Section 2

EXTENDED BENTON HARBOR BASIC

TABLE OF CONTENTS

INTRODUCTION

BASIC, a conversational programming language, 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. BASIC is an interpretive language, compact enough to run in a Heath H8 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

The 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.

Hardware Requirements

EXTENDED BENTON HARBOR BASIC (Ex. B. H. BASIC) runs on an H8 computer with a minimum of 16K bytes of random access memory. BASIC requires that you use a console terminal, its appropriate interface card, and a mass storage device such as a cassette recorder.

BASIC automatically measures the maximum amount of unbroken memory above the starting point at 8K (40,100 offset octal). It uses all available memory unless the high memory limit is configured otherwise during the system configuration procedure (see "Appendix A" on Page 2-73 in this Software Reference Manual).

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

Loading and Running BASIC

BASIC is distributed in binary format on cassette tapes or paper tape. It is loaded in accordance with the software configuration guide outlined in "Appendix A" on Page 2-73. Once a system BASIC tape is configured, you can load the configured tape, using the internal PAM-8 loader, and start it by pressing the GO key. EX. B.H. BASIC uses the asterisk (*) as a prompt character.

Command Completion

EX. B.H. BASIC employs command completion. This means that BASIC examines each character as you type it on the console keyboard, and when sufficient information is received to uniquely identify one particular command, BASIC finishes typing the command for you. For example, once the letters PR have been typed, the command PRINT is uniquely defined. Therefore, BASIC supplies letters INT and the required blank following the T.

BASIC also watches your spelling. As commands are being typed, letter combinations leading to nonexistant commands are not accepted and the console terminal bell is rung.

BASIC ARITHMETIC

Data Types

EX. B.H. 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 a 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 number format. This format, called exponential notation, expresses a value 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," and 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 compuations 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
```

*NOTE: This may be changed. See "CNTRL 1," Page 2-33.

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 VALUES

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 nto the range of 0-65,535, an illegal number error is generated.

STRING DATA (Extended BASIC Only)

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. String values in either programmed text or data must always be enclosed by quotation marks ("). 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.

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 2-76, and "String Variables" on Page 2-78.

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 2-42), "PRINT" (2-47), and "INPUT AND LINE INPUT" (Page 2-54).

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 (N_1, ... , N_8),

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)*

*NOTE: The $ indicates a string variable.

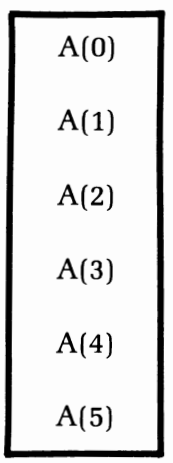
When you are referencing a subscripted variable, each element in the subscript list may consist of an arbitrarily complex expression so long as it evaluates to a numeric value within the allowable range for the indicated dimension. Thus, the subscripted variable A(5,5) would be dimensioned as:

```
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)
```

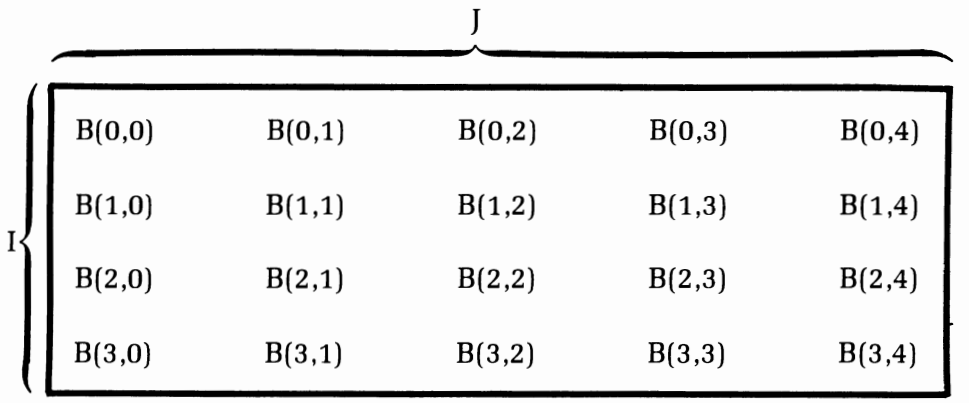
where I may assume the values of 0 to 5, allows reference to each of the six elements A(0), A(1), A(2), A(3), and A(5). A graphic representation of this 6-element, single-dimension array is shown below. Each box represents a memory location reserved for the value of the variable of the indicated name.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



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 I*J elements (i.e.: I =4 and J = 5) B(0,0), 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. IN EX.B.H. BASIC, a clear may be used to destroy an array, allowing you to use it again.

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 2-34.

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 variables 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 EX B. H. BASIC is its extensive use of expressions in situations when many other BASICs only permit variables or numbers. This feature permits you to perform sophisticated operations within a particular command or function. It is important to note that not all expressions are valid in a statement. The explanation describing the individual statement lists any limitations.

Arithmetic Operators

EX. B.H. BASIC performs exponentiation, multiplication, division, addition, and subtraction. BASIC also supports two unary operators (– and NOT). The asterisk (*) is used to signify 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 2-16.

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 6/3
 2

*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. NOTE: The plus operator (+) also performs string concatenation if both operands are string data. The following examples use the plus and minus operators:

```
*PRINT 3
 3
*PRINT 3+5
 8

*PRINT 10-3
 7

*PRINT "HEATH" + " " + "H8"
HEATH H8
*
```

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 expresssion.
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 (left-most 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 versions 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    65530 = 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, and line feed 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 2-8.

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 range of a concatenated string is 255 characters. For example:

```
*PRINT "THE HEATH" + " H8 COMPUTER"
THE HEATH H8 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"
 65535       (The relation shown is true)
*PRINT "ABC">"ABCD"
 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 "
 0           (The equality if 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 AN H8 COMPUTER"
```

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

```
*PRINT "THIS IS THE HEATH H8 COMPUTER"
```

when you type the RETURN key, immediately writes on the terminal:

```
THIS IS THE HEATH H8 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. 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 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 2-22 and "Statements Valid in the Command or Program Mode" on Page 2-31.

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 2-50.)

Line Numbers (Program Mode)

An integer number begins each line in a BASIC program. BASIC executes the program mode statements in numerical sequence, regardless of the input order. Statement numbers must lie in the range of 1 to 65,535. It is good programming practice to number lines in increments of 5 or 10 so you can insert additional statements when you are 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.

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, you can use a numeric or a string **expression**. For example, you can construct a computed GOTO by simply computing a value for a variable, X. The statement

```
GOTO X
```

then redirects the program to the computed line number.

The following three sections are organized as command mode statements, command and program mode statements. They can be found, respectively in: "Command Mode Statements" (Page 2-22), "Statements Valid in the Command or Program Mode" (Page 2-31), and "Program MODE Statements" (Page 2-53).

To simplify some practical descriptions in these sections and those following, the notation below is 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. "sep" indicates a separator. Separators such as the comma and the semi-colon are used to delineate certain portions of BASIC statements.
5. "[ ]" brackets indicate optional portions of a statement, depending on the exact function desired.
6. "var" indicates a variable. This may be a numeric or string variable, depending upon the example.
7. "name" indicates a string used to identify a date, a program, or a language record.

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
```

When BUILD is executed, the initial line number iexp1 is displayed on the terminal. Any test 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. CONTROL-C terminates BUILD.

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

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 CONTROL-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
```

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.

DUMP

The DUMP statement saves the current program text on the mass storage media connected to the load/dump port. This is usually paper tape or cassette. The current program is saved; however, no variables are saved. The specific program name is written with the data so the user may reload the program by the specified name. The form of the DUMP statement is:

```
*DUMP "name"
```

Make the tape drive ready before entering the DUMP statement. BASIC starts the drive, writes the data, and stops the drive. The CONTROL-C can be used to abort the DUMP while in progress. However, if a DUMP is aborted, an incomplete file exists on the tape.

The DUMP statement is the same as the dump statement for Extended Benton Harbor BASIC version 10.01.00 except that it produces a file type 4 (See "Appendix F"). A program dumped using the DUMP statement can only be loaded using the LOAD statement.

The string "name" may be up to 80 ASCII characters. The normal string ASCII characters are permitted. An example of a DUMP is:

```
*DUMP "STARTREK VER 1.0 03/11/77"
```

This statement dumps the program Startrek version 1.0 dated the 11th of March 1977.

FDUMP

The FDUMP statement is a combination of the DUMP and PUT statements in that it dumps both the program text and the variables (file type 6). The "name" that you give to the program is written on the tape so that you can reload the program and variables in the future using the specified name. The form of the FDUMP statement is:

```
*FDUMP "name" CR
```

The string "name" may consist of up to 80 ASCII characters. Any normal ASCII character string is permitted. Make sure the tape drive is ready before you enter the FDUMP statement. BASIC starts the drive, writes the data, and stops the drive. You can use the CONTROL-C to abort the FDUMP routine; however, the file will be incomplete. A program dumped using the FDUMP statement can only be loaded using the FLOAD statement.

FLOAD

The FLOAD statement is a combination of the LOAD and GET statements (Pages 25 and 26) in that it loads both the program and its variables (file type 6) at the same time. Since the program and its variables are stored in mass storage under a specified name, you can load them from storage using the specified name. The form of the FLOAD statement is:

```
FLOAD "name" CR

SURE?
```

A Y reply to the question "SURE?" causes BASIC to scan the mass storage device until it finds a program (file type 6) whose name matches the specified string "name." It then destroys the current program in memory and loads the new program text and program variables. Any other response cancels the FLOAD routine. If the name in the mass storage device is longer than the specified name you enter, a match on the supplied characters in the string "name" is valid. Thus, a program may be dumped (FDUMPed) with extra information entered in the name (such as program version). This lets you load a program without entering the extra information. The FLOAD command enters the LOCK mode after it has loaded the program text and the program variables.

GET

The GET statement loads variables (file type 5), previously stored on tape, into memory. The current variables in memory are destroyed, but the program text is not affected. Since the variables are stored in mass storage under a specified name, you can load them from storage using their specified name. The form of

the GET statement is:

```
GET "name" ↵

SURE?
```

A Y reply to the question "SURE?" causes BASIC to scan the mass storage device until it finds a variable (file type 5) whose name matches the specified string "name." It then destroys current variables in memory and loads the new variables. Any other response cancels the GET routine. If the name in mass storage device is longer than the specified name you enter, a match on the supplied characters in the string "name" is valid. Thus, a program may be dumped (PUT) with extra information entered in the name (such as program version number). This lets you load a program without entering the extra information. The GET command enters the LOCK mode after it has loaded the program variables.

LOAD

The LOAD statement discards the current program in memory. A specified program is loaded from the mass storage device connected to the load/dump port. The form of the LOAD statement is:

```
LOAD "name"
```

The string "name" may consist of up to 80 ASCII characters. The normal string ASCII characters are permitted. BASIC scans the mass storage device until it finds a program whose name matches the specified string. Before destroying the stored information, the user is asked "SURE?." A "Y" reply causes LOAD to proceed. Any other response cancels LOAD.

