BASIC/1000D
Multi-User Real-Time BASIC

Reference Manual

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

Update 4 October 1981
MANUAL PART NO. 92060-90016
Printed in U.S.A. April 1980

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

Sixth Edition	Feb 1980	
Update 1	Apr 1980	
Reprinted	Apr 1980	(Update 1 incorporated)
Update 2	Oct 1980	
Update 3	Apr 1981	
Update 4	Oct 1981	Include RTE-6/VM information

PREFACE

Multi-User Real-Time BASIC provides an augmented real-time version of the BASIC language with which as many as four users may code and execute programs simultaneously from different terminals. The Multi-User Real-Time BASIC subsystem provides functions, subroutines, and statements which allow you to schedule tasks, control instrument subsystems, the plotter and magnetic tape devices, and provides many additional capabilities. It may be run under control of the RTE 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 Operating System. If a BASIC system has been generated and is available for your use, you will find all the information you need to create and run BASIC programs in this manual. These manuals are shown in the documentation maps which follow this preface.

Section I introduces Multi-User Real-Time 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 or other languages are described in Section VI. Section VII describes disc files and the statements and functions which manipulate files.

Section VIII tells you how to execute the Real-Time BASIC Interpreter. Section IX describes the commands used to communicate with the Interpreter once it is running. Debugging commands are described separately in Section X.

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

Section XVIII provides instructions on generating the Instrument Table tape which is required if the instrumentation subroutines are to be used.

Appendix A contains alphabetical summaries of all statements, commands, and library subroutines. Appendix B describes error messages, Appendix C contains the ASCII character set, and Appendix D provides instructions for loading the Multi-User Real-Time BASIC software. Appendix E contains information about HP-IB/BASIC data conversion and subroutine table requirements for calling HP-IB utility subroutines.

NOTE

All references to RTE include RTE-II, III, IVA, IVB, and 6/VM unless specifically noted.

CONTENTS

CONTENTS (continued)

ILLUSTRATIONS

TABLES

SECTION I
INTRODUCTION

1-1. FEATURES

Multi-User Real-Time BASIC is a subsystem designed for use on RTE disc systems and provides a simple, easy-to-use augmented real-time version of the BASIC language. As many as four users may efficiently employ Real-Time BASIC concurrently, each with a uniquely named copy of the Real-Time BASIC software. Interaction with Multi-User BASIC can be via local or remote terminal devices, keypunched cards, paper tape, magnetic tape, or disc.

Real-Time BASIC provides you with these capabilities:

- Conversational programming.
- Multiple peripheral device I/O including graphics display.
- Real-time and event task scheduling.
- Dynamic program debugging aids.
- Fast access disc file storage for programs and data.
- Bit manipulation.
- Scheduling of BASIC, FORTRAN, and Assembly language programs.
- Instrumentation I/O and device subroutine simulation.
- User defined subroutines and functions.
- Character string manipulation.
- Program statement character editing and line resequencing.

1-2. CONVERSATIONAL PROGRAMMING

BASIC is an English-like programming language that is easy to learn and use. You enter programs directly into the Real-Time BASIC subsystem from a keyboard device. The BASIC Interpreter checks each statement as it is entered. If the statement contains an error, a message is printed which defines the error and you can correct it immediately. This type of interaction between you and the Interpreter is called conversational programming.

Conversational interaction allows you to test your programs step-by-step as they are being prepared. You are in constant touch with the system, its functioning, and its results. Programming and debugging are completed quickly, easily, and efficiently.

1-3. MULTIPLE PERIPHERAL DEVICE I/O

Multi-User Real-Time BASIC can provide a wide selection of input/output capabilities. It can be used with either hardcopy or display screen terminals, line printers, tape punches, and magnetic tape units. Data can be displayed on a hardcopy graphic plotter or TV monitor. The Interpreter also makes use of the fast-access disc storage capabilities of the RTE Operating System under which it operates.

1-4. REAL-TIME AND EVENT TASK SCHEDULING

Multi-User Real-Time BASIC is called *real-time* because 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.

BASIC includes 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. PROGRAM DEBUGGING AIDS

Multi-User Real-Time BASIC provides commands that enable you to debug a program while it is running. The path of flow through a program can be displayed, the values of variables can be displayed and modified, and subroutine calls can be simulated.

