGHT$ (sexp, iexp)

The RIGHT STRING function returns the right-most "iexp" characters of the string expression (sexp). If "iexp" equals 0, the null string is returned. For example:

```
*PRINT RIGHT$("HELLO,  THIS IS A TEST",10) ↵
 IS A TEST
*
```

THE LEN FUNCTION, LEN (sexp)

The LEN function returns the length of the string expression "sexp". For example:

```
*PRINT LEN("HOW LONG IS THE STRING?") ↵
 23
```

THE MATCH STRING FUNCTION, MATCH (sexp1,sexp2,iexp)

The MATCH function searches the string sexp1 for any substrings matching sexp2. The search starts with character iexp in the string sexp1. If iexp = 1, the search starts at the first character in sexp1. If iexp = 2, the search starts at the second character in sexp1, and so forth. MATCH returns the character number of the start of the substring in sexp1, if one was found, and a 0 if it was not found. For example:

```
*PRINT MATCH("THIS IS A RATHER LONG STRING","TH",2) ®
 13
*
```

Note that MATCH found the TH in RATHER, not in THIS. Since the MATCH call specified a search to start with the second character, BASIC started searching at the "HIS IS . . .", thereby ignoring the T in "THIS".

THE MIDDLE STRING FUNCTION, MID$ (sexp, iexp1 [,iexp2])

The MIDDLE STRING function returns the right-hand substring of the string expression "sexp" starting with the "iexp1" th character from the left-hand side (the first character is 1). The return continues for "iexp2" characters or to the end of the string if the optional terminating expression "iexp2" is omitted. For example:

```
*PRINT MID$("HELLO, THIS IS A TEST",10,10) ®
IS IS A TE
*
```

THE NUMERIC VALUE FUNCTION, VAL (sexp)

The NUMERIC VALUE function returns the numeric value of the number encoded in the string expression (sexp). For example:

```
*PRINT VAL (".0032E-1") ®
 3.00000E-04
*
```

GENERAL TEXT RULES

BLANKS AND TABS

BASIC programs are generally "free format." That is, blanks (spaces and TABS) may be included freely with the following restrictions.

1. Variable names, keywords, and numeric constants may not contain imbedded blanks or tabs.

2. Blanks or tabs may not appear before a statement number.

LINE INSERTION

You can insert lines into a BASIC program by simply typing an appropriate line number followed by the desired line of text. This is done in response to the command mode prompt (an asterisk). Except when it it running a program, BASIC remains in the command mode, showing a single asterisk (*) as a prompt. NOTE: The text should immediately follow the last digit of the line number. Although intervening blanks are allowable, they waste memory. BASIC automatically inserts a blank when listing the text. For example:

```
*100PRINT "HEATH BASIC"
 LIST ↵
 100 PRINT "HEATH BASIC"
```

Each time you type a statement with a line number, BASIC performs some simple syntactical checks before inserting the line into your program. BASIC checks to see if all the keywords are spelled correctly, and translates them to upper case. It makes sure that all function calls are immediately followed by an open parenthesis "(". BASIC makes several other checks of the line to check for simple syntax errors. If the line is determined to be incorrect, the message

```
SYNTAX ERROR
```

will be typed, and the line will not be inserted into your program. Note that this preliminary syntax check will not detect all possible errors; BASIC may accept the line when you type it and then detect an error later when you execute your program.

LINE LENGTH

A line in Extended BENTON HARBOR BASIC is restricted to 100 characters.

LINE REPLACEMENT

Replace existing program lines by simply typing the line number and the new text. This is the same process you use to insert a new line. The old line is completely lost once the new line is entered.

LINE DELETION

Delete lines by typing the line number immediately followed by a carriage-return. You can leave blank lines by typing the single space before typing the carriage-return.