The LOAD statement is the same as the load statement for Extended Benton Harbor BASIC version 10.01.00 except that the variables in memory are not destroyed when the program (file type 4) is loaded. However, the variables will be cleared to zero if you use the RUN command to execute the program after it is loaded. Therefore, use the CONTINUE statement when you want to run the program without destroying the variables. The LOAD command enters the LOCK mode after it has loaded the program text.

NOTE: If the name on the mass storage device is longer than the specified name, a match on the supplied characters in the string "name" is valid. Thus, a program may be dumped with extra information entered in the name such as program version number and data. The program can then be loaded without it. For example:

```
*DUMP "STARTREK VER 1.0 03/11/77"
```

This program, Startrek version 1.0 dated 11 March 1977, may be loaded by

```
*LOAD "STARTREK"
 SURE? Y
```

A match is found between the first eight characters of the DUMP string "STAR-TREK" and the eight characters of the load command "STARTREK." If a null is used as the load string, the next program on the tape is loaded. Therefore, the statement

```
*LOAD ""
```

loads the next BASIC program appearing on the mass storage.

Mass storage media should be made ready before LOAD is executed. BASIC starts and stops the mass storage device. CONTROL-C may be used to abort the load part way through. Use a SCRATCH command to clear the results of an aborted load.

During a load, either one of the following two error messages may be generated:

```
SEQ ERR and
CHKSUM ERR.
```

A sequence error (SEQ ERR) is generated if the file records are not in sequence. For example, if two consecutive label records are read an error is generated, as a BASIC file consists of a label file followed by a data file. The form of the sequence error is

```
SEQ ERR
```

Type a blank after the SEQ ERR message. This will clear the error. The entire file must be reread.

A checksum error (CHKSUM ERR) is generated if the computed CRC for the record in question does not match the CRC included in the record. The form of the checksum error message is

```
CHKSUM ERR   IGNORE?
```

A Y in response to the question "ignore" aborts the error message and the next consecutive record is read. Do not ignore the checksum error unless there is no other way to recover the data. If a checksum error is flagged, the chances are very good that the data in the designated record is faulty. If you attempt to use such bad data, it may cause BASIC to crash.

LOCK

The LOCK statement protects your program by preventing the execution of the following command mode statements:

BUILD	SCRATCH
DELETE	CLEAR
LOAD	GET
RUN	FLOAD

OLDLOAD

The OLDLOAD statement lets you load program (file type 2) that were dumped using the old version (10.01.00) of Extended Benton Harbor BASIC. The OLDLOAD statement, unlike the new LOAD statement, destroys the program variables currently in memory. The form of the OLDLOAD statement is:

```
*OLDLOAD "name" ®
```

PUT

The PUT statement is a form of dump statement in that it saves the program variables on tape using a type 5 file (see "Appendix F"). It does not save the program, only the variables. The "name" that you give to the variables is written on the tape so you can reload the variables in the future using the specified name. The form of the PUT statement is:

```
*PUT "name" ®
```

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 termi

nated. 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 2-69) 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 CONTROL-C (CNTRL-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.

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.)
```

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 (CR)
```

VERIFY

The VERIFY statement permits you to check a file placed on mass storage without affecting the current program. The VERIFY command responds by indicating the name of the file found, and if the file is correct. A form of the VERIFY command is:

```
*VERIFY "name"
```

The string "name" can be the name of the record the user desires to verify or it may be a null (""); in which case, BASIC verifies the first record encountered. For example,

```
*VERIFY "STARTREK"
 FOUND STARTREK VER 1.0 03/11/77
 FILE OK
*
```

In the above example, the file containing the Startrek dump is verified. Note, that the name of the file is printed immediately as soon as the label record is encountered. The FILE OK message is printed after the data record is read and verified. VERIFY performs a checksum on the contents of all data in the file. Using the VERIFY command does not destroy any program data in memory.

During a VERIFY, one of two error messages may be generated. They are:

```
SEQ ERR and
CHKSUM ERR.
```

A sequence error (SEQ ERR) is generated if the file records are not in sequence. For example, if two consecutive label records are read an error is generated, as a BASIC file consists of a label file followed by a data file. The form of the sequence error is

```
SEQ ERR
```

Type a blank after the SEQ ERR message. This will clear the error. The entire file must be reread.

A checksum error (CHKSUM ERR) is generated if the computed CRC for the record in question does not match the checksum included in the record. The form of the CRC error message is

```
CHECKSUM ERR - IGNORE?
```

A Y in response to the question (IGNORE?) aborts the error message and the record is considered valid. Do not ignore the checksum error unless there is no other way to recover the data. If a checksum error is flagged, the chances are very good that the data in the designated record is faulty.

NOTE: The command VERIFY is not available if BASIC is patched to use an ASR console terminal as the load dump device.

Statements Valid in the Command or Program Mode

The statements in this section may be used 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.

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 forms of the CLEAR statement are:

```
*10 CLEAR
     or
*10 CLEAR varname
```

All variables, arrays, string buffers, etc., are cleared before 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.

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 CONTROL-B characters. The line number of the routine is specified as "iexp2." When a CONTROL-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:

```
10 CNTRL 0,500
20 FOR A=1 TO 9
30 PRINT A,A*A,A*A*A
40 NEXT A
50 END
500 PRINT "THAT TICKLES"
510 RETURN

*RUN

 1           1              1
 2           4              8
 3           9              27    (CONTROL-B typed)

THAT TICKLES
 4           16             64    (CONTROL-B typed)

THAT TICKLES
 5           25             125
 6           36             216
 7           49             343
 8           64             512
 9           81             729

END AT LINE 50
*
```

During the execution of the program containing these three statements, a CONTROL-B from the keyboard momentarily interrupts the regular execution of 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 M = 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 2-60.")
CNTRL 2,2	Turn display on with update (to monitor a register or memory location).

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 turns the hardware clock on and off. CNTRL 4,0 turns the clock off and CNTRL 4,1 turns it on. Once the clock has been turned off, clock dependent functions such as PAUSE iexp, and PAD(cannot be used. Turning the clock off increases execution speed approximately 11%.

NOTE: The CNTRL 1 through CNTRL 4 commands permanently reconfigure loaded BASIC. A new program does not clear them to the original state. You can reset them to their original state by using the appropriate form of the Control statement in the Command mode.

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	30 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
 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
 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 assigned the value 20 before exiting the loop.

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

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

*RUN
 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.

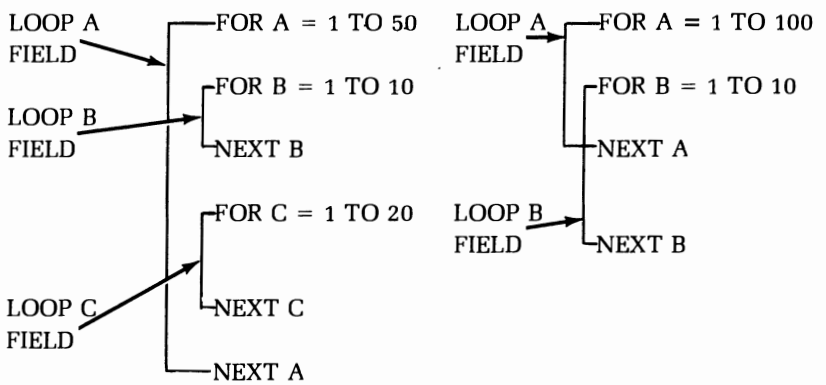
If the STEP expression and the word STEP are omitted from the FOR statement, a step of +1 is the default value. Since +1 is an extremely common step value, the STEP portion of the statement is frequently omitted. For example:

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

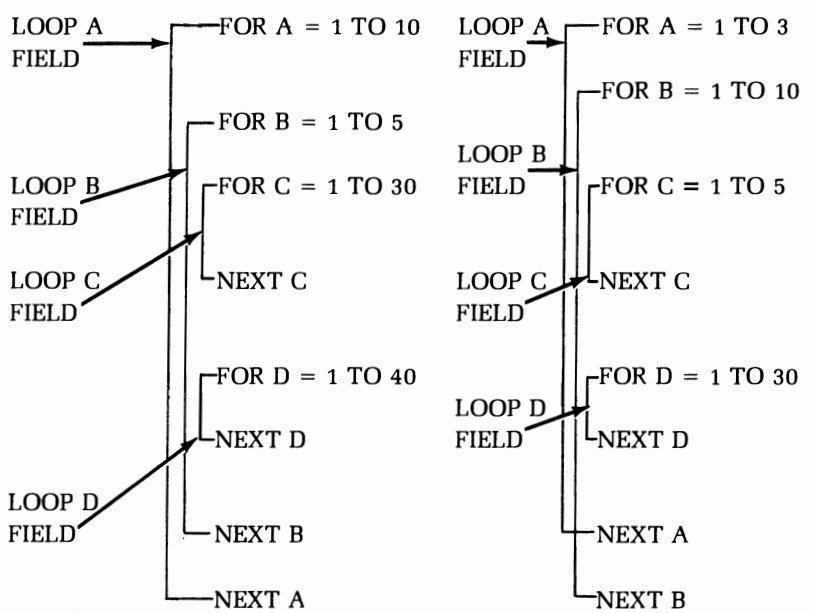
Nesting is a technique frequently used in programming. It consists of placing one or more loops completely inside another loop. The field or operating range of the loop (the lines from the FOR statement to the corresponding NEXT statement) must not cross the field of another loop. The following two examples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

LEGAL NESTING ILLEGAL NESTING

Two-Level Nesting



Three-Level Nesting



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 in EXTENDED BENTON HARBOR BASIC.

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 dependant 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
```

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 STRING.)
WORK = nnnn     (Bytes used by expression and function evaluation.)
FREE = nnnn     (Total number of free bytes.)
```

For example, running the program

```
*10 GOSUB 10
*RUN
```

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.

```
! ERROR - MEM OVR AT LINE 10
*FREE
 TEXT = 9
 SYMB = 0
 FORL = 0
 GSUB = 7232
 WORK = 0
 STRN = 0
 FREE = 16
*
```

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 iexp
```

where iexp is 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 iexp
```

When this statement is executed, program control transfers to the line number pecified by the integer expression "iexp." 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 │ iexp          or
                    │ GOTO │
IF expression       └ THEN ┘ statement
```

The expression frequently consists of two variables combined by the relational operators described in "Relational Operators" (Page 2-14). In the first form, if the result of the expression is true, control passes to the specified line number (iexp). 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

END AT LINE 40
*
```

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

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

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" (Extended BASIC). 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
AND
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.

LIST

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

```
LIST
LIST [iexp]
LIST [iexp1], [iexp2]
```

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 ("iexp") is specified, EX. B.H. BASIC lists that single line. BASIC lists the indicated line and the balance of the program lines. You can use a CONTROL-O or CONTROL-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. Refer to 'Editing Commands" (Page 2-65) or to "Appendix B" (Page 2-76) for a complete explanation of these functions.

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
 5    6    11    .833333

 END AT LINE 50

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

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

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 iexp2, . . . . . ,iexpn
```

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 iexp2 thru iexpn. If the expression "iexp1" evaluates to 1, for example, control is passed to the line number given by the expression "iexp2." If the expression "iexp1" evaluates to 3, for example, control is passed to line number given by the expression "iexp4." 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 iexp2, . . . . ,iexpn
```

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 iexp2 thru iexpn. If the expression "iexp1" evaluates to 1, for example, control is passed to the line number given by the expression "iexp2." If the expression "iexp1" evaluates to 3, for example, control is passed to line number given by the expression "iexp4." 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.