1-6. FILE CAPABILITIES

If you need or want a data base external to particular programs, Multi-User Real-Time BASIC provides a file capability allowing flexible yet straightforward manipulation of large volumes of data stored on disc files. Extensions to the READ, PRINT, and IF statements provide you with facilities for reading from or writing onto mass storage files and/or peripheral units.

Internally, files are organized as a collection of records each of 128 16-bit words. Thus, each record of a file may contain up to 64 numeric quantities. A string data item will occupy 1 + INT [(n + 1)/2] words, where n is its length in characters and INT truncates the quotient of the expression in brackets to an integer value.

When manipulated on a record-by-record basis, a file appears as a collection of subfiles which are the records. The ability to reference any record of the file directly allows you to partition your data and alter any group without disturbing the rest of the file.

BASIC, FORTRAN, and Assembly language programs can use the same data files but BASIC requires a special format to which programs in the other languages must conform if BASIC programs are to use the files. The file must be type 1 with 128 word fixed length records. Each word in the record must be initialized with all bits equal to 1. In RTE-IVB and 6/VM, BASIC does not recognize type 1 file extents.

1-7. ENVIRONMENT

1-8. HARDWARE

The BASIC Interpreter operates within the RTE hardware environment consisting of an HP 1000 M, E, or F-Series Computer System. (Refer to the appropriate system Programming and Operating Manual for equipment configurations and memory requirements.)

Peripheral devices required for BASIC are a system console and a disc drive. Optional devices include a line printer, card reader, photoreader, plotter, TV monitor, HP 2313 and HP 6940 Subsystems, and additional discs and terminals.