ERRORS

BASIC detects many different error conditions. When an error is detected, a message of the form:

```
! ERROR-(ERROR MESSAGE) [at line NNNNN]
```

is typed. BASIC returns to the command mode (if it is not already in the command mode), ringing the console terminal bell. If BASIC is in the command mode, the "at line NNNN" portion of the error message is omitted. For example:

```
!PRINT 1/0 CR
! ERROR - ATTEMPTED DIVIDE BY ZERO
*10PRINT 1/0 CR
*RUN CR

! ERROR - ATTEMPTED DIVIDE BY ZERO AT LINE 10
*
```

NOTE: If a line of BASIC contains an error, you can correct it by retyping the entire line. Once the line number is typed, the contents of the old line are lost. To delete a line, type the line number and follow it with a carriage-return.

Error Messages

The following error table describes the Error Messages generated by Extended BENTON HARBOR BASIC. This error table discusses only those errors which are detected directly by BASIC. When you are dealing with HDOS files, there are many errors which are detected by HDOS. They are printed in the BASIC error message format described above, but their meanings are discussed in Chapter 1, the HDOS Manual.

Recovering from Errors

When it detects an error, BASIC enters the command mode with the variables and stacks as they were at the time of the error. Thus, the user can use PRINT and LET statements to examine and alter variable contents. Likewise, a GOTO statement can be used to set the "next statement" pointer to any desired statement number. Often, a combination of these techniques allows the user to continue a program with the error corrected.

NOTE: If the program text is modified in any way, the GOSUB and FOR stacks are purged. If an error occurred in a GOSUB routine for a FOR-loop, the entire program must be restarted.

ERROR MESSAGES

AN ILLEGAL CHARACTER WAS ENCOUNTERED

This message indicates a syntax error in the line. BASIC saw a character that was not legal in a BASIC statement.

ATTEMPTED DIVIDE BY ZERO

Your program tried to divide a number by 0.

CAN'T FIND VARIABLE NAME MENTIONED IN NEXT STATEMENT

BASIC has not seen a matching FOR-loop for the variable named in the NEXT statement. This error can be caused by improper FOR-loop nesting.

CTL-B STRUCK

The CTRL-B key was struck and no CTRL-B line number had been set up. See the CNTRL 0,n command for more information.

CTL-C STRUCK

The CTRL-C key was struck, interrupting the program.

DATA EXHAUSTED

A READ statement was executed when there was no data remaining in DATA statements to satisfy the READ. You either have too many READ requests or too few DATA statements.

DATA LOCK ENGAGED.

This operation is illegal when BASIC is in the LOCKed state. See the LOCK and UNLOCK commands for more information.

END

Your program executed an END statement. This is a normal way of terminating execution, and not an error. If your program has no END statement, BASIC will invent one at line 65535.

FILE ALREADY EXISTS

You tried to SAVE to a file name which is already present on that device. Either SAVE to a different file name, UNSAVE (delete) the existing file name, or use the REPLACE command.

FILE IS NOT OPEN

You tried to do file I/O (PRINT #chan, or INPUT #chan) to a channel which has no open file associated with it. You must OPEN a file before it can be used for INPUT or PRINTing.

FLOATING POINT OVERFLOW (Number too large)

An arithmetic calculation produced a number larger than 1 × 10↑37.

ILLEGAL FORMAT FOR FILE NAME

A file name specified in an OPEN statement contained too many characters to be valid. There should be no blanks or extraneous characters in a file name string.

ILLEGAL NUMBER VALUE

A number appeared in an illegal format or syntax. If this error occurs during a READ or INPUT statement, check the value being READ or INPUTted to see if it is valid.

ILLEGAL OR UNKNOWN STATEMENT NUMBER

A reference was made to a statement that does not exist, or to an illegal statement number. Statement numbers must be between 1 and 65534.

ILLEGAL USAGE

This statement may not be used in this mode. You have tried to use an "execution mode only" command in immediate mode, or an "immediate mode only" command in an executing program.

NO CORRESPONDING GOSUB FOR THIS RETURN STATEMENT

A RETURN statement was encountered when the GOSUB stack was empty.

