HEWLETT hp PACKARD

HP 92065A BASIC/1000M RTE/M Real-Time BASIC Language

Reference Manual

HP 92065A BASIC/1000M RTE-M Real-Time BASIC Language

Reference Manual

(This manual reflects information that is compatible with Software Revision Code 1726.)

HEWLETT hp PACKARD

HEWLETT-PACKARD COMPANY
11000 WOLFE ROAD, CUPERTINO, CALIFORNIA, 95014

Library Index Number
2RTEM.320.92065-90001

PART NO. 92065-90001

Change 1: 7/77
Printed in U.S.A. 2/77

LIST OF EFFECTIVE PAGES

Changed pages are identified by a change number adjacent to the page number. Changed information is indicated by a vertical line in the outer margin of the page. Original pages do not include a change number and are indicated as change number 0 on this page. Insert latest changed pages and destroy superseded pages.

Change 0 (Original)	Feb 1977
Change 1	Jul 1977

PREFACE

The HP 92065A Real-Time BASIC subsystem provides functions, subroutines, and statements which allow you to schedule tasks, control instruments, the plotter and magnetic tape devices, and provides many additional capabilities. It runs under control of the HP 92064A RTE-M Operating System.

This manual is a reference guide to the BASIC language, the BASIC system commands, and the subroutines available with the system. You should be familiar with the RTE-M Operating System. If a BASIC system has been generated and is available for your use, you will find the information you need to create and run BASIC programs in this manual. If you must generate the BASIC system yourself, you should be familiar with the RTE-M System Generation Manual. This manual and other manuals that will assist you in becoming familiar with the RTE-M Operating System are shown in the documentation map which follows this preface.

Section I introduces BASIC and describes some of its general features. Sections II through VII describe the BASIC programming language. Expressions are defined in Section II and statements in Section III. Section IV describes statements in relation to strings and special characteristics of string variables and constants. Section V describes functions, lists the functions provided with BASIC, and tells you how to define your own functions. Both BASIC subroutines embedded in a BASIC program and external subroutines written in BASIC and other languages are described in Section VI. Section VII describes input and output operations to logical devices and the statements and functions which manipulate logical unit I/O features.

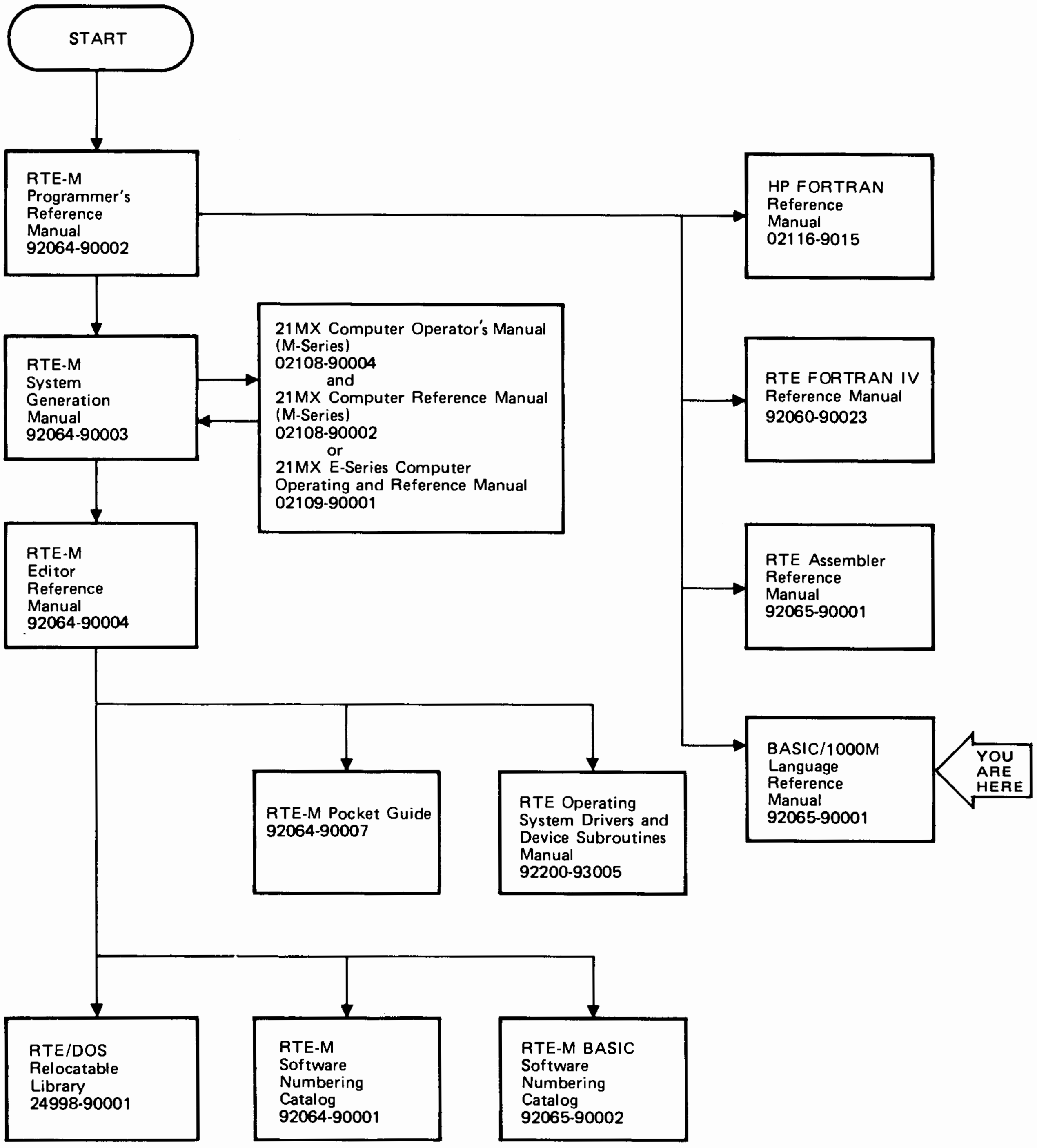
Section VIII tells you how to execute the BASIC Interpreter. Section IX describes the commands used to communicate with the Interpreter once it is running.

Sections X through XII deal with the subroutines and statements which schedule tasks and control specific hardware. Section X describes real-time task scheduling and the subroutine calls BASIC provides for this purpose. Bit manipulation functions are described in Section XI. Both commands and subroutine calls used to read, write, and control magnetic tape devices are described in Section XII. Section XIII provides information about generating the Branch and Mnemonic Tables which are required if external subroutines are used with BASIC and describes the HP 2313/91000 Subsystem subroutine calls and configuration, the HP 6940 Subsystem configuration and routines, and the HP 7210 Plotter subroutine calls.

Appendix A contains the HP Character Set. Appendix B contains summaries of all statements, commands, and library subroutines. Appendix C contains a summary of error messages.

The components of RTE-M Real-Time BASIC can be ordered by part number information which is given in the HP 92065A Software Numbering Catalog, HP Part No. 92065-90002.

DOCUMENTATION MAP



START
RTE-M Programmer's Reference Manual 92064-90002
RTE-M System Generation Manual 92064-90003
21MX Computer Operator's Manual (M-Series) 02108-90004 and 21MX Computer Reference Manual (M-Series) 02108-90002 or 21MX E-Series Computer Operating and Reference Manual 02109-90001
RTE-M Editor Reference Manual 92064-90004
HP FORTRAN Reference Manual 02116-9015
RTE FORTRAN IV Reference Manual 92060-90023
RTE Assembler Reference Manual 92065-90001
BASIC/1000M Language Reference Manual 92065-90001
YOU ARE HERE
RTE-M Pocket Guide 92064-90007
RTE Operating System Drivers and Device Subroutines Manual 92200-93005
RTE/DOS Relocatable Library 24998-90001
RTE-M Software Numbering Catalog 92064-90001
RTE-M BASIC Software Numbering Catalog 92065-90002

7700-24

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

INTRODUCTION

SECTION I

1-1. FEATURES

HP 92065A Real-Time BASIC is a BASIC language subsystem designed for use under control of the HP 92064A RTE-M Operating System. The BASIC subsystem provides an easy-to-use version of the BASIC programming language for the RTE-M environment. Interaction with BASIC can be via a keyboard terminal, mini cartridge tape, keypunched card, paper tape, or magnetic tape devices.

BASIC provides you with the following capabilities:

- Conversational programming
- Multiple peripheral device I/O including graphics display.
- Real-time and event task scheduling.
- Bit manipulation.
- User defined subroutines and functions.
- Character String manipulation.

1-2. CONVERSATIONAL PROGRAMMING

BASIC is a programming language that is easy to learn and use. You enter program statements, line-by-line, directly into the BASIC subsystem from an input device. The BASIC Interpreter checks each statement as it is entered. If the statement contains an error, a message is displayed that defines the error so that you may correct it immediately. This type of interaction between you and the BASIC Interpreter is called conversational programming.

Conversational programming allows you to test your programs step-by-step as you prepare them. You are in constant contact with the system, its functioning, and its results. You can complete programming and debugging quickly, easily, and efficiently.

1-3. MULTIPLE PERIPHERAL DEVICE I/O

BASIC provides a wide selection of input/output capabilities and it can be used with either hardcopy or display screen terminals, line printers, paper tape punches, magnetic tape units, and mini cartridge tape devices. Data can be displayed on a hardcopy graphic plotter or TV monitor.

1-4. REAL-TIME AND EVENT TASK SCHEDULING

BASIC operates in "real-time"; that is, the order of processing may be governed by time or by the occurrence of external events rather than by a strict sequence defined in the program itself. Because these events can occur in random order and require different amounts of processing, conflicts may arise between tasks. BASIC is capable of resolving these conflicts.

A task is defined as a group of BASIC statements initiated by a call to one of the BASIC scheduling subroutines (e.g., START, TRAP, etc.) and terminated by a RETURN statement (see Section X).

The BASIC subsystem provides statements that assign execution priority to tasks, and statements to schedule execution of tasks as a function of time. The user can also connect task subroutines to event interrupts such as contact closures. Each task subroutine that is to be repeated during the course of system operations specifies the interval between successive executions of the task.

1-5. ENVIRONMENT

The minimum hardware and software requirements for support of the RTE-M BASIC subsystem are listed in the following paragraphs.

1-6. HARDWARE

The BASIC subsystem operates within the RTE-M hardware environment. (Refer to the appropriate system Programming and Operating Manual for equipment configurations.)

Minimum requirements are:

- HP 21MX Series computer.
- 24K words minimum memory (the BASIC subsystem occupies approximately 10K words).
- Floating Point Hardware.
- System Console (HP 2644/2645 Terminal).

Optional devices include a line printer, card reader, paper tape reader, plotter, TV monitor, HP 2313 and HP 6940 subsystems, mini cartridges, and terminals.

A typical system configuration is depicted in Figure 1-1.

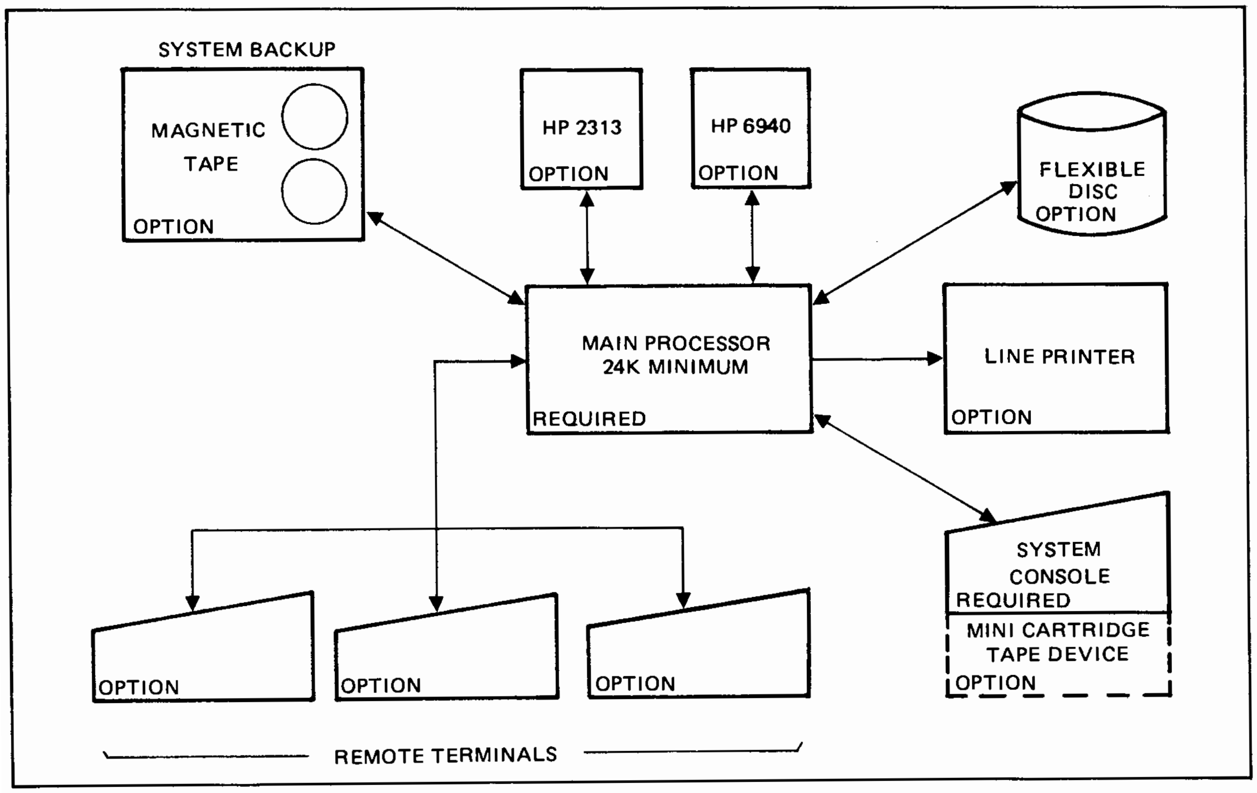


Figure 1-1. RTE-M Typical System Configuration

1-7. SOFTWARE

The BASIC subsystem is an option that executes under control of the RTE-M Operating System.

The BASIC subsystem consists of the following modules and components:

- BASIC, the main program, and a set of subroutines used for control and for input/output. The module names are:

BASCM	main program module
BASC1	statement syntax checking module
BASC2	listing module
BASC3	pre-execution module
BASC4	execution module
BASC5	command syntax checking and execution module
BASC8	slow statements module (includes PAUSE and STOP)

- BASIC Library.
- User-defined Branch and Mnemonic Tables, used to link BASIC to subroutines and functions.
- Standard Branch and Mnemonic Tables (supplied), used to link BASIC to required functions in the FORTRAN IV Library.
- Trap Table module, used for keeping track of real-time tasks and traps.

The BASIC subsystem part numbers are listed in the HP 92065A Software Numbering Catalog.

Figure 1-2 illustrates the organization of the BASIC components in RTE-M system memory.

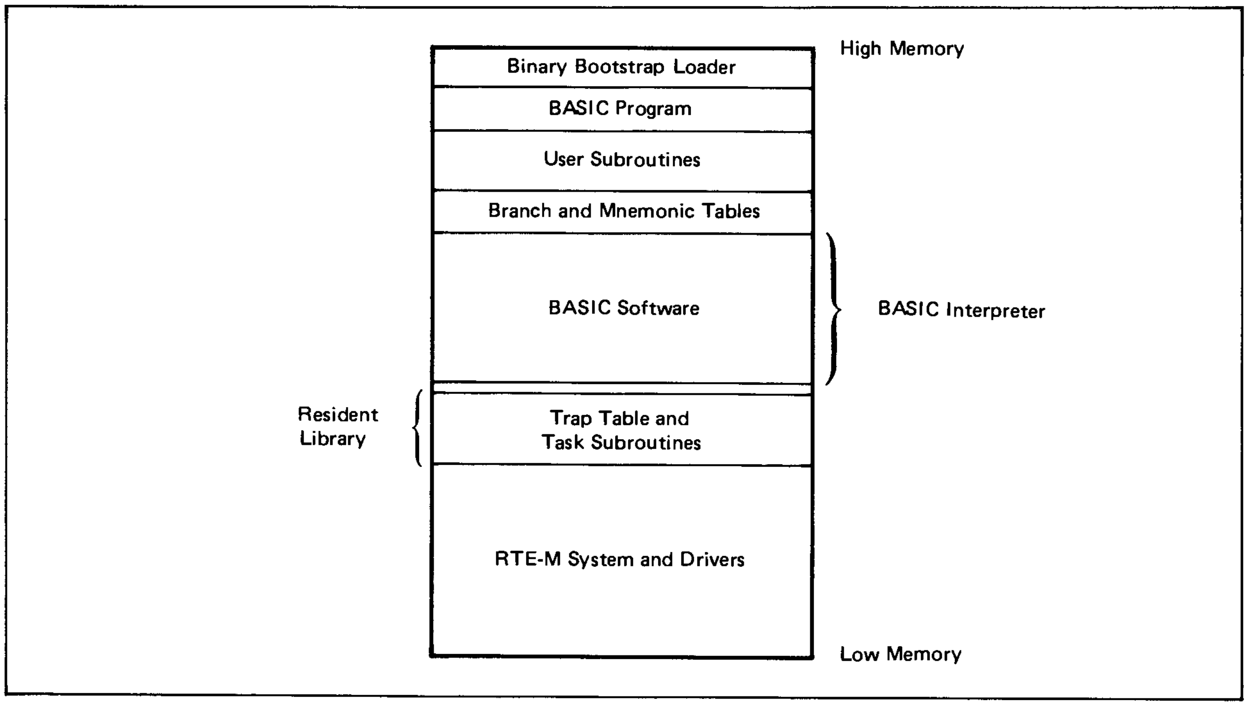


Figure 1-2. RTE-M Memory Allocation with BASIC

1-8. BASIC COMMANDS

BASIC commands instruct the BASIC Interpreter to perform certain control functions. Commands differ from the statements used to write a program in the BASIC language.

A command instructs the Interpreter to perform some action immediately. A statement is an instruction to perform an action only when the program is executed. A statement is always preceded by a statement number; a command never is.

Any BASIC command can be entered following the BASIC prompt character, >. Each command is a single word that must be typed in its entirety. (DELETE is an exception, you may type DEL.) If misspelled, the computer will return an error message. Some commands have parameters to further define command operation.

For example, BYE is a command that you use to terminate the BASIC Interpreter and return to the operating system. It has no parameters. Another command, LIST, prints the program currently being entered. It may have parameters to specify that only part of the program is to be listed, or to indicate a particular list device.

1-9. BASIC STATEMENTS

Statements are used to write a BASIC program that will subsequently be executed. Each statement performs a particular action. Every statement you enter becomes part of the current program and is kept until explicity deleted or you exit from BASIC with the BYE command.

A statement is always preceded by a statement number. This number is an integer between 1 and 9999. The statement number indicates the order of the statements in the program. Statements are ordered by BASIC from the lowest to the highest statement number. Because this order is maintained by the Interpreter, it is not necessary for you to enter statements in execution order so long as the numbers are in that order.

Following each statement, you must press the RETURN key to inform the Interpreter that the statement is complete. The Interpreter generates a linefeed and prints the prompt character, >, on the next line to signal that the statement is accepted. If an error is made upon entering the statement, the Interpreter prints an error message.

BASIC statements have a free format. This means that blanks are ignored in most cases (exceptions: string literals, CALL statements). For example, all of the following statement forms are equivalent.

```
>30    PRINT     S
>30 PRINTS
>30PRINTS
>   30 PRINTS
> 3 0 P R I N T S
```

1-10. BASIC PROGRAMS

Any statement or group of statements that can be executed constitutes a program. A program may consist of only two statements. The following is an example of such a program:

```
>100 PRINT 35+5
>110 END
```

The first statement number is 100. PRINT is the keyword or instruction that tells the Interpreter the kind of action to perform. In this case, it prints the result of the arithmetic expression that follows the keyword. The expression is 35+5. It is evaluated by the Interpreter, and when the program is executed, the result is printed. The END statement (statement number 110) indicates the end of the BASIC program.

Usually, a program contains more than two statements. The following four statements are a program:

```
>10 INPUT A,B,C,D,E
>20 LET S = (A+B+C+D+E)/5
>30 PRINT S
>40 END
```

This program, which calculates the average of five numbers is shown in the order of its execution. It could be entered in any order if the statement numbers assigned to each statement were not changed.

For example, the following program executes exactly like the program above:

```
>20 LET S = (A+B+C+D+E)/5
>10 INPUT A,B,C,D,E
>30 PRINT S
>40 END
```

It is generally a good idea to number statements in increments of 10. This allows room to intersperse additional statements as needed.

1-11. CORRECTION OF TYPING ERRORS

You may use the following keys to correct typing errors:

- RUBOUT deletes the current line you are typing; on some terminals a DEL key is used instead.
- BACKSPACE deletes a character. If your terminal does not have a key labeled BACKSPACE, you may use Control H (H^c). Press the control key (CNTL), hold it down and press the H key. The backspace is printed as an underline, _ or a back arrow, ←. If you want to delete more than one character, repeat BACKSPACE or H^c for each character to be deleted.

Terminals which have both upper and lower case characters should be locked into upper case mode, if possible.

1-12. LOGICAL UNIT NUMBERS

Logical unit numbers, abbreviated LU in the remainder of this manual, are decimal integers between 0 and 63 that are used to address I/O devices. Certain logical unit numbers must always refer to specific devices. These are as follows:

1	System Console
2	System Mass Storage (upward compatible)
3	Auxiliary Mass Storage (upward compatible)
4	Standard Output Device
5	Standard Input Device
6	Standard List Device

The standard devices may be:

output	paper tape punch, magnetic tape, mini cartridge
input	paper tape reader, card reader, terminal, mini cartridge
list	line printer or terminal

The remaining logical unit numbers (7 through 63) may be assigned to any type of peripheral device.

Logical unit number 0 is not associated with any particular device, but is used essentially to turn off an input or output statement. That is, an I/O statement that references LU 0 is ignored.

1-13. SYNTAX CONVENTIONS

The following syntax conventions are used in this manual to specify command and statement formats:

UPPER-CASE BLOCK LETTERS	Literals that must be specified exactly as shown.
lower-case italics	Type of information to be supplied by you; most parameters are in this form.
[*,parameter*]	Optional parameters are enclosed in brackets.
parameter 1 *parameter 2* *parameter 3*	One and only one of the stacked parameters must be specified.
$\begin{bmatrix} \textit{parameter 1} \\ \textit{parameter 2} \\ \textit{parameter 3} \end{bmatrix}$	All bracketed parameters are optional, only one may be specified.
[*,param1* [*,param2*]]	Series of optional parameters; the last parameter may be omitted with no indication; embedded parameters must be supplied.
. . .	Ellipsis indicates that the previous parameter or series of bracketed parameters can be repeated.

BASIC Language Syntax Conventions

EXPRESSIONS

SECTION II

An expression combines constants, variables, or functions with operators in an ordered sequence. When evaluated, an expression must result in a value. An expression that, when evaluated, is converted to an integer is called an integer expression. Constants, variables, and functions represent values; operators tell the computer the type of operation to perform on these values.

Some examples of expressions are:

(P + 5)/27	*P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divide operator. Parentheses group those portions of the expression to be evaluated first.* *If P = 49, this example is an integer expression with the value 2.*
(N − (R + 5)) − T	*N, R, and T must all have been assigned values. + and − are the add and subtract operators. The innermost parentheses enclose the part evaluated first.* *If N=20, R=10, and T=5, the value of the integer expression is zero.*

2-1. CONSTANTS

A constant is either numeric or a literal string.

2-2. NUMERIC CONSTANTS

A numeric constant is a positive or negative decimal number including zero. It may be written in any of the following three forms:

- As an integer - a series of digits with no decimal point.
- As a fixed point number - series of digits with one decimal point preceding, following, or embedded within the series.
- As a floating point number - an integer or fixed point number followed by the letter E and an optionally signed integer.

Examples of Integers:

```
 1234
-70
 0
```

Examples of Fixed Point Numbers:

```
 1234.
 1234.56
-.0123
```

2-3. FLOATING-POINT NUMBERS

In the floating point notation, the number preceding E is a magnitude that is multiplied by some power of 10. The integer after E is the exponent, that is, it is the power of 10 by which the magnitude is multiplied.

The exponent of a floating point number is used to position the decimal point. Without this notation, describing a very large or very small number would be cumbersome:

```
1E+35 = 100000000000000000000000000000000000
1E−35 = .00000000000000000000000000000000001
```

Examples of Floating-Point Numbers:

1E+23	$=1 \times 10^{23} = 100000000000000000000000$
1.0E23	(same as above)
.001E26	(same as above)
1.02E+4	$=1.02 \times 10^{4} = 10200.$
1.02E−4	=.000102

Within the computer, all these constants are represented as floating-point real numbers whose precision is 6 or 7 digits and whose size is between 10^{-38} and 10^{38}.

2-4. LITERAL STRINGS

A literal string consists of a sequence of characters in the ASCII character set enclosed within quotes. The quote is the only character excluded from the character string.

Examples of Literal Strings:

"ABC"	"" *(a null, empty, or zero length string)*
"!!WHAT A DAY!!"	" " *(a string with two blanks)*
" X Y Z "	

Blank spaces are significant within a string.

2-5. VARIABLES

A variable is a name to which a value is assigned. This value may be changed during program execution. A reference to the variable acts as a reference to its current value. Variables are either string or numeric. Further, numeric variables are either simple or subscripted.

Simple numeric variables are a single letter (from A to Z) or a letter immediately followed by a digit (from 0 to 9):

```
A    A0
P    P5
X    X9
```

A variable of this type always contains a numeric value that is represented in the computer by a real floating-point number.

If a variable names an array, it must be subscripted. Only the alphabetic characters A through Z may be used to name an array. When a variable is subscripted, the variable name is followed by one or two subscript values enclosed in parentheses. If there are two subscripts, they are separated by a comma. A subscript may be an integer constant or variable, or any expression that is evaluated to an integer value:

```
A(1)        A(N,M)
P(1,1)      P(Q5,N/2)
X(N+1)      X(10,10)
```

A simple variable and a subscripted variable may have the same name with no implied relation between the two. For example, a simple variable named A is totally distinct from a subscripted variable named A(1,1).

Simple numeric variables can be used without being declared. Subscripted variables must be declared with a DIM statement (see Section III) if the array dimensions are greater than 10 rows, or 10 rows and 10 columns. The first subscript is always the row number, the second the column number. The subscript expressions must result in a value between 1 and the maximum number of rows and columns.

A variable may also contain a string of characters. This type of variable, a string array, is identified by a variable name consisting of a letter and $:

```
A$          P$
```

The value of a string variable is always a string of characters, possibly null or zero length. If the string array contains a single character, it need not be declared with a DIM statement (see Section III). String arrays differ from numeric arrays in that they have only one dimension. You may optionally use two subscripts which refer to the first and last characters in the substring you want to reference (See Section IV, String Arrays). You may also use one subscript to refer to the first character of the substring. In this case, the last character of the substring will be the last character of the string. Examples of subscripted string array names (substrings) are:

```
A$(1,3)    Z$(N,N+M)    A$(10)
```

2-6. FUNCTIONS

A function names an operation that is performed using one or more parameter values to produce a single value result. A numeric function is identified by a three-letter name followed by one or more formal parameters enclosed in parentheses. If there is more than one, the parameters are separated by commas. The number and type of the parameters depends on the particular function. The formal parameters in the function definition are replaced by actual parameters when the function is used.

Since a function results in a single value, it can be used anywhere in an expression where a constant or variable can be used. To use a function, the function name followed by actual parameters in parentheses (known as a function call) is placed in an expression. The resulting value is used in the evaluation of the expression.

Examples of common functions:

SQR(x)	where x is a numeric expression that results in a value ≥ 0. When called, it returns the square root of x. For instance, if N=2, SQR(N+2) = 2.
ABS(x)	where x is any numeric expression. When called, it returns the absolute value of x. For instance, ABS(-33) = 33.

BASIC provides many built-in functions that perform common operations such as finding the sine,taking the square root, or finding the absolute value of a number. The available functions are listed in Section V. In addition, you may define and name your own functions should you need to repeat a particular operation. How to write functions is described in Section V, Functions.

2-7. OPERATORS

An operator performs a mathematical or logical operation on one or two values resulting in a single value. Generally, an operator is between two values, but there are unary operators that precede a single value. For instance, the minus sign in A - B is a binary operator that results in subtraction of B from A; the minus sign in -A is a unary operator indicating that A is to be negated.

The combination of one or two operands with an operator forms an expression. The operands that appear in an expression can be constants, variables, functions, or other expressions.

Operators may be divided into types depending on the kind of operation performed. The main types are arithmetic, relational, and logical (or Boolean) operators.

The arithmetic operators are:

+	Add (or if unary, positive)	$A + B$ or $+A$
–	Subract (or if unary, negative)	$A - B$ or $-A$
*	Multiply	$A \times B$
/	Divide	$A \div B$
↑ or ∧	Exponentiate	A^B

In an expression, the arithmetic operators cause an arithmetic operation resulting in a single numeric value.

The relational operators are:

=	Equal	$A = B$
<	Less than	$A < B$
>	Greater than	$A > B$
<=	Less than or equal to	$A \leq B$
>=	Greater than or equal to	$A \geq B$
<> or #	Not equal	$A \neq B$

When relational operators are evaluated in an expression they return the value 1 if the relation is found to be true, or the value 0 if the relation is false. For instance, A = B is evaluated as 1 if A and B are equal in value, as 0 if they are unequal.