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

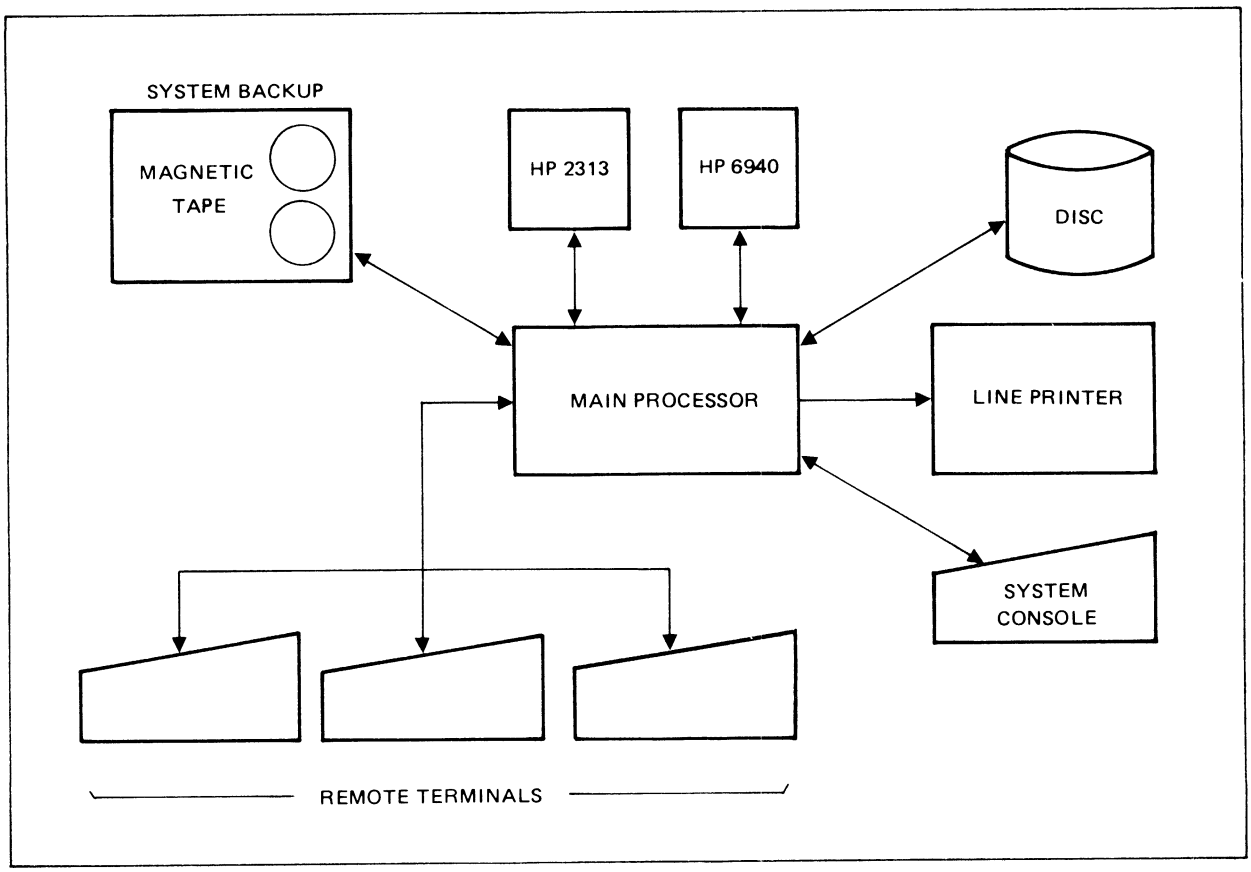


Figure 1-1. Typical System

1-9. SOFTWARE

The BASIC Interpreter is an option which runs under the RTE operating system. It requires a memory partition of about 16K words — 12K for the Interpreter and at least 4K for your BASIC program. If TRAP and Task Scheduling is used, SSGA must be accessed. The subsystem consists of the following modules and components:

- BASIC, the main program and all disc resident segments used for control and I/O.

  Segment 1 - Statement syntax checking.
  Segment 2 - Program and error listing.
  Segment 3 - Pre-execution processing, building symbol tables and intermediate code.
  Segment 4 - Execution of Programs.
  Segment 5 - Command execution.
  Segment 6 - Command execution.
  Segment 7 - Tracing, debugging and subroutine simulation.
  Segment 8 - Execution of PAUSE, STOP, END, ASSIGN, and CHAIN statement and BYE command (closing segment).

  Each segment is loaded from the disc as required by the BASIC main program.

- Branch and Mnemonic Tables, used to link BASIC to subroutines and functions. These tables are binary disc files, not relocatable modules and are created by a separate table generator program, RTETG.

- Disc Resident User-Written subroutines.

- Trap Table Module, used for keeping track of all real-time tasks and traps.

Figure 1-2 illustrates the layout of BASIC in the RTE system memory.