OUT OF RAM SPACE

There is insufficient free RAM space to continue with this program. This error usually occurs when you DIMension a large array. If you cannot determine the cause of the memory overflow, use the FREE command to display the amounts being used by the various tables. If you have specified CNTRL 4,1, you can free up some RAM space by specifying CNTRL 4,0.

STOP

A STOP statement was encountered. This is a normal condition, and not an error.

STRING LENGTH EXCEEDS 255 CHARACTERS

The maximum length of a string in BASIC is 255 characters.

SUBSCRIPT OUT OF RANGE

The program specified a subscript value larger than the declared limit for that dimension. Either your array was declared too small or your program incorrectly computed the subscript.

SYNTAX ERROR

There is an error in the statement's syntax.

TOO MANY OR TOO FEW ARGUMENTS SPECIFIED

An incorrect number of arguments was specified for a call to a built-in function or a user-defined function.

TOO MANY OR TOO FEW SUBSCRIPTS SUPPLIED

The number of subscripts in the array reference do not match the number of dimensions declared.

TYPE CONFLICT (ILLEGAL mix of string and number values)

The program attempted an operation illegally mixing string and number values, or supplied a numeric argument to a function requiring a string argument, or vice versa. This error can also occur if you try to INPUT or READ a string value into a numeric variable.

UNDEFINED FUNCTION

This user-defined function has not been defined. Your program must execute the DEF FN statement before you attempt to call that function.

APPENDIX A

A Summary of BASIC

For additional details, refer to the page number that is given with each of the following topics.

See Page

Subscripted Variables 5-12

Subscripted variables are named like variables, but are followed by dimensions in parentheses. Subscripted variables are of the form:

An(N_1,N_2, ... , N_x) For example: A(1,2,7) or A6(1,5).

You must use a DIMENSION statement to define the range and number of allowable subscripts for a variable.

Arithmetic Operators 5-14

Listed in order of priority. Operators on the same line have equal precedence. Parenthetical operations are performed first. Precedence is left to right if all other factors are equal.

SYMBOL	EXPLANATION
–	Unary negation logical complement
↑	Exponentiation.
* /	Multiplication division
+ –	Addition subtraction

Relational Operators 5-18

SYMBOL	EXPLANATION
=	Equal to
<	Less than
< =	Less than or equal to
>	Greater than
> =	Greater than or equal to
< >	Not equal to

See Page

Boolean Operators 5-19

Boolean operators perform the Boolean (logical) operations on two integer operands. The operands must evaluate to integers in the range of 0 to 65535. The operators are:

NOT	Logical complement, bit by bit
OR	Logical OR, bit by bit
AND	Logical AND, bit by bit

String Variables 5-21

String variables may be either subscripted or nonsubscripted. They take the same form as numeric or Boolean variables but are followed by a dollar sign ($) to indicate a string variable. For example: A$ A6$ A$(1,2,7) or A6$(1,5).

String Operators 5-22

String expressions may be operated on by the relational operators as well as the plus (+) symbol. The plus symbol is used to perform string concatenation.

Line Numbers 5-25

When it is used in the program mode, BASIC requires that each line be preceded by an integer line number in the range 1 to 65534.

The Command Mode 5-23

The command mode does not use line numbers. Statements are executed when a carriage-return is typed.

Multiple Statements on One Line 5-25*

BASIC permits multiple statements on one line. Each statement is separated from the others by a colon (:). DATA statements may not appear on lines with other statements.

*See "Basic Statements."