Logical or Boolean operators are:

AND	Logical "and"	A AND B
OR	Logical "or"	A OR B
NOT	Logical complement	NOT A

Like the relational operators, the evaluation of an expression using logical operators results in the value 1 if the expression is true, the value 0 if the expression is false.

Logical operators are evaluated as follows:

A AND B	= 1 (true) if A and B are both $\neq 0$; = 0 (false) if A = 0 or B = 0
A OR B	= 1 (true) if A $\neq 0$ or B $\neq 0$; = 0 (false) if both A and B = 0
NOT A	= 1 (true) if A = 0; = 0 (false) if A $\neq 0$

2-8. EVALUATING EXPRESSIONS

An expression is evaluated by replacing each variable with its value, evaluating any function calls and performing the operations indicated by the operators. The order in which operations are performed is determined by the hierarchy of operators:

↑ or ∧	(highest)
NOT	
* /	
+ −	
Relational (=, <, >, < =, > =, < >)	
AND	
OR	(lowest)

The operator at the highest level is performed first followed by any other operators in the hierarchy shown above. If operators are at the same level, the order is from left to right. Parentheses can be used to override this order. Operations enclosed in parentheses are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair are performed first.

For instance: 5 + 6*7 is evaluated as 5 + (6 × 7) = 47
7/14*2/5 is evaluated as ((7/14)×2)/5 = .2

If A=1, B=2, C=3, D=3.14, E=0

then: A+B*C is evaluated as A + (B*C) = 7
A*B+C is evaluated as (A*B) + C = 5
A+B−C is evaluated as (A+B)−C = 0
(A+B)*C is evaluated as (A+B)*C = 9

When a unary operator immediately follows another operator of higher precedence, the unary operator assumes the same precedence as the preceding operator. For instance,

B ↑ −B ↑ C is evaluated as $(B^{-B})^{C}$ = 1/64 or .015625

In a relation, the relational operator determines whether the relation is equal to 1 (true) or 0 (false):

(A*B) < (A−C/3) is evaluated as 0 (false) since A*B=2 which is not less than A−C/3=0

In a logical expression, other operators are evaluated first for values of zero (false) or non-zero (true). The logical operators determine whether the entire expression is equal to 0 (false) or 1 (true):

E AND A−C/3	is evaluated as 0 (false) since both terms in the expression are equal to zero (false).
A+B AND A*B	is evaluated as 1 (true) since both terms in the expression are different from zero (true).
A=B OR C=SIN(D)	is evaluated as 0 (false) since both expressions are false (0).
A OR E	is evaluated as 1 (true) since one term of the expression (A) is not equal to zero.
NOT E	is evaluated as 1 (true) since E=0.

STATEMENTS

SECTION III

This section describes statements used in writing a Real-Time BASIC program. Statements must be preceded by a line number and are terminated by pressing the RETURN key when entered. Statements are executed in numeric sequence (unless branching occurs), but may be entered in any sequence.

Unlike Assembly language, FORTRAN, and other programming languages, BASIC statements are interpreted at the time they are entered; thus a compile stage is not required. Invalid statements are immediately rejected. Statements are not executed, however, until the program is executed with the RUN command (see Section IX).

Table 3-1. lists some statements used in writing a program and briefly describes each. Detailed explanations of each statement are provided in the remainder of the section. Additional statements related to specific programming objectives are introduced and explained in subsequent sections of this part of the manual. A complete list of Real-Time BASIC statements and their uses is provided in Appendix B.

3-1. LET

This statement assigns a value to one or more variables. The value may be in the form of an expression, a constant, a string, or another variable of the same type.

Format

When the value of the expression is assigned to a single variable, the formats are:

[LET] *variable* = *expression*

When the same value is to be assigned to more than one variable, the formats are:

[LET] *variable* = *variable* = . . . = *variable* = *expression*

In this statement, the equal sign is an assignment operator. It does not indicate equality, but is a signal that the value on the right of the assignment operator be assigned to the variable on the left. If any ambiguity exists between the relational operator "=" and the assignment operator, the equal sign is treated as a relational operator.

Table 3-1. Statements

STATEMENTS	FUNCTION
LET	Assigns the value of an expression to a variable. The word LET may be omitted.
REM	Introduces remarks and comments in the program listing.
GOTO	Transfers control to a specified statement.
GOTO . . . OF	Multibranch GOTO transfers control to one of a list of statements, depending on the value of an integer expression.
END/STOP	END indicates the last program statement and terminates execution of the current program. STOP terminates execution of the current program.
FOR . . . NEXT	Allows repetition of a group of statements between FOR and NEXT. The number of repetitions is determined by the initial and final values of a FOR variable, and an optional STEP specification.
IF . . . THEN	Evaluates a conditional expression and specifies action to be taken if condition is true.
PRINT	Prints the contents of a list of numeric or string expressions on the list device, or to a specified logical unit number.
READ/DATA/RESTORE	Assigns constants and string literals from one or more DATA statements to the variables specified in the READ statement. Treats contents of all DATA statements as a single data list.
INPUT	Requests user input to one or more variables by printing a prompt and accepts string or numeric data from the terminal.
DIM	Defines the size of arrays.
COM	Allows a program to store data in memory for retrieval by a subsequent BASIC program.
PAUSE	Stops program execution without terminating the program.
WAIT	Causes an executing program to stop for a specified number of milliseconds before continuing.

When a variable to be assigned a value contains subscripts, these are evaluated first from left to right, then the expression is evaluated and the value assigned to the array element.

If a value is assigned to more than one variable, the assignment is made from right to left. For instance, in the statement A=B=C=2, first C is assigned the value 2, then B is assigned the current value of C, and finally A is assigned the value of B.

Examples

```
10 LET A = 5.02
20 A=5.02
```

The variable A is assigned the value 5.02. Statements 10 and 20 have the same result.

```
30 X = Y7 = Z = Z1 = 0
```

Each variable X, Y7, Z, and Z1 is set to zero. This is a simple method for initializing variables at the start of a program.

```
35 LET M=2
40 LET A(M) = N = 9
```

First M is assigned the value 2 in line 35. In line 40 N is assigned the value 9, then the array element A(2) is assigned the value 9.

```
50 N = 0
60 LET N = N+1
70 LET A(N) = N
```

Statements 50 through 70 set the array element A(1) to 1. By repeating statements 60 and 70, each array element can be set to the value of its subscript.

3-2. REM

This statement allows the insertion of a line of remarks in the listing of the program. The remarks do not affect program execution.

Format

REM *any characters*

Like other statements, REM must be preceded by a statement number.

The remarks introduced by REM are saved as part of theReal-TimeBASIC program, and printed when the program is listed or punched. They are, however, ignored when the program is executed.

Remarks are easier to read if REM is followed by spaces, or a punctuation mark as in the examples.

Examples

```
>LIST
  10   REM: THIS IS AN EXAMPLE
  20   REM: OF REM STATEMENTS.
  30   REM -- ANY CHARACTERS MAY FOLLOW REM: "//**!!&&&,ETC.
  40   REM...REM STATEMENTS ARE NOT EXECUTED
>
```

3-3. GOTO

GOTO overrides the normal sequential order of statement execution by transferring control to a specified statement. The statement to which control transfers must be an existing statement in the current program.

Format

GOTO *statement number label*

GOTO *integer expression* OF *statement number label* [, *statement number label*, . . .]

GOTO may have a single *statement number label*, or may be multi-branched with more than one label. If the multi-branch GOTO is used, the value of the *integer expression* determines the label in the list to which control transfers. It is rounded to the nearest integer. GOTO may be entered as GO TO.

If the GOTO transfers to a statement that cannot be executed (such as REM or DIM), control passes to the next sequential statement after that statement. GOTO cannot transfer into or out of a function definition (see Section V). If it should transfer to the DEF statement, control passes to the line following the function definition.

The statement number labels in a multi-branch GOTO are selected by numbering them sequentially starting with 1, such that the first label is selected if the value of the expression is 1, the second label if the expression equals 2, and so forth. If the value of the expression is less than 1 or greater than the number of labels in the list, then the GOTO is ignored and control transfers to the statement immediately following GOTO.

Examples

```
 5Ø GOTO 1ØØ
 6Ø GOTO A OF 1ØØ, 2ØØ, 3ØØ
```

The first statement sends the sequence of execution to line number 100. The second statement directs control to either line number 100, 200, or 300 depending on the current value of A

The example below shows a simple GOTO in line 200 and a multi-branch GOTO in line 600.

```
>LIST
  1ØØ  LET I=Ø
  2ØØ  GOTO 6ØØ
  3ØØ  PRINT I
  4ØØ  REM THE VALUE OF I IS ZERO
  5ØØ  LET I=I+1
  6ØØ  GOTO I+1 OF 3ØØ,5ØØ,8ØØ
  7ØØ  REM THE FINAL VALUE OF I IS 2
  8ØØ  PRINT I
  9ØØ  END
>RUN
Ø
2
```

When run, the program prints the initial value of I and the final value of I.

3-4. END/STOP

The END and STOP statements are used to terminate execution of a program.

Format

```
END

STOP
```

The END statement consists of the word END; the STOP statement of the word STOP.

END and STOP have identical functions; the only difference is that the highest numbered statement in a program must be an END statement. STOP may be used simply to halt program execution at a given point.

Examples

```
200 IF A # 27.5 THEN 350
 •
 •
300 STOP
 •
350 LET A = 27.5
 •
 •
500 IF B # A THEN 9999
 •
 •
9999 END
```

3-5. FOR . . . NEXT

The looping statements FOR and NEXT allow repetition of a group of statements. The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are repeated is determined by the value of a simple numeric variable specified in the FOR statement.

Format

FOR *variable* = *initial expression* TO *final expression* [STEP *step expression*]

The *variable* is set to the value resulting from the *initial expression.* The *variable* is incremented after each time the loop is executed. When the value of the *variable* passes the value of the *final expression,* the looping stops. If STEP is specified, the *variable* is incremented by the value resulting from the *step expression* each time the group of statements is repeated. This value can be positive or negative, but should not be zero. If a *step expression* is not specified, the variable is incremented by 1.

The NEXT statement terminates the loop:

NEXT *variable*

The *variable* following NEXT must be the same as the *variable* after the corresponding FOR.

When FOR is executed, the variable is assigned an initial value resulting from the expression after the equal sign,and the final value and any step value are evaluated. Then the following steps occur:

1. The value of the FOR variable is compared to the final value; if it exceeds the final value (or is less when the STEP value is negative), control skips to the statement following NEXT.
2. All statements between the FOR statement and the NEXT statement are executed.
3. The FOR variable is incremented by 1, or if specified, by the STEP value.
4. Return to step 1.

Your program should not execute the statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce undesirable results.

FOR loops can be nested if one FOR loop is completely contained within another. They must not overlap.

Examples

Each time the FOR statement executes, a value for R is entered and the area of a circle with that radius is computed and printed.

```
>LIST
  1Ø  REM : RADIUS EXAMPLE
  2Ø     FOR A=1 TO 5
  3Ø     INPUT R
  4Ø     PRINT "AREA OF CIRCLE WITH RADIUS ";R;" IS ";3.14159*R↑2
  5Ø     NEXT A
  6Ø  END
>RUN
?1
AREA OF CIRCLE WITH RADIUS 1          IS 3.14159
?2
AREA OF CIRCLE WITH RADIUS 2          IS 12.5664
?4
AREA OF CIRCLE WITH RADIUS 4          IS 5Ø.2654
?8
AREA OF CIRCLE WITH RADIUS 8          IS 2Ø1.Ø62
?16
AREA OF CIRCLE WITH RADIUS 16         IS 8Ø4.247

BASIC READY
```

The FOR loop executes six times, decreasing the value of X by 1 each time:

```
>LIST
  10     FOR X=0 TO -5 STEP -1
  20     PRINT X-5
  30     NEXT X
  40  END
>RUN
-5
-6
-7
-8
-9
-10

BASIC READY
>
```

3-6. IF . . . THEN

IF . . . THEN statements are used to test for specified conditions and to specify program action depending on the test results. When a condition is found by the program to be true, then program action indicated by the statement is performed. When a condition is found by the program to be untrue, program action simply continues to the next statement.

Format

IF *expression* THEN *statement*
statement number label

The IF . . . THEN *statement* relationship is often described as a conditional transfer. Possible statement transfers that may be used with the IF . . . THEN condition are:

IF. . . CALL
GOSUB
GOTO
INPUT
LET
PAUSE
PRINT
PRINT #
READ
READ #
RESTORE
RETURN
STOP
WAIT

Note that the word THEN is omitted from the statement in the above operations.

Because numbers are not always represented exactly in the computer, the = operator should be used carefully in IF . . . THEN statements. Whenever possible, < = or > = should be used instead of =.

Examples

```
 10 IF A=B THEN 30
```

is equivalent to IF A = B GOTO 30

```
 10 IF A=B PRINT C
```

In the following example, if X > 10, the message in statement 40 is executed. Otherwise, the message in statement 60 is executed.

```
 >LIST
   10  LET N=10
   20  READ #1;X
   30  IF X <= N THEN 60
   40  PRINT "X IS MORE THAN";N
   50  GOTO 80
   60  PRINT "X IS LESS THAN OR EQUAL TO";N
   70  GOTO 20
   80  END
```

Note that the relational operator is optional in logical evaluations:

```
  5 IF X PRINT A$
 75 IF Y GOTO  99
```

3-7. PRINT

PRINT causes data to be output at the terminal. The data to be output is specified in a print list following PRINT.

Format

PRINT [*print list*]

The *print list* consists of items separated by commas or semicolons. The list may be followed by a comma or a semicolon. If the list is omitted, PRINT causes a skip to the next line. Items in the list may be numeric expressions, numeric or string variables, string literals, or tabbing functions.

The contents of the print list is printed. If there is more than one item in the print list, commas or semicolons must separate the items. The choice of a comma or semicolon affects the output format.

The output line is divided into five consecutive fields: four of 15 characters and one of 12 characters, for a total of 72 characters. The fields begin in columns 0, 15, 30, 45, and 60. When a comma separates items, each item is printed starting at the beginning of the next available field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

The separator between items can be omitted if one or both of the items is a quoted string. In this case, a semicolon is inserted automatically.

A carriage return and linefeed are output after PRINT has executed, unless the output list is terminated by a comma or semicolon. In this case, the next PRINT statement begins on the same line.

If an expression appears in the print list, it is evaluated and the result is printed. Any variable must have been assigned a value before it is printed. Each character between quotes in a string constant is printed.

See Section VII for information about other forms of the PRINT statement.

Examples

When items are separated by commas, they are printed in up to five fields per line; separated by semicolons, they directly follow one another. In the example below, the items are numeric, so each item is assigned a minimum of six characters.

```
>LIST
  10  LET A=B=C=D=E=15
  20  LET A1=B1=C1=D1=E1=20
  30  PRINT A,B,C1,C
  40  PRINT A;B;C1;C;D;E;A1;D1;E1
  50  PRINT A,B;C,D
  60  END
>RUN
15               15               20               15
15     15     20     15     15     15     20     20     20
15               15     15        15
```

In the example below, the first PRINT statement evaluates and then prints three expressions. The second PRINT skips a line. The third and fourth PRINT statements combine a string constant with a numeric expression. No fields are used in the print line for string constants unless a comma appears as separator. The fourth PRINT statement prints output on the same line as the third because the third statement is terminated by a comma.

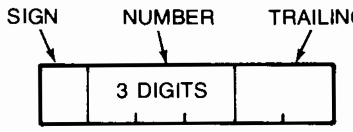
```
>LIST
  10  LET A=B=C=D=E=15
  20  LET A1=B1=C1=D1=E1=20
  30  PRINT A*B,B/C/D1+30,A+B
  40  PRINT
  50  PRINT "A*B =";A*B,
  60  PRINT "THE SUM OF A AND B IS ";A+B
  70  END
>RUN
225              30.05            30

A*B =225         THE SUM OF A AND B IS 30
```

3-8. NUMERIC OUTPUT FORMATS

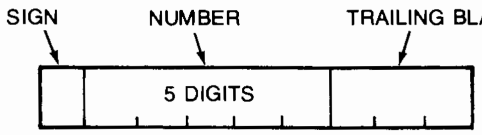
Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least two positions to the right containing blanks. The width is always a multiple of three; the minimum width is six characters.

Integers

An integer with a magnitude less than 1000 requires a field width of six characters:



An integer with a magnitude between 1000 and 32767 inclusive requires a field width of nine characters:



Examples of integers:

The integers below are less than 1000 and greater than −1000:

```
>LIST
  10  PRINT 1;999;30;-300;+295
  20  END
>RUN
1       999    30     -300   295
```

These integers are between 1000 and 32767 or between −1000 and −32767:

```
>LIST
  10  PRINT 1000;+32751;-32767;32767
  20  END
>RUN
1000     32751    -32767   32767
```

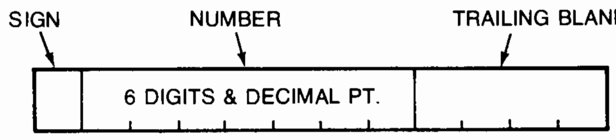
These integers are mixed in magnitude, but none are greater than 32767 or less than −32767:

```
>LIST
  10  PRINT 1;1000;999;+32751;20;-32767;-300;25687;+286;5000
  20  END
>RUN
1       1000     999    32751    20     -32767   -300   25687    286
5000
```

If an integer has a negative sign it is printed; a positive sign is not printed.

Fixed-Point Numbers

A fixed point number requires a field width of 12 positions. If the magnitude of the number is greater than or equal to .09999995 and less than 999999.5, or is less than .1 but can be printed with six significant digits, the number is printed as a fixed-point number. Trailing zeros are not printed, but a trailing decimal point is printed to show the number is not exact. The number is left-justified in the field with trailing blanks. The sign is printed only if it is negative.

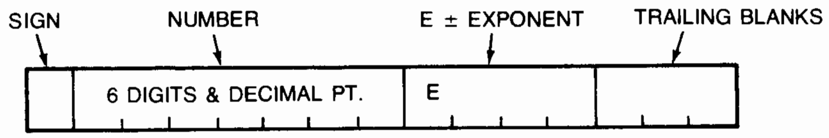


Examples of fixed-point numbers:

```
>LIST
  10   PRINT 999999.;.1;.000044
  20   END
>RUN
999999.     .1          .000044
```

Floating-Point Numbers

Any number, integer or fixed-point, with a magnitude greater than the magnitude of the numbers presented above, is printed as a floating-point number using a total field width of 15 positions:



Examples of floating-point numbers:

```
BASIC READY
>10 PRINT 2345678;.0000044
>20 END
>RUN
2.34568E+06    4.40000E-06

BASIC READY
>10 PRINT 23456789;.00000044
>20 END
>RUN
2.34568E+07    4.40000E-07

BASIC READY
>10 PRINT .00003943;.0000257895
>20 END
>RUN
3.94300E-05    2.57895E-05
```

3-9. TAB FUNCTION

The TAB function moves the print position to a specified column.

Format

TAB *(integer expression)*

The print position is moved to the column specified by the *integer expression*. Print positions are numbered from 0 to 71. If the *integer expression* is less than the current position, nothing is done. If the expression is greater than 71, the print position is moved to the beginning of the next line.

Example

```
5 PRINT A;TAB(25);B;TAB(50);C
```

3-10. READ/DATA/RESTORE

Together, the READ, DATA, and RESTORE statements provide a means to input data to a BASIC program. The READ statement reads data specified in DATA statements into variables specified in the READ statement. RESTORE allows the same data to be read again.

Format

READ *variable list*
DATA *constant* [, *constant*, ...]
RESTORE [*statement number label*]

Parameters

variable list	list of variables separated by commas.
constant	numeric or string constant.
statement number label	identifies a DATA statement.

Constants in the DATA statement are assigned to variables in the READ statement according to their order; the first constant to the first variable, and so forth.

When a READ statement is executed, each variable is assigned a new value from the constant list in a DATA statement.

More than one DATA statement can be specified. All the constants in the combined DATA statements comprise a data list. The list starts with the DATA statement having the lowest statement label and continues to the statement with the highest label. DATA statements can be anywhere in the program; they need not precede the READ statement, nor need they be consecutive. DATA statements do not execute, but merely specify data.

If a variable is numeric, the next item in the data list must be numeric; if a variable is a string, the next item in the data list must be a string constant. It is possible to determine the type of the next item with the TYP function (see Section V).

A pointer is kept in the data list showing which constant is the next to be assigned to a variable. The RUN command sets the pointer at the first DATA statement. It is advanced consecutively through the data list as constants are assigned. The RESTORE statement can be used to access data constants in a non-serial manner by specifying a particular DATA statement to which the pointer is to be moved.

When the RESTORE statement specifies a label, the pointer is moved to the first constant in the specified statement. If the statement is not a DATA statement, the pointer is moved to the first following DATA statement. When no label is specified, the pointer is restored to the first constant of the first DATA statement in the program.

Examples

The data in statement 10 is read in statement 20 and printed in statement 30:

```
>LIST
  10  DATA 3,5,7
  20  READ A,B,C
  30  PRINT A,B,C
  40  END
>RUN
3                  5                  7
```

Note the use of RESTORE in this example. It permits the second READ to read the same data into a second set of variables:

```
>LIST
  10  DIM A$[3],B$[3]
  20  DATA 3,5,7
  30  READ A,B,C
  40  READ A$,B$
  50  DATA "ABC","DEF"
  60  RESTORE
  70  READ D,E,F
  80  PRINT A$;B$,A;B;C;D;E;F
  90  END
>RUN
ABCDEF            3      5      7      3      5      7
```

3-11. INPUT

The INPUT statement allows you to input data to your program from the terminal.

Format

INPUT *variable list*

Parameters

variable list list of variables separated by commas.

The INPUT statement requests data to be input from your terminal for subsequent assignment to a variable. When the INPUT statement is encountered, the program comes to a halt and a question mark is printed on the terminal. The program does not continue execution until the input requirements are satisfied.

Only one question mark is printed for each INPUT statement. The statements:

```
10 INPUT A, B2, C5, D, E, F, G
```

and

```
20 INPUT X
```

each cause a single question mark to be printed. Note that the question mark generated by statement 10 requires seven input items, separated by commas, while that generated by statement 20 requires only a single input item.

When you run the program, if you enter data of the wrong type or other invalid input, two question marks (??) are printed. You may then type the correct input data.

If you want to terminate the program and return control to the BASIC Interpreter, type Control Q (Q^c), followed by a carriage return.

Example

```
>LIST
  10     FOR M=1 TO 2
  20     INPUT A
  30     INPUT A1,B2,C3,Z0,Z9,E5
  40     PRINT "WHAT VALUE SHOULD BE ASSIGNED TO R ";
  50     INPUT R
  60     PRINT A;A1;B2;C3;Z0;Z9;E5;"R= ";R
  70     NEXT M
  80   END
>RUN

?1
?2,3,4,5,6,7
WHAT VALUE SHOULD BE ASSIGNED TO R ?27
1       2       3       4       5       6       7       R= 27
?1.5
?2.5,3.5,4.5,6.,7.2
?8.1
WHAT VALUE SHOULD BE ASSIGNED TO R ?-99
1.5           2.5           3.5           4.5           6       7.2
8.1           R= -99
```

3-12. DIM

The DIM (dimension) statement defines the size of an array. DIM statements may also be used with strings (see Section IV).

Format

DIM *X(integer)*[, ...]

DIM *X(integer,integer)*[, . . .]

Parameters

X	array name (A through Z)
integer	dimension of array. (The first *integer* refers to rows and the second to columns).

The DIM statement defines the size of an array. 255 is the maximum dimension allowed. If a variable is subscripted and has not been defined in a DIM or COM statement, the size of the array is assumed to be 10. If the reference is to a two dimensional array, the array is assumed to be 10 by 10. An array may be dimensioned only once. More than one array can be named in a DIM statement; they are separated by commas.

There is no requirement to use all of the space reserved when you define the array. The maximum array size depends only upon the maximum available memory in the computer. The DIM statement can appear anywhere in a program and is not executed.

There is no way to initialize an array before execution. Values must be loaded by FOR loops or by reading from peripheral devices.

Examples

```
>LIST
  10  DIM F[2,3]
  20    FOR I=1 TO 2
  30      FOR J=1 TO 3
  40      LET F[I,J]=1
  50      NEXT J
  60    NEXT I
  70  END
>RUN
```

The size of the F array is defined and the array is initialized to contain all ones.

3-13. COM

The COM statement is used to pass data values between programs. Variables specified in a COM statement are placed in a common area so that values assigned to these variables in one program will be retained when loading in and executing another program. Both programs must include a COM statement.

Format

COM *variable list*

COM is an array whose last location is placed in a known fixed location in memory. Upon completion of the first program and the loading of the second program, the last location in the COM area is aligned with the last location of the second load.

Numeric bounds for arrays and strings are specified as in a DIM statement. Because a variable cannot be defined in two places at once, if the variable appears in a COM statement, it cannot also be defined in a DIM statement. An example of how the COM statement might appear in two successive programs follows.

	First Program 10 COM A(7)	Second Program 10 COM C(2),B(4)
Position in Memory	First Program	Second Program
xxx1	A(1)	
xxx2	A(2)	C(1)
xxx3	A(3)	C(2)
xxx4	A(4)	B(1)
xxx5	A(5)	B(2)
xxx6	A(6)	B(3)
xxx7	A(7)	B(4)

Remember, it is your responsibility to ensure proper access of common areas. If program common sizes differ, words outside the smaller common are destroyed during execution of the program with the smaller common block.

Common areas are not initialized to UNDEFINED as arrays declared in DIM statements are. You must not use Common area arrays before initialization or your results will be erroneous.

3-14. PAUSE

The PAUSE statement is used to stop the execution of a program without terminating the program.

Format

PAUSE [(*n*)]

Parameter

n optional parameter. If used, it must be enclosed in parentheses. The number *n* will be printed after PAUSE when the statement is executed.

The PAUSE statement stops a running program without terminating it, that is, without sending it to end of program. When a PAUSE statement is encountered and executed, the program is halted and the PAUSE is printed on the terminal. If you wish the program to continue, type GO, otherwise type Control Q (Q^c) followed by a carriage return thereby instructing the program to terminate and returning control to the BASIC Interpreter. BASIC is unable to execute real-time tasks during the time that a program is halted by a PAUSE statement.

One use of the numeric parameter is to identify the point of pause if more than one PAUSE statement is used within a program. The number specified is included in the PAUSE message printed on the terminal.

3-15. WAIT

The WAIT statement is used to introduce a program delay. When a WAIT statement is encountered, program execution is stopped for the number of milliseconds specified, then continued automatically.

Format

WAIT (*number of milliseconds*)

The WAIT statement introduces a program delay which allows instruments to achieve a steady state. The number following the word WAIT is the desired delay in milliseconds. Hence the statement:

WAIT (1000)

will delay the program one full second. The range of the number of milliseconds that the program can wait is from 0 to 32767: the maximum delay is therefore 32.767 seconds. A positive real number is converted to an integer number of milliseconds. If the value specified is negative or zero, the WAIT statement is ignored.

The time delay produced by WAIT is not precise.

Example

```
>LIST
  10  LET Y=5000
  20  LET Z=1
  30  PRINT #Z;"STATEMENT 20"
  40  WAIT (Y)
  50  PRINT #Z;"STATEMENT 40"
  60  GOTO 20
  70  END
>RUN
```

STRINGS

SECTION IV

A string is a set of characters delimited with quotation marks, such as "DDDDDE" or "45T,#". BASIC contains special variables and language elements for manipulating string quantities. This section explains how to use the string features of BASIC. There is little difference in the form of statements referencing numeric quantities and those referencing strings. One important difference, however, is the use of subscripts which is explained later.

Lower-case alphabetic characters can be input from or output to user terminals having this capability. When lower-case characters are output to a terminal not capable of printing them, most terminals will print such characters as the upper-case equivalent. Lower-case characters are automatically converted to upper-case by the system, except when they occur in strings or REM statements.

The examples and comments in this section emphasize modifications in statement form or other special considerations in handling strings.

If you are familiar with the concepts "string", "string variable", and "substring", skip directly to paragraph 4-5.

4-1. STRING

A string is a set of characters enclosed by quotation marks or the null string (no characters). See paragraph 4-5 for detailed information about string dimensions.

Typical Strings:	"ABCDEFGHIJKLMNOP"
	"12345"
	"BOB AND TOM"
	"MARCH 13, 1970"
Null String:	" "

Quotation marks cannot be used within a string because quotation marks are used as string delimiters.

Apostrophes and control characters are legal as string characters.

A null string has no value, as distinguished from a blank space which has a value.

Strings are manipulated in string variables. For example:

```
 100 A$   =  "THIS IS A STRING"
     ↑                 ↑
   string            string
 variable

 200 B$   =  A$(1,10)
     ↑          ↑
   string    substring
 variable    (defined later)

 300 C$   =  ""
     ↑        ↑
   string    null string
 variable
```

4-2. STRING VARIABLE

A string variable consists of a single letter (A to Z) followed by a $, and is used to store strings. A$,Z$,M$ are typical string variables.