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. There are two forms of PAUSE. The first one is:

```
PAUSE
```

Once the PAUSE statement is executed, no further statements are executed until you type a console terminal character. You can terminate PAUSE by typing any key. The character will not be echoed, but it is good practice to consistently use one character such as space to terminate PAUSE. The second form of the PAUSE statement is:

```
PAUSE [iexp]
```

If iexp is specified, PAUSE waits 2 times iexp milliseconds. After 2 times iexp milliseconds pass, normal program execution continues.

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 continuing execution.

WARNING

CAUTION: You can damage BASIC when using the POKE statement (Page 2-46), causing a failure which could result in loss of program material and/or require reloading BASIC itself. The POKE statement should be confined to areas of memory, not used by the BASIC interpreter.

POKE

The POKE statement is used to write values into an assigned H8 memory location. The form of the POKE statement is:

```
POKE iexp1, iexp2
```

The low-order 8-bits of iexp2 are inserted into memory location iexp1. NOTE: iexp1 and iexp2 must be given as decimal numbers. If you wish to use octal numbers for ease in referencing to binary notation, you must use a separate octal to decimal subroutine or function to generate these numbers.

PORT

The PORT statement lets you control alternate peripheral devices, such as an H14 line printer. You may connect the device to your computer system with either an H8-4 or H8-5 Serial I/O Card. This description of the PORT command and the sample exercise are valid when your H8 computer system operates under multiple peripheral devices.

When the Extended Benton Harbor BASIC (version 10.03.00) cassette tape was initially loaded, you were directed to repeatedly press the space bar on your console control terminal after pressing the GO key on the H8 front panel Pressing the space bar signals the computer system that you want this terminal to become your "boss" terminal. The H8 initially recognizes the console assigned port address 350Q as the "boss." If no terminal is attached at location 350Q, the system software selects the terminal at location 372Q as the "boss."

With the PORT statement you can PERManently reconfigure your operating system to recognize an alternate terminal as the "boss." The PORT statement will also let you selectively assign an alternate terminal INPUT or OUTput operations. The format for the PORT statement is:

```
PORT mode address[,baud]*
```

The mode lets you select either of three operations: INPUT, OUTput or PERManent. The address is a port location. When the device is attached to an H8-4 Serial Interface Card, you must specify the baud rate.

Since the PORT command assigns console control functions away from the original console, you cannot enter commands or data from the original console.

*"Appendix G" (Page 2-104) lists program "DEMO:PORT COMMAND" that uses the PORT command with several variations of the same format. Both the address and the baud rate can be an arbitrarily complex integer expression. For instance, port 372Q could be expressed as P1 and set equal to 3*64+7*8+2.

You may still use the control-characters (CTRL-C, CTRL-S, etc.) from either the onsole or an alternate terminal. However, if you PERManently reassign console control, then the control-characters can be used only from the new "boss" console.

PRINT

The PRINT statement is used to output **data** to the console terminal. The form of the PRINT statement is:

```
PRINT [nexp1 sep1 . . . nexpn(sepn)]
```

The expressions and separators contained within the brackets are optional. When used without these optional expressions and separators, the simple PRINT statement outputs a blank line followed by a carriage-return line feed.

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 2-48). 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 H8"
*RUN
 THIS IS A HEATH H8

 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 @,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 exp1, . . . , expn
```

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
 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 expression 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 DATA statement contains more expressions than the READ statement has variables.

DATA statements may contain arbitrarily complex expressions to represent the data values. Each value expression is separated from other value expressions by a comma. A field in the DATA statement may be left null by means of two adjacent commas. This causes the associated variable to retain its 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 expressions 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 an expression contained in a DATA statement is bad, 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 expressions 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
*
```

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 2-50) 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 modes.

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 statement is used when data is to be READ from the terminal keyboard or from a mass storage device **working through the console terminal**. The form of the INPUT statement is:

```
INPUT prompt;var1, . . . , varn
```

If the first element in the list following the INPUT statement is a string, INPUT assumes it is a PROMPT and types the string in place of a question mark (?). If no prompt string is desired but the first variable is a string variable, a leading semicolon is inserted. For example:

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

This statement tells BASIC that the data to come from the console terminal is to be placed in a dimensioned string named S3.

Data input from the console terminal has a format identical to the DATA statement.

NOTE: Responses to string inputs must be enclosed in quotes.

Expressions may be supplied and null fields cause the variable to retain its previous value. If the user response does not supply sufficient data to complete the INPUT statement, another "?" prompt is issued, requesting more data input at the terminal. **CAUTION:** If you supply too much data, it will be ignored. The next INPUT statement issues a fresh READ to the terminal.

The response to the LINE INPUT statement cannot be continued on another line, as they are terminated by the return key.

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, but the string should not be enclosed in quotes.

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
 1             2             3
 END AT LINE 30
*15STOP

*RUN

 STOP AT LINE 15
*PRINT A
 1
*15                          Stop deleted

*RUN
 1             2             3

 END AT LINE 30
*
```

PREDEFINED FUNCTIONS

Introduction

There are 26 predefined functions in EX. B.H. BASIC. These functions perform standard mathematical operations such as square roots, logarithms, string manipulation, and special features. Each function has an abbreviated three- or four-letter name, followed by an argument in parentheses. As these functions are predefined, you may use them throughout a program when they are required. Predefined functions use numeric expressions (nexp), integer expressions (iexp), and string expressions (sexp). Function key words are automatically followed by an open parenthesis "(".

The abbreviation (narg) is used to indicate 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)
 5.5                    or,

*PRINT ABS(SIN(3.5))
 .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 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 error 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 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:

$$LOG_a\ N = LOG_e N / LOG_e a$$

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 2-60; for an addition example.)

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

THE PEEK FUNCTION, PEEK (iexp)

The PEEK function returns the numeric value of the byte at memory location iexp.

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 (0)

The POSITION function returns the current terminal printhead (cursor) position. Although the numeric argument (0) is ignored, it must be present to complete the function. 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             15             29   33   37
*
```

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 programmer 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 FOR A=0 TO 2
 20 PRINT RND(1)
 30 NEXT
 40 END

*RUN
 .93677
 .566681
 .53128

 END AT LINE 40
*RUN
 .564484
 .787262
 .332306

 END AT LINE 40
*20PRINT RND(0)

*RUN
 .332306
 .332306
 .332306

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

*RUN
 6.25305E-02
 6.25305E-02
 6.25305E-02

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

*RUN
 .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 END
*RUN
```

When this program is executed, the H8 front panel LEDs respond to the H8 keypad numeric entries. The program ends after nine H8 keypad entries.

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

*
```

USER DEFINED FUNCTION, USR (narg)

This function calls a user-supplied machine language function. The user-supplied function should be stored in high memory above BASIC. You should place the starting address of your machine language function at location USRFCN. (See "Appendix C.") A RET instruction exits from the user-defined function. An example of the USR function is given in "Appendix E," Page 2-99.

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 the 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

Extended 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 D" on Page 0-46 of this "Software Reference Manual." 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$(100)          } STR$ function
100
*PRINT "100"              } Normal string printing
100
*
```

THE ASCII FUNCTION, 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 D" (on Page 0-46) of the "Software Reference Manual." 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
*
```

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

The MIDDLE STRING function returns the right-hand substring expression sexp" starting with the iexp1" the 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. Fo 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 (".003E-1")
 3.00000E-04
*
```

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
```

EDITING COMMANDS

EXTENDED BENTON HARBOR BASIC provides several commands you can use to halt program execution, erase characters, delete lines, add lines, and provide other editing functions. A great number of these editing commands are common to all the Heath H8 Software packages. Their operation is covered in detail in the "Console Driver" section (Page 0-36) of the "Introduction" to your H8 Software Reference Manual.

Control-C, CTRL-C

CONTROL-C is a general-purpose cancel key. It can be used to stop a mass storage input or output operation, stop program execution, stop a listing, and to stop a program during an input statement. Using CONTROL-C results in the CTL-C error message. NOTE: A CONTROL-C causes the program to terminate at the end of a current statement.

You can continue program execution by using the CONTINUE statement.

Inputting Control

The following control characters take effect when you are inputting information from the terminal.

BACKSPACE, BKSP/CTRL-H

The BACKSPACE key (or a control-H) causes a one-character backspace. The backspace code is echoed to the terminal so devices with backspace capability physically backspace. Attempting to backspace into column zero is illegal and causes a terminal bell code to be echoed. NOTE:. Backspace can be changed at configuration. See "Product Installation" on Page 0-20 of the Software Reference Manual.

The RUBOUT key causes BASIC to discard the current line being inputted. A carriage-return and line feed is sent to the consolde terminal, and the user may now re-enter the entire line. NOTE: Rubout can be changed at configuration. See "Product Installation" on Page 0-20 to 0-24 of this "Software Reference Manual."

Outputting Control

The following control characters take effect only when you are outputting information to the console terminal. They should not be used when you are inputting information via the console terminal, as they affect characters being echoed to the console.

OUTPUT SUSPENSION AND RESTORATION, CTRL-S AND CTRL-Q

Type CONTROL-S to suspend the output and to suspend program execution. This command is particularly useful when you are using a video terminal; you can use the CONTROL-S (suspend) feature each time a screen is nearly filled and information at the top will be lost due to scrolling.

By typing CONTROL-Q, you permit BASIC to continue execution and outputting information to the terminal. CONTROL-Q cancels the CONTROL-S function.

The DISCARD FLAG, CTRL-O and CTRL-P

Type a CONTROL-O to toggle the DISCARD FLAG. Setting the DISCARD FLAG stops output on the terminal but does not halt program execution until you retype CONTROL-O or until you type a CONTROL-P to clear the DISCARD FLAG. CONTROL-O is often used to discard the remainder of long listings and other similar outputs. BASIC clears the discard flag when it returns to the command mode or when an INPUT statement is executed, so that the prompt will appear.

Command Completion

When you are inputting information from the console terminal in the command mode, or in response to an INPUT command, BASIC checks the incoming characters for the initial characters of a keyword. As soon as enough characters of a keyword are entered to uniquely identify it, and it is distinguished from a variable name, BASIC completes the keyword into the terminal. For example, to enter the command SCRATCH, type SC. Since SC uniquely determines SCRATCH and SC is not a legal variable name, BASIC types the characters RATCH immediately following the SC. Striking the backspace key backspaces over the entire word SCRATCH.

Function keywords are automatically followed by an open parenthesis "(". Other keywords are immediately followed by a blank.

Enforced Lexical Rules

BASIC enforces two lexical rules during input.

1. Two adjacent alphabetical characters must start a keyword. For example, XX is illegal as no keyword starts with "XX" and "XX" is an illegal variable name. This rule is excepted when following a REMARK statement or when the characters are contained within quotes (indicating a string).

2. A quoted string must be closed; every quote character must have a mate on the same line.

Should a character be typed which is in violation of these rules, a bell code is echoed and the character is ignored.