Command Mode Statements

COMMAND	FORM	DESCRIPTION	SEE Pg.
BUILD	BUILD iexp1, iexp2	Automatically generates program line numbers starting at iexp1 in steps of iexp2.	5-27
BYE	BYE	Exits BASIC, returns to HDOS command mode.	5-28
CONTINUE	CONTINUE	Resumes program execution.	5-28
DELETE	DELETE [iexp1, iexp2]	Deletes program lines between iexp1 and iexp2	5-29
LIST	LIST [iexp1] [,iexp2]	Lists the entire program on the console terminal. Lists the line iexp1 or the range of lines iexp1 through iexp2.	5-29
OLD	OLD "fname"	Loads file "fname" into BASIC. Clears variables.	5-30
REPLACE	REPLACE "fname"	Saves current program as file "fname." Replaces "fname" if it already exists.	5-30
RUN	RUN	Start execution of current program. Preclears all variables, stacks, etc..	5-31
SAVE	SAVE "fname"	Saves current program as file "fname". Will not replace any pre-existing "fname".	5-32
SCRATCH	SCRATCH SURE?Y ↵	Clears all program and data storage area. Any response to SURE but Y cancels SCRATCH.	5-32

Command and Program Mode Statements

COMMAND	FORM	DESCRIPTION	SEE Pg.
CHAIN	CHAIN "fname"[,linnum]	Loads new program "fname" into BASIC and continues execution at linnum. If no line number is specified, start execution at first line number. Does not affect variables or open files.	5-33
CLEAR	CLEAR [varname]	Clears all variables, arrays, string buffers, etc. Optionally clears named variable (varname). Specifies functions and arrays as V).	5-34
CLOSE	CLOSE #chan 1 [,#chan n]	Close an HDOS file. "#chan" is the number assigned to the opened file.	5-35
CONTROL	CNTRL iexp1, iexp2	CNTRL 0 sets a GOSUB to line iexp2 when a CTRL-B is typed.	5-37
		CNTRL 1 sets iexp2 digits before exponential format is used.	5-38
		CNTRL 2 controls the H8 front panel. If iexp2: = 0, display off; if iexp2 = 1, display on without update; if iexp2 = 2, display on with up date. (NOTE: has no effect on the H89).	5-38
		CNTRL 3 sets the width of a print zone to iexp2 columns.	5-39
		CNTRL 4 controls the state of the HDOS system overlay. iexp2 = 0, swap overlay. iexp = 1, keep overlay in memory. (Command Mode only).	5-39
DIMENSION	DIMvarname(iexp1 [,. . .,iexpn]) [,varname2(....)]	Defines the maximum size of variable arrays.	5-40

COMMAND	FORM	DESCRIPTION	SEE Pg.
FOR/NEXT	FOR var = nexp1 TO nexp2 [STEP nexp3]		5-41
	NEXT var	Defines a program loop. Var is initially set to nexp1. Loop cycles until NEXT is executed; then var is incremented by nexp3 (default is +1). Looping continues until var > nexp2 (or less then nexp2 if STEP is negative). The statement after NEXT is then executed.	
FREE	FREE	Displays the amount of memory assigned to tables and text.	5-45
FREEZE	FREEZE "fname"	Saves BASIC interpreter, current program, and data values on file "fname". All files must be closed before FREEZE.	5-47
GOSUB/ RETURN	GOSUB iexp RETURN	Transfers execution sequence of program to line iexp (the beginning of a subroutine). RETURN returns execution sequence to the statement following the calling GOSUB.	5-47
GOTO	GOTO iexp	Unconditionally transfers the program execution sequence to the line iexp.	5-49
IF/THEN	IF expression THEN iexp IF expression THEN statement	If the expression is true, control passes to iexp line or to "statement." If the relation is false, control passes to the next independent statement.	5-49
LET	LET var = nexp LET var$ = sexp	Assigns the value nexp (or sexp in the case of strings) to the variable var (or var$). LET keyword is optional.	5-50