String variables must be declared before being used if they contain strings longer than one character. See the String DIM statement, paragraph 4-5.When a string variable is declared, its "physical" length is set. The "physical" length is the maximum size string that the variable can accommodate. For example:

```
71Ø DIM A$(72),B$(2Ø),C$(5Ø)
```

During execution of a program, the "logical" length of a string variable varies. The "logical" length of the variable is the actual number of characters that the string variable contains at any point. For example:

```
1ØØ   DIM A$[72]                          Sets physical length of A$
2ØØ   LET A$="SAMPLE STRING"              Logical length of A$ is 13
3ØØ   LET A$="LONGER SAMPLE STRING"       Logical length of A$ is now 20
```

4-3. SUBSTRING

A substring is a single character or a set of contiguous characters from within a string variable. The substring is defined by a subscript string variable.

A substring is defined by subscripts placed after the string variable. Characters within a string are numbered from the left starting with one. Subscripts must be positive, non-zero, and less than or equal to 255. Non-integer subscripts are rounded to the nearest integer.

Two subscripts, separated by a comma, specify the first and last characters of the substring. For example:

```
100   DIM Z$[72]
200   LET Z$="ABCDEFGH"
300   PRINT Z$[2,6]
```

prints the substring

```
BCDEF
```

A single subscript specifies the first character of the substring and implies that all characters following are part of the substring. For example:

```
300   PRINT Z$[3]
```

prints the substring

```
CDEFGH
```

Two equal subscripts specify a single character substring. For example:

```
>300   PRINT Z$(2,2)
```

Prints the substring

```
B
```

If subscripts specify a substring larger than the physical length of the original string, blanks are appended.

4-4. STRINGS AND SUBSTRINGS

A string can be made into a null string. This is done by assigning it the value of a substring whose second subscript is one less than its first. For example:

```
100 A$ = B$(6,5)
```

A$ now contains a null string.

This is the only case in which a smaller second subscript is acceptable in a substring.

Substrings can become strings. For example:

```
100 A$ = "ABCDEFGH"
200 B$ = A$(3,5)
300 PRINT B$
```

prints the string

CDE

because the substring of A$ is now a string in B$.

Substrings can be used as string variables to change characters within a larger string. For example:

```
100 A$ = "ABCDEFGH"
200 A$(3,5) = "123"
300 PRINT A$
```

prints the string

AB123FGH

Strings,substrings, and string variables can be used with relational operators. They are compared and ordered as entries are in a dictionary. See Appendix A for the ranking of non-alphabetic characters. For example:

```
100 IF A$ = B$ THEN 2000
200 IF A$ <= "TEST" THEN 3000
300 IF A$(5,6) >= B$(7,8) THEN 4000
```

See the STRING IF statement description in this section.

4-5. STRING DIM

Format

DIM *string variable (number of characters in string)*

The string DIM statement reserves storage space for strings longer than 1 character; also for arrays.

The number of characters specified for a string in its DIM statement must be expressed as an integer from 1 to 255.

Each string having more than 1 character must be mentioned in a DIM statement before it is used in the program.

Strings not mentioned in a DIM statement are assumed to have a length of 1 character.

The length mentioned in the DIM statement specifies the maximum number of characters which may be assigned; the actual number of characters assigned may be smaller than this number. See the LEN Function, paragraph 4-11, for further details.

Array dimension specifications may be used in the same DIM statement as string dimensions (example statement 45 below).

Example

```
35 DIM A$(72), B$(6Ø)
4Ø DIM Z$( 1Ø)
45 DIM N$(2), R(5,5), P$(8)
```

4-6. STRING ASSIGNMENT

Format

```
                                    "string literal"
[LET]      string variable      =   string variable
         substring variable
                                    substring variable
```

The string assignment statement establishes a value for a string; the value may be a literal value in quotation marks, or a string or substring value.

An input line may contain a total of 80 characters. One string assignment statement is allowed per input line. Thus, a string may contain 80 characters minus assignment statement elements such as the string variable name, equal sign, quotation marks, and any blanks inserted between these elements. String variables having more than 1 character must be mentioned in a DIM statement (see paragraph 4-5).

Special purpose characters, such as A^c, H^c, D^c, Y^c or quotation marks may not be string characters.

If the source string is longer than the destination string, the source string is truncated at the appropriate point from the right.

Example

```
2ØØ LET A$ = "TEXT OF STRING"
21Ø B$ = "*** TEXT !!!"
22Ø LET C$ = A$( 1,4)
23Ø D$ = B$(4)
24Ø F$(3,8)=N$
```

Strings may be concatenated to include up to 255 characters. For example:

```
  5 DIM A$(240)
100 A$= "60 characters"
101 A$(61,120)=' "60 characters"
102 A$(121,180)= "60 characters"
103 A$(181,240)= "60 characters"
```

4-7. STRING INPUT

Format

$$\text{INPUT} \begin{matrix} \textit{string variable} \\ \textit{substring variable} \end{matrix}, \ldots$$

The string INPUT statement allows string values to be entered from the user terminal.

Placing a single string variable in an INPUT statement allows the string value to be entered without enclosing it in quotation marks.

If multiple string variables are used in an INPUT statement, each string value must be enclosed in quotation marks, and the values separated by commas. The same convention is true for substring values. Mixed string and numeric values must also be separated by commas.

If a substring subscript extends beyond the boundaries of the input string, the appropriate number of blanks are appended at the right.

Numeric variables may be used in the same INPUT statement as string variables (example statement 55 below).

Example

```
50 INPUT R$
55 INPUT A$,B$,C9,D7
60 INPUT A$(1,5)
65 INPUT B$(3)
```

4-8. PRINTING STRINGS

Format

$$\text{PRINT} \begin{matrix} \textit{string variable} \\ \textit{substring variable} \end{matrix} \left[\begin{bmatrix} ; \\ , \end{bmatrix} \begin{matrix} \textit{string variable} \\ \textit{substring variable} \end{matrix} \begin{bmatrix} ; \\ , \end{bmatrix} \ldots \right]$$

A string PRINT statement causes the current value of the specified string or substring variable to be output to the user's terminal device. The terminal device may be any ASCII output device.

String and numeric values may be mixed in a PRINT statement (example statements 115 and 125 below).

Specifying only one substring parameter causes the entire substring to be printed. For instance, in the example below, if the value of B3 = 642 and C$ = "WHAT IS YOUR NAME?", example statement 120 prints:

WHAT IS

while statement 115 prints

YOUR NAME?END OF STRING 642

Numeric and string values may be "packed" in PRINT statements without using a "semicolon", as in example statement 115.

Example

```
105 PRINT A$
110 PRINT A$, B$, Z$
115 PRINT C$(8) "END OF STRING" B3
120 PRINT C$(1,7)
125 PRINT "THE TOTAL IS: ";X5
```

4-9. READING STRINGS

Format

READ *string variable* / *substring variable* [, *string variable* / *substring variable* , . . .]

A string READ statement causes the value of a specified string or substring variable to be read from a DATA statement.

The dimension (length) of a string variable to be assigned more than 1 character must be declared in a DIM statement before attempting to READ its value.

String or substring values read from a DATA statement must be enclosed in quotation marks, and separated by commas. See paragraph 4-12 in this section.

Only the number of characters specified in the DIM statement may be assigned to a string. Blanks are appended to substrings extending beyond the string dimensions.

Mixed string and numeric values may be read (example statement 310 below); see TYP (0), paragraph 5-1, for a description of a data type check which may be used with DATA statements.

Example

```
300 READ C$
305 READ X$, Y$, Z$
310 READ Y$(5), A,B,C5,N$
315 READ Y$(1,4)
```

4-10. STRING IF

Format

IF *string var. relational oper. string var.* THEN *statement number label* / *statement*

A string IF statement compares two strings. If the specified condition is true, control is transferred to the statement number specified or the statement is executed. Statements allowed with IF are listed in paragraph 3-6.

Strings are compared one character at a time, from left to right; the first difference determines the relation. If one string ends before a difference is found, the shorter string is considered the smaller one.

Characters are compared by their ASCII representation.

If substring subscripts extend beyond the length of the string, null characters (rather than blanks) are appended.

String compares may appear only in IF. . .THEN statements and not in conjunction with logical operators.

Strings may not use Boolean expressions.

Example

```
340 IF C$<D$ THEN 800
350 IF C$>D$ THEN 900
360 IF C$=D$ THEN 1000
370 IF N$(3,5)<R$(9) THEN 500
380 IF A$(10)="END" THEN 400
390 IF A$#B$ PRINT A$
```

4-11. LEN FUNCTION

Format

statement type LEN *(string variable)*. . .

The LEN function supplies the current (logical) length of the specified string, in number of characters.

DIM merely specifies a maximum string length. The LEN function allows you to check the actual number of characters currently assigned to a string variable.

Example

```
469 PRINT LEN(A$)
479 PRINT LEN(X$)
489 PRINT "TEXT"; LEN(A$); B$, C
499 IF LEN(P$) #5 THEN 600
509 LET X$(LEN(X$)+1) = "ADDITIONAL SUBSTRING"

  •
  •
  •
  •

600 STOP
609 PRINT "STRING LENGTH = "; LEN(P$)
```

4-12. STRINGS IN DATA STATEMENTS

Format

DATA *"string literal"* [, *string literal"*. . .]

The DATA statement specifies data in a program (numeric values may also be used as data).

String values must be enclosed by quotation marks and separated by commas.

String and numeric values may be mixed in a single DATA statement. They must be separated by commas (example 520 below).

A DATA statement input line may contain a total of 80 characters. Thus, a string literal may contain 80 characters minus the word DATA, quotation marks, and any blanks, or commas and other string literals included within the input line.

Example

```
500 DATA "NOW IS THE TIME."
510 DATA "HOW", "ARE", "YOU,"
520 DATA 5.172, "NAME?", 6.47,5071
```

4-13. PRINTING STRINGS TO A PERIPHERAL DEVICE

Format

PRINT # *logical unit number* ; *string variable* / *substring variable* / *"string literal"* [, . . .]

The PRINT # statement prints string or substring variables or string literals on a specified logical unit number.

String and numeric variables may be mixed on output to a device (example statement 360 below).

See Section VII for more information about printing output to a peripheral device.

Example

```
350 PRINT #4; "THIS IS A STRING."
355 PRINT #8; C$, B$, X$, Y$, D$
360 PRINT #7; X$, P$, "TEXT", 27.5,R7
365 PRINT #N; P$ N, A(5,5), "TEXT"
```

4-14. READING STRINGS FROM A PERIPHERAL DEVICE

Format

READ # *logical unit number* ; *string variable* / *substring variable* , *string variable* / *substring variable* [, . . .]

The READ # statement reads string and substring values from a specified logical unit number.

String and numeric values may be mixed on input from a device and in a READ number statement; they must be separated by commas.

See Section VII for more information about reading input from a peripheral device.

Example

```
710 READ #1; A$, B$
715 READ #4; C$, A1, B2, X
720 READ #5; C$(5),X$(4,7),Y$
730 READ #N; C$, V$(2,7),R$(9)
```

FUNCTIONS

SECTION V

A function is the mathematical relationship between two variables, X and Y, for example, that returns a single value of Y for each value of X. The independent variable is called an argument; the dependent variable is the function value. To illustrate, in the statement:

```
100 LET Y = SQR(X)
```

X is the argument; the function value is the square root of X; and Y takes the value of the positive root.

Two types of functions are used in BASIC: system defined functions and user-defined functions.

5-1. SYSTEM-DEFINED FUNCTIONS

Real-Time BASIC provides a variety of functions that perform common operations such as finding the sine, taking the square root, or finding the absolute value of a number. The resulting value of a function is always numeric and can be used in the evaluation of an expression. Available system-defined functions are listed below:

ABS(*x*)	The ABS function gives the absolute value of the expression (*x*).
ATN(*x*)	ATN is the arctangent function. ATN returns the angular argument of *x* in radians adjusted to the appropriate quadrant.
COS(*x*)	The COS function returns the cosine of *x* expressed in radians.
EXP(*x*)	EXP gives the value of the constant *e* raised to the power of the expression (*x*).
IERR(*x*)	This function returns the error code value which may have been set by a user-defined subroutine or function. See Section VI. In this function, *x* is a dummy argument.
INT(*x*)	The integer function, INT, provides the largest integer $<=x$.
LEN(*x*$)	Determines length (no. of characters) in character string identified by string variable *x*$. See Section IV.
LOG(*x*)	Gives base 10 logarithm of variable or expression.
LN(*x*)	LN provides the logarithm of a positive expression to the base *e*.
OCT(*x*)	This function prints the octal equivalent of an integer value. The maximum possible range of the returned variable is $0\text{-}177777_8$. If *x* is outside the range of −32768 to 32767, 77777_8 is returned.
RND(*x*)	RND generates a random number greater than or equal to zero and less than 1. The expression *x* may have any value. A sequence of random numbers is repeatable if the initial reference to the RND function contains a negative argument value and is followed by a reference to RND containing a positive argument value. A random sequence can be achieved with positive arguments.

SERR(x)	Sets the error code which may be queried with IERR(x). See Section VI.
SGN(x)	SGN returns 1 for $x > 0$, 0 for $x = 0$, and -1 for $x < 0$.
SIN(x)	The SIN function gives the sine of x expressed in radians.
SQR(x)	SQR provides the square root of x. The value of x must be greater than zero.
SWR(x)	The SWR function returns the logical value, one or zero, of the Switch Register bit position specified by x (range = 0 through 15).
TAB(x)	The TAB function is used to advance the print position the number of positions specified by x. The value of x may be equal to 0 through 71. See Section III.
TAN(x)	The TAN function returns the tangent of x expressed in radians.

TIM(x)	The TIM function returns the current minute, hour, day or year.
	
	x = 0, TIM(x) = current minutes (0 to 59)
	x = 1, TIM(x) = current hour (0 to 23)
	x = 2, TIM(x) = current day (1 to 366)
	x = 3, TIM(x) = current year (four digits).
	x = −1, TIM(x) = current seconds (0 to 59)
	x = −2, TIM(x) = current tens of milliseconds.

TYP(x)	The TYP function references the DATA statements and returns the following response: 1 = number, 2 = string, 3 = "out of data" condition. The value of argument x must be zero.

5-2. USER-DEFINED FUNCTIONS

A user-defined function is one that you define for use in your program. It is called and used the same way that a system-defined function is. The DEF statement is used to define a new function, that is to equate the function to a mathematic expression.

Format

DEF FN*x*(*y*) = *expression*

Parameters

x	stands for a letter (A-Z) that completes the name of the function. Only 26 user-defined functions may be specified: FNA through FNZ.
y	stands for the variable to which the function is to be applied. Any number, string, or variable may be used in this position.
expression	provides a formula such as X*X or X ↑TAN(X). Whenever the function is called in the program, this formula will be evaluated.

Example

```
>10 DEF FNA(Y)=Y/10
>20 PRINT FNA(100)
>30 END
>RUN
10
```

When FNA (100) is called for in statement 20, the formula defined for FNA is evaluated to determine the value printed. Note that the results of the function may be used in computation:

```
35 LET X = FNA(M1) + 14 -FNA(12)
```

An operand in the program may be used in the defining expression, however, such circular definitions as the one below cause infinite looping.

```
10 DEF FNA(Y) = FNB(Y)+1
20 DEF FNB(X) = FNA(X)-1
```

SUBROUTINES

SECTION VI

It is often preferable to make use of the same procedure several times within a program. Rather than re-writing the procedure each time it is to be used, you can simply refer to a given segment of code (a subroutine) whenever that segment is needed. The GOSUB/RETURN statement sequence is used when a subroutine is located within your own program.

There are also times when the inclusion of subroutines outside of your program is desirable. In this case, the CALL statement is required. External subroutines are completely separate from your program and from the BASIC Interpreter. They are routines, accessed via a memory directory, and must have been specified by a special subroutine configuration process as described in Section XIII.

6-1. GOSUB/RETURN

GOSUB transfers control to the beginning of a simple subroutine. A subroutine consists of a collection of statements that may be executed from more than one location in the program. In a simple subroutine, there is no explicit indication in the program as to which statements constitute the subroutine. A RETURN statement in the subroutine returns control to the statement following the GOSUB statement.

Format

GOSUB *statement number label*

GOSUB *integer expression* OF *statement number label* [, *statement number label, . . .*]

RETURN

GOSUB may have a single *statement number label,* or may be multi-branched with more than one label separated by commas. In a multi-branch GOSUB, the particular label to which control transfers is determined by the value of the *integer expression* which is rounded to the nearest integer. The RETURN statement consists simply of the word RETURN.

A single-branch GOSUB transfers control to the statement indicated by the label. A multi-branch GOSUB transfers to the statement label determined by the value of the integer expression. As in a multi-branch GOTO, if the value of the expression is less than 1 or greater than the length of the list, no transfer takes place.

When the sequence of control within the subroutine reaches a RETURN statement, control returns to the statement following the GOSUB statement. RETURN statements may be used at any desired exit point in a subroutine. There may be more than one RETURN statement per GOSUB.

Within a subroutine, another subroutine can be called. This is known as nesting. When a RETURN is executed, control transfers back to the statement following the last GOSUB executed. Up to 20 GOSUB statements can occur without an intervening RETURN; more than this causes a terminating error.

Examples

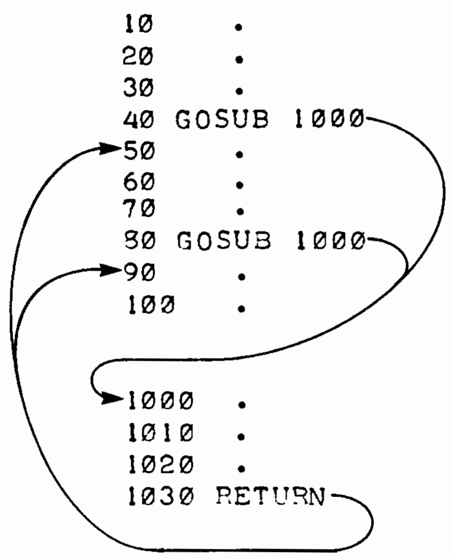
In the first example, line 20 contains a simple GOSUB statement; the subroutine is in lines 50 through 70, with RETURN in line 70.

```
>LIST
  10   LET B=90
  20   GOSUB 50
  30   PRINT "SINE OF B IS ";A
  40   GOTO 80
  50   REM: THIS IS THE START OF THE SUBROUTINE
  60   LET A=SIN(B)
  70   RETURN
  80   REM: PROGRAM CONTINUES WITH NEXT STATEMENT AFTER 20
  90   END
>RUN
SINE OF B IS .893993
```

The GOSUB statement can follow the subroutine to which it transfers as in the example below.

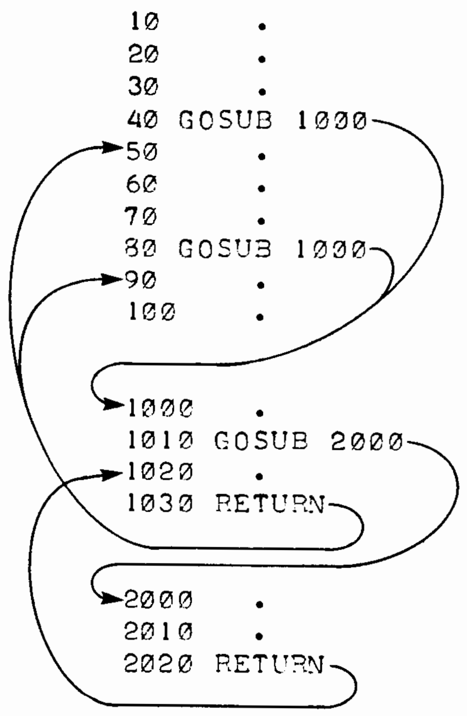
```
>LIST
  10   LET B=90
  20   GOSUB 110
  30   REM: THIS IS THE START OF SUBROUTINE
  40   LET A=SIN(B)
  50   RETURN
  60   REM: OTHER STATEMENTS CAN APPEAR HERE
  70   REM: THEY WILL NOT BE EXECUTED
  80   LET A=24
  90   LET B=50
  100   PRINT A;B
  110   GOSUB 30
  120   PRINT A
  130   REM: A SHOULD EQUAL .893993
  140   PRINT B
  150   REM: B SHOULD EQUAL 90
  160   END
>RUN
.893993
90
```

It is also possible for any one subroutine to be called from several places in the coding of any one program. The logical flow of this situation looks like this:



```
10      .
20      .
30      .
40 GOSUB 1000
50      .
60      .
70      .
80 GOSUB 1000
90      .
100     .

1000    .
1010    .
1020    .
1030 RETURN
```

Taking the same situation one step further, it is permissable for a subroutine to, in turn, call another subroutine:



```
10      .
20      .
30      .
40 GOSUB 1000
50      .
60      .
70      .
80 GOSUB 1000
90      .
100     .

1000    .
1010 GOSUB 2000
1020    .
1030 RETURN

2000    .
2010    .
2020 RETURN
```

Subroutines should be entered only with GOSUB statements rather than GO TO's to avoid unexpected RETURN errors (which cause the program to stop execution).

This sequence shows logically nested GOSUB's:

```
10 INPUT
20 GOSUB 100
•
•
•
100 IF C>0 THEN 120
110 LET C=-C
120 GOSUB 200
130 RETURN
•
•
•
200 LET A=SQR(C)
210 LET C=SQR(A)
220 RETURN
300 END
```

The order in which this program is executed is:

when C>0:
10
20
100
120
200
210
220
130
statements after 20

when C<=0:
10
20
100
110
120
200
210
220
130
statements after 20

6-2. CALL

The CALL statement is used to identify and execute an external subroutine at a given point within a program. CALL is optional, you may simply use the subroutine name and parameter list. After the subroutine executes, control returns to the statement following the CALL unless there is a FAIL return.

Format

[CALL]*subroutine name(parameter list)* [FAIL: *statement*]

Parameters

subroutine name	name of the routine as entered into the system during system generation or when loaded on-line.
parameter list	list of variables or constants to be passed to the subroutine or variables into which the subroutine places information for the calling program. Spaces are not allowed between the subroutine name and the left parenthesis.
FAIL: *statement*	optional subroutine failure return. See paragraph 6-3.

To execute the subroutine calling sequence, you need to determine the following:

- the name of the subroutine
- the number of parameters in the call
- the meaning of the contents of each parameter
- the values acceptable in each parameter.

Usually this information is provided in the documentation supplied with the subroutine.

Examples

Figures 6-1 and 6-2 contain examples of routines written in FORTRAN which may be called from BASIC.

Constant numbers, string literals, and expressions cannot be used as parameter values when calling a subroutine if the parameter is defined as a return variable (type V, see Section XIII).

```
FTN,L,M
      INTEGER FUNCTION NUM(I)
C
C
C
C
C THIS FUNCTION  RETURNS THE NUMERIC VALUE OF THE FIRST CHARACTER
C OF THE STRING EXPRESSION ACCORDING TO THE STANDARD CHARACTER CODE.
C
C   FOR EXAMPLE:
C
C      10 PRINT NUM("A")
C      20 END
C
C      >RUN
C
C      65
C
C
C
C THE FUNCTION'S DESCRIPTION THAT MUST BE INPUT TO THE TABLE
C GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
C TABLE IS AS FOLLOWS:
C
C   NUM(R), INTG, ENT=NUM
C
C   WHERE:    R      INDICATES REAL PARAMETER(STRINGS ARE ALWAYS REAL)
C
C
C
C
C
C
      DIMENSION I(2)
C
C RIGHT JUSTIFY CHARACTER BY DIVIDING
C
C RIGHT HALF OF THE FIRST WORD OF A STRING IS THE CHARACTER COUNT
C AND MUST NOT BE DISTURBED.
C
      NUM =I(2)/256
      RETURN
      END
```

Figure 6-1. Preparing a FORTRAN Function for Use by BASIC Program

```
FTN,L,M
      SUBROUTINE CHRS(I,J)
C
C
C
C
C THIS SUBROUTINE CAUSES THE NUMERIC VALUE OF THE FIRST PARAMETER
C TO REPLACE THE FIRST CHARACTER OF THE SECOND PARAMETER WHICH
C IS A STRING VARIABLE.
C
C
C   FOR EXAMPLE:
C
C      1Ø DIM A$(1Ø)
C      2Ø A$="YBCDE"
C      3Ø CHRS(65,A$)
C      4Ø PRINT A$
C      5Ø END
C
C      >RUN
C
C      ABCDE
C
C
C
C THE FUNCTION DESCRIPTION THAT MUST BE INPUT TO THE TABLE
C GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
C TABLE IS AS FOLLOWS:
C
C      CHRS(I,RVA), END=CHRS
C
C      WHERE:  I      INDICATES AN INTEGER VARIABLE PASSED TO 'CHRS'
C              RVA    INDICATES A REAL VARIABLE (STRINGS ARE ALWAYS
C                     SPECIFIED AS REAL) RETURNED FROM 'CHRS'
C
C
C
C
C
C
C
      DIMENSION J(2)
C
C PLACE CHARACTER IN FIRST CHARACTER POSITION OF STRING 'J'
C
C THE RIGHT HALF OF THE FIRST WORD OF A STRING IS THE CHARACTER
C COUNT AND MUST NOT BE DISTURBED.
C
      J(2)=IAND(J(2),377B)
      J(2)=IOR(I*256,J(2))
      RETURN
      END
```

Figure 6-2. Preparing a FORTRAN Subroutine for Use by BASIC Program

6-3. THE FAIL ERROR OPTION

Some of the externally defined subroutines supplied with the BASIC Interpreter make error checks at execution time. For example, the TRNON routine checks both the time schedule table and the trap table for overflow before adding a new entry. If an execution time error is detected, an appropriate error message is printed, the ERRCD flag is set, the program is aborted, and the BASIC Interpreter returns to command input mode.

You may avoid aborting your program by using the FAIL option as part of the subroutine call statement. Any statement which can appear in an IF statement can be added to the end of a subroutine CALL statement following the word FAIL:.

For example:

```
100 CALL TRNON(2000,122536)FAIL: GO TO 9000
```

If the called subroutine detects an error during execution, the error message is printed but the Interpreter executes the statement following FAIL: instead of aborting the program. The error message format is:

ERROR *n* IN LINE *xxx* where *n* is the ERRCD value.

The FAIL: option may be used with the following routines:

SETP	Task Control Statements
TRNON	
START	
ENABL	
DSABL	
RDBIT	HP 6940 Calls
RDWRD	
WRBIT	
WRWRD	
DAC	
MPNRM	
SENSE	
AISQV	HP 2313 Calls
SGAIN	
RGAIN	
AOV	
NORM	
PACER	

These routines are described in Sections X and XIII.

If ERRCD equals zero, the FAIL statement is not executed.

6-4. THE IERR FUNCTION

Since the action desired may depend on which error occurred, the function IERR is supplied to interrogate the ERRCD flag. It is a BASIC function and must be used as an operand in an expression. It returns the value of ERRCD. IERR requires one dummy parameter which is ignored. Any call to another external subroutine or execution of a PRINT statement resets the value of IERR(*x*).

Example

```
100 CALL TRNON(2000,124515)FAIL:GOTO 9000   Specify task 2000 to be executed
•                                           at 12:45 and 15 sec. If error, go to
•                                           9000.
•
9000 IF IERR(X) = 1 GOTO 9100               If error is 1, go to 9100.
9010 IF IERR(X) = 2 GOTO 9200               If error is 2, go to 9200.
```

6-5. THE SERR FUNCTION

You may use the SERR function to set the ERRCD flag to a particular value in a subroutine. For example, the statement:

110 I= SERR(N) (I is a dummy variable)

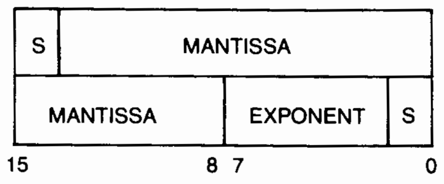
sets the ERRCD flag to the value of N. After execution of your subroutine you can examine the error code by using the IERR function. The value of I is unchanged.

The CALL statement initializes ERRCD to 0, however, you should initialize it at the beginning of your program and reset it to zero after you have detected an error in a routine and taken appropriate action to avoid leaving it set in case there are no more CALLs. You initialize the error code as follows:

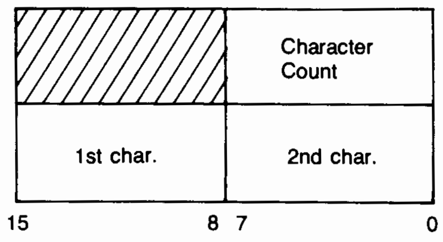
10 I= SERR (0)

6-6. PARAMETER CONVERSION

BASIC has two data types: number (real) and string. The format of real data is:



and for string is:



The leftmost half of the 1st word may contain information used internally by BASIC.

If you want to pass parameters to or from external subroutines, you must be aware of the internal representation of these two data types. BASIC converts real data variables and arrays to integer and vice versa, and thus provides you with the ability to transfer data between BASIC and subroutines that use integer type data. For example, if the subroutine passes an integer array to BASIC, you must specify that parameter as an integer returned array when generating the Branch and Mnemonic Tables. (See Section XIII.)

Some RTE-M EXEC calls use string parameters but do not follow the BASIC string format convention. If you want to call one of these subroutines, you must first write an interface routine to convert the BASIC string to the necessary format. Figure 6-3 contains a FORTRAN routine which converts the program name to the FORTRAN string format.