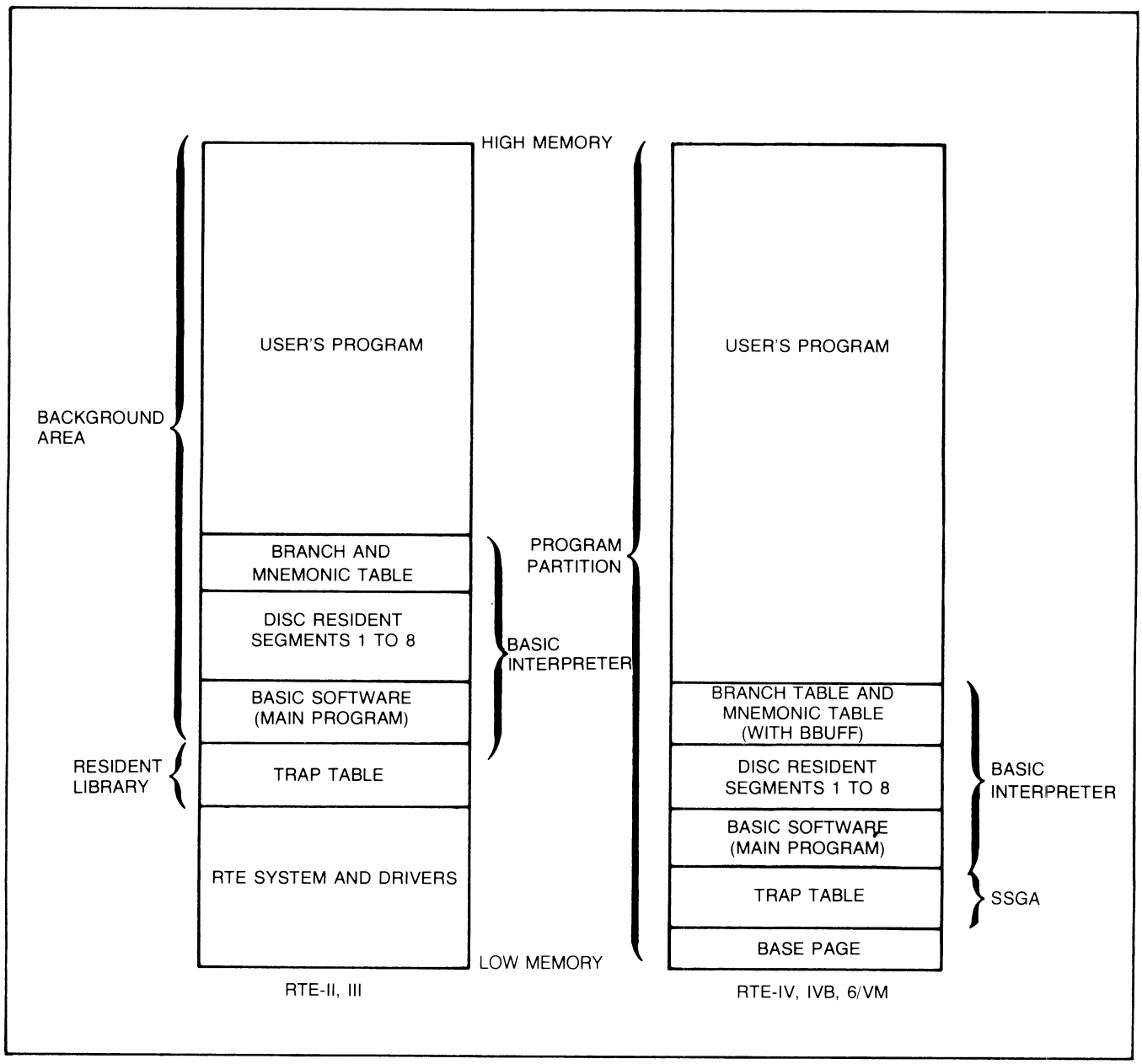


Figure 1-2. RTE Memory Layout with BASIC

3-7a. PRINT USING

PRINT USING statements are used to direct output to specific devices and to format that output. The LU# of the output device is specified in the statement. The format of the data to be output is specified in the format part of the string following PRINT USING, and the data is specified in a print list following the format part.

Format

PRINT [#*lu*] USING format part [;*using list*]

Parameter

lu	an optional numeric constant or variable which may be the LU# of the device, or the type 0 file in the sixth position in the FILES statement. "*lu*" may not be a standard file.
format part	either a format string (represented as a literal string or a string variable) or a statement number referencing an IMAGE statement.
using list	an optional parameter list which may include; numeric expressions, string variables, or print function (TAB), separated by commas. These commas are delimiters only and have no formatting function. Note that string variables, not string expressions, are allowed.

NOTE

Use of the carriage control operators #, + and − on non-terminal devices such as line printers may produce results that are different from those on a terminal. This is caused by mechanical or programatic differences in the hardware or software.

GROUPS. A *group* of one or more *format specifications* may be enclosed in parentheses which must be preceded by a *repetition factor* between 1 and 132 inclusive (e.g., 2(/AX,D/) is equivalent to /AX,D/ //AX,D/). Within the parentheses, the specifications must be separated by commas or slashes and the *group must be set off from other specifications by a comma or slashes, just as if it were a single specification. Groups can be nested two levels deep.*

EXECUTION OF THE PRINT USING STATEMENT. Execution of the PRINT USING statement commences by examining the *format string*. The *carriage control* character, if present, is noted for termination processing, then each *format specification* is examined.

If the specification is either a string or a numeric specification, the next item from the *using list* is printed according to the specification. If the *using list* has already been exhausted or is not present, the statement terminates. If the item does not agree with the specification (i.e., string vs. numeric), an error message is printed and the program execution terminates.

If the specification is literal, the specified number of blanks (or the contents of the literal string) is simply printed; the *using list* is not examined.

If the the end of the *format string* is reached before the end of the *using list,* processing continues from the beginning of the *format string* but after the optional *carriage control* character (if the *format string* contains no string or numeric specifications, the statement terminates).

When all items from the *using list* have been printed the statement terminates (any remaining literal specifications are processed if the end of the *format string* has not been reached for the first time). Termination consists of printing, carriage return, and linefeed, modified by the *carriage control* character.

If the *format string* is empty or contains only blanks, output consists of only a carriage return, and linefeed.

Examples

```
10      PRINT #6 USING 100A$, (2+X), B$
        .
        .
        .
100     IMAGE 3DX,DD.D//SD.DDE

50      PRINT #17 USING "#,3(DD.D2X)";Z1,Z2,Z3
```

See Chapter 19 for additional information on formatted output.

When printing to a type 0 file representing the line printer, a blank or control character must follow the " in the format string because control characters are in the first column for line printers (e.g., IMAGE " THIS IS A LINE",5A).