COMMAND	FORM	DESCRIPTION	SEE Pg.
LOCK	LOCK	Protects your program by preventing you from executing the BUILD, BYE, CHAIN, UNFREEZE, DELETE, RUN, SCRATCH, and CLEAR command mode statements. Also prevents the entry or deletion of program text.	5-51
ON/GOSUB	ON iexp1 GOSUB iexp2,...,iexpn.	Permits a computed GOSUB. Iexp1 is evaluated and acts as an index to line numbers iexp2 thru iexpn, each pointing to a different subroutine.	5-52
ON/GOTO	ON iexp1 GOTO iexp2,...,iexpn	Permits a computed GOTO. Iexp1 is evaluated and acts as an index to line numbers iexp2 thru iexpn.	5-52
OPEN	OPEN sexp FOR READ AS FILE #iexp OPEN sexp FOR WRITE AS FILE #iexp	Opens file for read or write operations. "sexp" is a string expression for the file name. "#iexp" is the channel number assigned to the file to be opened.	5-52
OUT	OUT iexp1, iexp2	Outputs a number iexp2 to output port iexp1.	5-54
PAUSE	PAUSE (iexp)	Ceases program execution until a console terminal key is typed. Ceases program execution for 2 X iexp mS.	5-54
POKE	POKE iexp1, iexp2	Writes a number iexp2 into memory location iexp1.	5-55

COMMAND	FORM	DESCRIPTION	SEE Pg.
PRINT	PRINT [#chan,] (nexp1 sep1 . . . nexpn (sepn)	Prints the value of the expression(s) exp with a leading and trailing space. Expressions may be numeric or string. If the separator is a comma, the next print zone is used. If the separator is a semicolon, no print zones are used. No separator prints each expression on a new line. #chan specifies channel to write line to HDOS file. If no #chan is specified, line goes to console terminal.	5-55
READ/DATA	READ var1, . . . ,varn DATA exp1 . . ,expn	The READ statement assigns the values exp1 thru expn in the data to the variables var1 thru varn.	5-59
REMARK	REM	Text following the REM is not executed and is used for commentary only.	5-61
RESTORE	RESTORE	Causes the program to reset the DATA pointer, thus reusing data at the first DATA statement.	5-61
STEP	STEP iexp	Executes iexp lines of the program. Then returns BASIC to the command mode.	5-62
UNFREEZE	UNFREEZE "fname"	Restores BASIC program and variables from previously created FREEZE file.	5-62
UNLOCK	UNLOCK	Aborts the LOCK mode and restores the use of all command mode statements.	5-63
UNSAVE	UNSAVE "fname"	Deletes programs or files from the disk.	5-63

Program Mode Statements

COMMAND	FORM	DESCRIPTION	SEE Pg.
DEF	DEF FN varname (arg list) = exp	Defines a single-line program function created by the user.	5-63
END	END	Causes control to return to the command mode.	5-64
INPUT	INPUT [#chan,] prompt;var1,...,varn	Reads data from the console terminal, or from the HDOS file open on channel "chan", if #chan is specified. String data must be enclosed in quotes if it contains any commas (,).	5-64
LINE INPUT	LINE INPUT [#chan,] prompt;stringvar	Reads string data from the console terminal, or from the HDOS file open on channel "chan, if #chan, is specified. Data should not be enclosed in quotes; entire line is read into string variable.	5-64
STOP	STOP	Causes BASIC to enter the command mode when the statement containing STOP is executed.	5-66

Predefined Functions

FUNCTION	DEFINITION	SEE Pg.
ABS (nexp)	Returns the absolute value of nexp.	5-67
ASC (sexp)	Returns the ASCII code for the first character in the string sexp.	5-75
ATN (nexp)	Return the arctangent of nexp (radians).	5-68
CHR$ (iexp)	Returns the ASCII character iexp.	5-74
CIN (chan)	Reads a character from any open file, or from the console terminal (if chan = 0). If the value returned is positive, a character was read. If the value was negative, an end-of-file or no line was read.	5-68
COS (nexp)	Returns the cosine of nexp (radians).	5-68
EXP (nexp)	Returns e^{nexp}.	5-68
INT (narg)	Returns the integer value of narg.	5-68
LEFT (sexp, iexp)	Returns the left iexp charaters of the string sexp.	5-75
LEN (sexp)	Returns length of string expression sexp.	5-75
LNO (iexp)	Converts iexp to a line number.	5-69
LOG (nexp)	Returns the natural logarithm of nexp.	5-69
MATCH (sexp1, sexp2, iexp)	Finds the first occurrence of the substring sexp 2 in sexp1 starting at the iexp th character in sexp1. Returns index of start of substring if found, 0 if not found.	5-76
MAX (nexp1,...,nexpn)	Returns the maximum value of expressions nexp1 thru nexpn.	5-73