BASIC does not pass the character string address to a FORTRAN subroutine. It passes the address of the word containing the specified character.

```
FTN,L,M
      SUBROUTINE EXEC9(I)
C
C
C
C
C THIS SUBROUTINE CALLS THE RTE 'EXEC' TO SCHEDULE A PROGRAM WITH WAIT
C
C   FOR EXAMPLE:
C
C     10 DIM A$(6)
C     20 PRINT "INPUT PROGRAM NAME";
C     30 INPUT A$
C     40 CALL EXEC9(A$)                               BASIC program
C     50 END
C
C     >RUN
C
C     INPUT PROGRAM NAME?PROGA
C
C
C
C THE SUBROUTINE'S DESCRIPTION THAT MUST BE INPUT TO THE TABLE
C GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
C TABLE IS AS FOLLOWS:
C
C   EXEC9(RA),            INTG, ENT=EXEC9
C
C    WHERE:  RA         INDICATES REAL ARRAY PARAMETER( STRINGS
C                       ARE ALWAYS SPECIFIED AS REAL )
C
C
C
C
C
      DIMENSION I(4)
C
      CALL EXEC(9,I(2))                             FORTRAN Subroutine
      RETURN
      END
```

Figure 6-3. FORTRAN Subroutine to Convert String Parameter

LOGICAL UNIT INPUT/OUTPUT

SECTION VII

For situations that require permanent data storage external to a particular program, BASIC provides a logical unit I/O capability. This capability allows direct reading from and writing to peripheral I/O devices.

The statements you may use to perform I/O operations from and to peripheral devices are:

- READ #
- PRINT #
- IF EOF # . . . . THEN

These statements are described in the remainder of this section.

7-1. READ # STATEMENT

The READ # statement is used to read items from a peripheral device into numeric or string variables.

Format

READ # *lu number; variable list*

lu number	a number, variable, or expression whose value represents a logical unit number. This is the peripheral device from which the data is to be read.
variable list	a series of variables separated by commas. The rules governing this list are the same as those described for the READ statement described in Section III.

The READ # statement fills the variables in the list by performing a serial read operation on the device associated with the specified LU number.

String data and numeric data may be intermixed in the source data list, but the type of data in the variable list and the source data list must correspond in the correct order. That is, the destination for a string value must be a string variable and that for a numeric value must be a numeric variable.

If you attempt to read beyond a logical or physical end-of-file, an end-of-file error condition results. The program terminates unless an IF EOF # . . . . THEN statement (paragraph 7-3) transfers control to another statement.

Example

```
400 READ #5;A,B(3),R(1,3)
```

7-2. PRINT # STATEMENT

The PRINT # statement writes data items to a peripheral device. The items may be string or numeric data.

Format

PRINT # *lu number; print list*

lu number	a number, variable, or expression whose value represents a logical unit number. This is the peripheral device to which the data is to be written.
print list	a series of numeric expressions, numeric or string variables, or string literals.

The PRINT # statement performs essentially the same operation as the ordinary PRINT statement, except that data is written to a specified peripheral device. No line formatting takes place, the comma and semicolon act only as delimiters and may not be used as actual data unless the LU number refers to a teleprinter or lineprinter. (See the PRINT statement, paragraph 3-7.)

Examples

`30 PRINT #1;A,"SAMPLE",A$`	*The value of A, the string literal SAMPLE, and the string value of A$ are written to LU 1.*
`60 PRINT #9;"SUMMARY"`	*The string literal SUMMARY is written to LU 9.*
`150 PRINT #4;A,B,C`	*The values of the variables A, B, and C are written to the standard output device, LU 4.*
`21 PRINT #M;2,42,A,B,C,D(3,5)`	*The items in the print list are written to the peripheral device associated with the value of M.*

7-3. IF EOF # . . . . THEN STATEMENT

The IF EOF # . . . . THEN statement checks the status of a specified peripheral device so that if an end-of file (EOF) condition has been encountered in reading from a peripheral device, control is transferred to a specified statement.

Format

IF EOF # *lu number* THEN *statement number*

lu number	a number, variable, or expression whose value represents a logical unit number for the peripheral device whose status is to be checked.
statement number	the number of a BASIC statement to which control transfers on an EOF condition.

An end-of-file condition occurs when execution of a READ # statement encounters a logical or physical EOF.

Example

```
10 IF EOF #5 THEN 125
```

When a logical or physical EOF occurs on LU 5, control transfers to program statement 125.

STARTING UP BASIC

SECTION VIII

BASIC is a program, and as such must be made available for use on your system. Procedures for loading BASIC are provided in the HP 92064A RTE-M System Generation Manual. Once BASIC is loaded and ready for use, you need simply schedule BASIC for operation as described below.

8-1. SCHEDULING BASIC

The command to start the BASIC Interpreter is:

*RU,BASIC [*console*[*,list*[*,input*[*,output*]]]]

console	is the LU number of the keyboard device you are using. Default is LU 1, the system console.
list	is the LU number of the list device you want to use. Default is LU 1, the system console.
input	is the LU number of the input device you want to use. Default is LU 5, the standard input device.
output	is the LU number of the output device you want to use. Default is LU 4, the standard output device.

If you omit the optional parameters, the default device specified above automatically will be used by BASIC. You must type a comma in place of an omitted parameter if you specify subsequent parameters. If you want to alter the devices used, enter the LU numbers of the devices that you prefer.

The following is an example of the RUN command used to schedule BASIC. Once BASIC begins execution, a "ready" message is displayed and followed with a "greater than" symbol:

```
RU,BASIC
BASIC READY
>
```

8-2. USING BASIC

BASIC prompts for commands and program statements with the "greater than" symbol (>). Following the display of this prompt, you may enter any legal BASIC command or statement.

8-3. START UP OPTIONS

You may start BASIC directly from the console, or you may schedule BASIC from another program. The program may be written in FORTRAN, ALGOL, Assembly Language, or BASIC.

Examples

1. To start BASIC from the console, enter the following command:

```
*RU,BASIC
```

2. To initiate BASIC from another program, use the following calling sequences:

Assembly Language:

```
       JSB EXEC            Transfer control to RTE-M
       DEF *+6
       DEF .9              Request code (.10 if schedule without wait)
       DEF BASIC           Program name (BASIC).
       DEF P1
       DEF P2              Parameters.
       DEF P3
       DEF P4

                           Continue execution of program.
.9     DEC 9               Schedule with wait.
BASIC  ASC,3,BASIC         BASIC Interpreter name.
P1     DEC 1               Console logical unit number.
P2     DEC 6               List logical unit number.
P3     DEC 1               Input logical unit number.
P4     DEC 4               Output logical unit number.
```

FORTRAN:

```
DIMENSION NAME(3)              Store name of BASIC in integer array name.
NAME(1)=41101B
NAME(2)=51511B
NAME(3)=41440B
CALL EXEC(9,NAME,1,6,1,4)      Transfer control to RTE-M.
```

For additional information on scheduling BASIC from programs, see the RTE-M Programmers Reference Manual.

8-4. MULTI-TERMINAL MONITOR ENVIRONMENT

Because of memory space limitations, Hewlett-Packard recommends that you include support of the Multi-Terminal Monitor (MTM) only in an RTE-MIII System having at least 48K words of memory. In such a system, you may execute BASIC on up to four terminals. A separate copy of the BASIC Interpreter may be created for each terminal connected to the system. The HP 92064A RTE-M System Generation Manual and the HP 92064A RTE-M Programmers Reference Manual contain the information you need to generate a system that supports MTM and to create multiple copies of BASIC.

When scheduling BASIC in an MTM envrionment, you use the same RUN command syntax described in paragraph 8-1.

Note that the default values for RUN command parameters are those described in paragraph 8-1. Thus, when scheduling BASIC from an MTM terminal you should specify the LU number of your terminal in parameter 1. For example, to schedule a copy of BASIC from a terminal associated with LU 7, enter:

```
*RU,BAS07,7
```

If you do not specify an LU number in parameter 1, BAS07 will be scheduled at the system console (LU 1).

See paragraph 8-1 for additional information concerning the scheduling of BASIC.

BASIC INTERPRETER COMMANDS

SECTION IX

This section describes the BASIC commands. Unlike the statements discussed in earlier sections, commands are not part of a program, nor are they preceded by line numbers. When entered, a command is executed immediately.

Table 9-1 lists and defines the various operator commands available to you with BASIC. Detailed explanations of most of the commands are provided in the remainder of the section. Additional commands used in specific situations such as magnetic tape drive manipulation are introduced and explained in subsequent sections. A complete summary of of the commands and their uses is provided in Appendix B.

Table 9-1. BASIC Interpreter Commands

LOAD	Loads a source program from a peripheral device into memory.
SAVE	Stores the program currently in memory on a peripheral device.
MERGE	Merges a source program from a peripheral device with a program in memory.
DELETE	Deletes a program from memory.
RUN	Loads and executes a program from a peripheral device or executes a program currently in memory.
BYE	Terminates the execution of the BASIC Interpreter.
LIST	Lists the program currently in memory on a peripheral device.
*BR,BASIC	Breaks (interrupts) execution of the current program.
REWIND	Rewinds magnetic tape.*
SKIPF	Skips to end-of-file on magnetic tape.*
BACKF	Backspaces to end-of-file on magnetic tape.*
WEOF	Writes end-of-file on magnetic tape.*

*This command is described in Section XII

9-1. LOAD

The LOAD command enables you to load all or a portion of a source program from a peripheral device identified by a specified logical unit number.

Format

LOAD [*line-1,line-2*] [FROM *lu number* / *filename*]

line-1,line-2	the beginning and ending line numbers of the portion of the program you want loaded. If omitted, the entire program is loaded. The value of *line-2* must be equal to or greater than that of *line-1*.
FROM *lu number*	a number whose value represents a logical unit number. This is the peripheral device containing the program to be loaded. If omitted, the default is LU 5 or the LU number specified as the input device parameter in the RUN,BASIC command. If this parameter is specified, the keyword FROM must be included.
FROM *filename*	filename can be used only if the optional file handler is present. Filename is a NAMR specification for the flexible disc file in which the program is stored. The form of NAMR is:

file [*:sc* [*:crn* [*:type* [*file size*]]]]:

file	= file name
sc	= security code
crn	= cartridge label number if positive disc logical unit number if negative
type	= always type 4
file size	= number of blocks in file (one block = 128 words)

The LOAD command reads in all program statements between and including the line numbers specified. If no line numbers are specified, the entire program is loaded.

Once loaded, a program is ready for execution or editing.

Examples

`>LOAD`	*Loads from default input device.*
`>LOAD 150,250 FROM 8`	*Loads program statements 150 through 250 from LU 8.*

9-2. SAVE

The SAVE command stores the BASIC program currently in memory on a peripheral device.

Format

SAVE [*line-1,line-2*] $\begin{bmatrix}\text{ON } \textit{lu number} \\ \textit{filename}\end{bmatrix}$

line-1,line-2	the beginning and ending line numbers of the program to be saved. If no line numbers are specified, the entire program currently in memory is saved. The value of *line-2* must be equal to or greater than that of *line-1*.
ON *lu number*	a number whose value represents a logical unit number. This is the LU number of the peripheral device on which the program is to be saved. If omitted, the program will be saved on LU 4 or the LU number specified as the output device parameter in the RUN,BASIC command. If this parameter is specified, the keyword ON must be included.
ON *filename*	filename can be used only if the optional file handler is present. Filename is a NAMR specification for the flexible disc file onto which the program is to be stored. If the file does not already exist it is created for you. The form of NAMR is:

file [*:sc* [*:crn* [*:type* [*file size*]]]]:

file	= file name
sc	= security code
crn	= cartridge label number if positive disc logical unit number if negative
type	= always type 4
file size	= number of blocks in file (one block = 128 words)

The SAVE command stores only those statements between and including the line numbers you specify. If no line numbers are specified the entire program is saved.

A program stored using the SAVE command can be edited with the RTE-M Editor (EDITM) as well as with the BASIC Interpreter. Subsequently, you may load and execute the program as needed.

Examples

`>SAVE`	*The current source program is written to the default output device.*
`>SAVE 100,260`	*Lines 100 through 260 of the current source program are written to the default output device.*

9-3. MERGE

The MERGE command merges a source program or a portion of a source program on a peripheral device with a program residing in memory.

Format

MERGE [*line-1,line-2*] [FROM *lu number* / *filename*]

line-1,line-2	the beginning and ending line numbers of the program to be merged. If omitted, the entire program is merged. The value of *line-2* must be equal to or greater than that of *line-1*.
FROM *lu number*	a number whose value represents a logical unit number. This is the peripheral device from which a program is to be merged. If omitted, LU 5 or the LU number specified as the input device in the RUN, BASIC command is assumed. If this parameter is specified, the keyword FROM must be included.
FROM *filename*	filename can be used only if the optional file handler is present. Filename is a NAMR specification for the flexible disc file in which the program is stored. The form of NAMR is:

file [*:sc*[*:crn*[*:type*[*file size*]]]]:

file	= file name
sc	= security code
crn	= cartridge label number if positive disc logical unit number if negative
type	= always type 4
file size	= number of blocks in file (one block = 128 words)

You use the MERGE command to combine BASIC program statements from a peripheral device with BASIC program statements currently in memory.

MERGE will not replace line numbers in memory with duplicate line numbers from a peripheral device. If any line number from a device is encountered that duplicates a line number already in memory, the new line number is ignored. This is useful if you want to load a program that contains syntax errors. Simply type the corrected statement into memory using the same line number as the statement in the original program. Then, use the MERGE command to merge in the original program from a peripheral device. The corrected statements in memory will remain intact. The erroneous statements from the original program will be ignored.

Examples

`>MERGE`	*Merges source statements from the default input device.*
`>MERGE 120,190 FROM 8`	*Merges source statement lines 120 through 190 from LU 8.*

9-4. DELETE

The DELETE command enables you to remove a program or a portion of a program from memory.

Format

DELETE [*line-1,line-2*]

line-1,line-2	the beginning and ending line numbers of the portion of the program to be deleted. If no line numbers are specified, the entire program is deleted. The value of *line-2* must be equal to or greater than that of *line-1*.

DELETE effectively erases the program currently in memory. The line number parameters allow you to delete specific portions of a program. The DELETE command may be abbreviated; you may use DEL when entering the command.

A single statement may be deleted simply by typing the statement number and pressing RETURN.

Examples

`>DELETE`	*Deletes the current program from memory.*
`>DEL 50,425`	*Deletes statement numbers 50 through 425 of the current program.*
`>DEL`	*Deletes the current program from memory.*
`>45`	*Deletes line 45 of the current program.*

9-5. RUN

The RUN command enables you to load and execute a program or portion of a program in one operation.

Format

RUN [*line-1,line-2*] $\begin{bmatrix}\text{FROM } \textit{lu number}\\ \textit{filename}\end{bmatrix}$

line-1,line-2	the beginning and ending line numbers of the portion of the program to be executed. If no line numbers are specified, the entire program is executed. The value of *line-2* must be equal to or greater than that of *line-1*.
FROM *lu number*	a number whose value represents a logical unit number. This is the peripheral device from which the program to be executed is loaded. If omitted, it is assumed that the program is currently in memory. If this parameter is specified, the keyword FROM must be included.
FROM *filename*	filename can be used only if the optional file handler is present. Filename is a NAMR specification for the flexible disc file in which the program is stored. The form of NAMR is:

file [*:sc* [*:crn* [*:type* [*file size*]]]]:

file	= file name
sc	= security code
crn	= cartridge label number if positive disc logical unit number if negative
type	= always type 4
file size	= number of blocks in file (one block = 128 words)

The RUN command loads and executes a program or portion of a program. It is used for a program already in memory or for a program residing at an input device.

Examples

`>RUN`	*Executes the program currently in memory.*
`>RUN 50,75 FROM 12`	*Loads and executes line numbers 50 through 75 from LU 12.*

9-6. BYE

The BYE command is used to terminate execution of the BASIC Interpreter.

Format

```
BYE
```

Upon entry of the BYE command, the execution of the BASIC Interpreter is terminated immediately. Control is returned to the RTE-M Operating System or the program that scheduled BASIC. You should always use the BYE command (not *OFF,BASIC,1) to terminate your BASIC session.

9-7. LIST

The LIST command enables you to copy a program or portion of a program to a specific peripheral device.

Format

LIST [*line-1,line-2*] [ON *lu number* / *filename*]

line-1,line-2	the beginning and ending line numbers of the portion of the program to be listed. If omitted, the entire program is listed. The value of *line-2* must be equal to or greater than *line-1*.
ON *lu number*	a number whose value represents a logical unit number. This is the peripheral device on which the program will be listed. If omitted, LU 1 or the LU number for the list device specified in the RUN,BASIC command is assumed. If this parameter is specified, the keyword ON must be included.
ON *filename*	filename can be used only if the optional file handler is present. Filename is a NAMR specification for the flexible disc file onto which the program is to be stored. If the file does not already exist it is created for you. The form of NAMR is:

file [*:sc* [*:crn* [*:type* [*file size*]]]]:

file	= file name
sc	= security code
crn	= cartridge label number if positive disc logical unit number if negative
type	= always type 4
file size	= number of blocks in file (one block = 128 words)

The LIST command enables you to list a program or portion of a program in memory on a specific device. The program is listed in ascending sequence by statement number. You may request a partial list by specifying line numbers. Any FOR . . . . NEXT statement pairs are indented two spaces for easy identification of program loops.

Examples

`>LIST`	*Lists the current program in memory.*
`>LIST 120,180`	*Lists statements 120 through 180 from the current program in memory.*

9-8. *BR

The *BR command permits you to interrupt an executing BASIC program or the listing of a program. It is used as follows:

- When a program is executing and you wish to interrupt it, press any key at your terminal.
- The RTE-M system prompt is printed.
- Enter BR as follows:

Format

*BR,*interpreter program name*

interpreter program name	the name under which the BASIC Interpreter program exists within your RTE-M system. See the RTE-M System Generation Manual, Section V, for information about naming the BASIC Interpreter program in MTM and flexible disc drive environments.

The program is interrupted immediately. Control is returned to the Interpreter at this point and the message OPERATOR TERMINATION IN LINE *nnnn* is printed on the console, where *nnnn* is the program statement number where the break occurred.

To break execution of a program which is waiting for a response to an INPUT, READ #, or PAUSE statement, type Control Q (Q^c) and press RETURN.

Examples

`*BR,BASIC`	*Break (interrupt) the BASIC program currently being executed by the BASIC Interpreter.*
`13>BR,BAS13`	*Under MTM control, break (interrupt) the BASIC program currently being executed by a copy of the BASIC Interpreter named BAS13.*

REAL-TIME TASK SCHEDULING

SECTION X

10-1. INTRODUCTION

BASIC is said to operate in *real-time* because the order of processing is governed by time or by the occurrence of external events rather than by a strict sequence defined in the program itself. Because these events can occur in a random order and require different amounts of processing, a real-time system must be capable of resolving conflicts between tasks.

A task is defined as a group of BASIC statements initiated by a call to one of the BASIC scheduling subroutines (e.g. START, TRAP, etc.) and terminated by a RETURN statement. Each task is uniquely identified by the line number of the first statement in the task. If a task is initiated by executing line 2000 of the BASIC program, that task will be represented in all scheduling calls as 2000. No blanks are allowed in the name or between the subroutine name and the left parenthesis preceding the parameter list. The facilities provided by these routines require that the TRAP subroutine reside in the System Resident Library. If the scheduling routines are not to be used, the external reference to the TRAP entry point may be satisfied by the Dummy Trap subroutine. The Dummy Trap subroutine must be relocated together with the BASIC Interpreter.

10-2. METHODS OF INITIATING TASKS

A task can be initiated in three different ways:

- at a specified time of day,
- after a specified delay,
- upon the occurrence of an external event.

To initiate a task at a given time of day, use the TRNON routine. For example:

```
   100 CALL TRNON(2000,124505)
```

initiates task 2000 five seconds after 12:45 p.m.

The system clock must be set if it is to reflect the correct time-of-day. To do this, use the RTE operator command TM after loading the RTE Operating System.

To initiate a task after a delay, use the START routine. For example:

```
   100 CALL START(2000,15)
```

executes task 2000 fifteen seconds after statement 100 is executed. This form of scheduling allows you to schedule a task repetitively. For example:

```
   100   CALL START(2000,15)        Schedule task 2000 in 15 seconds.
   998   WAIT (500)                 Execute idle-loop.
   999   GOTO 998
   2000   CALL START(2000,10)       Schedule task 2000 again in 10 seconds.
   2010   PRINT "TASK 2000 AT 10 SECOND INTERVALS"  Execute task 2000.
   2020   RETURN
   3000   END
>RUN
TASK 2000 AT 10 SECOND INTERVALS
       •
       •
```

Lines 998 and 999 comprise an *idle-loop* and keep the program in execution mode indefinitely. When no task is executing, the program continuously cycles waiting for something to happen. All real-time programs should have some type of idle-loop. The idle-loop may be a useful statement such as a PRINT to continuously log the results of your calculations.

To terminate this example, press any key on the system console or your terminal. The RTE-M System responds by printing the system prompt character. Type BR followed by a comma and the BASIC Interpreter program name. Then press RETURN. The example program will be aborted and the Interpreter returned to Command mode. For information about the BASIC Interpreter program name, see the RTE-M System Generation manual, Section V.

One common error occurs when a task reschedules itself every few seconds. For example, if a task reschedules itself every two seconds and the START statement is at the end of the task, the delay will be two seconds plus the amount of time necessary to execute the task statements which precede the START statement and the time required for all interim processing by the operating system. Therefore it is recommended that the statement used to reschedule the task be the first in the task in order to avoid the cumulative effect of the time delays.

If you give a series of consecutive task initiation commands, you should not specify a delay of zero (immediate start) for each one. The first task encountered must run to completion before another program statement can be executed. Subsequent tasks are delayed by the amount of time required to initiate preceding ones. To remedy this situation it is recommended that you schedule each task with a one second delay, thus allowing enough time for all of the initiating statements to be executed before any of the tasks are executed.

To initiate a task in response to an external event, the task must be associated with a trap number. For example:

```
100 TRAP 5 GOSUB 2000
```

associates task 2000 with trap number 5. Once the trap number is associated with the task, various events can refer to the trap number by using HP 6940 SENSE calls (for events sensed by the HP 6940) or a TTYS call (for an auxiliary teleprinter event).

BASIC senses when an external event interrupt occurs, signals that the event has occurred, and records significant information about the event. The recording of an event takes place concurrently with the execution of a BASIC language statement. When statement execution completes, the BASIC Scheduler determines whether a task has been scheduled, compares its priority with the task currently executing and decides whether or not to suspend the current task in favor of the new one.

10-3. PRIORITIES

Since a program may contain up to 16 tasks, more than one task may try to execute at the same time. To resolve these conflicts you may assign a priority number to each task. Priorities are established to provide some delineation between actions which must occur immediately and actions which have a more relaxed requirement. A task priority may be from 1 (the highest) to 99 (the lowest). A task of higher priority can interrupt the processing of a lower priority task if the interrupt scheduling the higher priority task occurs while the lower priority task is executing.

The following statement sets the priority of task 2000 to 50:

```
110 CALL SETP(2000,50)
```

Any task whose priority has not been specified is given a priority of 99.

10-4. RESPONSE TIME

To determine how often and how quickly an event interrupt will be processed, you must consider the length of time required to complete execution of the BASIC statement being processed when the interrupt occurs. The amount of time required to process BASIC statements varies widely; a range of 0.5 to 3 milliseconds (ms) is typical. If it takes 3 ms to process a particular BASIC statement, and the event occurs after 1 ms has elapsed, then obviously the event will be noted but task execution will not begin for at least 2 ms. Since you do not know which statement will be executing when the interrupt occurs, the statement in your program with the longest execution time determines the maximum response time required to service an interrupt.

If you are doing real-time processing, it is recommended that you avoid statements which require operator intervention such as PAUSE, INPUT, and READ# *lu*. Interrupts may be lost while the program waits for a response from the operator.

Interrupt processing may also be delayed indefinitely or interrupts ignored as a result of the operator suspending the BASIC program with the RTE SUSPEND command or the operator entering RTE commands while BASIC is outputting to the system console or your terminal.

10-5. THE BASIC SCHEDULER

The BASIC Scheduler keeps track of tasks and tells the Interpreter when to initiate execution of each task. The Scheduler maintains information passed to it by the task scheduling statements and uses it to make decisions concerning the tasks. The information includes the following:

- line number of the first statement in the task.
- priority of the task.
- trap number associated with the task.
- whether task is enabled or disabled.
- the current state of the task.

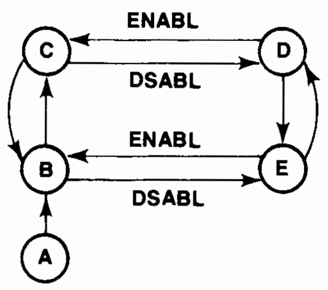
The BASIC Scheduler considers a task to be in one of five states at all times. For purposes of discussion, these states will be labeled A, B, C, D, and E. Refer to Figure 10-1.

When the BASIC Interpreter completes execution of a line of BASIC code, it transfers control to the Scheduler. The Scheduler examines the real-time clock and puts any time scheduled tasks that are ready for execution into State C (pending) or State D (pending/disabled). (Event scheduled tasks go to State C or D immediately upon occurrence of the event.) The Scheduler then determines which is the highest priority task in State C. If the priority of this task is higher than the currently executing task, the Scheduler puts it in State B (dormant) and tells the Interpreter to suspend the currently executing task and begin executing the highest priority pending task.

The BASIC Interpreter suspends the currently executing task and stores the line number of the next statement to be executed. It then stores the priority of the new task and begins executing the first line of that task. Task execution, once initiated, is independent of the Scheduler. The task can be interrupted but not disabled.

The Interpreter maintains the priority of the currently executing task which is determined when execution is initiated and does not change. It is independent of the priority kept by the Scheduler. Each time the Interpreter transfers control to the Scheduler after executing a statement, the priority is made available so the Scheduler can decide whether or not to suspend the currently executing task and initiate a new one. The Scheduler cannot change the priority of the currently executing task.

Once the Scheduler initiates a task, the execution of that task is controlled by the Interpreter. The Scheduler does not know which task is executing and is only concerned with which task should be initiated next. Thus, while a task is executing, it is possible that the same task may be initiated a second time. If its priority has been raised, the second initiation may interrupt the first execution of the task since the first retains the original priority.



State A — Undefined

A statement referencing the task has not yet been executed.

State B — Dormant

A statement referencing the task has been executed (the BASIC Interpreter has information about the task) but neither an external event nor the real-time clock has indicated that the task should be executed.

State C — Pending

An external event or the real-time clock indicate the task should be executed but execution has not yet begun. The task is actively vying for the computer resources. Execution is delayed until:

- the BASIC Interpreter completes execution of the current program statement, or
- a higher priority task completes execution.

State D — Pending/Disabled

A CALL DSABL has disabled the task which would otherwise be in State C (pending). The task is no longer vying for resources but once it is enabled, it will go directly to State C. A disabled task will not be initiated.

State E — Dormant/Disabled

A CALL DSABL has disabled the task which would otherwise be in State B (dormant). Once enabled, it will go directly to State B. If an interrupt indicates the task should be executed, it will go to State D (pending/disabled).

Figure 10-1. Task State Definitions

10-6. DSABL

The DSABL routine disables a specified task.

Format

CALL DSABL(*statement number label*) [FAIL: *statement*]

Parameter

statement number label	first statement number of the task to be disabled. If zero, all tasks are disabled. If negative, the task is removed from the task table (which controls the task), and any previous scheduling is nullified.
FAIL: *statement*	optional error return statement.

The DSABL routine tells the BASIC Scheduler not to initiate a task or tasks. If the argument is positive, the task beginning with that statement is placed in State D or E as appropriate. If it is 0, all tasks are placed in State D or E. If the argument is negative, the task beginning with the statement number equal to the absolute value of the argument is placed in State A. Any task scheduling statement referring to a disabled task enables it (puts it in State B or C). If the task was in State A, any previous priority or trap number is lost.

Example

```
350 CALL DSABL(2000)
```

Disables task 2000.

10-7. ENABL

The ENABL routine allows the scheduling of a previously disabled task.

Format

CALL ENABL(*statement number label*) [FAIL: *statement*]

Parameter

statement number label	first statement number of task to be enabled. If 0, all tasks are enabled.
FAIL: *statement*	optional error return statement.

A call to ENABL allows the initiation of a task which has been turned off previously by DSABL. A positive argument transfers the task from State D or E to State B or C. A zero argument transfers all tasks to State B or C. If the argument is negative, the SCHED-4 error message is printed (see paragraph 10-16) and the ERRCD flag is set to 4. It may be interrogated with IERR.

Note that if a task has been disabled by a DSABL call, it may also be enabled by any other call (except DSABL).

You may use DSABL and ENABL to prevent a task from being interrupted. For example:

`100 CALL START(1000,5)`	*Initiate task 1000 in 5 seconds.*
`•` `•` `1000 CALL DSABL(0)` `•` `•`	*Disable all other tasks. Task 1000 is initiated and then this statement is executed, it will not be disabled. Processing proceeds uninterrupted to 1100.*
`1100 CALL ENABL(0)`	*Enable all tasks.*

10-8. SETP

The SETP routine sets the priority of a task.

Format

CALL SETP(*statement number label, priority*) [FAIL: *statement*]

Parameters

statement number label	first statement in the task to have priority set.
priority	a number from 1 to 99. 1 is the highest priority and 99 is the lowest.
FAIL: *statement*	optional error return statement.

The priority of a task is used to resolve scheduling conflicts. Tasks with lower numbered priorities are selected for execution before tasks with higher numbered priorities. If no SETP call is made for a task its priority is 99.

Example

`500 CALL SETP(1500,45)`	*Task 1500 is given priority 45.*

10-9. START

The START routine schedules a task for processing after a specified delay.