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. A maximum of 80 characters can be input.

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. Failure to include commas between input items will result in corrupt input data.

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^) or QQ. If you are running from a multi-point terminal, type QQ only.

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 transferring to another program with CHAIN. COM areas must be equal in size for CHAINed or INVOKE'd programs so that common will remain properly aligned.

Format

COM *variable list*

Parameter *variable list* list of string or array variables

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 Multi-User Real-Time 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. x is a dummy argument.

INT(x) The integer function, INT, provides the truncated value of x; $x < 32767$.

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 argument x may have any value. A sequence of random numbers is repeatable upon each run if the argument is positive. A random sequence can be achieved upon each run by two RND calls. The first call is issued in the negative value for x, called a seed, followed by another call with a positive x. If the same seed is used for each run, the same random sequence is repeated. Also, the same random sequence is restarted after BREAK/RESUME commands are given during debugging.

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*. *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*. *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 determines the type of the next data item in the specified file. The three possible reponses are: 1 = next item is a number, 2 = next item is a character string, 3 = next item is "end of file", 4 = next item is "end of record". If *x* is zero, the TYP function references the DATA statements and returns the following response: 1 = number, 2 = string, 3 = "out of data" condition.

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.

6-3a. INVOKE

The INVOKE statement is used to schedule a second BASIC program from a calling program. The called program may also call another program and so on. When the currently executing, called program terminates, control is returned to its calling program.

Format

INVOKE { *string variable* / *string literal* } [,*statement number label*]

The *string variable* or *literal* is the name of a Real-Time BASIC program or the type 6 RTE program that has been saved with the File Manager 'SP' command. This may be a fully qualified file name (see Section VII, Files). If the optional *statement number label* is present, execution begins at the first executable statement at or after the label; the exact label need not be present in the called program. If omitted, execution begins at the first executable statement in the called program.

INVOKE calls the program identified by the string expression, and it stores the current program on disc. When the program called by INVOKE finishes execution, it terminates and automatically returns to the calling program. The called program may call another program, including the original calling program, with the INVOKE statement.

Basic data files remain open when one program INVOKES another.

Only variables declared in a COM statement are saved during a INVOKE operation. All variables and arrays of the current program that were not declared in COM are not available to the new program when the new program begins execution. All programs must contain the same size COM area and the same number of file positions in the FILES statements so that common will be properly aligned. Files declared in INVOKE'd programs correspond to files declared in the main program. Logical units cannot be defined in a INVOKE'd program unless it is already declared in the main program's FILES statement. Refer to the FILES statement, Section 7-3, for examples.

If the programs are CSAVEd, the time required to execute the INVOKE statement is reduced.

Any TRAPS previously enabled will remain enabled.

The calling program and its own local variables are saved on the disc on program tracks in the system area of the disc. The levels deep of INVOKE is determined by the size of the calling program and the number of available system tracks. When the available system tracks are used up, the message "BASIC WAITING TRACKS" is printed. The copy of Multi-User BASIC being used by the operator is disc track suspended until system tracks become available, or the copy of Multi-User BASIC being used by the operator is terminated with an OF command (this is not recommended).

When BASIC invokes a program, the LU's of your terminal and list device are passed. If the devices are interactive, the echo bit (K-bit) is set.

The non-BASIC program that is INVOKE'd must exist as a type six file for BASIC to schedule it. This means that after the program to be INVOKE'd is loaded, it must be SP'd with the File Manager SP command. For example, if EXMPL is the non-BASIC program to be INVOKE'd first load the program then type,

```
:SP, EXMPL
```

EXMPL is now stored as a type six file. BASIC first 'opens' the INVOKE'd program to determine its type. This type could be ASCII (type three or four) and BASIC assumes a BASIC program is to be INVOKE'd. If it is type six, then BASIC schedules the non-BASIC program.

When the user breaks the INVOKE'd program with the system break command (*BR,BASIC), BASIC returns control to the original program.

SECTION VII
FILES

For situations that require permanent data storage external to a particular program, Real-Time BASIC provides a data file capability. This capability allows flexible direct manipulation of large volumes of data stored in files.

The simplest approach to files is to treat them as serial storage devices. Visualize a file as a list of data items, ordered serially. You can read the data in a file and write data to a file with your programs quite easily without worrying about the internal structure of the file. Several programs may access the same file along with yours. Each program uses its own data pointer to mark its position in the file, and functions independently of the other programs.