General Text Rules

BLANKS

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

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

2. Blanks 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 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"
```

LINE LENGTH

A line in EXTENDED BENTON HARBOR BASIC is restricted to 100 characters. This restriction on line length is completely independent of the console width, which was established by the software configuration procedure (see Page 0-21 of the "Introduction" to this Software Reference Manual). NOTE: If the console terminal, for example, was set at a width of 34 characters, the console will display three completer lines for one line of BASIC.

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
! ERROR - /0
*10PRINT 1/0
*RUN

! ERROR - /0 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 "Basic Error Table" (Page 2-70) describes the error messages generated by EXTENDED BENTON HARBOR BASIC. An explanation is given after each error message in the Comments column.

Recovering from Errors

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

NOTE: If program text in an EX. B.H. BASIC program is modified in any way, the GOSUB and FOR stacks are purged. If an error occurred in a GOSUB routine, or a FOR LOOP, the entire program must be restarted.

BASIC ERROR TABLE

EXTENDED BASIC	COMMENTS
ARG CT	Argument count. Incorrect number of arguments supplied to a DEF defined function.
CNTRL-C	CONTROL-C. Program execution or other operation aborted by a CONTROL-C typed at the console terminal.
NO DAT	Data exhausted. A READ called for more data than is available in the DATA statement.
/0	Divide by zero. An attempt to divide by zero. NOTE: Either dividing by the number zero or dividing by an expression which evaluates to zero generates this error.
NUM	Illegal number. The line number referenced by a command or program statement is not used by BASIC.
USE	Illegal usage. A command or statement is used in the improper context.
NXT	Next variable missing. No FOR statement matching the accompanying NEXT statement.
OVRFL	Overflow. Memory space is filled by program text.
RTN	Return error. A RETURN is encountered without a calling statement.
STAT#	Statement number. The referenced statement number does not exist in the program.

EXTENDED BASIC	COMMENTS
SYN	Syntax error. Command, statement, or function uses incorrect separators, functions, etc..
MEM OV	Table overflow. One of the internal tables has grown too large for memory. Check GOSUBs, FOR loops, and DIM.
SUBS RANG	Subscript range. The subscript size of a dimensioned variable exceeded the size defined by the DIM statement.
SUBS CNT	Subscript count. The number of subscripts assigned to a variable exceeds the number defined in the DIM statement.
DIM?	Not dimensioned. The subscripted variable has not been dimensioned in a DIM statement.
ILL CHA	Illegal character. An improper character is assigned to a command or function. NOTE: An attempt to assign a string to a numeric variable results in an illegal character message.
FN ?	Function error. No single line function defined by a DEF statement was found when the FN function was encountered. NOTE: The DEF FN must be executed prior to executing the FN function.
TAPE	Tape error. An error in handling the mass storage device at the load dump port.

EXTENDED BASIC	COMMENTS
CNTRL-B	CONTROL-B error. A CONTROL-B entered from the console terminal, but no CONTROL-B processing in the program.
STR LE	String length error. The length of a string exceeded 255 characters.
TYP CNFLC	Type conflict. String data supplied for a numeric variable or numeric data supplied for a string variable.
LOCK	Data LOCK engaged.

APPENDIX A

Loading Procedures

Loading From the Software Distribution Tape

1. Load the tape in the reader.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A single beep indicates a successful load.
5. Insert any "Optional Patches." See Page 2-75.
6. Press GO on the H8 front panel.
7. Repeatedly press the space bar on your terminal keyboard until the display reads:

```
EXTENDED BENTON HARBOR BASIC
 COPYRIGHT WINTEK CORP., 01/77
 COPYRIGHT HEATH CO., 06/78
 ISSUE  #10.03.00.
```

8. Configure EX. B.H. BASIC as desired, answering the following questions. Prompt each question by typing its first character on the console terminal keyboard. (See "Product Installation" on Page 0-21 in the Introduction to your H8 Software Reference Manual.)

```
•AUTO NEW-LINE (Y/N)?
•BKSP = 00008/
•CONSOLE LENGTH = 00080/
•HIGH MEMORY = 16383/
•LOWER CASE (Y/N)?
•PAD = 4/
•RUBOUT = 00127/
•SAVE?
•
```

9. If you execute SAVE, have the tape transport ready at the DUMP port.
10. To use BASIC directly from the distribution tape, type the return key at any time. The Console Terminal will display:

```
EXTENDED BENTON HARBOR BASIC #10.03.00.
```

EX. B.H. BASIC is ready to use.

Loading From a Configured Tape

1. Load the tape in the tape transport.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A single beep indicates a successful load. Insert patches.
5. Press GO on the H8 front panel.
6. Repeatedly type the space key on your terminal until the display reads. (See PORT command, Page 2-46).

```
EXTENDED BENTON HARBOR BASIC # 10.03.00
```

BASIC is ready to use in the configured form.

Optional Patches

OPTION PATCH #1

2 Stop Bits For 110 Baud

This patch is inserted for systems which use a terminal device requiring 2 stop bits (such as the Teletype Model ASR 33). It is necessary only if the terminal device is interfaced via an H8-5 SERIAL I/O Card. If the terminal device is interfaced with an H8-4 MULTI PORT SERIAL I/O Card this patch is not necessary, as the software will detect the need for 2 stop bits. Do not use this patch for devices which can run with only one stop bit. See "Appendix A" in your H8 Software Reference Manual.

NOTE: When this patch is installed, all devices interfaced via H8-5 SERIAL I/O cards will be supplied two stop bits.

```
----
     040220  200
----
```

APPENDIX B

A Summary of BASIC

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

See Page

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 2-11

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. Ex BASIC only
* /	Multiplication division
+ –	Addition subtraction

Relational Operators 2-14

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

Boolean Operators 2-15

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

See Page

String Variables 2-17

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 2-18

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 2-21

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 65535.

The Command Mode 2-19

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

Multiple Statements on One Line 2-21*

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 PAGE
BUILD	BUILD iexp1, iexp2	Automatically generated program line numbers starting at iexp1 in steps of iexp2.	2-23
CONTINUE	CONTINUE	Resumes program execution.	2-23
DELETE	DELETE [iexp1, iexp]	Deletes program lines between iexp1 and iexp2.	2-24
DUMP	DUMP "name"	Saves current program "name" on mass storage media at load dump port; "name" is up to 80 ASCII characters.	2-24
GET	GET "name"	Loads program variables.	2-25
FDUMP	FDUMP "name"	Stores program text and variables.	2-25
FLOAD	FLOAD "name"	Loads program text and variables.	2-25
LOAD	LOAD "name"	Loads program "name" from mass storage media at load dump port; "name" is up to 80 ASCII characters. Current program is destroyed.	2-26
LOCK	LOCK	Protects your program.	2-28
PUT	PUT "name"	Saves program variables on tape.	2-28
RUN	RUN	Start execution of current program. Preclears all variables, stacks, etc.	2-28
SCRATCH	SCRATCH SURE?Y	Clears all program and data storage area. Any response to SURE but Y cancels SCRATCH.	2-29
UNLOCK	UNLOCK	Restores all command mode statements to normal usage.	2-29
VERIFY	VERIFY "name"	Performs a checksum on the mass storage record titled "name." No response if record is bad.	2-29

Command and Program Mode Statements

COMMAND	FORM	DESCRIPTION	SEE PAGE
CLEAR	CLEAR [varname]	Clears all variables, arrays, string buffers, etc. Optionally clears named variable (varname). Specified functions and arrays as V(.	2-31
CONTROL	CTRL iexp1, iexp2	CTRL 0 sets a GOSUB to line iexp2 when a CONTROL-B is typed.	2-31
		CTRL 1 sets iexp2 digits before exponential format is used.	2-33
		CTRL 2 controls the H8 front panel. If iexp2: = 0, display off; =1, display on without update; =2, display on with update.	2-33
		CTRL 3 sets the width of a print zone to iexp2 columns.	2-33
		CTRL 4 controls the H8 hardware clock. iexp2 = 0, clock off. iexp2 = 1, clock on.	2-33
DIMENSION	DIMvarname(iexp1 [, ... ,iexpn]) [,varname2(. . . .)]		
		Defines the maximum size of variable arrays.	2-34
FOR/NEXT	FOR var = nexp1 TO nexp2 [STEP nexp3]		
	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 than nexp2 if STEP is negative). The statement after NEXT is then executed.	2-35

COMMAND	FORM	DESCRIPTION	SEE PAGE
FREE	FREE	Displays the amount of memory assigned to tables and text	2-38
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.	2-39
GOTO	GOTO iexp	Unconditionally transfers the program execution sequence to the line iexp.	2-40
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.	2-41
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.	2-42
LIST	LIST[iexp1] [,iexp2]	Lists the entire program on the console terminal. Lists the line iexp1 or the range of lines iexp1 to iexp2.	2-43
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.	2-44

COMMAND	FORM	DESCRIPTION	SEE PAGE
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.	2-44
OUT	OUT iexp1, iexp2	Outputs a number iexp2 to output port iexp1.	2-45
PAUSE	PAUSE [iexp]	Ceases program execution until a console terminal key is typed. Ceases program execution for 2 × iexp mS.	2-45
POKE	POKE iexp1, iexp2	Writes a number iexp2 into memory location iexp1.	2-46
PORT	PORT mode [adrs[,baud]]	Reassigns the console terminal to a specified port. The mode is either INPUT, OUTput or PERManent.	2-46
PRINT	PRINT(nexp. sep 1 . . . 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 value on a new line.	2-47
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.	2-50

COMMAND	FORM	DESCRIPTION	SEE PAGE
REMARK	REM	Text following the REM is not executed and is used for commentary only.	2-51
RESTORE	RESTORE	Causes the program to reset the DATA pointer, thus reusing data at the first DATA statement.	2-51
STEP	STEP iexp	Executes iexp lines of the program. Then returns BASIC to the command mode.	2-52

Program Mode Statements

DEF	DEF FN varname (arg list) = exp		
		Defines a single-line program function created by the user.	2-53
END	END	Causes control to return to the command mode.	2-54
INPUT	INPUT prompt; var1, ... ,varn		
		Reads data from the console terminal. String data must be enclosed in quotes.	2-54
LINE INPUT	LINE INPUT prompt; var1, . . . ,varn		
		Read string data from the terminal. String data for LINE INPUT is not enclosed in quotes.	2-54
STOP	STOP	Causes BASIC to enter the command mode when the statement containing STOP is executed.	2-55

Predefined Functions

FUNCTION	DEFINITION	SEE PAGE
ABS (nexp)	Returns the absolute value of nexp.	2-56
ATN (nexp)	Returns the arctangent of nexp (radians).	2-56
COS (nexp)	Returns the cosine of nexp (radians).	2-57
EXP (nexp)	Returns e^{nexp}.	2-57
INT (narg)	Returns the integer value of narg.	2-57
LOG (nexp)	Returns the natural logarithm of nexp.	2-57
PAD (0)	Returns the value of the H8 front panel key pressed. Includes key debounce.	2-58
PEEK (iexp)	Returns the numeric value at memory location iexp.	2-58
PIN (iexp)	Returns the data input from port iexp.	2-58
POS (0)	Returns the current terminal printhead (cursor) position (by column number).	2-58
RND (narg)	Returns a random number. If narg >0, RND is next in the series. If narg = 0, RND is the previous random number. If narg<0, RND algorithm uses narg as a new seed.	2-58
SEG (narg)	Returns the correct eight-bit number to display narg (0-9) on the H8 LEDs.	2-60
SGN (narg)	Returns +1 if narg is positive. Returns −1 if narg is negative. Returns 0 if narg is zero.	2-60
SIN (nexp)	Returns the sine of nexp (radians).	2-60
SPC (iexp)	Positions printhead (cursor) iexp columns to the right.	2-62
SQR (narg)	Returns the square root of narg.	2-61

FUNCTION	DEFINITION	SEE PAGE
TAB (iexp)	Position printhead (cursor) to the iexp th column.	2-62
USR (narg)	Calls a user-written machine language function to evaluate narg.	2-61
MAX (nexp1, . . . ,nexpn)	Returns the maximum value of expressions nexp 1 thru nexpn.	2-61
MIN (nexp 1, . . . ,nexpn)	Returns the minimum value of expressions nexp 1 thru nexpn.	2-61
TAN (nexp)	Returns the tangent of nexp (radians).	2-62
CHR$ (iexp)	Returns the ASCII character iexp.	2-62
STR$ (narg)	Returns narg encoded into ASCII with leading and trailing blanks as in the print statement.	2-63
\SC (sexp)	Returns the ASCII code for the first character in the string sexp.	2-63
LEFT$ (sexp, iexp)	Returns the left iexp characters of the string sexp.	2-63
RIGHT$ (sexp, iexp)	Returns the right iexp characters of the string sexp.	2-63
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.	2-63
VAL (sexp)	Returns the numeric value of the number encoded in the string.	2-64
LEN (sexp)	Returns the length of sexp.	2-64

Editing Commands

COMMAND	FUNCTION	SEE PAGE
CONTROL-C	General-purpose cancel. Returns BASIC to monitor mode from any operation or program execution.	2-65
CONTROL-S	Suspends the output to the console terminal and suspends program execution.	2-66
CONTROL-Q	Restores the output to the console terminal and restores program execution.	2-66
CONTROL-O	Toggles the output discard flag. Does not stop program execution.	2-66
CONTROL-P	Clears the discard flag set by CONTROL-O.	2-66
CONTROL-H	Causes a one-character backspace.	2-65

APPENDIX C

BASIC Utility Routines

The following pages contain a description of several utility routines included in EXTENDED BENTON HARBOR BASIC. They can be used with user-written machine language routines called by the USR function. See "Appendix D" for Utility Routine entry points.