COMMAND	FUNCTION	SEE Pg.
MID$ (sexp, iexp1) [,iexp2]	Returns the substring of the string sexp starting with the iexp1 th character and ending with the iexp2 th character if iexp2 is specified. If not specified, returns iexp1 th character to the end.	5-76
MIN (nexp1,...,nexpn)	Returns the minimum value of expressions nexp1 thru nexpn.	5-73
PAD (0)	Returns the value of the H8 front panel key pressed. Includes key debounce. Returns a 0 on an H89.	5-70
PEEK (iexp)	Returns the numeric value at memory location iexp.	5-70
PIN (iexp)	Returns the data input from port iexp.	5-70
POS (chan)	Retuns the current file or console printhead (cursor) position (by column number).	5-70
RND (narg)	Returns a random number. If narg >0, RND is next in the series. If narge = 0 RND is the previous random number. If narg <0, RND algorithm uses narg as a new seed.	5-70
RIGHT$ (sexp, iexp)	Returns the right iexp characters of the string sexp.	5-75
SEG (narg)	Returns the correct eight-bit number to display narg (0-9) on the H8 LEDs. Has no effect on an H89.	5-72
SGN (narg)	Returns +1 if narge is positive. Returns −1 if narg is negative. Returns 0 if narg is zero.	5-72
SIN (nexp)	Returns the sine of nexp (radians).	5-72
SPC (iexp)	Positions printhead (cursor) iexp columns to the right.	5-74
SQR (narg)	Returns the square root of narg.	5-73

COMMAND	FUNCTION	SEE Pg.
STR$ (narg)	Returns narg encoded into ASCII with leading and trailing blanks as in the print statement.	5-75
TAB (iexp)	Position printhead (cursor) to the iexp th column.	5-74
TAN (nexp)	Returns the tangent of nexp (radians).	5-73
VAL (sexp)	Returns the numeric value of the number encoded in the string.	5-76

APPENDIX B

ASCII CODES

DECIMAL TO OCTAL TO HEX TO ASCII CONVERSION

DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0	000	00	NUL	32	040	20	SPACE	64	100	40	@	96	140	60	'
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	PERIOD	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DELETE

NUL	Null; Tape Feed,
SOH	Start of Heading; Start of Message
STX	Start of Text; End of Address
ETX	End of Text; End of Message
EOT	End of Transmission; Shuts off TWX machines
ENQ	Enquiry; WRU
ACK	Acknowledge; RU
BEL	Rings Bell
BS	Backspace; For at Effector
HT	Horizontal TAB
LF	Line Feed or Space (New Line)
VT	Vertical TAB
FF	Form Feed (PAGE)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1; Reader on
DC2	Device Control 2; Punch on
DC3	Device Control 3; Reader off
DC4	Device Control 4; Punch off
NAK	Negative Acknowledge; Error
SYN	Synchronous Idle (SYNC)
ETB	End of Transmission Block; Logical End of Medium
CAN	Cancel (CANCL)
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

Note that these characters (Octal 000 through 037), can be generated from the combination CTRL and the character in the same row, but in the third or fourth column (Octal 100 through 137 or 140 through 177).
That is, BEL is Control/G or /g, and CAN is Control/X or /x.

INDEX

NOTE: Numbers printed in a bold type face refer to examples of the indicated statement or function.