You may also envision files as structured data bases, internally organized as a collection of records — each record consisting of 128 16-bit words. Thus each record of a file may hold up to 64 numerical quantities. A string data item occupies $1 + (n+1)/2$ words where n is its length in characters.

To use a file you should be familiar with the CREATE and PURGE commands and with the statements listed below:

- FILES
- READ#
- PRINT#
- ASSIGN
- IF END

These commands and statements as applicable to files are defined in the remainder of this section.

7-1. FILE CHARACTERISTICS

In order to create and use files, you must understand the following characteristics of Real-Time BASIC files. The conventions for file creation are the same as RTE File Manager conventions.

- A file name may contain from 1 to 6 characters. The first character may not be a number. Leading and trailing blanks are ignored. Embedded blanks are not allowed. Any printable ASCII character except the plus (+), hyphen or minus (−), comma (,), and colon (:) may be used.

- A file may be assigned a security code to control read/write access. The security code may be a number between −32767 and +32767. A positive code write protects the file. When accessing the file, you may supply a positive or negative version of the positive security code in order to write on the file. A negative security code both read and write protects the file. You must provide the negative code to read or write on a file protected by a negative code. If you do not want to protect the file, assign a zero security code.

  Two ASCII characters may be used in lieu of a positive security code. The first character may not be a number.

- Each file is assigned a type number. For a complete description of all file types, see the Batch-Spool Monitor Reference Manual. The types you will be using with BASIC are: type 0, type 1, type 4, and type 10.

  A type 0 file defines an I/O device. You must create type 0 files with the File Manager CR command. After you create a type 0 file, you can use the file name to reference the device it defines.

A type 1 file contains data. You must create this type of file with a Real-Time BASIC CREATE command if you are using it with BASIC programs. In RTE-IVB and 6/VM type 1 files are extendable. BASIC does not recognize type 1 file extents. Extended files should be stored into another file without extents (e.g., :ST,BIG:::1,BIGG:::1:-1,BN) before running BASIC.

A type 4 file is created when you SAVE a program. You may also create a type 4 file with the File Manager or Interactive Editor and store source programs or commands in it.

A type 10 file is created when you CSAVE a program.

- When you create or access a file, you can specify the cartridge reference. The cartridge reference can be a positive integer corresponding to the label of a currently mounted cartridge or a negative logical unit number referencing a disc. The file will be created on or accessed from the specified cartridge. If you specify a zero, the cartridges are accessed in the order in which they appear in the File Manager Cartridge Directory.

NOTE

When obtaining input from devices via type 0 files, you must ensure that the data is of the correct file type. BASIC cannot determine if a type 0 file is of type 3, 4, or 10 format while it reads the records into memory. Invalid information may corrupt the Interpreter's own internal buffers and cause BASIC to execute improperly. This attention to file formats is especially necessary for the RUN, LOAD, and MERGE commands and the CHAIN statement.

7-2. CREATE AND PURGE

The CREATE command is used to create a file for use by a program and the PURGE command is used to remove a file. These commands are described in Section IX, paragraphs 9-6 and 9-7.

7-3. FILES STATEMENT

Every file that is to be accessed by a program must be identified in the program's FILE statement.

Format

FILES *filename*$_1$ [, *filename*$_2$, . . . *filename*$_n$ [:*security*[:*cartridge*]]]

Parameters

filename$_n$	name of file to be referred to by number corresponding to position in FILES statement, or an asterisk indicating file will be assigned later or zero indicating position refers to a logical unit.
security	optional security code which may be supplied with each filename.
cartridge	optional cartridge reference which may be supplied with each filename (either positive cartridge reference number or negative LU number).

The FILES statement declares which files will be used in a program. Up to four FILES statements can appear in a program, but only 16 files total can be declared. The files are assigned numbers (from 1 to 16) in the order in which they are declared in the program. In the examples below MATH is file #1 and #9, FILE27 is #7, and DATA is #10. File position #3 will be assigned to a filename with the ASSIGN statement. File position #4 is specified as zero to indicate LU4.

Example

```
75 READ #1,3; A, B1, C(I,J+2)
80 READ # I,J
```

Positions file pointer to record J without reading any data.

7-9. SERIAL FILE PRINT STATEMENT

The serial file PRINT # statement writes data items on a file, starting at the current position of the pointer. The items may be numeric or string expressions.