```
BASIC - WINTEK BASIC INTERPRETER.                                HEATH X8ASM V1.0  02/18/77
SELECTED SOURCE LISTING.                                          13:12:38  01-APR-77  PAGE     1

000.000                    2          XTEXT   MTR

                          74  ***      BASIC - *WINTEK* BASIC INTERPRETER.
                          75  *
                          76  *        J. G. LETWIN, 09/76, FOR *WINTEK* CORPORATION.
                          77  *                                 LAFAYETTE, IN.

                          79  ***      COPYRIGHT 09/1976, *WINTEK* CORPORATION,
                          80  *                           902 N. 9TH ST.
                          81  *                           LAFAYETTE, IN. 47901

                          83  **       LOW-MEMORY CELLS USED BY BASIC
                          84
040.064                   85           ORG     7*34.+.UIVEC
                          86
040.064                   87           DS      2               ACCX TYPE
040.066                   88  ACCX     DS      4
040.072                   89           DS      2               ACCY TYPE
040.074                   90  ACCY     DS      4
                          91
                          92
```

```
                                95  **      USR - CALL USER ASSEMBLY LANGUAGE FUNCTION.
                                96  *
                                97  *       THE *USR* FUNCTION IS ACTUALLY A CALL TO A USER-WRITTEN ROUTINE
                                98  *       WHICH MUST HAVE BEEN PREVIOUSLY LOADED INTO MEMORY BY THE USER.
                                99  *
                               100  *       THE ADDRESS OF THE FUNCTION'S ENTRY POINT MUST BE IN *USRFCN*.
                               101  *
                               102  *       BASIC MUST HAVE BEEN PREVIOUSLY CONFIGURED SO THAT THE USER
                               103  *       FUNCTION RESIDES IN MEMORY ABOVE THE STACK POINTER (HIGH MEMORY)
                               104  *       OR ELSE BASIC WILL OVERLAY THE FUNCTION WITH DATA.
                               105  *
                               106  *       THE FUNCTION IS ENTERED WITH A POINTER TO THE SINGLE ARGUMENT (IN
                               107  *       FLOATING POINT), AND MAY RETURN A FLOATING POINT OR STRING ARGUMENT.
                               108  *       IF NO RETURN VALUE IS PLACED IN *ACCX*, THEN THE ORIGINAL ARGUMENT
                               109  *       REMAINS THERE, AND USR( RETURNS ITS ARGUMENT AS ITS VALUE.
                               110  *
                               111  *       ENTRY   (BC) = *ACCX
                               112  *       EXIT    (ACCX) CONTAINS VALUE
                               113  *       USES    ALL
                               114
                               115
                               116
___.___  000 000               117  USRFCN  DW      0               ADDRESS OF USER FUNCTION ENTRY POINT
                               118  *                               IF = 0, USR( IS NOT LEGAL
```

```
                        121  **      ERROR PROCESSING.
                        122  *
                        123  *       THESE ERROR PROCESSORS ARE ENTERED WHEN AN ERROR IS DETECTED.
                        124  *
                        125  *       CONTROL PASSES DIRECTLY BACK TO COMMAND MODE.
                        126
                        127
---'---                 128  ERR.CC  EQU     *               CONTROL-C
                        129
---'---                 130  ERR.CB  EQU     *               CNTRL-B
                        131
---'---                 132  ERR.DE  EQU     *               DATA EXHAUSTED
                        133
---'---                 134  ERR.DO  EQU     *               /0
                        135
---'---                 136  ERR.IN  EQU     *               ILLEGAL NUMBER
                        137
---'---                 138  ERR.IU  EQU     *               ILLEGAL USAGE
                        139
---'---                 140  ERR.NV  EQU     *               NEXT VARIABLE MISSING
                        141
---'---                 142  ERR.OV  EQU     *               OVERFLOW
                        143
---'---                 144  ERR.RE  EQU     *               RETURN ERROR
                        145
---'---                 146  ERR.SL  EQU     *               STRING LENGTH
                        147
---'---                 148  ERR.SN  EQU     *               STATEMENT NUMBER
                        149
---'---                 150  ERR.SY  EQU     *               SYNTAX ERROR
                        151
---'---                 152  ERR.TC  EQU     *               TYPE CONFLICT
                        153
---'---                 154  ERR.TO  EQU     *               TABLE OVERFLOW
                        155
---'---                 156  ERR.SR  EQU     *               SUBSCRIPT RANGE
                        157
---'---                 158  ERR.SC  EQU     *               SUBSCRIPT COUNT
                        159
---'---                 160  ERR.ND  EQU     *               NOT DIMENSIONED
                        161
---'---                 162  ERR.IC  EQU     *               ILLEGAL CHARACTER
                        163
---'---                 164  ERR.UD  EQU     *               UNDEFINED FUNCTION
                        165
---'---                 166  ERR.TP  EQU     *               TAPE ERROR
```

```
                    169  **      CVX - COPY VALUE INTO 'X' ACCUMULATOR.
                    170  *
                    171  *       CVX COPIES A 4 BYTE VALUE INTO THE X ACCUMULATOR.
                    172  *
                    173  *       ENTRY   (DE) = ADDRESS OF VALUE
                    174  *       EXIT    COPIED
                    175  *       USES    A,F
                    176
                    177
---.---             178  CVX     EQU     *



                    180  **      CXY - COPY (ACCX) INTO (ACCY)
                    181  *
                    182  *       ENTRY   NONE
                    183  *       EXIT    NONE
                    184  *       USES    A,F,D,E
                    185
                    186
---.---             187  CXY     EQU     *



                    189  **      CXV - COPY X TO VALUE.
                    190  *
                    191  *       CXV COPIES THE CONTENTS OF THE 'X' ACCUMULATOR INTO A MEMORY
                    192  *       LOCATION.
                    193  *
                    194  *       ENTRY   (DE) = TARGET ADDRESS
                    195  *       EXIT    COPIED
                    196  *       USES    A,F
                    197
                    198
---.---             199  CXV     EQU     *



                    201  **      IFIX - SPLIT NUMBER INTO INTEGER AND FRACTION.
                    202  *
                    203  *       IFIX FIXES ((DE)) INTO AN INTEGER.
                    204  *
                    205  *       ENTRY   (DE) = ADDRESS OF NUMBER
                    206  *       EXIT    (DE) = INTEGRAL PART OF 0<=N<=65535
                    207  *               TO ERR.IN OTHERWISE
                    208
                    209
---.---             210  IFIX    EQU     *
```

```
BASIC - WINTER BASIC INTERPRETER.                          HEATH X8ASM V1.0  02/18/77
UTILITY SUBROUTINES                                         13:12:43  01-APR-77  PAGE     5


                        212  **      IFLT - FLOAT NUMBER.
                        213  *
                        214  *       ENTRY   (DE) = VALUE
                        215  *       EXIT    (ACCX) = NUMBER VALUE
                        216  *               (DE) = #ACCX-1
                        217
                        218
     ---*---            219  IFLT    EQU     *



                        221  **      TDI - TYPE DECIMAL INTEGER.
                        222  *
                        223  *       TDI TYPES AN INTEGER AS A 5 PLACE NUMBER. LEADING ZEROS ARE
                        224  *       SUPPRESSED.
                        225  *
                        226  *       ENTRY   (DE) = NUMBER
                        227  *       EXIT    TYPED
                        228  *       USES    A,F,D,E
                        229
                        230
     ---*---            231  TDI     EQU     *



                        233  **      XCY - EXCHANGE (ACCX) WITH (ACCY)
                        234  *
                        235  *       ENTRY   NONE
                        236  *       EXIT    NONE
                        237  *       USES    A,F
                        238
                        239
     ---*---            240  XCY     EQU     *



                        242  **      ZRO - ZERO MEMORY.
                        243  *
                        244  *       ZRO ZEROS A FIELD OF MEMORY.
                        245  *
                        246  *       ENTRY   (HL) = ADDRESS
                        247  *               (DE) = COUNT
                        248  *       EXIT    NONE
                        249  *       USES    A,F,D,E,H,L
                        250
                        251
     ---*---            252  ZRO     EQU     *
```