Format

CALL START(*statement number label, sec*) [FAIL: *statement*]

Parameters

statement number label	first statement number of task to be initiated.
sec	number of seconds until execution is initiated.
FAIL: *statement*	optional error return statement.

If you CALL START with *sec* = 0 it is the same as using GOSUB, the task executes immediately and runs to completion. All subsequent tasks will be delayed by the execution time of this task unless they have a higher priority.

Example

```
655 CALL START(3300,35)
```

Task 3300 will be scheduled in 35 seconds.

10-10. TIME

The TIME routine returns the time according to the system real-time clock.

Format

CALL TIME(*time*)

Parameter

time	a variable equal to the time-of-day to the nearest tenth of a second. *time* is expressed as the number of seconds past midnight.

The following program converts the *time* parameter to hours, minutes, and seconds in the format *hh:mm:ss* and prints the converted result every five seconds on LU 17.

```
>LIST
  10   LET L=17
  20   CALL START(100,6)
  29   WAIT (100)
  30   GOTO 29
  100   CALL START(100,5)
  110   CALL TIME(T1)
  120   LET S1=INT(T1/60)
  130   LET S=INT(T1-S1*60)
  140   LET H=INT(S1/60)
  150   LET M=INT(S1-H*60)
  160   PRINT #L;H;TAB(2);":";M;TAB(5);":";S
  170   RETURN
  180   END
>RUN
9     :49    :3
9     :49    :8
9     :49    :13
9     :49    :18
9     :49    :23
9     :49    :28
```

10-11. TRAP STATEMENT

The TRAP* statement associates a trap number with a task which then may be associated with a hardware interrupt.

Format

TRAP *trapn* GOSUB *statement number label*

Parameters

trapn	trap number, a constant between 1 and 16 inclusive.
statement number label	first statement number of the associated task.

The trap number is a parameter in the HP 6940 SENSE routine, auxiliary teleprinter TTYS routine and the HP-IB SRQSN routine. For example:

CALL SENSE(*chan,nbit,bit,trapn*)
CALL TTYS(*lu,trapn*)
CALL SRQSN(*lu,trapn*)

When an interrupt to either of these routines occurs, the task associated with the *trapn* number is executed and the task is run. The TRAP association statement must already have been executed.

Only one trap number may be associated with each task and vice versa. Any attempt to associate more than one trap number to a statement number causes a SCHED-3 error (see paragraph 10-16), and the ERRCD flag to be set to 3. You may interrogate the ERRCD flag with IERR.

There are two methods of changing the association between a trap number and a task. Assume an association has been made as follows:

```
750 TRAP 5 GOSUB 1000
```

The first method is to simply assign a new task statement number as follows:

```
100 TRAP 5 GOSUB 2000
```

This forces the old task (1000) into State A (undefined, see figure 10-1), and nullifies any interrupts that have occurred to trap number 5. All future interrupts to trap number 5 will be transferred to statement 2000.

The second method is to use a negative statement number as follows:

```
100 TRAP 5 GOSUB -2000
```

The minus sign indicates you want to save any interrupts that have occurred to trap number 5. These interrupts will be transferred to statement 2000. The task at statement 1000 is forced to State A. All future interrupts to trap number 5 will be transferred to statement 2000.

*Within the Multiple Terminal Monitor environment, only one copy of the BASIC Interpreter can utilize TRAP and task scheduling calls at one time.

If the error TRAP-1 is printed, the trap number is negative, the task was not found at syntax time, or the GOSUB part of the statement is missing.

Examples

```
1ØØ TRAP 3 GOSUB 1ØØØ
11Ø TRAP 3 GOSUB 2ØØØ
```

After executing statement 110, trap number 3 is associated with task 2000 only.

Do not change two values at once, you will get ambiguous results.

```
1ØØ TRAP 3 GOSUB 1ØØØ
11Ø TRAP 4 GOSUB 2ØØØ
12Ø TRAP 3 GOSUB 2ØØØ
```

Statement 120 causes error SCHED-3 (see paragraph 10-16) since task 2000 is associated with two trap numbers.

```
5   TRAP 7 GOSUB 17Ø
1Ø  SENSE(5,4,1,7)
•
•
•
16Ø GOTO 16Ø
17Ø CALL WRBIT(2,4,1)
18Ø PRINT "RELAY CLOSED"
19Ø RETURN
```

Trap 7 transfers to statement 170.

A contact closure on bit 4 of channel 5 on a HP 6940 event sense card traps to statement 170.

This statement writes a bit on a channel.

A message is printed and control returns to the statement following the one completed before the interruption.

```
4Ø TRAP 7 GOSUB 96Ø
5Ø SENSE(5,I,J,7)
6Ø CALL WRBIT(2,4,Ø)
•
•
•
96Ø PRINT "RELAY CLOSED"
97Ø TRAP 7 GOSUB 1ØØØ
98Ø CALL WRBIT(2,4,1)
99Ø RETURN
1ØØØ CALL WRBIT(2,4,Ø)
1Ø1Ø PRINT "RELAY OPEN"
1Ø2Ø STOP
```

Set trap 7 to task 960.

First time SENSE interrupt occurs it traps to statement 960.

Task prints message.

Changes trap 7 so it is associated with task 1000.

The next time statement 50 is executed, the interrupt will trap to statement 1000.

10-12. TRNON

The TRNON routine executes a task at a specified time.

Format

CALL TRNON(*statement number label,time*) [FAIL: *statement*]

Parameters

statement number label	first statement of task to be initiated at specified time.
time	variable containing six digit number equal to *hhmmss*, the time in hours, minutes, and seconds. Hours must be based on a 24 hour clock.
FAIL: *statement*	optional error return statement.

The call to TRNON starts a task when the real-time clock equals the *time* parameter.

Example

```
100  LET  T=120015                  Initialize T to 12 p.m. and 15 seconds.
110  LET  I=5                       Set increment = 5 seconds.
120  CALL  TRNON(1000,T)            Schedule Task 1000 at 12 p.m. 15 sec.
999  GOTO  999         Idle-loop.
1000  LET  T=T+I       Increment T by 5 seconds.          If sec = 60,
1010  IF  T-INT(T/100)*100>=60  LET  T=T+40               increment by 40 (minute by 1).
1020  IF  T-INT(T/10000)*10000>=6000  LET  T=T+4000
1030  IF  T>=240000  LET  T=T-240000                      If min = 60, increment hr by 1
1040  CALL  TRNON(1000,T)                                 (min + sec by 4000).
•                                   Reschedule task.      If hr = 24, reset to 0 hours.
•
•                                   Execute task code.
1900  RETURN                        End of task.
```

10-13. TTYS

The TTYS routine allows an auxiliary teleprinter to interrupt the BASIC system.

Format

CALL TTYS(*lu*,*trapn*)

Parameters

lu	logical unit number of the teleprinter.
trapn	trap number associated with a TRAP statement

A user at an auxiliary teleprinter can interrupt BASIC by pressing any key and the trap associated with the logical unit number of the teleprinter will be executed. The routine does not service LU 1, the system console. If the error TTY-1 is printed, the logical unit number is less than 7. (LU 1 through 6 are reserved for standard devices.)

Example

```
10 TRAP 5 GOSUB 100
20 CALL TTYS(10,5)
29 WAIT(100)
30 GOTO 29
100 TIME(T)
110 PRINT#10;T
120 RETURN
```

Associate trap number 5 and task 100.

Define trap to be executed when interrupt occurs on LU 10. Insert idle-loop.

When a key closure generates an interrupt on LU 10, task 100 prints the number of seconds past midnight.

10-14. PROGRAM EXAMPLE

Figure 10-3 represents the structure of the working program shown in figure 10-4. The routine shown in figure 10-2 generates the data required by the program in figure 10-4.

Line 999 in figure 10-4 defines an idle-loop. In order to get some idea of how much time is spent in looping, a counter can be installed as part of the loop. Different variables are used for each task since one task may destroy the contents of variables used by another task. When a variable is to be shared by more than one task, you must analyze the implications of interrupts breaking into the statement sequence.

```
4 LET L4=4                                                          27.8551      10.6984
5 LET L6=6                                                          25.2468      9.84983
10 LET C1=20                                                        26.9359      4.74023
20 LET C2=1                                                         32.7251      6.6409
30 LET C3=10                                                        32.4648      8.21229
40 LET C4=3                                                         26.0227      5.04531
50 LET C5=5                                                         23.1503      7.74416
60 LET C6=.1                                                        23.0191      10.4607
70 LET M1=36                                                        12.6957      13.2719
100 FOR I=1 TO 100                                                  12.1831      9.58012
110 LET X=C6*I                                                      8.38165      19.0826
120 GOSUB 1000                                                      3.45362      18.99
130 NEXT I                                                          -.724108     24.6351
140 PRINT# L4;999                                                   -1.18706     22.219
150 STOP                                                            -5.77463     29.8881
1000 LET Y=C1*COS(C2*X)+C3*SIN(C4*X)+C5-2*C5*RND(X)                 -7.25732     29.5723
1005 PRINT# L4;Y                                                    -10.3784     28.426
1010 LET P=Y+M1                                                     -8.12697     21.0262
1020 IF P<M1 GOTO 1200                                              -15.2137     19.6844
1030 IF P>M1 GOTO 1300                                              -12.8658     16.6032
1040 PRINT# L6;TAB(M1);"X"                                          -6.25166     14.049
1050 RETURN                                                         -5.85369     16.0063
1200 PRINT# L6;TAB(P);"X";                                          -8.8297      6.51914
1210 PRINT# L6;TAB(M1);"I"                                          -4.75087     4.82018
1220 RETURN                                                         -7.68372     .951558
1300 PRINT# L6;TAB(M1);"I";                                         -6.94039     -6.74977
1310 PRINT# L6;TAB(P);"X"                                           -12.7432     -7.43395
1390 RETURN                                                         -13.3699     -9.90004
1400 END                                                            -16.9722     -13.6216
                                                                    -18.1321     -15.5273
                                                                    -20.8142     -11.5115
                                                                    -17.0846     -14.503
                                                                    -22.8928     -9.28221
                                                                    -24.363      -8.73984
                                                                    -23.5851     -8.01055
                                                                    -26.8436     -2.87477
                                                                    -21.9709     -2.86945
                                                                    -28.5017     -4.23474
                                                                    -22.5021     -9.50069
                                                                    -14.2186     -8.39849
                                                                    -10.5363     -16.9804
                                                                    -10.8942     -14.0962
                                                                    -.624686     -18.1544
                                                                    2.30019      -19.4267
                                                                    6.92991      -23.0569
                                                                    8.10668      -26.4572
                                                                    12.2503      -26.6669
                                                                    13.2217      -28.2566
                                                                    15.5333      -23.6533
                                                                    10.9137      999
                                                                    8.84168
```

NOTE: This routine generates the data at the right. The generated data is then fed into the following routine which demonstrates the scheduling capabilities of BASIC.

Figure 10-2. Task Scheduling Program Example (Part 1)

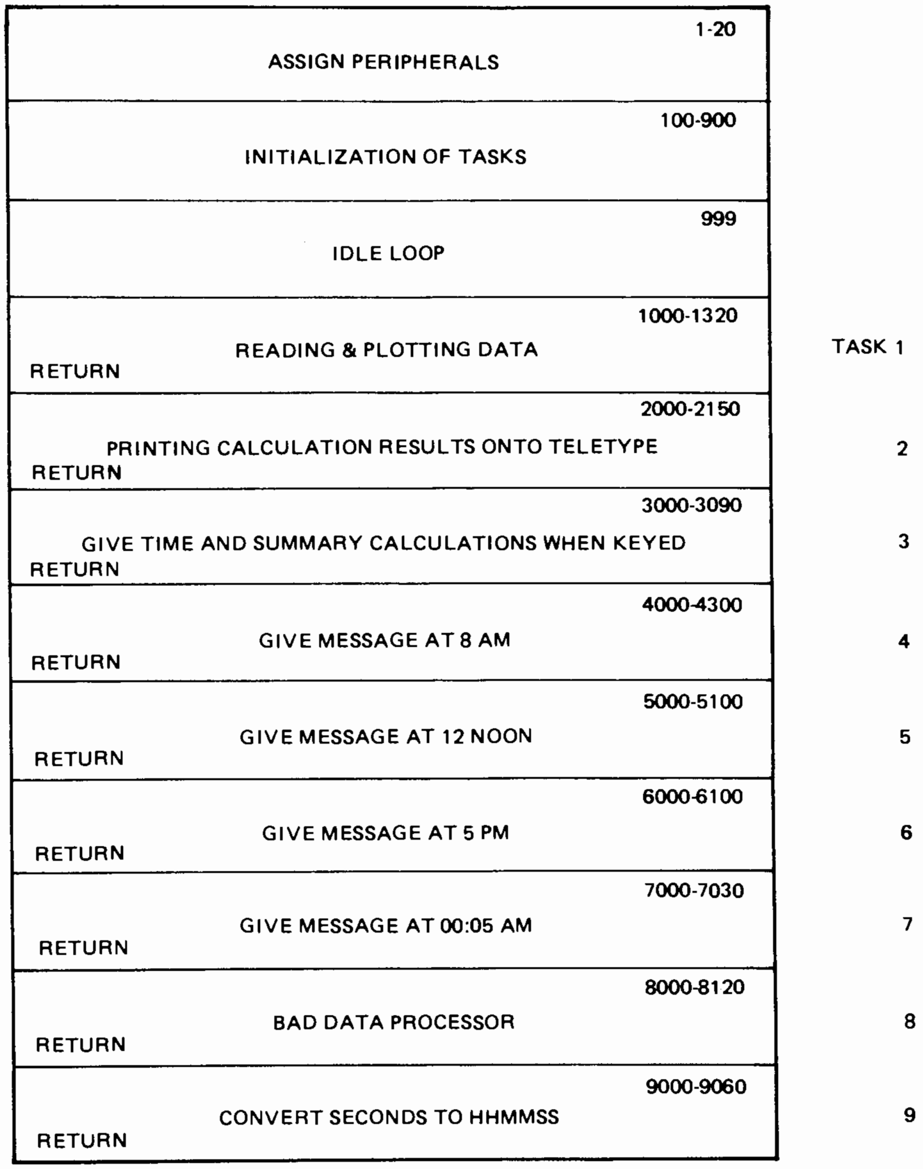


Figure 10-3. Structure of Program Example in Figure 10-4.

```
1 LET L1=1
2 LET L2=11
3 LET L3=11
4 LET L4=4
5 LET L5=5
6 LET L6=6
10 LET L0=11
20 LET D=100
```

Initialize variables which assign outputs to specific devices.

```
100 REM******** SET UP TASK 1000 ****
110 LET D1=1
120 LET A1=0
140 LET N1=0
160 SETP(1000,50)
170 START(1000,D1)
```

Lines 100 through 900 schedule tasks 1000 through 7000.

Set priority = 50.

Tell Scheduler to initiate task 1000 D1 seconds from now.

Since D1=1, task 1000 executes 1 second after statement 170 executes. This allows time for all other tasks to be initiated.

```
200 REM******** SET UP TASK 2000 ****
210 LET S2=0
220 LET N2=0
230 LET D2=10
240 SETP(2000,70)
250 START(2000,D2)
```

Execute task 2000 in 10 seconds (D2=10.)

```
300 REM********SET UP TASK 3000 ****
310 SETP(3000,5)
320 TRAP 1 GOSUB 3000
330 TTYS(11,1)
```

Set priority to 5.

Associate trap 1 with task 3000.

Associate LU 11 with trap 1. Pressing any key on LU 11 interrupts BASIC and starts task 3000.

```
400 REM********SET UP TASK 4000 ****
420 SETP(4000,99)
430 TRNON(4000,80000)
```

Start task 4000 at 8 a.m.

```
500 REM********SET UP TASK 5000 ****
510 SETP(5000,60)
520 TRNON(5000,120000)
```

Start task 5000 at 12 noon.

```
600 REM********SET UP TASK 6000 ****
610 SETP(6000,1)
620 TRNON(6000,170000)
```

Start task 6000 at 5 p.m.

```
700 REM********SET UP TASK 7000 ****
710 TRNON(7000,105)
```

Start task 7000 at 1 min. 5 sec. after midnight.

```
900 REM********IDLE LOOP ***********
999 GOTO 999
```

Forces program to continue executing while waiting for tasks to be scheduled. Control returns here if no task is scheduled.

```
1000 REM
1002 REM ** TASK 1000 ** READ DATA AND PLOT IT*
1004 REM
1010 START(1000,D1)
1020 READ# L5;X1
1030 IF X1>36 OR X1<-36 GOTO 8000
1040 LET A1=A1+X1
1050 LET N1=N1+1
1060 LET P1=36+X1
1070 IF P1>36 GOTO 1200
1080 IF P1<36 GOTO 1300
1090 PRINT# L6;TAB(36);"X"
1100 RETURN
1200 PRINT# L6;TAB(36);"I";
1210 PRINT #L6;TAB(P1);"X"
1220 RETURN
1300 PRINT# L6;TAB(P1);"X";
1310 PRINT# L6;TAB(36);"I"
1320 RETURN
```

Note that task 1000 and 2000 will try to execute simultaneously. Task 1000 has a higher priority than task 2000 so will be scheduled first or may interrupt task 1000 which must wait.

The tasks defined in lines 1000 to 9060 are identical to subroutines.

```
2000 REM
2002 REM ** TASK 2000 ** DATA COMPRESSION ROUTINE*
2004 REM
2010 START(2000,D2)
2015 GOSUB 9000
```

Figure 10-4. Task Scheduling Program Example (Part 2)

```
2020 LET A2=A1/N1
2030 LET S2=S2+A1
2040 LET N2=N2+N1
2050 LET B2=A2/N2
2060 PRINT# L4;A2,N1,B2,N2,T
2070 LET A1=0
2080 LET N1=0
2100 PRINT#L1
2110 PRINT# L1;"AT TIME ";T;TAB(0);
2120 PRINT# L1;" THE AVERAGE IS ";B2;TAB(0);
2130 PRINT# L1;" FROM ";N2;TAB(0);" DATA POINTS."
2140 PRINT# L1;"THE AVERAGE FOR THE LAST PERIOD WAS ";A2
2145 PRINT# L1
2150 RETURN
3000 REM
3002 REM ** TASK 3000 ** EVENT SCHEDULED TASK **
3004 REM
3010 GOSUB 9000
3015 PRINT# L3
3020 PRINT# L3;"THE CURRENT TIME IS ";T;TAB(0);
3025 PRINT# L3;".  SO FAR WE HAVE"
3040 LET N3=N2+N1
3050 LET A3=(S2+A1)/N3
3070 PRINT# L3;N3;TAB(0);" DATA POINTS WITH AN AVERAGE OF ";A3
3080 PRINT# L3
3090 RETURN
4000 REM
4002 REM ** TASK 4000 **
4004 REM
4010 FOR I4=1 TO 8
4020 PRINT# L2;""
4030 WAIT(D)
4040 NEXT I4
4100 PRINT# L2;"GOOD MORNING "
4200 PRINT#L2;"I HOPE YOU HAD A GOOD NIGHTS REST "
4300 RETURN
5000 REM
5002 REM ** TASK 5000 **
5004 REM
5010 FOR I5=1 TO 12
5020 PRINT# L2;""
5030 WAIT(D)
5040 NEXT I5
5050 PRINT# L2;"TIME FOR LUNCH, LET'S EAT."
5100 RETURN
6000 REM
6002 REM ** TASK 6000 **
6004 REM
6010 FOR I6= 1 TO 5
6020 PRINT# L2;""
6030 WAIT(D)
6040 NEXT I6
6050 PRINT# L2;"TIME TO GO HOME, SEE YOU TOMORROW."
6100 RETURN
7000 REM
7002 REM ** TASK 7000 **
7004 REM
7010 PRINT# L2;"EITHER YOU ARE WORKING LATE, OR";
7020 PRINT# L2;" YOU FORGOT TO SET THE TIME"
7030 RETURN
```

Figure 10-4. Task Scheduling Program Example (Part 2) (Continued)

```
8000 REM
8002 REM ** THIS ROUTINE STOPS WHEN OUT OF DATA **
8004 REM
8010 PRINT "EITHER YOU HAVE RUN OUT OF DATA, OR YOU HAVE";
8020 PRINT " READ A BAD DATA POINT."
8120 STOP
9000 REM ** SUBROUTINE TO CONVERT TIME **
9005 TIME(T9)
9010 LET S9=INT(T9/60)
9020 LET S=T9-S9*60
9030 LET H=INT(S9/60)
9040 LET M=S9-H*60
9050 LET T=INT((H*100+M)*100+S)
9060 RETURN
9999 END
```

Figure 10-4. Task Scheduling Program Example (Part 2) (Continued)

10-15. TABLE PREPARATION

In order to use the task scheduling subroutines, you must add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in the HP 92064A RTE-M System Generation manual. A list of the RTMTG commands required to add the subroutines described in this section is given there.

10-16. ERROR MESSAGES

The following errors may result from execution of the task scheduling routines:

ERROR SCHED-2 IN LINE *nnnn*	Task table overflow.
ERROR SCHED-3 IN LINE *nnnn*	Impossible to resolve combination in TRAP statement.
ERROR SCHED-4 IN LINE *nnnn*	Attempt to ENABL or DSABL non-exixtent entry.
ERROR SCHED-5 IN LINE *nnnn*	Time schedule table overflow.
ERROR SCHED-6 IN LINE *nnnn*	Time to execute scheduled task, but its entry has been deleted from the Task Table.

In all cases the ERRCD flag is set to the error number. You may use the FAIL: option and IERR to interrogate the flag. See paragraphs 6-3 and 6-4.

BIT MANIPULATION OPERATIONS

SECTION XI

BASIC performs all arithmetic operation in 32-bit floating point format (i.e., two 16-bit computer words). In instrument-related systems it is frequently necessary to manipulate the internal floating point number as though it were a 16-bit integer. This capability is especially important when the BASIC program communicates with instruments that require special bit patterns as input and/or output parameters. The following integer bit manipulation routines are designed to allow the BASIC programmer to perform inclusive OR, exclusive OR, NOT, AND, shift, set bit, clear bit, and test bit operations. These functions may be incorporated in the BASIC system at generation time by placing the proper name, entry point and parameter conversion in the Branch and Mnemonic table (see the HP 92064A RTE-M System Generation manual).

11-1. BIT MANIPULATION WORD FORMAT

Each word within the computer can be thought of as a 16-bit shift register. The bits are numbered from right to left as shown in Figure 11-1. Each set of three bits is weighted 4-2-1. For example, a bit pattern of 101 has an octal value of 5.

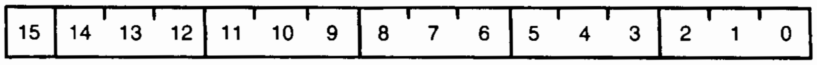


Figure 11-1. 16-Bit Word

For the purposes of bit manipulation, there are no sign bits or any significance other than a positional relationship.

In all cases of bit manipulation functions the operations are performed bit-by-bit. There is no carry from one set of bits to adjacent bits. Thus, in an OR function, for example, there is no carry from column-to-column.

11-2. AND

The AND function logically multiplies two words bit-by-bit.

Format

A = AND (*arg1, arg2*)

Parameters

arg1	first value to be ANDed.
arg2	second value to be ANDed.
result A	returned value of the operation.

The AND statement requires that both of the values be "1" in order to have the product be "1". Refer to the following truth table:

arg1	0011
arg2	0101
result	0001

Example

```
10 X = ISETC("762")
20 Y = ISETC("543")
30 CALL AND(X,Y)
40 PRINT OCT(Z)
50 END
```

ISETC function described in paragraph 11-9.

Results

```
X   =762   =111-110-010
Y   =543   =101-100-011
Z   =542   =101-100-010
```

11-3. IBCLR (Bit Clear)

This function sets selected bit positions to zero. The position to be set to zero is determined by the number given in the *bit posit* parameter where the bits are referenced right to left starting with zero. Refer to Figure 11-1 for the bit positions. Note that only one bit can be set to "0" at a time.

Format

A = IBCLR (*value, bit posit*)

Parameters

value	starting value.
bit posit	position of the bit to be cleared. The least significant bit is zero. If *bit posit* exceeds 15, the value of *result* is set to the starting value *(value).*
result A	returned value.

Example

```
10 X = ISETC("767")
20 Z = IBCLR(X,8)
30 PRINT OCT(Z)
40 END
```

Results

```
X   =767   =111-110-111
                ↓
Z   =367   =011-110-111
```

Bit position 8 is cleared (set to zero).

11-4. IBSET (Bit Set)

This function sets selected bit positions to "1". The position to be set to one is determined by the number given in the *bit posit* parameter where the bits are referenced right to left starting with zero. Refer to Figure 11-1 for the bit positions. Note that only one bit can be set to "1" at a time.

Format

A = IBSET (*value, bit posit*)

Parameters

value	starting value.
bit posit	position of the bit to be set. The least significant bit is zero. If *bit posit* exceeds 15, the value *result,* is set to the starting value (*value*).
result A	returned value.

Example

```
10 X = ISETC("452")
20 Z = IBSET(X,7)
30 PRINT OCT (Z)
90 END
```

Results

```
X   =452   =100-101-010
              ↓                    Bit position 7 is set to 1.
Z   =652   =110-101-010
```

11-5. IBTST (Bit Test)

This function tests a selected bit in a word. It is used to return the value of a certain bit within a word without disturbing the original word.

Format

A = IBTST (*value, test posit*)

Parameters

value	value to be tested.
test posit	the position of the bit is to be tested. The least significant bit is zero. If *test posit* is greater than 15 or less than zero, *result* is set to zero.
result A	value of the tested bit (either a "1" or "0").

Example

```
10 Z = IBTST(X,15)
20 PRINT OCT(Z)
30 END
```

X = the value to be tested.
15 = the left-most bit, or bit 15.
Z = a "1" or "0" depending on the value of bit 15 in word X.

11-6. IEOR

The IEOR function executes the modulo-two sum (Exclusive OR) between two words, bit-by-bit.

Format

A = IEOR (*arg1, arg2*)

Parameters

arg1	first value to be Exclusive OR'ed.
arg2	second value to be Exclusive OR'ed.
result A	returned value of the operation.

The exclusive OR operation is primarily used in compare operations. When performing the exclusive OR, the result will be "1" if one and only one of the bits is "1". Refer to the following truth table:

arg1	0011
arg2	0101
result	0110

Example

```
10 X = ISETC("564")
20 Y = ISETC("371")
30 Z = IEOR(X,Y)
40 PRINT OCT(Z)
50 END
```

Results

```
X  =564   =101-110-100
Y  =371   =011-111-001
Z  =615   =110-001-101
```

11-7. NOT

This function complements a word bit-by-bit. It causes the complement of a value to appear in the result. That is, a "1" in *value* becomes a "0" in *result* and vice versa.

Format

A = NOT (*value*)

Parameters

value	value to be complemented.
result A	returned value of the operation.

Example

```
10 CALL ISETC("1762",X)
20 CALL NOT(X,Z)
30 PRINT OCT(Z)
40 END
```

Results

```
X  =  1762   =0-000-001-111-110-010
Z  =176015   =1-111-110-000-001-101
```

11-8. OR

The OR function executes the logical sum (Inclusive OR) between two words bit-by-bit.

Format

A = OR (*arg1, arg2*)

Parameters

arg1	first value to be Inclusive OR'ed.
arg2	second value to be Inclusive OR'ed.
result A	returned value of the operation.

The Inclusive OR operation is primarily used in compare operations. When performing the Inclusive OR, the result is "1" if either of the bits in argument 1 or 2 is "1". Refer to the following truth table.

arg1	0011
arg2	0101
result	0111

Example

```
10 X = ISETC("461")
20 Y = ISETC("577")
30 Z = IOR(X,Y)
40 PRINT OCT(Z)
50 END
```

Results

```
X  =461  =100-110-001
Y  =577  =101-111-111
Z  =577  =101-111-111
```

11-9. ISETC (Set to Octal)

ISETC converts the octal number to its floating point equivalent. Therefore, whenever using the PRINT statement to print the number, use the OCT option. Otherwise, the decimal equivalent of the number is printed. For example, if you use the octal number 177777, and the PRINT statement without the OCT option, the decimal number −1 is printed.

Format

A = ISETC$\left(\begin{matrix}\textit{"octal numb"}\\ \textit{string variable}\end{matrix}\right)$

Parameters

octal numb	six-character or less octal number enclosed in quotes (" ") which is to be set into *variable*. Any characters other than 0 through 7 will cause an error message: ERROR -26 IN LINE *nnnn*.
string variable	variable containing a string of numeric characters.
variable A	parameter that receives the octal number.

Example

```
10 X = ISETC("27765")
20 PRINT OCT(X)
30 END
```

Results

X is printed as 27765.

11-10. ISHFT (Register Shift)

This function shifts the bit contents of a variable left or right a given number of positions. It shifts the entire word as though the word were a shift register. The bits shifted out of the word are lost. Replacement bits coming into the word are zeros.

Format

A = ISHFT (*value, shift numb*)

Parameters

value	argument to be shifted.
shift numb	direction of shift and number of positions.
numb <0	shift right "n" positions.
numb >0	shift left "n" positions.
numb =0	no shifting.
numb < − 15 *numb* > + 15	result is set to zero.
result A	returned value of the operation.

Example

```
10 X = ISETC("276")
20 Z = ISHFT(X,2)
30 PRINT OCT(Z)
40 END
```

Shifts all bits two positions left.