Format

PRINT # $\begin{matrix} file\ number \\ lu \end{matrix}$; *print list* [, END]

Parameters

file number	a number, variable, or expression whose value is between 1 and 16, indicating a file's position in the FILES statement.
lu	if a file position corresponding to the *file number* does not exist in the program or the file position contains a zero (i.e. FILES 0,FILA), the number is interpreted as a logical unit number (*lu*).
print list	series of numeric expressions, numeric or string variables, or string literals.
END	optional constant which prints EOF on the file.

The PRINT # statement performs essentially the same operation as the ordinary PRINT statement, except that data is written to a file or logical unit rather than to a terminal. No line formatting of files 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). When printing to the line printer using PRINT #6, BASIC inserts a leading space in column 1 when not a type 0 file. The PRINT USING (Section 3-7a) or PRINT to a type 0 file should be used to control the line printer.

Writing of the first value begins wherever the file pointer is positioned and new records are used as necessary. Writing onto a file overlays whatever may have been in that area, including end-of-file marks. To ensure that a file actually ends with the last item written, the special constant END may be placed at the end of the print list. END is significant only when it is the last item written on a file.

The END constant should not be used when the PRINT# statement refers to an LU. The interpreter cannot distinguish the difference between a file number or an LU when the line is evaluated by the parsing routine. Therefore, a run-time error will occur.

If printing is attempted beyond the physical end of the file, an end-of-file condition occurs. The IF END statement can be used to specify action in this case, or the program will terminate.

Since character strings vary in length and each string must be wholly contained within a record, some space in each record may be left unused. You can calculate the number of words occupied by any string with the formula described in paragraph 1-6.

If the file named in the FILES statement is a type 0 file, the data is printed to the device corresponding to the file. See Section IX for more information about type 0 files.

You should not do a serial read after PRINT without resetting the pointer. For information about modifying records, see paragraph 7-12.

```
20 FILES FILA
30 PRINT #1; A,"STRING",A$
```
The value of A, the string "STRING", and A$ are printed on FILA.

```
20 FILES LINEP
   .
   .
   .
60 PRINT #1; "SUMMARY"
```
LINEP is a type 0 file corresponding to the line printer.

The string literal SUMMARY is printed on the line printer.

```
100 FILES FILEA,FILB,FILC,0,FILD
      .
      .
      .
150 PRINT #4; A,B,C
```
The variables A, B, C are punched on the paper tape punch, LU 4.

```
200 PRINT#(I+J);  2,42,A,B,C,D(3,5),END
```
The items are printed on the file in position I+J followed by an end-of-file mark.

To advance a page (top-of-form) on a line printer, set up a type 0 file to the line printer as follows:

```
10 FILES 0,LP
20 PRINT #2;"1"
30 END
```

where: LP is a type 0 file for the line printer.

7-10. PRINTING A RECORD

The PRINT # statement may also be used to write items to a designated record.

Format

PRINT # *file number, record number; print list*

Parameters

file number	see paragraph 7-9.
record number	a number, variable, or expression indicating on which record the list is to be printed. Ignored if *file number* refers to LU number or type 0 file.
print list	see paragraph 7-9.

Execution of a print record statement performs the same task as serial PRINT with the exception that the operation is limited to one record. The entire list of data must fit in one record. An attempt to write more than one record or more than the record can hold results in an end-of-file condition. The write begins at the beginning of the record, which is scratched of previous data. Again, the use of the IF END statement will avoid program termination at the end of the write.

OPERATOR COMMANDS

SECTION IX

This section describes the Multi-User Real-Time 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 Multi-User Real-Time BASIC. Detailed explanations of most of the commands are provided in the remainder of the section. Additional commands used in specific situations such as program debugging are introduced and explained in subsequent sections. The TABLES command is described in Section VIII. A complete list of the commands and their uses is provided in Appendix A.

File characteristics (filename, security, cartridge) are described in Section 7-1.