```
                        **      H8 FLOATING POINT FORMAT:
                        *
                        *      SINGLE-PRECISION FLOATING POINT NUMBERS ARE REPRESENTED
                        *      BY A 4-BYTE VALUE. THE NUMBER CONSISTS OF A 3 BYTE
                        *      TWO'S COMPLEMENT MANTISSA, AND A ONE-BYTE BIASED BINARY
                        *      EXPONENT.  THE NUMBER FORMAT IS:
                        *
                        *
                        *        N+0     LEAST SIGNIFICANT MANTISSA BYTE
                        *        N+1     MID   SIGNIFICANT MANTISSA BYTE
                        *        N+2     MOST  SIGNIFICANT MANTISSA BYTE
                        *        N+3     BIASED EXPONENT
                        *
                        *
                        *      EXPONENT:
                        *      ---------
                        *
                        *      EACH FLOATING POINT NUMBER CONTAINS A BINARY EXPONENT.
                        *      THE EXPONENT CONTAINS A BIAS OF 128 (200 OCTAL).
                        *
                        *      THUS AN EXPONENT OF 0 IS ENTERED AS 200Q, AN EXPONENT OF -10
                        *      IS CODED AS 166Q, ETC. THE NUMBER ZERO IS TRAEATED AS A
                        *      SPECIAL CASE: ITS EXPONENT (AND MANTISSA) IS ALWAYS 0.
                        *
                        *
                        *      MANTISSA:
                        *      ---------
                        *
                        *      THE MANTISSA OCCUPYS 3 BYTES, FOR A TOTAL OF 24 BITS. THE NUMBERS
                        *      ARE STORED IN TWO'S COMPLEMENT NOTATION. THE HIGH ORDER
                        *      BIT IS THE SIGN BIT, THE NEXT BIT (100Q) IS THE MOST SIGNIFICANT
                        *      DATA BIT. NOTE THAT FLOATING POINT NUMBERS SHOULD ALWAYS
                        *      BE NORMALIZED, SO THAT THE MOST SIGNIFICANT DATA BIT IS THE
                        *      OPPOSITE OF THE SIGN BIT'S VALUE.
                        *
                        *      THE FLOATING POINT ROUTINES SUPPLIED WILL NOT OPERATE ON NON-
                        *      NORMALIZED DATA.
                        *
                        *      EXAMPLES:
                        *      ---------
                        *
                        *      DECIMAL NUMBER          M3  M2  M1  EX
                        *
                        *        1.0                   000 000 100 201
                        *        0.5                   000 000 100 200
                        *        0.25                  000 000 100 177
                        *       10.0                   000 000 120 204
                        *        0                     000 000 000 000
                        *        0.1                   146 146 146 175
                        *       -1.0                   000 000 200 200
                        *       -0.5                   000 000 200 177
                        *      -10.0                   000 000 260 204
```

```
                    309  **      FPADD - FLOATING POINT ADD.
                    310  *
                    311  *       ACCX = ACCX + (DE)
                    312  *
                    313  *       ENTRY   (DE) = POINTER TO 4 BYTE FP VALUE
                    314  *       EXIT    ACCX = RESULT
                    315  *               SUPPLIED VALUE UNCHANGED
                    316  *       USES    A,F
                    317
                    318
       ---•---      319  FPADD   EQU     *


                    321  **      FPSUB - FLOATING POINT SUBTRACT.
                    322  *
                    323  *       FPSUB COMPUTES (DE) - ACCX
                    324  *
                    325  *       ENTRY   (DE) = POINTER TO 4 BYTE FP VALUE
                    326  *       EXIT    ACCX = RESULT
                    327  *               SUPPLIED VALUE UNCHANGED
                    328  *       USES    A,F
                    329
                    330
       ---•---      331  FPSUB   EQU     *


                    333  **      FPNRM - FLOATING POINT NORMALIZE.
                    334  *
                    335  *       FPNRM NORMALIZES THE CONTENTS OF (ACCX).
                    336  *
                    337  *       ENTRY   NONE
                    338  *       EXIT    (ACCX) NORMALIZED
                    339  *       USES    A,F
                    340
                    341
       ---•---      342  FPNRM   EQU     *


                    344  **      FPNEG - FLOATING POINT NEGATE.
                    345  *
                    346  *       FPNEG NEGATES THE CONTENTS OF ACCX.
                    347  *
                    348  *       ENTRY   NONE
                    349  *       EXIT    (ACCX) = -(ACCX)
                    350  *       USES    A,F
                    351
                    352
       ---•---      353  FPNEG   EQU     *
```

```
                        355  **      FPMUL - FLOATING POINT MULTIPLY.
                        356  *
                        357  *       ENTRY   (DE) = ADDRESS OF Y
                        358  *       EXIT    ACCX = ACCX * Y
                        359  *       USES    A,F
                        360
                        361
                        362  FPMUL   EQU     *


                        364  **      FPDIV - FLOATING POINT DIVIDE.
                        365  *
                        366  *       ACCX = ACCX/Y
                        367  *
                        368  *       ENTRY   (DE) = POINTER TO Y
                        369  *       EXIT    (ACCX) = RESULT
                        370  *       USES    A,F
                        371
                        372
                        373  FPDIV   EQU     *
```

```
BASIC - WINTEK BASIC INTERPRETER.                          HEATH X8ASM V1.0  02/18/77
ASCII/FLOATING CONVERSION ROUTINES.                    13:12:48  01-APR-77  PAGE     9


                        376  **      ATF - ASCII TO FLOATING.
                        377  *
                        378  *       ATF CONVERTS AN ASCII STRING INTO A FLOATING POINT VALUE
                        379  *       IN ACCX.
                        380  *
                        381  *       SYNTAX
                        382  *
                        383  *        NNNN [.NNN] [E [+-] NN]
                        384  *
                        385  *       ENTRY   (HL) = ADDRESS OF TEXT
                        386  *       EXIT    (HL) UPDATED
                        387  *               (ACCX) = VALUE
                        388  *       USES    A,F,H,L
                        389
                        390
                   ---*---  391  ATF     EQU     *



                        393  **      FTA - FLOATING TO ASCII.
                        394  *
                        395  *       FTA CONVERTS A FLOATING POINT NUMBER INTO AN ASCII
                        396  *       REPRESENTATION.
                        397  *
                        398  *       ENTRY   (ACCX) = VALUE
                        399  *               (HL) = ADDRESS TO STORE TEXT
                        400  *       EXIT    (A) = LENGTH OF STRING DECODED
                        401  *               (DE) = ADDRESS OF LAST BYTE
                        402  *       USES    A,F,D,E
                        403
                        404
                   ---*---  405  FTA     EQU     *
```

```
                                    408  **      FLOATING POINT VALUES.
                                    409  *
                                    410
                                    411          LON     G                   LIST GENERATED BYTES
                                    412
___.___  000 000 100                413  FP1.0   DB      0,0,100Q,201Q
         201
___.___  000 000 120                414  FP10.   DB      0,0,120Q,204Q
         204
___.___  146 146 146                415  FP0.1   DB      146Q,146Q,146Q,175Q
         175
___.___  000 000 000                416  FP0.0   DB      0,0,0,0
         000
___.___  022 170 233                417  NPI.2   DB      022Q,170Q,233Q,201Q      -PI/2
         201
___.___  022 170 233                418  NPI2    DB      022Q,170Q,233Q,203Q      -PI*2
         203
___.___  022 170 233                419  NPI     DB      022Q,170Q,233Q,202Q      -PI
         202
___.___  022 170 233                420  NPI.4   DB      022Q,170Q,233Q,200Q      -PI/4
         200
___.___  356 207 144                421  PI.4    DB      356Q,207Q,144Q,200Q       PI/4
         200
                                    422
                                    423
___.___                             425          END

ASSEMBLY COMPLETE
  425 STATEMENTS
    0 ERRORS
20312 BYTES FREE
```

APPENDIX D

Entry Points to Utility Routines

ADDRESS FOR
EXTENDED BENTON HARBOR BASIC
ISSUE # 10.05.XX

ACCX	040066	ACCY	040074	ATF	104144	CVX	073023
CXV	073052	CXY	073036	ERR.CB	070140	ERR.CC	070130
ERR.DO	070161	ERR.DE	070150	ERR.IC	070356	ERR.IN	070166
ERR.IU	070174	ERR.LK	070202	ERR.ND	070347	ERR.NV	070211
ERR.OV	070220	ERR.RE	070231	ERR.SC	070334	ERR.SL	070237
ERR.SN	070251	ERR.SR	070317	ERR.SY	070262	ERR.TC	070270
ERR.TO	070305	ERR.TP	071000	ERR.UD	070371	FP0.0	111332
FP0.1	111326	FP1.0	111316	FP10.	111322	FPADD	101201
FPDIV	103101	FPMUL	102144	FPNEG	102123	FPNRM	102023
FPSUB	102007	FTA	105060	IFIX	073366	IFLT	074024
NPI	111346	NPI.2	111336	NPI.4	111352	NPI2	111342
PI.4	111356	TDI	077121	USRFCN	111303	XCY	100015
ZRO	100047						

APPENDIX E

An Example of USR

The following program, written on an H8 computer system using a 24K memory, illustrates the USR function in EX. B.H. BASIC (version 10.05.00). The program calls a machine language subroutine (Page 2-100) that performs the simple task of sounding the H8 audio oscillator. The program was written under TED-8 and assembled by HASL-8. The object program could have been loaded into H8 memory by HASL-8. However, use the H8 keypad to enter the object code because the program is small. The program works as follows:

1. Load the BASIC software distrubution tape*. ("Appendix A").

2. SET the HIGH MEMORY limit at 24500.

3. ENTER the following BASIC program lines:

```
210 PRINT TAB(25);"U S R FUNCTION"
220 FOR A=1 TO 5
230 LET B=USR(2000)
240 PAUSE 1000
250 NEXT A
260 END
```

4. Use the H8 front panel to enter the object code from the machine language subroutine (DEMO: USR Page 2-100) starting from memory location 140.000.

5. When BASIC calls the USR function, it finds the starting address of the subroutine at USRFCN. Therefore, use the H8 front panel to enter the starting address (140.000) into USRFCN.

   Enter the low byte (000) into memory location 111303.

   Enter the high byte (140) into memory location 111304.

6. Use the warm start (040103) to reenter BASIC.

7. RUN the program. A warning message and a series of short "Beeps" will be generated by the H8 computer system.

*You must use a distribution tape so that you can set HIGH MEMORY limits.