Results

```
X   = 276   =010-111-110
Z   =1370   =001-011-111-000
```

11-11. BRANCH AND MNEMONIC TABLE PREPARATION

In order to use the routines described in this section, you must enter the names of the functions and other information in the Branch and Mnemonic Tables. The procedure for doing this is described in the HP 92064A RTE-M System Generation manual with a list of the precise commands required.

MAGNETIC TAPE INPUT/OUTPUT

SECTION XII

The magnetic tape drive enhances BASIC's effectiveness by providing sorting, manipulation, storage, and retrieval capabilities. The magnetic tape drive allows the system to save information that is too voluminous for storing in memory, or on paper tape. There are two primary advantages to using magnetic tape over paper tape:

- The speed and capacity of a magnetic tape.
- The ability to rewind and reread material solely on command from the computer.

Data is written to the tape drive as a set of contiguous data called a *record*. Each tape record is a direct result of a command from the program to write a record onto a magnetic tape. Sets of consecutive records all associated with the same logical program and function are called *files*. It is possible to have more than one file on a single tape, and, except for the fact that they are on the same physical reel, they do not necessarily have to be related. At the end of each of the files, the program can write a special record called an end-of-file (EOF) which signifies the end of a file.

You should not mix READ and WRITE commands as you progress through a tape. Since records are not spaced a precise distance apart, extraneous data may be left on the tape if you write a record and then read the record which follows it. Always write onto a tape consecutively and then rewind and read it later. It is unacceptable to update a tape by replacing records.

12-1. MAGNETIC TAPE OPERATOR COMMANDS

There are several operator commands available that allow you to manipulate the tape drive from the system console. The command format consists of a control word and the logical unit number of the magnetic tape device.

Format

Command	Purpose
REWIND *lu number*	Rewind the magnetic tape all the way back to the beginning.
WEOF *lu number*	Write an end-of-file mark on the tape.
SKIPF *lu number*	Skip to the end of the current file and stop at the beginning of the next file on the magnetic tape.
BACKF *lu number*	Backspace past the previous file mark and stop.
lu number	The logical unit number associated with the magnetic tape device.

12-2. MAGNETIC TAPE CALLS

Descriptions of all magnetic tape calls available in BASIC are provided in the remainder of this section. Figure 12-2 contains a sample program which uses the routines described here.

12-3. MTTRT

The MTTRT routine writes a record onto a tape.

Format

CALL MTTRT(*lu,array,numb,eof,length*)

Parameters

lu	logical unit number of magnetic tape device.
array	first element of an array of data to be written.
numb	number of elements requested to be transferred from memory to the tape.
eof	dummy variable always set to zero on return.
length	actual number of elements transferred from memory to tape.

The number of elements requested to be transferred is always repeated in the *length* parameter since a tape WRITE always writes the specified number of variables. If there is a problem with writing on a bad portion of tape, the statement causes a skip over the bad portion of tape and rewrites on good tape.

Example

```
355 CALL MTTRT(13,A(1),100, E, N)
```
Writes 100 elements of array A on LU 13.

12-4. MTTRD

The MTTRD routine reads a record from a tape into an array.

Format

CALL MTTRD(*lu,array,numb,eof,length*)

Parameters

lu	logical unit number of the magnetic tape device.
array	first element of an array into which data is read.
numb	number of elements requested to be read from the tape.
eof	variable set to 1 if EOF is encountered during a READ operation, set to 0 otherwise.
length	actual number of elements transferred from the tape to memory. Useful when the record read is shorter than the value specified by *numb*.

The routine transfers up to *numb* elements. If the record length is smaller than *numb*, only the record is transferred. If the record is larger than *numb*, the remaining data in the record is lost.

The *length* parameter provides the actual number of elements transferred to the array. *Length* can never exceed *numb*; therefore, if *length* = *numb*, the record may have been too long.

After the last record in a file is read, the next READ returns an end-of-file indication (if there is one) in parameter *eof*.

Example

```
500 CALL MTTRD(12,B(1),200,E,N)
```
Read 200 elements into array B from LU 12.

12-5. MTTPT

The MTTPT routine positions the tape forward or backward a certain number of files and/or records.

Format

CALL MTTPT(*lu,fspace,rspace*)

Parameters

lu	logical unit number of the magnetic tape device.
fspace	number of files to skip. If positive, skips forward, if negative, skips backward.
rspace	number of records to skip. If positive skips forward, if negative, skips backward.

Forward positioning is accomplished by reading files or records until the number you request have been skipped or an EOF is read.

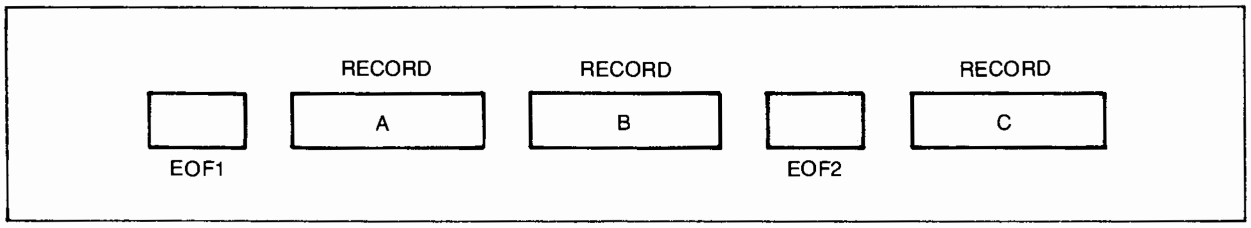


Figure 12-1. Record Positioning Example Using MTTPT

Examples (Refer to figure 12-1)

1. File position is immediately after record C, and *fspace* = −1 (backspace 1 file). File position moves to immediately after record B before EOF2.

2. File position is immediately after record C; *fspace* = −2 (backspace 2 files) and *rspace* = 1 (forward space 1 record). File position moves to immediately after EOF1 before record A.

3. File position is immediately after record C, and *rspace* = −3 (backspace 3 records). File position moves between record A and B.

12-6. MTTFS

The MTTFS routine writes an end-of-file or rewinds the tape.

Format

CALL MTTFS(*lu,func*)

Parameters

lu	logical unit number of the magnetic tape device.
func	specific tape drive control or function:
	0 = WRITE a 4 inch gap.
	1 = WRITE an EOF mark.
	2 = REWIND the tape but leave the tape, when finished, available for use again (at load point).
	3 = REWIND the tape and when the rewind is complete, make the tape device no longer available to the system (rewind and unload).

Examples

`435 CALL MTTFS(12,1)`	*WRITE an EOF on LU 12.*
`86Ø CALL MTTFS(11,2)`	*REWIND LU 11. Control returns to program immediately.*
`755 CALL MTTFS(11,3)`	*REWIND LU 11 and make tape device unavailable.*
`95Ø CALL MTTFS(1Ø,Ø)`	*Erase 4 inches of tape on LU 10.*

12-7. TAPE MANIPULATION ERRORS

The following message is given when tape errors occur:

ERROR MAGTP-*x* IN LINE *nnnn*

If *x* equals:

1	An illegal logical unit number has been specified for a tape.
2	An illegal request for tape manipulation has been made.
3	A WRITE request has been given but the write-ring (file-protect-ring) is not in place on the reel.

The line in the BASIC program where the error occurred is supplied as *nnnn*.

12-8. BRANCH AND MNEMONIC TABLE ENTRIES

During system generation, the routines described in this section must be entered in the Branch and Mnemonic Tables if the magnetic tape drive calls are to be used. The procedure for doing this is described in the HP 92064A RTE-M System Generation manual.

12-9. SAMPLE PROGRAM USING MAGNETIC TAPE

Figure 12-2 contains a sample program which demonstrates some of the previously described routines. The tape is rewound, file records are written on it, it is rewound again and then read until an end-of-file is encountered. The tape is then backspaced two records to allow the last data record to be reread.

```
>LIST
  10   REM - MAGNETIC TAPE EXAMPLE
  20   DIM V[100],W[100]
  22   FOR I=1 TO 100
  24   LET V(I)=I
  26   NEXT I
  30   REM - REWIND THE TAPE UNIT TO LOAD POINT
  40   CALL MTTFS(8,2)
  50   REM - WRITE FIVE RECORDS
  60     FOR I=1 TO 5
  70     CALL MTTRT(8,V[1],100,X,E)
  80     NEXT I
  90   REM - WRITE END-OF-FILE MARK
  100   CALL MTTFS(8,1)
  110   REM - REWIND THE TAPE UNIT TO LOAD POINT
  120   CALL MTTFS(8,2)
  130   REM - NOW READ ALL THE RECORDS
  140   CALL MTTRD(8,W[1],100,E,N)
  150   IF E=1 THEN 170
  160   GOTO 130
  170   REM - - BACKSPACE TWO RECORDS
  180   CALL MTTPT(8,0,-2)
  190   END
>
```

Figure 12-2. Tape Control Sample Program

BASIC CALLS TO EXTERNAL SUBROUTINES

SECTION XIII

The BASIC Subsystem requires Branch and Mnemonic Tables if your programs make calls to subroutines and functions external to BASIC. Further, for instrument subsystems such as the HP 2313 and HP 6940, BASIC requires an Instrument Table.

You create these tables using table generator programs. Once the tables are loaded into RTE-M, the BASIC Subsystem uses them for the transfer of program execution between the BASIC Interpreter and the external subroutine or function called.

In most instances, BASIC does not directly call an external subroutine. Instead, a BASIC program call is made via the Branch and Mnemonic Table to a BASIC callable subroutine which re-issues the call in a format acceptable to the external subroutine library.

For example, the HP 2313 subroutine AISQV (paragraph 13-8) is a BASIC callable subroutine. When a call is issued to this subroutine, AISQV reformats the call and passes it on to call the AISQW subroutine (FORTRAN callable subroutine) from the ISA FORTRAN Extension Package library.

13-1. BRANCH AND MNEMONIC TABLE

External subroutines and functions written in BASIC, FORTRAN, ALGOL, or HP Assembler language are defined in a Branch and Mnemonic Table. The RTE-M Table Generator (RTMTG) is used to create the Branch and Mnemonic Table. Detailed instructions for generating this table are given in Section V of the HP 92064A RTE-M System Generation Manual.

Once you generate this table, you may load it into memory as a relocatable module either during RTE-M System generation or on-line using the relocating loader, RTMLD.

13-2. INSTRUMENT TABLE

To support the HP 2313 and HP 6940 Subsystems with your RTE-M Operating System, you must include an Instrument Table during system generation. The Instrument Table contains all of the information about the HP 2313 and 6940 Subsystems required by the operating system.

Currently, you cannot execute the RTE Instrument Table Generator program in RTE-M. You must generate the Instrument Table on an RTE-II or RTE-III Operating System. The RTE-II or RTE-III Table Generator program produces a file containing the Instrument Table that you load into your RTE-M System. Before loading the Instrument Table file, be sure that the HP 2313 and HP 6940 cards are in the proper order as described in paragraph 13-17 and 13-31.

Instructions for executing the Instrument Table Generator program are given in the HP ISA FORTRAN Extension Package Reference Manual.

13-3. HP 2313/91000 DATA ACQUISITION SUBSYSTEM

The following paragraphs describe the BASIC calls used to communicate with the HP 2313/91000 Data Acquisition Subsystem. Information about generating a system containing the instrument (subsystem) subroutines is provided in paragraph 13-15.

13-4. MEASUREMENT OF ANALOG INPUT

The HP 2313 Subsystem is capable of measuring single-ended high-level inputs or differential high- or low-level inputs, with 12-bit resolution. The system can be equipped with a plug-in Programmable Pacer for timing measurements with 50 nanosecond precision. High-level input is ±10.24 volts full scale and low-level input is programmable in eight ranges from ±10 millivolts to ±800 millivolts full scale. Measurement ranges on the various low-level channels are preassigned in a table in memory so no distinction has to be made between high- and low-level channels when programming specific measurements.

The HP 91000 Plug-In 20 KHz Analog-to-Digital Interface Subsystem is capable of measuring 16 high-level single-ended or 8 high-level differential inputs. Each plug-in card constitutes a subsystem and can be programmed with the appropriate commands described in this section.

13-5. ANALOG OUTPUT

The HP 2313 Subsystem can also be used for analog outputs. A dual 12-bit digital-analog converter card, providing two analog outputs, can be installed in any of the working card spaces provided in the basic or expanded measurement subsystem. Analog output speeds can be timed precisely by the pacer option.

13-6. HP 2313 SUBSYSTEM SUBROUTINES

A series of subroutines are used to perform various 2313 operations. These subroutines are called in the same way as other BASIC subroutines:

CALL *name(parameters as required)*

Each call is listed in alphabetical order by name.

13-7. AIRDV (Random Scan)

The AIRDV subroutine reads analog input in a random manner.

Format

CALL AIRDV (*numb,achan,volt,error*)

Parameters

numb	number of channels to be read. If *numb* is negative, the readings are paced by the system pacer (see PACER call).
achan	array whose contents are channel numbers and whose positional relationship is the same as the voltage in *volt.* If you designate a channel number in *achan,* the corresponding voltage appears in the corresponding position in the *volt* array.
volt	first location of an array where the voltages to be read are placed.
error	variable set to one of the following values upon return from the routine: 0 = No error. 1 = Overload has occurred. The voltage available at the sensor multiplied by the gain you specified exceeds the voltage range of the analog-to-digital converter. 2 = Pace error. The subsystem was not ready when a pace pulse occurred. Normally this is caused by too rapid a pace rate or failing to turn off the pacer after a paced operation.

Example

Assume the channel number array A has been initialized to the channel numbers. The values in the V array correspond on a one-to-one basis to the channel numbers in the A array.

```
10 DIM V(75),A(75)
20 LET N=75
30 CALL AIRDV(N,A(1),V(1),E)
```

Reads 75 voltages into the V array.

13-8. AISQV (Sequential Scan)

The AISQV routine reads analog input sequentially.

Format

CALL AISQV (*numb,schan,volt,error*)

Parameters

numb	number of channels to be read. This, in conjunction with the starting address in *schan* defines the starting place and number of channels to be addressed. If *numb* is negative, *numb* readings are taken on the channel and are paced by the system pacer (see PACER call).
schan	starting channel number of the sequential scan (the first reading is taken on this channel). If *schan* is negative, *numb* readings are taken on channel *schan*.
volt	first location of an array where the voltages to be read are placed.
error	variable set to one of the following values upon return from the routine: 0 = No error. 1 = Overload has occurred. The voltage available at the sensor multiplied by the gain you specify exceeds the voltage range of the analog-to-digital converter. 2 = Pace Error. The subsystem was not ready when a pace pulse occurred. Normally this is caused by too rapid a pace rate or failing to turn off the pacer after a paced operation.

Example

```
100 LET N = 100
110 LET C = 50
120 DIM V(10,10)
130 CALL AISQV(N,C,V(1,1),E)
```

Read 100 voltages into the V array beginning with the 50th channel.

13-9. AOV (Digital to Analog Conversion)

The AOV routine converts digital information to analog voltage output.

Format

CALL AOV(*numb,achan,volt,error*)

Parameters

numb	the number of channels to be output. If *numb* is negative, the analog output is paced by the system pacer.
achan	array containing channel numbers corresponding to voltages in *volt* array.
volt	array of voltages to be output to the channels defined by *achan*.
error	variable set to one of the following values upon return from the routine: 0 = No error. 1 = Overload has occurred. The gain you specified is outside the range of the digital-to-analog converter. 2 = Pace error. The subsystem was not ready when a pace pulse occurred. Normally this is caused by too rapid a pace rate or failing to turn off the pacer after a paced operation.

Example

```
300 CALL AOV(25,C(1),V(1),E)
```

Convert data from channels in the C array to analog voltages and store in V array.

13-10. NORM

The NORM routine normalizes the subsystem, resets it to a home or known state.

Format

CALL NORM [(*unit*)]

Parameter

unit	HP 2313/91000 unit number. This number is defined as part of the Instrument Table. See paragraph 13-2. If the unit number is going to be supplied, the NORM subroutine must be modified in the Branch and Mnemonic Tables to accommodate the parameter. If not, it is assumed to be one.

Unless specifically excluded, all numbers are floating point numbers. Either the actual number or a reference to the number may be given.

Example

```
200 LET Z = 2                    Set subsystem unit equal to 2.
210 CALL NORM(Z)                 Normalize the HP 2313 Subsystem.
```

All DACs (Digital to Analog Converters) are set to 0. The HP 2313 pacer and the LAD (Last Address Detector) is turned off. The gain settings on the LLMPX channels are not changed.

13-11. PACER

The PACER routine sets the pace rate of the HP 2313 system. The HP 91000 Subsystem pacer is not controlled by this call.

Format

CALL PACER(*rate*,*mult*,*start*[,*unit*])

Parameters

rate	basic pacer rate in microseconds. Basic rate must be between 0 and 255 inclusive.
mult	rate multiplier. This parameter specifies one of eight ranges (0 - 7) which is a power of ten multiplier applied to the basic pace rate.

For example:

rate = 25 (25 microseconds)

mult = 3 results in a pace period of 25 milliseconds.

start external and internal start/stop control for the pacer.

Value	Change Period or Start/Stop	External Start/Stop
0	Immediately	Disable
1	Next pace pulse	Disable
2	Immediately	Enable
3	Next pace pulse	Enable

unit HP 2313 unit number. This number is defined as part of the instrument table. See paragraph 13-15. If *unit* is to be supplied, the PACER subroutine must be modified in the Branch and Mnemonic Tables.

If *start* is zero, a 10-millisecond delay is inserted by the PACER routine. This allows time for the analog input or output statement to be executed so that no pace error occurs.

Example

```
100 REM NORMALIZE SYSTEM
110 CALL NORM
200 REM RATE = 1*10↑3 MICROSECONDS
210 CALL PACER(1,3,0)
300 REM TAKE 20 READINGS FROM CHANNEL 5
310 CALL AISQV(20,-5,V(1),E)
400 REM TURN OFF PACER
410 CALL PACER(0,0,0)
```

Perform a paced reading on a single channel with a period of 1 millisecond.

13-12. RGAIN

The RGAIN routine tells you what the gain setting is on a particular channel.

Format

CALL RGAIN(*chan,gain*)

Parameters

chan	the channel number which must be greater than or equal to 1.
gain	location which upon return contains the discrete number 1, 12.5, 25, 50, 100, 125, 250, 500, or 1000 representing the setting of the gain amplifier on that particular channel.

The gain for any particular channel is initially set during system configuration. This call allows you to determine the present gain setting of any channel.

Example

```
400 CALL RGAIN(55,X)
```

Determine present gain on channel 55 and store it in X.

13-13. SGAIN

The SGAIN routine sets the gain for all channels in a group.

Format

CALL SGAIN(*chan,gain*)

Parameters

chan	LLMPX channel number which must be greater than or equal to 1.
gain	value of the gain to which you wish a specified channel to be set. It should be one of the discrete values: 12.5, 25, 50, 100, 125, 250, 500, or 1000. These are the only values available on the multiplexers. If the gain specified is not one of these, it will be set to the next lowest gain in relation to its absolute value. If it is < 12.5, 12.5 is used. For example, 5 or -5 will be set to 12.5, -999 will be set to 500, and 1001 will be set to 1000.

In multiple channel groups all of the channels in the group have the same gain, so changing the gain on a single channel sets all of the channels in the group to the new gain. This does not apply if the groups have been specified as one channel per group during configuration.

Example

```
500 CALL SGAIN(CH,12.5)
```

The channel CH is set to 12.5.

13-14. HP 2313/91000 SUBSYSTEM ERRORS

All error messages generated by the previous subroutines take the following form:

ERROR *name-numb* IN LINE *xx*

where *name* is the module name, *numb* is the error type, and *xx* is the line in which the error occurs.

The following is a list of HP 2313/91000 Subsystem errors (*name-numb* values):

HLMPX/LLMPX

ADC-1 Driver timeout

ADC-2 Parameter value outside defined range

ADC-3 Attempt to set gain on HLMPX channel or HP 91000 Subsystem.

DAC

AOV-1 Driver timeout

AOV-2 Addressed channel undefined.

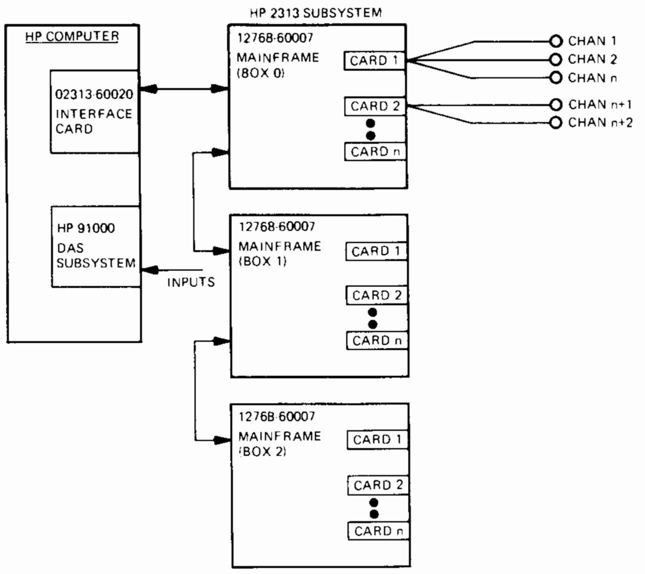
13-15. HP 2313/91000 TABLE PREPARATION

In order to use the HP 2313/91000 Subsystem subroutines you must provide information about the subsystem which is incorporated in the Instrument Table (see paragraph 13-2).

You must also add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in the HP 92064A RTE-M System Generation Manual. A list of the RTMTG commands required to add these subroutines is given there.

13-16. HP 2313 SUBSYSTEM CONCEPT

Figure 13-1 illustrates the concept of the HP 2313 Subsystem configuration. A subsystem is defined as the equipment occupying a computer I/O slot. Note that in the figure, the HP 2313 is one subsystem (including all three boxes) having one I/O slot number, and the HP 91000 DAS Card is another subsystem occupying an I/O slot number. The BASIC software treats the HP 91000 exactly like an HP 2313 Subsystem. The only difference is that there are no low-level multiplexers or Digital-to-Analog cards in the HP 91000.


HP 2313 SUBSYSTEM
HP COMPUTER
02313-60020 INTERFACE CARD
HP 91000 DAS SUBSYSTEM
INPUTS
12768-60007 MAINFRAME (BOX 0)
12768-60007 MAINFRAME (BOX 1)
12768-60007 MAINFRAME (BOX 2)
CARD 1
CARD 2
CARD n
CHAN 1
CHAN 2
CHAN n
CHAN n+1
CHAN n+2

1. The HP 2313 HLMPX and LLMPX input cards are divided as follows:

High-Level Multiplexer	16-Differential inputs
	32-Single ended inputs
Low-Level Multiplexer	16-Differential

2. All HP 2313 HLMPX and LLMPX cards are numbered sequentially for programming purposes.
3. The HP 91000 DAS has either 8 HLMPX differential inputs or 16 HLMPX single-ended inputs.

Figure 13-1. HP 2313 Subsystem Configuration

13-17. HP 2313 CARD CONFIGURATION

The configuration of the various types of cards in an HP 2313 Subsystem must follow the conventions given in the HP ISA FORTRAN Extension Package Reference Manual. If expansion is anticipated, configuration can include blank slots to accommodate future cards. The blank slots will then have channel numbers assigned to them, but cannot be accessed until the hardware is added.

13-18. HP 2313/91000 CHANNEL NUMBERING

As a result of physically placing all high-level multiplexer (HLMPX) and low-level multiplexer (LLMPX) cards in the required order, and then specifying how many of each type there are, all of the multiplexer channels will be assigned a number starting with 1 for the first specified channel and ending with *n* for the last. It is your responsibility to record the division line between channel numbers as shown in the following example.

CHANNEL NUMBERS	TYPE
1 2 : 32	HLMPX single-ended
33 34 : 48	HLMPX differential
49 50 : 64	Blank
65 66 : 72	LLMPX Gain = 25
73 74 75 76 77 78 79 80	LLMPX Gain = 100 1 channel groups

Detailed channel numbering information is given in the HP ISA FORTRAN Extension Package Reference Manual.

13-19. SETTING GAIN

Gain must be specified for low-level multiplexer channels. At system generation time a gain is assigned to a group of channels which is used until changed by the SGAIN command. Low-level channels can be configured into groups of 1 to N channels where N is the total number of low-level channels in a subsystem.

All of the channels in a group have the same gain, so if the gain of one channel in a group is changed by the SGAIN command, all channels in that group are changed to the specified gain. If programming the gain of each channel independently is required, then the groups would consist of only 1 channel. It might be noted that the execution of the SGAIN command does not have any direct effect on the hardware, but simply changes the group-gain entry in the configuration table. When a reading is taken, this table is read to determine the gain with which to set the corresponding multiplexer card, and the appropriate conversion factor to compute the actual terminal voltage. Therefore, the SGAIN command only has to be issued to change the table and does not have to be issued following a system normalize command or before each reading.

13-20. HP 6940 MULTIPROGRAMMER SUBSYSTEM

The HP 6940 Multiprogrammer is the master control unit for bidirectional data transfers (i.e., output data distribution/input data multiplexing). The HP 6940 can be used in a single-unit system employing from one to fifteen plug-in input/output cards, or in a multi-unit system consisting of up to 8 HP 6940 master units each with up to fifteen HP 6941A extender units. Each extender unit can also accommodate up to fifteen input/output cards. The digital I/O capabilities include direct or isolated digital input and interrupting event sense input, solid-state digital output, relay register output, stepping motor control, voltage and current DAC, programmable timers, and pulse counters.

13-21. HP 6940 SUBSYSTEM SUBROUTINES

A series of subroutines are used to operate the subsystem. They vary depending on the characteristics of the particular device. These subroutines are called in the same way as other BASIC subroutines:

CALL *name(parameters as required)*

Each call is listed in alphabetical order by name.

13-22. DAC

The DAC subroutine converts digital information to analog.

Format

CALL DAC(*chan,value*)

Parameters

chan	number of the analog output channel.
value	either the voltage or current (depending on the hardware) to be output by the DAC. (Current is in milliamperes.)

DAC causes the desired analog voltage or current value (based on *value*) to be output on the channel defined by *chan*.

Example

```
100 CALL DAC(1,10)
```

Outputs 10 volts on channel 1.

13-23. MPNRM

The MPNRM routine clears the event sense mode and erases the channel/bit to trap number correspondence. This call should be issued before any SENSE calls to insure there are no residual definitions from previous programs. This command also negates any previous SENSE calls.

Format

```
CALL MPNRM
```

13-24. RDBIT

The RDBIT subroutine checks the state of a specified bit on a channel.

Format

CALL RDBIT(*chan,nbit,bit*)

Parameters

chan	number of the channel being read.
nbit	the bit position starting from the right with bit 0.
bit	state of *nbit*, either 0 or 1.

RDBIT reads from the specified HP 6940 channel the state of a certain bit. Bit positions having a 1 or 0 may represent relay contact opening or closing. The precise state is a function of the particular hardware and its interface with the computer.

Digital input can be performed in two modes, either with or without wait for the input card flag. The flag is set by making the channel number negative in the RDBIT call.

Examples

`100 CALL RDBIT(-1,3,0)`	*Performs digital input with wait from channel 1 bit 3.*
`200 CALL RDBIT(1,2,B)`	*Reads a bit from channel 1, bit position 2 into location B without wait.*

13-25. RDWRD (Read Channel)

The RDWRD subroutine reads the contents of a channel into a word.

Format

CALL RDWRD(*chan,word*)

Parameters

chan	number of the channel being read.
word	location into which the results of the information from the channel will be placed.

The RDWRD routine reads the HP 6940 interface board on the specified channel and places the twelve input bits into the twelve low-order positions of the location defined by *word*. The four high-order positions are zeroed. It is possible to read any digital signal up to 12 bits wide. The signals may represent any type of device in any combinations.

Digital input can be performed in two modes, either with or without wait for the input card flag. The flag is set by making the channel number negative in the RDWRD call.

Examples

`100 CALL RDWRD(1,W)`	*Reads a complete word of information from channel 1 and places the information in location W.*
`200 CALL RDWRD(-1,W)`	*Digital input is performed with the wait flag set.*

13-26. SENSE

The SENSE (Event Sense) routine senses a change in the bit pattern.

Format

CALL SENSE(*chan,nbit,bit,trapn*)

Parameters

chan	number of the channel being sensed.
nbit	bit position starting from the right with bit 0.
bit	state of *nbit*, either 0 or 1.
trapn	trap number to which the sensing is directed.

The SENSE subroutine allows you to ask the hardware to constantly monitor for the presence of a specified condition. Specifically, the two conditions that can be sensed are:

- a given bit position becoming 1,
- a given bit position becoming 0.

As soon as one of the conditions is met, an interrupt is set and the associated TRAP is initiated. The condition being met in one direction, and thereby causing the interrupt, does not imply any interrupt in the opposite direction. If the SENSE is upon a given bit becoming 0 and that bit changes from the 0 condition, no action results in BASIC.

The SENSE command is designed for the HP 69434 Event Sense Card only. If the SENSE command is addressed to any other card, the following message is given:

ERROR A6940-2 IN LINE *nnnn*

where *nnnn* is the line number.

The HP 69434A card is operated in the "not equal" mode (jumper W3 in position D on the card). In this mode, an event is detected when the external data is "not equal" to the reference bit (parameter *bit* in the SENSE call).