```
                                                    HEATH ASM #104.02.00.
                                                    Page  1

                        ***********************************************************
                        *                                                         *
                        *       DEMO:  USR                                        *
                        *                                                         *
                        *       SOUNDS H8 AUDIO OSCILLATOR                        *
                        *         AND TURNS ON L.E.D. DISPLAY WHEN USR            *
                        *       ARGUMENT IS NOT A ZERO.                           *
                        *                                                         *
                        ***********************************************************
                        *
                        ***     SYSTEM DEFINITIONS
                        *
140.000                         ORG     140000A         ORIGINATE SUBROUTINE
002.140                 HORN    EQU     2140A           PAM8 SUBROUTINE
000.053                 DLY     EQU     0053A           DELAY:  PAM8 -- SUB.
040.010                 .MFLAG  EQU     40010A          USER FLAG OPTIONS
040.013                 FPLEDS  EQU     40013A          FRONT PANEL L.E.D. LOCATIONS
000.002                 UO.DDU  EQU     2A              DISABLE DISPLAY UPDATE
                        *************************************************************************
                        *
                        *       THE FOLLOWING ENTRY POINTS TO EXTENDED BENTON HARBOR BASIC
                        *       IS FOR ISSUE #10.05.XX (APPENDIX D).  THESE ADDRESSES ARE
                        *       VARIABLE AND WILL BE DIFFERENT FROM OTHER ISSUES RELEASED.
                        *       ALWAYS CHECK THE VERSION NUMBER BEFORE USING THE UTILITIES.
                        *
102.023                 FPNRM   EQU     102023A         BASIC -- FLOATING POINT NORM.
111.346                 NPI     EQU     111346A         -3.14 FLOATING POINT
073.023                 CVX     EQU     73023A          COPIES 4 BYTE VALUE INTO 'X'
                        *************************************************************************
                        *
                        *       TURN ON H8 SPEAKER AND MAKE NOISE
                        *
140.000  076 372       USR     MVI     A,500/2         SET UP 500 MS BEEP
140.002  315 140 002           CALL    HORN            MAKE NOISE
                        *************************************************************************
                        *
                        *       TEST IF USR(ARGUMENT) EQUALS ZERO!
                        *
                        *       UPON ENTRY TO THE USR FUNCTION, REGISTER PAIR 'BC'
                        *       CONTAINS THE ADDRESS OF FLOATING POINT ACCUMULATOR
                        *       (ACCX).  THIS INFORMATION LETS YOU EASILY
                        *       INCREMENT AND TEST IF THE EXPONENT IS ZERO.
                        *
140.005  003                   INX     B               INCREMENT 'ACCX'
140.006  003                   INX     B                 UP TO THE
140.007  003                   INX     B                   EXPONENT
140.010  012                   LDAX    B               (A)=EXP 'ACCX'
140.011  247                   ANA     A               SET CONDITION CODE
140.012  310                   RZ                      IF ZERO - RETURN TO BASIC
                        *************************************************************************
                        *
                        *       IF THE ARGUMENT IS ZERO, THEN WE EXIT FROM THE
                        *       ROUTINE.  OTHERWISE, WE CONTINUE THE PROGRAM AND LIGHT
                        *       THE FRONT PANEL DISPLAYS.
                        *
140.013  076 002               MVI     A,UO.DDU        DISABLE NORMAL UPDATING
140.015  062 010 040           STA     .MFLAG            DONE
```

```
                        ** ************************************************************************
                        *
                        *       DISPLAY 'USR' MESSAGE ON L.E.D.'S
                        *
140.020 001 013 040             LXI     B,FPLEDS        INITIALIZE L.E.D. ADDRESS
140.023 021 065 140             LXI     D,DATA          ADDRESS OF L.E.D. PATTERN
140.026 032             LED.ON  LDAX    D               MOVE 'DATA' INTO 'A'
140.027 002                     STAX    B               TRANSFER 'DATA' INTO FPLED
140.030 023                     INX     D                  SET 'D' FOR NEXT ROUND
140.031 003                     INX     B                  SET 'B'
140.032 171                     MOV     A,C             TEST IF END
140.033 376 024                 CPI     24Q                OF DATA
140.035 332 026 140             JC      LED.ON          NO -- DO AGAIN!
140.040 076 372                 MVI     A,500/2         500 MS DELAY
140.042 315 053 000             CALL    DLY             .... WAIT ....
140.045 315 053 000             CALL    DLY
140.050 315 053 000             CALL    DLY
                        ** ************************************************************************
                        *
                        *       LOAD 'ACCX' WITH NEGATIVE 'PI' (-3.14..) BEFORE EXIT
                        *
140.053 021 346 111             LXI     D,NPI           -3.14..
140.056 315 023 073             CALL    CVX             COPIES 4 BYTES INTO 'ACCX'
140.061 315 023 102             CALL    FPNRM           NORMALIZE 'ACCX'
                        ** ************************************************************************
                        *
                        *       RETURN TO BASIC
                        *
140.064 311                     RET
                        ** ************************************************************************
                        *
                        *       ARRAY STORAGE FOR FRONT PANEL L.E.D. DATA
                        *          'H-8 USR FNC'
                        *       SEE YOUR PAM-8 PROGRAM LISTING WHEN YOU WANT
                        *       TO DECODE L.E.D. DISPLAYS.
                        *
140.065 222 376 200     DATA    DB      222Q,376Q,200Q
140.070 203 244 336             DB      203Q,244Q,336Q
140.073 234 326 316             DB      234Q,326Q,316Q
140.076 000                     END     USR

00099 Statements Assembled
20399 Bytes Free
 No Errors Detected
```

APPENDIX F

System Data Format

EXTENDED BENTON HARBOR BASIC (version 10.03.00) contains the three original BASIC file types (001, 002, and 003) plus the file types (004, 005, and 006) from version 10.02.00.

001 Memory Image.

002 BASIC Programs using EXTENDED BENTON HARBOR BASIC version 10.01.00.

003 Compressed Text.

004 BASIC Programs using EXTENDED BENTON HARBOR BASIC version 10.02.00.

005 BASIC Data using EXTENDED BENTON HARBOR BASIC version 10.02.00.

006 BASIC Programs and Data using EXTENDED BENTON HARBOR BASIC version 10.02.00.

The three original file types (001, 002, and 003), which are explained in the "Tape Files" section of the H8 Software Reference Manual, do not change. The three file types from version 10.02.00 are explained in the following paragraphs.

BASIC PROGRAMS (Type = 004)

This file type is used by BASIC (version 10.02.00) when you load and dump **programs**. It is the same as file type 002 except that it uses a different format to dump and load the program. This file always has a label record (#0), a table record (#1) that contains a table describing what data has been stored, and a data record (#2) containing the actual data (program text).

BASIC DATA (Type = 005)

This file type is the same as file type 004 except that it is used to load and dump **data** (program variables).

BASIC PROGRAM AND DATA (Type = 006)

This file type is the same as file type 004 except that it is used to load and dump both the **program text** and **program variables** together.

READING THE DISPLAYS

When the H8 computer is reading or writing data on a tape transport, the front panel displays are continually displaying data about the tape operation. The information contained in the "Reading The Displays" section of the H8 Software Reference Manual explains how to read the displays. However, the data-type LED (the one on the right) displays the type of data being read or written. This information is displayed as:

DISPLAY	DATA/TYPE
1	Memory Image
2	BASIC Program Text (version 10.01.00)
3	Compressed Text
4	BASIC Program Text (version 10.02.00)
5	BASIC Program Variables (version 10.02.00)
6	BASIC Program Text and Program Variables (version 10.02.00)

APPENDIX G

Special Features

Extended Benton Harbor BASIC version 10.03.00 provides special features that let you use the H8-4 Multiport Serial I/O Card. These features include the selection of alternate ports and variable baud rates. The H8-4 circuit board design lets you take advantage of these new features. For example, when the H8 is interfaced to an alternate terminal or a line printer, the device drivers are supplied with the software.

The information presented in this Appendix supplements the material presented in section five on the PORT, GET, and PUT commands. The PORT command lets you communicate with any peripheral device connected to your H8 microcomputer with either an H8-4 or H8-5 Serial I/O Card. This is the only function that the PORT command provides.

The PUT statement lets you store program variables onto a cassette tape. The GET statement lets you load program variables from a cassette tape. The example program using PUT and GET (Page 2-111) shows a simple application for this feature.

"Penny Match," another sample program, provides the techniques and command usage you need to use the PORT command. The "System Configuration" section of this Manual (Page 0-6) provides the information that will let you select the proper port configuration. The port allocation scheme of your computer system must be identical with the allocation required by the sample programs.

Port Command

OUTput

The OUTput mode directs console output, using the PRINT statement. You can direct output to each peripheral device connected to your H8 computer system. The form of the PORT OUTput command is:

```
PORT OUT address[,baud]
```

For example, the following command sequence would list lines 100 through 350 of your program on the H14 printer (port address 340Q) at 4800 baud. Control is returned to the original console immediately after the printer lists the program lines.

```
*PORT OUT 224,4800 : LIST 100,350
```

A PORT OUTput command remains in effect until you execute another PORT OUTput command or BASIC returns to the command mode.

INPUT

The INPUT mode directs an alternate terminal to read data using an INPUT statement. You can read data from each peripheral device connected to your H8 computer system. The form of the PORT INPUT command is:

```
PORT INPUT address[,baud]
```

When a PORT INPUT command is used to assign input operations from an alternate terminal, it remains in effect until another PORT INPUT command is executed or BASIC returns to the command mode. For instance, an H36 assigned port address 350Q would input all data after the following program statement was executed.

```
*100 PORT INPUT 232,300
```

PERManent

The PERManent mode is similar to the INPUT and OUTput modes, except that all EXTENDED BENTON HARBOR BASIC operations are PERManently assigned to the new port. The form of the PORT PERManent command is:

```
*PORT PERM address[,baud]
```

For instance, the following command unconditionally transfers control to the console connected at port address 372Q.

```
*PORT PERM 250
```

NOTE: A baud rate is not specified for the H8-5 Serial I/O and Cassette Interface Card because the circuit board is permanently wired for a fixed baud rate.

The console device that originates a PORT PERM command is totally ignored until a PORT PERM command issued from the new console returns control. A warm start (i.e. 040103A) always returns you to port 350Q first, and then to 372Q if nothing is assigned address 350Q.

The next example requires that you use two console devices. Device A is an H9 connected to your H8 system with an H8-5 circuit board. Device B is an H36 connected with an H8-4 circuit board. The H36 is assigned address 350Q and immediately becomes the "boss" when the system is configured. The program illustrates control between an H36 and an H9.

```
*LIST ↵
10 PRINT "THIS IS AN H36 TEST "
20 PORT OUT 250 : PRINT
30 PRINT "THIS IS AN H9 TEST!!"
40 PORT INPUT 250
50 LINE INPUT "ENTER FOUR WORDS ";A$,B$,C$,D$
60 PORT PERM 232,300 : PRINT
70 PRINT A$;" ";B$;" ";C$;" ";D$;"."
80 END

*RUN ↵

     (. . Printed at PORT 350Q . . )


THIS IS AN H36 TEST!

     (..Printed at PORT 372Q..)

THIS IS AN H9 TEST!!
ENTER FOUR WORDS
?THIS ↵
?COMPLETES ↵
?THE ↵
?TEST ↵

     (..Printed at PORT 350Q..)

THIS COMPLETES THE TEST.
END AT LINE 80
*
```

Example Program Using PORT

PENNY MATCH

Penny Match is a simple computer game used to show the PORT command in EXTENDED BENTON HARBOR BASIC (Version 10.03.00). The program illustrates a few of the subtle distinctions between the H8-5 and the H8-4 Serial I/O Cassette Interface Card and the software that controls the H8-4 Serial I/O Card.

Some minimum hardware requirements must be met to play the game. Let's assume you own an H8 with 16K of memory; this makes you boss. Your H8 is interfaced to an H36 console using the H8-4 Serial I/O Card at port location 350Q. Your friend is using an H9 video terminal interfaced to your computer system with an H8-5 Serial I/O and Cassette Interface Card. The H9 is configured for port location 372Q, which is the HEATH standard configuration.

NOTE: Check the "Systems Configuration section" (Page 0-6) of this "Software Reference Manual" if you are not sure of the correct port locations for your H8 computer system.

```
100 REM       DEMO:  PORT COMMAND
102 REM
104 REM
110 REM       P1=PORT 350Q ASSIGNED TO H36
115 REM
120 LET P1=3*64+5*8
125 REM
130 REM       P2=PORT 372Q ASSIGNED TO H9
135 REM
140 LET P2=3*64+7*8+2
145 REM
150 REM       J AND K RESERVED FOR SCORE
155 REM
160 LET J=0 : LET K=0
165 REM
170 REM       P1 AND P2 ARE CONVERTED TO
180 REM       A DECIMAL EQUIVALENT BY
185 REM       THE LET STATEMENT
186 REM
190 PRINT : PRINT : PRINT
192 REM
194 REM
200 REM       PRINT INTRODUCTORY MESSAGES
201 REM       ON BOTH THE H36 AND THE H9.
205 REM
210 PRINT "WELCOME TO THE GAME OF CHANCE!"
220 PRINT ".....  MATCH  PENNIES  ....."
230 PRINT : PRINT : PRINT
240 PRINT "WAIT A SECOND, I'LL FLIP A COIN."
250 PRINT : PRINT : PRINT
260 PORT OUT P2 : PRINT : PRINT : PRINT
270 PRINT "WELCOME TO THE GAME OF CHANCE!"
280 PRINT "HEADS I WIN -- TAILS YOU LOSE!"
290 PRINT : PRINT : PRINT
292 REM
294 REM
300 REM       STARTING POINT FOR GAMES!!!
302 REM
305 REM        THE DESTINATION OF A BRANCH
306 REM        IS DETERMINED BY THE VALUE
307 REM        OF T1....
308 REM
310 LET T1=0
320 PRINT : PRINT : PRINT
330 PRINT "FLIP A COIN!!!!"
340 PRINT : PRINT  : PRINT
350 PAUSE 1000
360 PRINT "WHAT DID YOU GET???"
370 REM
380 REM
390 REM
```

```
400 REM          INPUT A GUESS (HEAD/TAIL)
405 REM
406 REM          INPUT A GUESS FROM P2
408 REM
410 PORT INPUT P2
420 LINE INPUT "( HEAD/TAIL ) ";A$
430 LET B$=LEFT$(A$,1)
440 IF T1=50 THEN GOSUB 860
450 PRINT : PRINT : PRINT
460 IF B$="H" GOTO 500+T1
470 IF B$="T" GOTO 500+T1
480 PRINT "INVALID ENTRY!!!   TRY AGAIN."
490 GOTO 420
492 REM
494 REM
495 REM
500 REM          INPUT A GUESS FROM P1
502 REM
504 REM
510 PRINT "WAIT A SECOND, I'LL FLIP A COIN."
520 LET T1=50 : LET C$=B$ : PORT PERM P1,300
530 PRINT "I FLIPPED A ";A$;". NOW ITS YOUR TURN."
540 GOTO 420
542 REM
544 REM
550 REM          TOTAL SCORES -- AND PRINT A
555 REM          VICTORY MESSAGE.
556 REM
558 REM
560 IF B$=C$ THEN LET J=J+1 : GOSUB 910
570 IF B$<>C$ THEN LET K=K+1 : GOSUB 960
580 REM
590 REM
600 REM             PRINT RESULTS ON BOTH TERMINALS
602 REM
 04 REM
610 PRINT : PRINT : PRINT
620 PRINT "T O T A L S: PORT ";P1;" HAS WON"
630 PRINT TAB(11);J;" OUT OF ";J+K;" GAMES."
640 PRINT : PRINT : PRINT
650 PORT PERM P2
660 PRINT : PRINT : PRINT
670 PRINT "T O T A L S: PORT ";P2;" HAS WON"
680 PRINT TAB(11);K;" OUT OF ";J+K;" GAMES."
690 REM
```

```
700 REM         DECIDE IF END OF GAME...
702 REM
704 REM         CHECK STATUS OF P2.
706 REM
710 LINE INPUT "PLAY AGAIN? (YES/NO) ";D$
720 PRINT : PRINT : PRINT
730 IF D$="NO" GOTO 820
732 REM
734 REM         CHECK STATUS OF P1.
736 REM
740 PORT PERM P1,300
750 LINE INPUT "PLAY AGAIN? (YES/NO) ";D$
760 PORT OUT P2
770 IF D$="NO" GOTO 790
775 REM         IF BOTH AGREE -- DO AGAIN.
780 GOTO 300
785 REM         SAY GOOD-BYE
790 PRINT "NO MORE FOR ME!!!"
800 PORT PERM P1,300
810 END
820 PORT PERM P1,300
830 PRINT "I DON'T WANT TO PLAY AGAIN!"
840 END
842 REM
844 REM
850 REM  ----------  SUBROUTINES  --------
852 REM
854 REM
860 PORT OUT P2 : PRINT : PRINT : PRINT
870 PRINT "I FLIPPED A ";A$
880 PRINT : PRINT : PRINT
890 PORT OUT P1,300
900 RETURN
902 REM
904 REM
910 PRINT "THE PLAYER USING THE H36 WON!!"
920 PORT OUT P2 : PRINT : PRINT : PRINT
930 PRINT "THE PLAYER USING THE H9 LOST..."
940 PORT OUT P1,300
950 RETURN
952 REM
954 REM
960 PRINT "THE PLAYER USING THE H36 LOST!"
970 PORT OUT P2 : PRINT : PRINT : PRINT
980 PRINT "THE PLAYER USING THE H9 WON!!!"
990 PORT OUT P1,300 : RETURN
992 REM
994 REM
996 REM  ------ END  SUBROUTINES  --------
```

Example Program Using PUT and GET

This example of a simple application demonstrates the use of the EXTENDED BENTON HARBOR BASIC "PUT" and "GET" commands.

Mr. Sands owns a grocery store and desires to keep track of the number of hours worked each week by his employees. Therefore, he wrote the following three programs.

```
10 REM PROGRAM #1
20 REM CREATE ARRAYS FOR EMPLOYEE NUMBER, EMPLOYEE NAME, AND
30 REM HOURS WORKED THIS WEEK.
40 REM INITIALIZE HOURS WORKED TO ZERO.
50 REM
60 INPUT "ENTER NUMBER OF EMPLOYEES? ";N
70 DIM E$(N),N$(N),H(N)
80 FOR I = 1 TO N
90 LINE INPUT "ENTER EMPLOYEE NUMBER? ";E$(I)
100 LINE INPUT  "ENTER EMPLOYEE NAME? ";N$(I)
110 LET H(I) = 0
120 NEXT I
130 END
```

```
10 REM PROGRAM #2
20 REM UPDATE HOURS WORKED THIS WEEK FOR EACH EMPLOYEE.
30 REM
40 FOR I = 1 TO N
50 PRINT "ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER ";
60 PRINT E$(I)
70 INPUT H1
80 LET H(I) = H(I)+H1
90 NEXT I
100 END
```

```
10 REM PROGRAM #3
20 REM PRINT A REPORT OF THE HOURS WORKED THIS WEEK BY EACH
30 REM EMPLOYEE.
40 REM
50 PRINT
60 PRINT "EMPLOYEE","HOURS","EMPLOYEE"
70 PRINT "NUMBER","WORKED","EMPLOYEE"
80 PRINT
90 FOR I = 1 TO N
100 PRINT E$(I),H(I),N$(I)
110 NEXT I
120 END
```

Before running program #2, except after running program #1, Mr. Sands uses the "GET" command to load his variables. After running program #2, Mr. Sands uses the "PUT" command to save his variables for use the next day.

Following is a view over Mr. Sands' shoulder as he ran his programs for the week 07/24/78 through 07/28/78.

```
07/24/78 SESSION

     *LOAD "PROGRAM #1"
     SURE?YFOUND PROGRAM #1
     *UNLOCK
     *RUN
     ENTER NUMBER OF EMPLOYEES? 5
     ENTER EMPLOYEE NUMBER? 1
     ENTER EMPLOYEE NAME? RANDY SANDS
     ENTER EMPLOYEE NUMBER? 2
     ENTER EMPLOYEE NAME? TOM STEVENS
     ENTER EMPLOYEE NUMBER? 3
     ENTER EMPLOYEE NAME? MAX HAMILTON
     ENTER EMPLOYEE NUMBER? 4
     ENTER EMPLOYEE NAME? JOYCE PARKER
     ENTER EMPLOYEE NUMBER? 5
     ENTER EMPLOYEE NAME? SANDY MILLER

     END AT LINE  130
     *LOAD "PROGRAM #2"
     SURE?YFOUND PROGRAM #2
     *CONTINUE
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 1
     ?8
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 2
     ?4
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 3
     ?4
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 4
     ?8
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 5
     ?8

     END AT LINE  100
     *PUT "HOURS WORKED 07/24/78"
     *


07/25/78 SESSION

     *LOAD "PROGRAM #2"
     SURE?YFOUND PROGRAM #2
     *UNLOCK
     *GET "HOURS WORKED 07/24/78"
     SURE?YFOUND HOURS WORKED 07/24/78
     *CONTINUE
     ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 1
     ?8
```

```
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 2
      ?4
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 3
      ?4
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 4
      ?8
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 5
      ?8

      END AT LINE  100
      *PUT "HOURS WORKED 07/25/78"
      *

07/26/78 SESSION

      *LOAD "PROGRAM #2"
      SURE?YFOUND PROGRAM #2
      *UNLOCK
      *GET "HOURS WORKED 07/25/78"
      SURE?Y FOUND HOURS WORKED 07/25/78
      *CONTINUE
      ENTER NUMBER OF HOURS WORKED TOADY FOR EMPLOYEE NUMBER 1
      ?8
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 2
      ?4
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 3
      ?4
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 4
      ?8
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 5
      ?8

      END AT LINE  100
      *PUT "HOURS WORKED 07/26/78"
      *

07/27/78 SESSION

      *LOAD "PROGRAM #2"
      SURE?YFOUND PROGRAM #2
      *UNLOCK
      *GET "HOURS WORKED 07/26/78"
      SURE?YFOUND HOURS WORKED 07/26/78"
      *CONTINUE
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 1
      ?8
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 2
      ?4
      ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 3
      ?4
```

```
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 4
?8
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 5
?8

END AT LINE  100
*PUT "HOURS WORKED 07/27/78"
*
```

07/28/78 SESSION

```
*LOAD "PROGRAM #2"
SURE?Y FOUND PROGRAM #2
*UNLOCK
*GET "HOURS WORKED 07/27/78"
SURE?YFOUND HOURS WORKED 07/27/78
*CONTINUE
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 1
?8
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 2
?4
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 3
?0
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 4
?8
ENTER NUMBER OF HOURS WORKED TODAY FOR EMPLOYEE NUMBER 5
?7.5

END AT LINE  100
*UNLOCK
*LOAD "PROGRAM #3"
SURE?YFOUND PROGRAM #3
*CONTINUE

EMPLOYEE      HOURS         EMPLOYEE
NUMBER        WORKED        NAME

1             40            RANDY SANDS
2             20            TOM STEVENS
3             16            MAX HAMILTON
4             40            JOYCE PARKER
5             39.5          SANDY MILLER

END AT LINE  120
*
```

INDEX

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