Example

Time the duration of the "on" (1) condition of bit 4 on channel 2.

```
300 TRAP 5 GOSUB 500                 Define TRAP 5.
310 TRAP 6 GOSUB 600                 Define TRAP 6.
320 SENSE(2, 4, 1, 5)                Sense when bit 4 of channel 2 changes from 0 to
          •                          1.
          •                          Execute TRAP 5.

500 CALL TIME(X)                     Record time when bit changes. Then sense when
510 SENSE(2, 4, 0, 6)                turns "off" (1 to 0) and execute TRAP 6.
520 RETURN
600 CALL TIME(Y)                     Record time when turns off.
610 LET Z = Y - X                    Compute interval between on and off.
620 PRINT "TIME INTERVAL = ";Z  Print interval.
630 SENSE(2,4,1,5)                   Sense "on" again.
640 RETURN
```

13-27. WRBIT

The WRBIT routine writes a bit onto a channel.

Format

CALL WRBIT(*chan,nbit,bit*)

Parameters

chan	number of the channel.
nbit	bit position starting from the right with bit 0.
bit	state of *nbit*, either a 0 or 1.

WRBIT writes a single bit on the specified channel of the HP 6940. The bit can turn a light on or off, close or open a relay, or perform any other discrete function.

Digital output can be performed in two modes, either with or without wait for the output card flag. The flag is set by making the channel number negative in the WRBIT call.

Examples

```
100 CALL WRBIT(1,3,0)                Write a 0 bit onto channel 1 position 3 without
                                     wait.
200 CALL WRBIT(-1,3,0)               Perform digital output with wait.
```

13-28. WRWRD (Write Channel)

The WRWRD routine writes the contents of a word onto a channel.

Format

CALL WRWRD(*chan*,*word*)

Parameters

chan	number of the channel.
word	location from which the information is written.

WRWRD writes on the channel specified, the contents of the word stated by the parameter *word*. This provides a 12-bit parallel digital output which corresponds to the 12 least significant bits (right-most bits) of *word*. Depending upon the associated hardware, lights will be operated, relays closed, etc.

Digital output can be performed with or without wait for the output card flag. The flag is set by making the channel number negative in the WRWRD call.

Examples

`100 CALL WRWRD(1,W)`	*Writes the lower 12 bits of information from the location W onto channel 1 without wait.*
`200 CALL WRWRD(-1,W)`	*Performs the same output with wait.*

13-29. HP 6940 SUBSYSTEM ERRORS

All error messages generated by the previous subroutines take the following form:

ERROR *name-numb* IN LINE *xx*

where *name* is the module name, *numb* is the error type, and *xx* is the line in which the error occurs.

The following is a list of HP 6940 errors.

INPUT/OUTPUT CALLS

A6940-1	Driver timeout.
A6940-2	Parameter value outside defined range.

DAC CALL

DAC-1	Driver timeout.
DAC-2	Parameter value outside defined range.

13-30. HP 6940 TABLE PREPARATION

In order to use the HP 6940 subroutines you must provide information about the subsystem which is incorporated in the Instrument Table (see paragraph 13-2).

You must also add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in the HP 92064A RTE-M System Generation Manual. A list of the RTMTG commands required to add the HP 6940 subroutines is given there.

13-31. HP 6940 CARD CONFIGURATION

The configuration of the various types of cards in an HP 6940 Subsystem must follow the conventions given below. If expansion is anticipated, configuration can include blank slots to accommodate future cards. The blank slots then have channel numbers assigned to them, but cannot be accessed until the hardware is added.

Figure 13-2 illustrates the concept of the HP 6940 Subsystem configuration. The cards must be installed in the order described in the HP ISA FORTRAN Extension Package Reference Manual.

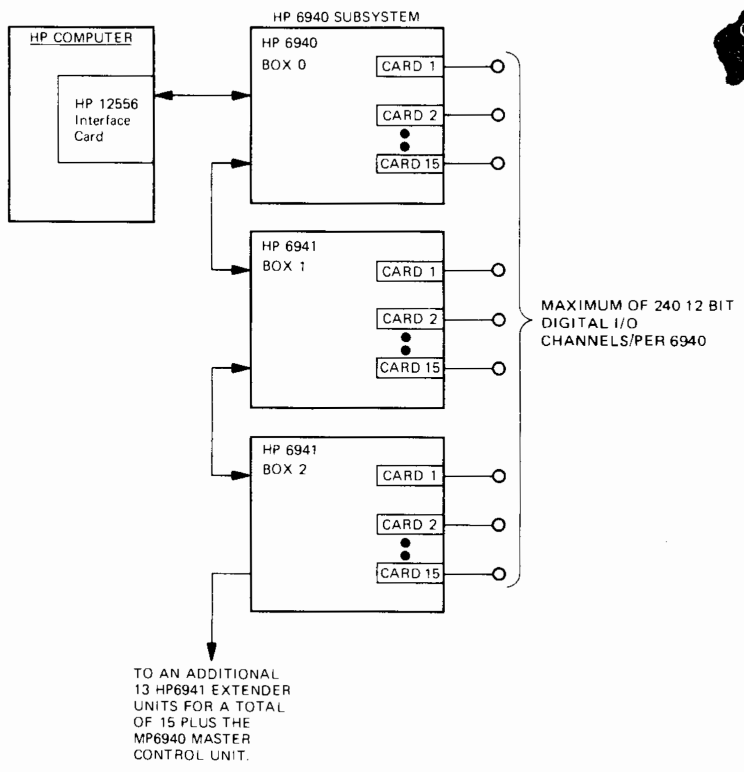

HP 6940 SUBSYSTEM
HP COMPUTER
HP 12556 Interface Card
HP 6940 BOX 0
CARD 1
CARD 2
CARD 15
HP 6941 BOX 1
CARD 1
CARD 2
CARD 15
HP 6941 BOX 2
CARD 1
CARD 2
CARD 15
MAXIMUM OF 240 12 BIT DIGITAL I/O CHANNELS/PER 6940
TO AN ADDITIONAL 13 HP6941 EXTENDER UNITS FOR A TOTAL OF 15 PLUS THE MP6940 MASTER CONTROL UNIT.

Figure 13-2. HP 6940 Subsystem Configuration

13-32. HP 6940 EXPANSION

The system may be expanded by adding additional HP 6940 Subsystems or by adding HP 6941 Extenders to existing subsystems. The card order applies to each HP 6940 added. The card order also applies to the slots beginning with slot 401 in the 6940 through slot 414 in a HP 6941, if one is added. Remember that Event Sense cards are not allowed in the HP 6941.

13-33. HP 6940 CHANNEL NUMBERING

The channels are numbered sequentially starting after the last **HP 2313** channel number. If there is no **HP 2313**, the first **HP 6940** channel is 1.

Figures 13-3 and 13-4 illustrate two channel numbering schemes and the effect of expanding the system in one of the two ways described above. Assume that the system shown in the examples has 96 HP 2313 channels.

Detailed channel numbering information is given in the HP ISA FORTRAN Extension Package Reference Manual.

	CHANNEL NUMBERS	6940/6941 SLOT NUMBERS	CARD TYPE
1st 6940	97	400	Event Sense
	98	401	Event Sense
	99	402	Digital I/O
	.	.	.
	.	.	.
	108	411	Digital I/O
	109	412	Voltage DAC
	110	413	Current DAC
	111	414	Current DAC
2nd 6940	112	400	Event Sense
	113	401	Digital I/O
	.	.	.
	.	.	.
	126	414	Digital I/O

Figure 13-3. Channel Numbers for Additional 6940

	CHANNEL NUMBERS	6940/6941 SLOT NUMBERS	CARD TYPE
1st 6940	97	400	Event Sense
	98	401	Event Sense
	99	402	Event Sense
	100	403	Blank cards for
	101	404	future Event Sense
	102	405	expansion
	103	406	Digital I/O
	.	.	.
	.	.	.
	111	413	Digital I/O
	112	414	Blank (future Digital I/O)
6941	113	400	Voltage DAC
	114	401	Current DAC

Figure 13-4. Channel Numbers for Addition of a 6941 Extender

13-34. HP 7210 PLOTTER

The plotter draws precise lines on a paper surface that is a maximum size of 15 by 10 inches. The lines are drawn by a pen on the plotter which moves from a desired point to some other desired point in either a pen-up (no line) or pen-down position. To facilitate the drawing of these lines, several routines have been written for your use with BASIC programs. The routines can be separated into two categories:

- those that draw characters (alphabetic characters and special symbols), and
- those that draw lines.

Figure 13-5 at the end of this section contains a program which uses most of these routines, and figure 13-6 contains a program which uses some of the special routines (e.g. scale, axis, lines).

The HP 7210 Plotter routines interface with the RTE Driver, DVR10. There are two versions of this driver. The Complete Plotter Driver DVR10 supports both character and line drawing routines. The Minimum Plotter Driver DVR10 supports only line drawing routines. Refer to the HP 92064A Software Numbering Catalog for the module names and part numbers of these drivers.

13-35. AXIS

The AXIS routine plots one axis (horizontal or vertical) of a graph with a specified axis label, a specified length, and a specified value at each inch marker.

Format

CALL AXIS(*x,y,"label",length,angle,min val, inc val*)

Parameters

x	horizontal coordinate.
y	vertical coordinate.
label	axis label (Enclose in quotes as a literal string.)
length	length of axis in inches.
angle	angle of axis in degrees. 0° is horizontal, angle increases in counter-clockwise direction.
min val	minimum value of axis (may be calculated with SCALE routine).
inc val	incremental value (may be calculated with SCALE routine).

Parameter *length* can be positive to place label counterclockwise to axis (as in y axis) and negative to place clockwise (as in x axis).

SCALE must be called before AXIS if points on the graph are scaled by SCALE. AXIS calls the SYMB routine to plot the labels 0.14 inches high.

Numbers are printed with .07 inch character height. Since X and Y always refer to the origin of the axis, leave at least .5 inch for axis label and numbers.

Example

```
100 CALL AXIS(0,0,"PSI",10,90,1,1)
```

Plots the Y axis with the label "PSI" on the counterclockwise side, 10 inches long at 90 degrees.

13-36. FACT

The FACT (factor) routine sets the ratio between the horizontal and vertical axis.

Format

CALL FACT(x,y)

Parameters

x	horizontal axis
y	vertical axis

FACT and SFACT are interrelated routines. Once SFACT is used to establish the graph limits (paper size), FACT does not need to be called unless you want to change the scale relationship.

Initially the horizontal and vertical axis in FACT are automatically set to 1 (e.g. the magnitude of 2 equals two inches in both directions). If you wish to have something plotted at half size, and SFACT has been set to 15 X 10, set FACT (.3333,.5) in the program. x and y are multiplied by 1000 plotter increments per inch when the new scaling factor is established.

The following equations give the actual translation from the X,Y in the plot call to the value actually put out by the plotter.

$$X_{out} = 1000.0 * X_{in} * X_{fact}$$

$$Y_{out} = 1000.0 * Y_{in} * Y_{fact}$$

X_{in} is X in PLOT call. X_{fact} is X in FACT call (set to 1 originally). Y_{in} is Y in PLOT call. Y_{fact} is Y in FACT call (set to 1 originally).

13-37. LINES

The LINES routine plots a line and/or symbols through successive data points in arrays previously scaled by the SCALE routine.

Format

```
CALL LINES(x(1),y(1),numb,scale,cntrl,symb)
```

Parameters

x	array scaled for the abscissa.
y	array scaled for the ordinate.
numb	number of points to be plotted.
scale	integer that specifies the point to be scaled. 1 = every point. 2 = every second point. *n* = every *n*th point.
cntrl	control value: 0 = line plot only. 1 = symbol at every point with line. −1 = symbol at every point. 2 = line and symbol at every second point. −2 = symbol at every second point. *n* = line and symbol at every *n*th point. −*n* = symbol at every *n*th point.
symb	number of centered symbol to be plotted (see SYMB routine).

Since the LINES routine requires the adjusted minimum and delta values produced by the SCALE routine, SCALE must be called before LINE for each graph.

Before calling this routine, the pen should be moved to the point 0,0 on the graph, and the origin should be set at that point with LLEFT.

13-38. LLEFT

The LLEFT routine lifts the pen and moves it to the lower-left corner.

Format

```
CALL LLEFT
```

When LLEFT is called, the internal *x* and *y* references are set to zero. This provides a defined reference for starting a plot. The pen is left in the up position at the completion of this call. This means that a call to PLOT must be made to put the pen down.

13-39. NUMB

The NUMB routine plots a number, with or without the decimal point, at a specified height, location, and angle.

Format

CALL NUMB(*x,y,height,numb,angle,digits*)

Parameters

x	horizontal coordinate of lower left corner of number.
y	vertical coordinate.
height	height of number in inches.
numb	number to be plotted.
angle	angle at which the number is to be plotted.
digits	number of digits. 0 = print the decimal point of an integer. −1 = suppress the decimal point. *n* = number of digits to print to the right of the decimal point (maximum of seven).

Example

```
100 CALL NUMB(5.32,8.79,0.1,0.0,-1)
110 CALL NUMB(6.3,8.79,0.1,J,0.0,-1)
120 CALL NUMB(7.16,8.79,0.1,K,0.0,-1)
```

Plot three numbers .1 inches high, with decimal point suppressed, at 8.79 inches above 0,0 and 5.32,6.3, and 7.16 inches to the right of 0,0.

13-40. PLOT

The PLOT routine moves the pen from an origin to a destination.

Format

CALL PLOT (*x,y,pram*)

Parameters

x	horizontal coordinate in inches.
y	vertical coordinate in inches.
pram	constant or variable name set equal to one of the following: −2 = move with pen down; consider the point where the pen stops (x,y) as the new origin. −3 = move with the pen up; consider the point where pen stops as new origin. +2 = move with the pen down; origin unchanged. +3 = move with the pen up; origin unchanged.

If the LLEFT routine is called first to establish the lower-left corner, the pen is left up. The PLOT routine must be called as follows to put the pen down:

CALL PLOT(0,0,2)

The parameters of the routine determine whether the origin is the original reference of the lower-left corner of the paper or the last known position from the previous plot.

Example

Plot a rectangle 8.5" by 11" starting at the origin (assuming the pen starts at the origin and that scaling has been set at 15 by 10 inches by calling LLEFT and SFACT(15,10)).

`100 CALL PLOT(0,0,2)`	*Sets the pen down.*
`110 CALL PLOT(11.,0.,2)`	*Moves the pen from X,Y =0,0 to X,Y = 11,0.*
`120 CALL PLOT(11.,8.5,2)`	*Moves the pen from X,Y = 11,0 to X,Y = 11,8.5.*
`130 CALL PLOT(0.,8.5,2)`	*Moves the pen from X,Y = 11,8.5 to X,Y = 0,8.5.*
`140 CALL PLOT(0.,0.,2)`	*Moves the pen from X,Y = 0,8.5 to the origin.*

13-41. PLTLU

The PLTLU routine defines the logical unit number of the plotter for all the plotter calls.

Format

CALL PLTLU(*lu*)

Parameter

lu	logical unit number of the plotter.

The PLTLU routine must be the first routine called to establish the logical unit number.

13-42. SCALE

The SCALE routine scales an array of numbers to fit a specified size graph; the values generated are used by the LINES and AXIS routines.

Format

CALL SCALE(*array(1),length,num pts,scale*)

Parameters

array	an array of values.
length	length of axis in inches.
num pts	number of points to be plotted.
scale	integer specifying the points to be scaled:
	1 = every point.
	2 = every second point.
	n = every *n*th point.

Separate calls are required for X and Y axis.

The adjusted minimum value is a number less than or equal to the minimum data value. The adjusted delta value is the result of subtracting the minimum data value from the maximum data value, divided by the length of the axis and adjusted to provide one-inch increments that will cover the data. The adjusted scale values are used by the LINES and AXIS routines.

The adjusted values are stored following the array. The minimum value of Y is stored in Y(*np***k*+1) where *np* is the number of points to be plotted; the delta value is stored in Y(*np***k*+2). Therefore, the array must be dimensioned (*k*+2) locations larger than (*np***k*), which is the number of locations necessary for data points. Normally, *k* equals 1, so an array Z of ten data points would be dimensioned as Z(12).

Example

`10 DIM X(52),Y(52)`	*Dimension X and Y arrays.*
`   •`	
`   •`	
`100 CALL SCALE(X(1),6.5,50,1)`	*Scale every point in a 50 point array, fitting X values on a 6.5 inch X axis and Y values on a 10 inch Y axis.*
`110 CALL SCALE(Y(1),10.0,50,1)`	

13-43. SFACT

The SFACT routine sets or adjusts the plotter for the particular size paper being used.

Format

CALL SFACT(*width,height*)

Parameters

width	width scale for horizontal movements.
height	height scale for vertical movements.

SFACT and FACT are interrelated routines. Once SFACT is used to establish the graph limits (paper size) FACT does not need to be called unless you wish to change the scale size. Initially the width and height parameters in SFACT are automatically set to 10 inches by 10 inches.

SFACT should be used as follows:

a. Place the paper on the plotter bed.

b. Manually adjust the pen and carriage travel to the lower left and then upper right corners of the paper (image area if desired).

c. Call SFACT with the parameters set to the paper size that was set up in step b.

After the plotter is adjusted and SFACT has been called with the proper dimensions, the horizontal and vertical dimensions in other calls are set for a one-to-one relationship.

Example

```
10 CALL PLTLU(13)
20 CALL LLEFT
25 CALL SFACT(15,10)
30 CALL PLOT(0,0,2)
40 CALL PLOT(5,5,2)
```

The pen moves five inches horizontally and five inches vertically for a paper size of 15 by 10 inches.

13-44. SYMB

The SYMB routine is used when characters are to be plotted.

Format

```
                  string variable
CALL SYMB(x,y,size,"char"          ,theta,pram)
                  integer
```

Parameters

x	horizontal starting coordinate.
y	vertical starting coordinate.
size	character height in inches.
"char"	actual characters to be plotted enclosed in quotes (string literal).
string variable	string variable containing characters to be printed.
integer	any single integer 0 through 25 (without quotes) representing a special character.
theta	number of degrees counterclockwise that the line of plotting is to be rotated.
pram	constant or variable name set equal to one of the following: 1 = plot the characters between the quotes or in the string variable. −1 = draw the special character and leave the pen up. −2 = draw the special character and leave the pen down.

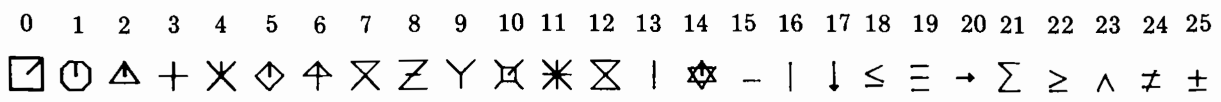
The SYMB routine is used to write characters on the paper if *pram* = +1. If *pram* = −1 or −2, the special character represented by the ASCII number is drawn. If the pen is left down, moving the pen to the place where the next character is to be drawn causes the special characters to be connected like points along a graph. If the pen is left up, the points are drawn without intervening connecting lines. When drawing graphs, it is recommended that symbols that are centered be used. The coordinates specified by you refer to the lower left corner of an imaginary box enclosing the character to be drawn. If you wish to continue to call PLOT and have the set of ASCII characters be contiguous, place the value 9999 as the *x* and *y* coordinates. Be aware that the *x* and *y* coordinates from one subroutine are not transmitted to the other subroutines. You can use WHERE (see paragraph 13-46) to supply missing information when you transfer from the PLOT to the SYMB routine, etc.

The following are the ASCII characters which may be enclosed within quotes:

A through Z and 0 through 9.

@ [ ] \ ↑ ← ! " # $ % & ' ( ) * , - . / 0 : ; < = > ? "space"

These symbols are centered and have their ASCII reference number immediately above the symbol:



13-45. URITE

The URITE routine lifts the pen and moves it to the upper right corner of the paper to facilitate unloading the paper at the end of a plot. This routine does not affect any of the coordinates.

Format

```
CALL URITE
```

13-46. WHERE

The WHERE routine indicates the current position of the plotter pen.

Format

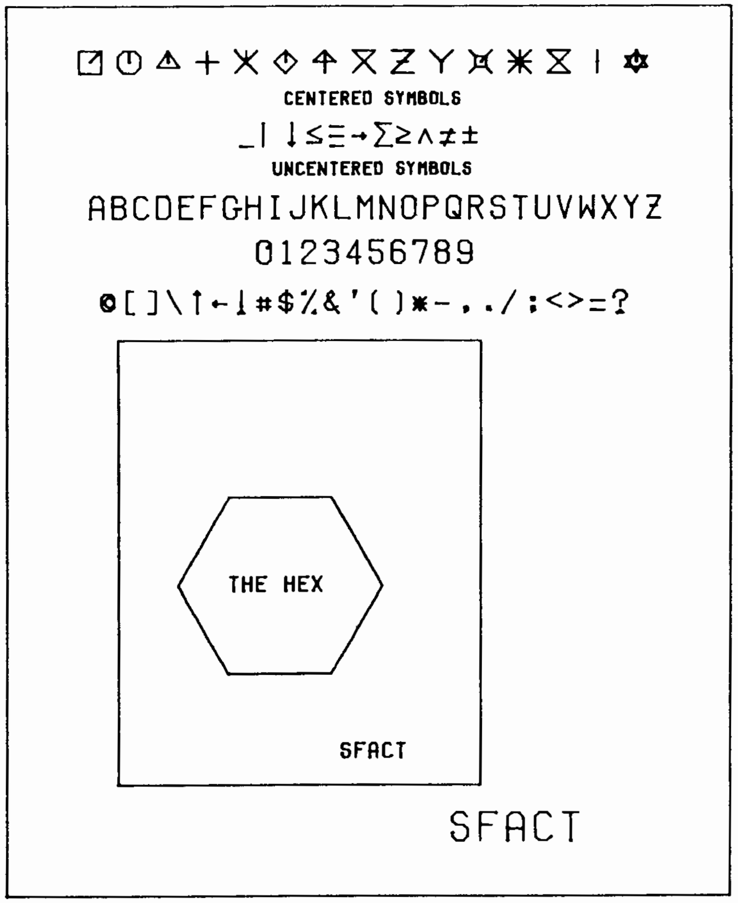
CALL WHERE(*x*,*y*)

Parameters

x	variable in which the horizontal coordinate is returned.
y	variable in which the vertical coordinate is returned.

13-47. TABLE PREPARATION

In order to use the plotter subroutines, you must add the name of the routines to the Branch and Mnemonic Tables. The procedure for doing this is described in the HP 92064A RTE-M System Generation Manual. The required RTMTG commands are provided there.



```
10  DIM A$[30]
20  LET Z=1
30  LET A=D=X=0
40  PRINT "PLOT DEMO"
```

```
50  CALL SFACT(15,10)
60  CALL LLEFT
70    FOR A=1 TO 2
80    CALL PLOT(1,1,3)
90    CALL PLOT(7.5,1,2)
100    CALL PLOT(7.5,9,2)
110    CALL PLOT(1,9,2)
120    CALL PLOT(1,1,2)
130    CALL SYMB(5,1.5,.25,"SFACT",0,1)
140    CALL SFACT(30,20)
150    CALL PLOT(3,3,-3)
160    NEXT A
```

To demonstrate the effect of "SFACT" on plot size, this section draws a box twice. The first time "SFACT" is set to (15,10). The second time it is set to (30,20) causing the second box drawn to be scaled one-half of the original larger box. Note that the different scale factor applies also to the size of the lettering.

```
170  CALL SFACT(60,40)
180  LET X=0
190  LET Y=0
200    FOR B=1 TO 5
210      FOR A=1 TO 15
220      CALL SYMB(X,Y,.25,15,D,-2)
230      LET X=Y=9999
240      NEXT A
250    LET D=D+60
260    NEXT B
270  CALL SYMB(0,3,.5,"THE HEX",0,1)
```

The scale factor is again reset and the "underline" character is used to draw a hexagon. This demonstrates the ability to draw characters in any orientation. Note that the initial setting of the pen is given to place the pen at the proper starting point and that all subsequent characters are drawn from the last-used position by employing coordinates of (999,999) thereby producing a solid line.

```
280  CALL LLEFT
290  CALL SFACT(15,10)
300  LET X=1.75
310  LET Y=8.5
320    FOR C=0 TO 14
330    CALL SYMB(X,Y,.2,C,0,-1)
340    LET X=Y=9999
350    NEXT C
```

As the position of the pen is unknown, the pen is brought to the known position of the lower-left-hand corner. The magnitude is made equal to inches by (15,10) and then the pen is sent to the starting position in the first "SYMB" execution. Thereafter, the last pen position is accepted for the next character. The special characters 0 through 14 are first plotted.

```
360  LET X=3.1
370  LET Y=7.75
380    FOR C=15 TO 25
390    CALL SYMB(X,Y,.2,C,0,-1)
400    LET X=Y=9999
410    NEXT C
```

The same logic is used to plot characters 15 through 25.

```
420  LET A$="CENTERED SYMBOLS"
430  CALL SYMB(3.5,8.125,.1,A$,0,1)
440  LET A$="UNCENTERED SYMBOLS"
450  CALL SYMB(3,4,7.5,.1,A$,0,1)
```

The two lines are labeled.

```
460  CALL SYMB(1.75,7.1,.2,"ABC",0,1)
470  CALL SYMB(9999,9999,.2,"DEF",0,1)
480  CALL SYMB(9999,9999,.2,"GHIJKLMNOPQRSTUVWXYZ",0,1)
490  CALL SYMB(3.25,6.7,.2,"0123456789",0,1)
500  CALL SYMB(1.85,6.25,.2,"@[]\↑←|",0,1)
510  CALL SYMB(9999,9999,.2,"#$%&'()*,./;<>=?",0,1)
```

All remaining characters are plotted by first positioning the pen at the beginning of each new line desired.

```
520  CALL WRITE
530  PRINT "END OF PLOT"
540  END
```

The pen is moved out of the way to the upper-right corner.

Figure 13-5. Plotter Control Sample Program #1

```
1Ø   REM
2Ø   REM      721Ø PLOTTER EXAMPLE
3Ø   REM
4Ø   LET N=36
5Ø   CALL PLTLU(18)
6Ø   CALL LLEFT
7Ø   CALL SFACT(15,1Ø)
8Ø   CALL PLOT(.5,Ø,-3)
9Ø   REM
1ØØ   REM      PLOT AXES FOR X AND Y
11Ø   REM
12Ø   CALL AXIS(Ø,5,"RADIANS",-14,Ø,Ø,1)
13Ø   CALL AXIS(Ø,1,"VOLTAGE",8,9Ø,-.4,.1)
14Ø   LET F=Ø
15Ø   LET F=F+1
16Ø   REM
17Ø   REM      PLOT CURVES
18Ø   REM
19Ø   DIM X[142],Y[142]
2ØØ   IF F=2 CALL PLOT(Ø,1,-3)
21Ø     FOR I=Ø TO 14 STEP .1
22Ø     LET X[I*1Ø+1]=I
23Ø     IF F=1 LET Y[I*1Ø+1]=SIN(I)*4
24Ø     IF F=2 LET Y[I*1Ø+1]=COS(I)*4
25Ø     NEXT I
26Ø   REM
27Ø   REM      SCALE ARRAY
28Ø   REM
29Ø   CALL SCALE(X[1],14,14Ø,1)
3ØØ   CALL SCALE(Y[1],1Ø,14Ø,1)
31Ø   REM
32Ø   REM      PLOT THE ARRAY
33Ø   REM
34Ø   CALL LINES(X[1],Y[1],14Ø,1,1,F)
35Ø   IF F<2 GOTO 15Ø
4ØØ   REM
41Ø   REM      PRINT PLOT NUMBER
42Ø   REM
43Ø   CALL SYMB(12.2,6.5,.15,"PLOT #",Ø,1)
44Ø   CALL NUMB(9999,9999,.15,N,Ø,-1)
5ØØ   REM
51Ø   REM      PRINT LEGEND
52Ø   REM
53Ø   CALL SYMB(12.2,1.5,.2,"LEGEND",Ø,1)
54Ø   CALL PLOT(12.2,1.4,3)
55Ø   CALL PLOT(13.3,1.4,2)
56Ø   CALL SYMB(12.2,1.1,.15,1,Ø,-1)
57Ø   CALL SYMB(9999,9999,.15," SINE WAVE",Ø,1)
58Ø   CALL SYMB(12.2,.6,.15,2,Ø,-1)
59Ø   CALL SYMB(9999,9999,.15," COSINE WAVE",Ø,1)
6ØØ   CALL WRITE
61Ø   STOP
9999   END
```

Figure 13-6. Plotter Control Sample Program #2

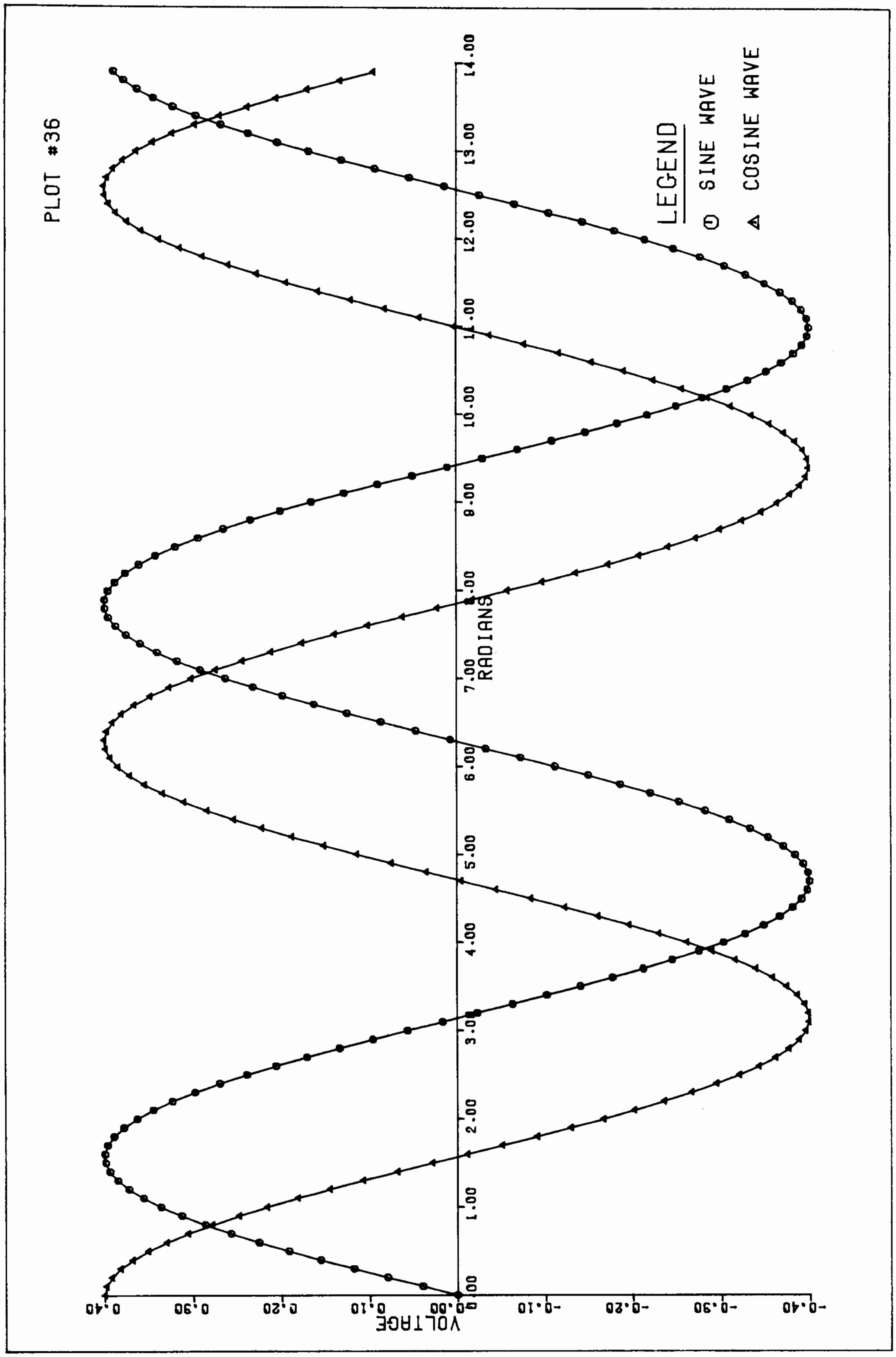


Figure 13-7. Plotter Control Sample Program #2 (Plot)

HP CHARACTER SET FOR COMPUTER SYSTEMS

APPENDIX A

Effect of Control key *

000-037B | 040-077B | 100-137B | 140-177B

b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
BITS				COLUMN →								
b4	b3	b2	b1	ROW ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

32 CONTROL CODES

Upshifted Lower Case

64 CHARACTER SET

96 CHARACTER SET

128 CHARACTER SET

EXAMPLE: The representation for the character "K" (column 4, row 11) is.

	b7	b6	b5	b4	b3	b2	b1
BINARY	1	0	0	1	0	1	1
OCTAL	1		1			3	

$$\text{BINARY}\quad 1\ \underbrace{0\ 0\ 1}_{1}\ \underbrace{0\ 1\ 1}_{3}$$

* Depressing the Control key while typing an upper case letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

9206- 1A

HEWLETT-PACKARD CHARACTER SET FOR COMPUTER SYSTEMS

This table shows HP's implementation of ANS X3.4-1968 (USASCII) and ANS X3.32-1973. Some devices may substitute alternate characters from those shown in this chart (for example, Line Drawing Set or Scandanavian font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16 bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word, add the two values. For example, "AB" produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Mnemonic	Graphic[1]	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	$\mathrm{N_U}$	Null
1	000400	000001	SOH	$\mathrm{S_H}$	Start of Heading
2	001000	000002	STX	$\mathrm{S_X}$	Start of Text
3	001400	000003	ETX	$\mathrm{E_X}$	End of Text
4	002000	000004	EOT	$\mathrm{E_T}$	End of Transmission
5	002400	000005	ENQ	$\mathrm{E_Q}$	Enquiry
6	003000	000006	ACK	$\mathrm{A_K}$	Acknowledge
7	003400	000007	BEL	🔔	Bell, Attention Signal
8	004000	000010	BS	$\mathrm{B_S}$	Backspace
9	004400	000011	HT	$\mathrm{H_T}$	Horizontal Tabulation
10	005000	000012	LF	$\mathrm{L_F}$	Line Feed
11	005400	000013	VT	$\mathrm{V_T}$	Vertical Tabulation
12	006000	000014	FF	$\mathrm{F_F}$	Form Feed
13	006400	000015	CR	$\mathrm{C_R}$	Carriage Return
14	007000	000016	SO	$\mathrm{S_O}$	Shift Out } Alternate Character Set
15	007400	000017	SI	$\mathrm{S_I}$	Shift In } Alternate Character Set
16	010000	000020	DLE	$\mathrm{D_L}$	Data Link Escape
17	010400	000021	DC1	$\mathrm{D_1}$	Device Control 1 (X-ON)
18	011000	000022	DC2	$\mathrm{D_2}$	Device Control 2 (TAPE)
19	011400	000023	DC3	$\mathrm{D_3}$	Device Control 3 (X-OFF)
20	012000	000024	DC4	$\mathrm{D_4}$	Device Control 4 ($\overline{\mathrm{TAPE}}$)
21	012400	000025	NAK	$\mathrm{N_K}$	Negative Acknowledge
22	013000	000026	SYN	$\mathrm{S_Y}$	Synchronous Idle
23	013400	000027	ETB	$\mathrm{E_B}$	End of Transmission Block
24	014000	000030	CAN	$\mathrm{C_N}$	Cancel
25	014400	000031	EM	$\mathrm{E_M}$	End of Medium
26	015000	000032	SUB	$\mathrm{S_B}$	Substitute
27	015400	000033	ESC	$\mathrm{E_C}$	Escape[2]
28	016000	000034	FS	$\mathrm{F_S}$	File Separator
29	016400	000035	GS	$\mathrm{G_S}$	Group Separator
30	017000	000036	RS	$\mathrm{R_S}$	Record Separator
31	017400	000037	US	$\mathrm{U_S}$	Unit Separator
127	077400	000177	DEL	▓	Delete, Rubout[3]

9206-1B

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number Sign, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent
38	023000	000046	&	Ampersand, And Sign
39	023400	000047	'	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus
44	026000	000054	,	Comma, Cedilla
45	026400	000055	–	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	} Digits, Numbers
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	
78	047000	000116	N	Upper Case Alphabet, Capital Letters
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	Left (opening) Bracket
92	056000	000134	\	Backslash, Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret, Circumflex; Up Arrow[4]
95	057400	000137	_ ←	Underline; Back Arrow[4]

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	`	Grave Accent[5]
97	060400	000141	a	
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	
110	067000	000156	n	Lower Case Letters[5]
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	Left (opening) Brace[5]
124	076000	000174	¦	Vertical Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

9206-1C

Notes: [1]This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as "||", "@", or space.

[2]Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on a 2640 terminal.

[3]Delete may be displayed as "__", "@", or space.

[4]Normally, the caret and underline are displayed. Some devices substitute the up arrow and back arrow.

[5]Some devices upshift lower case letters and symbols (` through ~) to the corresponding upper case character (@ through ^). For example, the left brace would be converted to a left bracket.

RTE SPECIAL CHARACTERS

Mnemonic	Octal Value	Use
SOH (Control A)	1	Backspace (TTY)
EM (Control Y)	31	Backspace (2600)
BS (Control H)	10	Backspace (TTY, 2615, 2640, 2644, 2645)
EOT (Control D)	4	End of file (TTY, 2615, 2640, 2644, 2645)

9206-1D

SUMMARY OF STATEMENTS, COMMANDS AND SUBROUTINES

APPENDIX B

B-1. STATEMENT SUMMARY

This summary of BASIC statements provides the statement names in alphabetical order with a brief description and a reference to the paragraph or paragraphs containing a complete statement description.

STATEMENT	DESCRIPTION	PARAGRAPH REFERENCE
CALL	Calls for execution of an external subroutine, optionally passing parameters to the subroutine.	6-2
COM	Declares a common block to contain specified variables used in common by more than one program.	3-13
DATA	Provides data to be read by READ statements.	3-10 4-12
DEF	Defines a function.	5-2
DIM	Reserves storage for arrays and sets upper bounds on the number of elements.	3-12
	DIM also reserves storage for strings and sets their maximum character length.	4-5
END	Terminates execution of the current program. Last statement in the program must be END.	3-4
FOR . . . . NEXT	Allows repetition of a group of statements between FOR and NEXT (a program loop). The number of repetitions is determined by the initial and final values of a FOR variable, and by an optional step specification.	3-5
GOTO	Transfers control to a specified statement label.	3-3
GOTO . . . OF	Multibranch GOTO transfers control to one of a list of statement labels depending on the value of an integer expression.	3-3
GOSUB	Causes execution of a subroutine beginning at a specified statement label. Following execution of a RETURN statement in the subroutine, control returns to the statement following GOSUB.	6-1
GOSUB . . . OF	Multibranch GOSUB executes one of a list of subroutines depending on the value of an integer expression.	6-1
IF EOF #*lu* . . . THEN	Specifies action to be taken when an end-of-file condition occurs on input from a peripheral device (see READ #*lu*).	7-5
IF . . . THEN	Evaluates a conditional expression and specifies action to be taken if the condition is true. The condition is a numeric or string expression. The action may transfer to a statement label or may be a single executable statement.	3-6 4-10

STATEMENT	DESCRIPTION	PARAGRAPH REFERENCE
INPUT	Requests user input to one or more variables by printing a question mark (?) prompt. Following the prompt, string or numeric data is accepted from the terminal.	3-11 4-7
LET	Introduces assignment statement that assigns one or more values to a variable or array element. The word LET may be omitted from the assignment statement.	3-1 4-6
NEXT	Terminates a loop introduced by a FOR statement. Specifies a variable that must match the FOR variable.	3-5
PAUSE	Stops program execution without terminating the program. Prints a PAUSE message on your terminal.	3-14
PRINT	Prints the contents of a list of numeric or string expressions on the list device.	3-7 4-8
PRINT #*lu*	Prints the contents of a list of numeric or string variables or expressions to a specified logical unit number.	4-13 7-9
READ	Assigns constants and string literals from one or more DATA statements to the variables specified in READ. Treats contents of all DATA statements as a single data list.	3-10 4-9
READ #*lu*	Reads one or more items from a specified logical unit number.	4-14 7-7
REM	Introduces remarks and comments in the program listing.	3-2
RESTORE	Resets the data pointer to the beginning of the program or to the first DATA statement following a specified label.	3-10
RETURN	Returns control from a GOSUB subroutine to the statement following the last previous GOSUB statement.	6-1
STOP	Terminates execution of the program run.	3-4
TRAP	Associates a trap number with a task.	10-11
WAIT	Causes an executing program to stop for a specified number of milliseconds.	3-15

B-2. COMMAND SUMMARY

Each command is listed by name in alphabetical order followed by a brief description and a reference to the paragraph containing a complete description of the command.

COMMAND	DESCRIPTION	PARAGRAPH REFERENCE
BACKF	Backspaces magnetic tape past previous file mark on specified logical unit number.	12-1
*BR,BASIC	Breaks execution of a BASIC program.	9-8
BYE	Terminates execution of the BASIC Interpreter.	9-6
DELETE or DEL	Deletes one or a range of more than one statement from the current program.	9-4
LIST	Lists the contents of the current program at a specified peripheral device or to the default output device.	9-7
LOAD	Loads all or a portion of a source program.	9-1
MERGE	Merges a source program with a program in memory.	9-3
REWIND	Rewinds magnetic tape on a specified logical unit number.	12-1
RUN	Executes the current program or loads and executes a program from a specified peripheral device, or the default input device.	9-5
SAVE	Saves the current program at a specified peripheral device or the default output device.	9-5
SKIPF	Skips magnetic tape to end of current file on specified logical unit number.	12-1
WEOF	Writes an EOF mark to magnetic tape on specified logical unit number.	12-1

B-3. SUBROUTINE SUMMARY

Each subroutine is listed in alphabetical order followed by a brief description and a reference to the paragraph containing a complete description of the routine.

SUBROUTINE	DESCRIPTION	PARAGRAPH REFERENCE
AIRDV	Reads HP 2313 analog input in a random manner.	13-7
AISQV	Reads HP 2313 analog input sequentially.	13-8
AOV	Converts HP 2313 digital value to analog output.	13-9
AXIS	Plots an axis of a graph.	13-35
DAC	Converts HP 6940 digital value to analog output.	13-22
DSABL	Disables a specified task.	10-6
ENABL	Enables a specified task, permits scheduling of a previously disabled task.	10-7
FACT	Sets the ratio between the horizontal and vertical axis of a graph.	13-36
LINES	Plots a line and/or symbol through successive data points in arrays.	13-37
LLEFT	Lifts the plotter pen and moves it to the lower left plotter corner.	13-38
MPNRM	Clears the event sense mode and erases the channel/bit to trap number correspondence (HP 6940).	13-23
MTTFS	Writes an end-of-file or rewinds magnetic tape.	12-6
MTTPT	Positions a magnetic tape forward or backward a specified number of files and/or records.	12-5
MTTRD	Read a magnetic tape data record into an array.	12-4
MTTRT	Writes a record to magnetic tape.	12-3
NORM	Normalizes the HP 2313 Subsystem; that is, resets it to a home or known state.	13-10
NUMB	Plots a number, with or without decimal point, at a specified height, location, and angle.	13-39
PACER	Sets the pace rate of the HP 2313 Subsystem.	13-11
PLOT	Moves the plotter pen from an origin to a destination point.	13-40
PLTLU	Defines the logical unit number of the plotter for all subsequent plotter calls.	13-41
RDBIT	Reads (or checks the state) of a specified bit on a channel (HP 6940).	13-24
RDWRD	Reads the contents of a channel into a word (HP 6940).	13-25

SUBROUTINE	DESCRIPTION	PARAGRAPH REFERENCE
RGAIN	Reads the gain on a particular channel (HP 2313).	13-12
SCALE	Scales an array of numbers to fit a specified graph size.	13-42
SENSE	Sets up link between event sense and a specified trap. Senses a change in the bit pattern (HP 6940).	13-26
SETP	Sets the priority of a task.	10-8
SFACT	Sets or adjusts the graph limits (paper size) and/or the scale size for the plotter.	13-43
SGAIN	Sets the gain for all channels in a group (HP 2313).	13-13
START	Schedules a task for processing after a specified delay.	10-9
SYMB	Writes characters on a plot operation.	13-44
TIME	Returns the time according to the system real-time clock.	10-10
TRNON	Executes a task at a specified time.	10-12
TTYS	Sets up a link between a trap number and a teleprinter logical unit number.	10-13
URITE	Lifts and moves the plotter pen to the upper right corner to facilitate paper removal.	13-45
WHERE	Requests the current position of the plotter pen.	13-46
WRBIT	Writes a bit onto a channel (HP 6940).	13-27
WRWRD	Writes the contents of a word onto a channel (HP 6940).	13-28

SUMMARY OF ERROR MESSAGES

APPENDIX C

Four types of errors may cause error messages: command errors, statement syntax errors, pre-execution errors, and execution errors resulting from program execution. The numeric error codes and their associated error messages are listed in Table C-1.

COMMAND ERRORS

Command error messages are printed following the command that caused the error.

SYNTAX ERRORS

When a syntax error in a statement is detected, an error message is printed. You may type a carriage return and enter the statement correctly.

PRE-EXECUTION ERRORS

These errors are detected following a RUN command but before execution of the program. If no errors are detected, the program will be executed. Otherwise, the pre-execution phase terminates with no attempt to run the program.

Whenever possible, the line number in which the error occurred will be appended to the message in the form: IN LINE *nnnn*, where *nnnn* is the line number of the statement that caused the error.

EXECUTION ERRORS

These errors are detected during program execution and printed as they occur; program execution terminates.

The line number where the error occurred is appended to all run error messages in the form: IN LINE *nnnn*, where *nnnn* is the line number of the statement that caused the error.

Table C-1. BASIC Error Messages

ERROR NUMBER	MESSAGE	PHASE
01	ILLEGAL EXPONENT	Syntax Check
02	NOT A FORTRAN FUNCTION	Syntax Check
03	MISSING ASSIGNMENT OPERATOR	Syntax Check
04	NOT A SUBROUTINE CALL	Syntax Check
05	MISSING OR BAD FUNCTION NAME	Syntax Check
06	MISSING OR BAD SIMPLE VARIABLE	Syntax Check
07	MISSING OR BAD TRAP NUMBER	Syntax Check
08	MISSING OR ILLEGAL 'THEN'	Syntax Check
09	MISSING OR ILLEGAL 'OF'	Syntax Check
10	MISSING OR ILLEGAL 'TO'	Syntax Check
11	MISSING OR ILLEGAL 'STEP'	Syntax Check
12	MISSING OR ILLEGAL SUBROUTINE	Syntax Check
13	WRONG NUMBER OF PARAMETERS	Syntax Check
14	MISSING OR ILLEGAL DATA ITEM	Syntax Check
15	ILLEGAL READ OR INPUT VARIABLE	Syntax Check
16	NO CLOSING QUOTE	Syntax Check
17	MISSING OR BAD LIST DELIMITER	Syntax Check
18	ILLEGAL PARAMETER	Syntax Check
19	ILLEGAL STRING VARIABLE	Syntax Check
20	PARAMETER NOT STRING	Syntax Check
21	MISSING OR ILLEGAL SUBSCRIPT	Syntax Check
22	STRING OR DIM LARGER THAN 255	Syntax Check
23	ILLEGAL STRING RELATIONAL OPERATOR	Syntax Check
24	STRING NOT PERMITTED	Syntax Check
25	MISSING LEFT PARENTHESIS	Syntax Check
26	MISSING RIGHT PARENTHESIS	Syntax Check
27	UNDECIPHERABLE OPERAND	Syntax Check
28	MISSING OR BAD ARRAY VARIABLE	Syntax Check

Table C-1. BASIC Error Messages (Continued)

ERROR NUMBER	MESSAGE	PHASE
29	ILLEGAL OR MISSING INTEGER	Syntax Check
30	SIGN WITHOUT NUMBER	Syntax Check
31	CHARACTERS AFTER STATEMENT END	Syntax Check
32	OUT OF STORAGE	Syntax Check
33	ARRAY TOO LARGE	Syntax Check
34	COM STATEMENT OUT OF ORDER	Pre-execution
35	FUNCTION DEFINED TWICE	Pre-execution
36	UNMATCHED FOR	Pre-execution
37	NEXT WITHOUT MATCHING FOR	Pre-execution
38	DIMENSIONS NOT COMPATIBLE	Pre-execution
39	LAST STATEMENT NOT 'END'	Pre-execution
40	VARIABLE DIMENSIONED TWICE	Pre-execution
41	ARRAY OF UNKNOWN DIMENSIONS	Pre-execution
42	ARRAY TOO LARGE	Pre-execution
43	OUT OF STORAGE	Pre-execution
44	SYMBOL TABLE OVERFLOW	Pre-execution
45	OUT OF STORAGE	Execute
46	GOSUBS NESTED 20 DEEP	Execute
47	RETURN WITH NO PRIOR GOSUB	Execute
48	OUT OF DATA	Execute
49	WRONG DATA TYPE	Execute
50	SUBSCRIPT OUT OF BOUNDS	Execute
51	REFERENCED STATEMENT NOT DATA	Execute
52	UNDEFINED STATEMENT ACCESSED	Execute
53	BAD DATA	Execute
54	BAD EXPONENT	Execute
55	SUB. OR FUNCT. TERMINATED ABNORMALLY	Execute

Table C-1. BASIC Error Messages (Continued)

ERROR NUMBER	MESSAGE	PHASE
56	TRAP TABLE FULL	Execute
57	ILLEGAL TRAP/SEQ NUMBER	Execute
58	SCHEDULED BUT DELETED TASK	Execute
59	TRAP TABLE BUSY	Execute
60	NEGATIVE STRING LENGTH	Execute
61	NON-CONTIGUOUS STRING	Execute
62	STRING OVERFLOW	Execute
63	UNDEFINED STATEMENT REFERENCE	Execute
64	NEGATIVE NUMBER TO REAL POWER	Execute
65	ZERO TO ZERO POWER	Execute
66	ZERO TO NEGATIVE POWER	Execute
67	OUT OF RANGE IN FUNCTION	Execute
68	LOG OF NEGATIVE ARGUMENT	Execute
69	EXP OF OUT OF RANGE	Execute
70	ILLEGAL FUNCTION	Execute
71	INVALID COMMAND	Command
72	INVALID LIMITS	Command
73	INVALID LU OR STATEMENT NUMBER	Command
74	INVALID STATEMENT NUMBER	Command
75	NO LU NUMBER REFERENCE FOUND	Syntax Check

NOTE: The file handler module will print any FMP error numbers as they occur.

APPENDIX D

HP-IB/BASIC DATA CONVERSION

In computer-based HP-IB systems, it is very often necessary to have complete control over manipulation of numeric data formats for a given HP-IB device. This is particularly true of a device that requires a stringent fixed data format in order to operate properly. It is equally desirable to have a free-field conversion capability that will automatically translate different representations of the same data value. Although normal I/O programming statements provide part of this capability, they are primarily for I/O and do not provide for simple memory-to-memory conversions. This appendix provides general data conversion techniques for use in BASIC programming. Included are examples of converting variables to strings, or strings to variables.

For details about HP-IB callable subroutines, refer to the *HP 59310A/B Interface Bus I/O Kit User's Guide,* HP Manual Part No. 59310-90064.

DATA CONVERSION REQUESTS

The general BASIC data conversion requests are as follows:

```
    100 DCODE(V1,A$,F$)
    100 DCODE(B$,V2,F$)
```

where

V1	=	variable to be converted.
A$	=	string to contain result. (Note that the string must be predefined as to size and content, as discussed later.)
B$	=	string to be converted.
V2	=	variable to contain result.
F$	=	format statement by which conversion will occur.

The format specification must be contained within parentheses and may be either of the following:

Fn.d	=	floating point form: n is the number of characters including sign and decimal point; d is number of digits following the decimal point.
En.d	=	E-format floating point form: n is the number of characters including sign, decimal point, E, and exponent sign; d is number of digits following the decimal point.

BINARY-TO-ASCII

When converting from a binary value to an ASCII string, certain conditions are assumed. The string variable where the converted results will be stored has been predefined. This means that the length of the string has been established (by a DIM statement) and that the contents of the string have been initialized. Thus, when conversion occurs the actual results are placed in the indicated string positions without regard to the string's attributes, such as length, current content, etc. This type of operation facilitates the piecemeal construction of strings as desired. However, this also means that the overall string requirements be anticipated by the user according to his application as demonstrated in the examples given later. (Also, refer to Section IV for an in-depth discussion of strings and Sections II, III, V, VI and VII for a discussion of normal language capability.) Two conversion format types are discussed in the following paragraphs.

Fn.d (F) FORMAT. When using binary-to-ASCII conversions, the Fn.d format performs certain special operations. This format specification generates the following:

$$\overbrace{\pm xxxxx.\underbrace{xxxx}_{d\text{-digits}}}^{n\text{-field}}$$

The n-field positions include sign and decimal point as well as the digits. Plus signs are suppressed in the result but minus signs are always supplied. When the magnitude of the converted value is less than the n-field specification, the resulting string is always right-justified with the decimal point in the indicated position. The remainder of the n-field is filled with blank spaces to the left. When the magnitude of the value matches the exact n-field width and the d-digit sub-field is zero, the decimal point is suppressed and the result is an integer string. (Note that rounding off always occurs in the least significant digit of the resulting string.) When the magnitude of the value exceeds the n-field specification, dollar signs ($) appear in the result. This indicates an impossible conversion format was specified by the user and a correction should be made in his program. (See example 1 below.)

EXAMPLE 1. Predefined strings and F-format conversions; (∧ = blank).

```
10 DIM A$(7),B$(6)              <define string length>
20 LET A$ = "xxx.xxx"           <initialize string content>
30 LET B$ = "(F7.3)"            <define format specification>
40 DCODE (V,A$,B$)              <perform conversion>
```

variable (V)	string result (A$)
1.234	∧∧1.234
12.34	∧12.340
123.456	123.456
1234.567	1234.57
-1.3579	∧-1.358
12345600	$$$$$$$

A special use of the F-format is the production of integer strings. The method consists of defining an F-format as Fn.0, where n is the exact number of integer digits to be produced. For example, 123.0 would result in an integer string when the F-format is specified as F3.0. (See example 2 below.)

EXAMPLE 2. F-format conversion to produce an integer string.

```
10 DIM A$(12)                   <define string length>
20 LET A$ = "INTEGER=xxxx"      <initialize string content>
30 DCODE (V,A$(9,12),"(F4.0)")  <perform conversion>
```

variable (V)	string result (A$)
1234.0	INTEGER=1234
-765.432	INTEGER=-765

En.d (E) FORMAT. Like the F-format, the En.d format conversion also provides special operations. This format specification generates the following:

$$\overbrace{\pm.\underbrace{xxxxxx}_{\text{d-digits}}E\pm xx}^{\text{n-field}}$$

The specified n-field positions include mantissa sign, decimal point, all digits, E, exponent sign, and exponent. A plus sign for the mantissa is always suppressed in the result but a minus sign is supplied. The decimal point is also supplied, followed by the d-digits. As in the F-format above, the least significant digit in the resulting string is rounded off. In the exponent part, the E and sign are always supplied, followed by the two-digit exponent. When the converted value requires fewer positions than indicated in the n-field, the result is right-justified and filled with blanks to the left. (See example 3.)

EXAMPLE 3. Use of substrings, literals, and E-format conversion.

```
10 DIM A$(25)                         <define string length>
20 LET A$="VALUE IS±.xxxxE±xx UNITS"  <initialize string content>
30 DCODE (V,A$(10,19),"(E10.4)")      <perform conversion>
```

Note the use of substring character positions 10 through 19 to indicate the position in the A$ string where the results are to be placed. Also, note the use of a string literal for the format specification instead of a string variable as in example 1.

variable (V)	**string result (A$)**
`1.234`	`VALUE IS  .1234E+01 UNITS`
`12.3`	`VALUE IS  .1230E+02 UNITS`
`123.456`	`VALUE IS  .1235E+03 UNITS`
`-.00123`	`VALUE IS -.1230E-02 UNITS`
`0`	`VALUE IS  .0000E+00 UNITS`

ASCII-TO-BINARY

ASCII-to-binary data conversions provide operations somewhat similar to the reverse conversions discussed in the preceding paragraphs. In general, the format specifications indicate n-field positions (columns) of the ASCII string to be converted. Leading and trailing blanks within the n-field are ignored and may be used as data delimiters by the user. Data conversions occur as described in the following paragraphs.

Fn.d (F) FORMAT. The Fn.d format describes the floating point form of the string to be converted. The n-field position establishes the bounds of the string. Within this field, the data conversion takes place according to the actual decimal point position in the string. If a decimal point is not in the string then it is assumed to exist according to the d-digit specification as indicated. (See example 4 below.)

EXAMPLE 4. ASCII-to-binary conversion by F-format; (∧= blank).

```
10 DIM A$(40),B$(6)                    <define string length>
20 READ #12; A$                        <input string via LU 12>
30 LET B$="(F7.3)"                     <define format specifications>
40 DCODE (A$,V,B$)                     <perform conversion>
```

ASCII string (A$)	result (V)
123.456	123.456
123.4∧∧	123.4
∧123.4∧	123.4
∧∧123.4	123.4
-00.123	-.123

En.d (E) FORMAT. The En.d format also uses the n-field to establish string bounds and the conversion occurs according to the decimal point position in the string. If a decimal point is not present, then it is assumed to be positioned as specified by the d-digit part of the format. (See example 5.)

EXAMPLE 5. ASCII-to-binary conversion by the E-format.

```
10 DIM A$(40)                          <define string length>
20 READ #12; A$                        <input string via LU 12>
30 DCODE (A$(5,16),"(E12.6)")          <perform conversion>
```

ASCII string (A$)	result (V)
DCV -.123456E+01	-1.23456
DCV +.123E+03	123.0
ABCD1.379E+00	1.379

INDEX

In this index, the page number for the principal reference of each item is printed in **boldface** type. Page numbers for all other references to the item are printed in standard typeface.

In this index, the page number for the principal reference of each item is printed in **boldface** type. Page numbers for all other references to the item are printed in standard typeface.

In this index, the page number for the principal reference of each item is printed in **boldface** type. Page numbers for all other references to the item are printed in standard typeface.

In this index, the page number for the principal reference of each item is printed in **boldface** type. Page numbers for all other references to the item are printed in standard typeface.

PART NO. 92065-90001
Printed in U.S.A. 2/77

HEWLETT PACKARD

Sales and service from 172 offices in 65 countries.
11000 Wolfe Road, Cupertino, California 95014