Table 9-1. Operator Commands

COMMAND	FUNCTION
LOAD	Loads a source program or a semi-compiled program from a file.
SAVE	Stores a program on disc as a source program.
CSAVE	Stores a program on disc in semi-compiled format.
MERGE	Merges a source program with a program in memory.
REPLACE	Replaces a source program on disc.
DELETE	Deletes a program from memory.
CREATE	Creates a data file on a device.
PURGE	Deletes a program or file from disc.
RENAME	Removes the name of a file on the disc and replaces it with a new name.
RESEQ	Renumbers the statements in a program.
RUN	Loads and executes a program.
LOCK	Locks a peripheral device to your program.
UNLOCK	Unlocks a locked device.
BYE	Terminates the use of BASIC.
LIST	Lists a program onto a file.
*BR,BASIC	Breaks execution of a program.
CALLS	Lists all of the mnemonics in the current Mnemonics Table.
TABLES	Specifies Branch and Mnemonic Table names. (Described in Section VIII also.)
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.*

*These commands are described in Section XIII.

NOTE: When obtaining input from devices via type 0 files, you must ensure that the data is of the correct file type. BASIC cannot determine if a type 0 file is of type 3, 4, or 10 format while it reads the records into memory. Invalid information may corrupt the Interpreter's own internal buffers and cause BASIC to execute improperly. This attention to file formats is especially necessary for the RUN, LOAD and MERGE commands and the CHAIN statement.

9-1. LOAD

The LOAD command enables you to load all or a portion of a source program or a semi-compiled program from a specified file.

Format

LOAD [*limits*] [*filename* [:*security* [:*cartridge*]]]

Parameters

limits	beginning and ending line numbers of the portion of the program you want loaded separated by a comma. Limits are omitted if the entire program is to be loaded.
filename	name of file containing the program or type 0 file corresponding to a device from which the program is to be loaded. The default is LU 5 or the LU number specified as the input parameter in the RUN,BASIC command.
security	optional security code. Must use if program saved with security code.
cartridge	optional cartridge reference (label or LU number).

The LOAD command reads in all program statements between and including the line numbers specified as limits. If limits are not specified, the entire program is loaded. Once loaded, a program is ready for execution or editing.

Examples

```
>LOAD                    Loads from LU 5 by default.
>LOAD 150,250 CARD       Loads from a file named CARD.
```

When a syntax error is encountered during the loading of a program, BASIC prints an error and returns to the command mode.

The correct procedure for recovering from syntax errors during LOAD is illustrated by the following example:

Given the file TEST::1132

```
10 LET A=1
20 LTE B=A
30 LET C=B
99 END
```

The load procedure is:

```
>LOAD TEST::1132
MISSING ASSIGNMENT OPERATOR IN LINE 20
    20 LTE B=A
>20 LET B=A
>MERGE TEST::1132,21

BASIC READY
>LIST
10 LET A=1
20 LET B=A
30 LET C=B
99 END
```

9-2. SAVE/CSAVE

The SAVE command stores the BASIC program currently in memory on the disc. CSAVE saves a program in semi-compiled form for faster loading.

Format

SAVE [*limits*] [*filename* [:*security* [:*cartridge*]]]
CSAVE [*filename* [:*security* [:*cartridge*]]]

Parameters

limits	beginning and ending line numbers of the program to be saved. If specified, limits must be separated by a comma. If no limits are specified, the entire program currently in memory is saved.
filename	name of the file in which the program is to be saved. If the file already exists, an error message is printed. If the file does not exist, it is created and the program is saved. The file is closed when the operation is completed. If no *filename* is specified, the program will be output to LU 4 (for SAVE only) or the LU number specified as the output parameter in the RUN,BASIC command.
security	optional security code.
cartridge	optional cartridge reference (label or LU number).

The SAVE and CSAVE commands store program statements on the disc. The SAVE command may store only the statements between and including the line numbers specified. If limits are not specified, the entire program is stored.

A SAVEd program can be edited with the Interactive Editor and can also be loaded by BASIC. CSAVE is used for faster loading and particularly where several large programs are CHAINed together. A CSAVEd program should always be backed up by a SAVEd version because it is in binary format. If any modification of the Interpreter or a new system generation takes place, the CSAVEd program becomes unloadable by the new version of BASIC or the RTE system. CSAVE is used generally for production programs where no modification is being made and speed of loading and chaining is important. A program cannot be CSAVEd to an LU or type 0 file.

A CSAVE file is a type 10 file with records possibly being greater than 128 words, each. If you wish to create another copy of a CSAVE file, you must use the CSAVE command. The RTE File Manager does not store or copy files with records greater than 128 words.

Rules for naming files and file characteristics are given in paragraph 7-1.

Once stored, the program can be loaded and executed as needed.

When a program containing external subroutines is CSAVEd, the names of the Branch and Mnemonic Tables are stored with the CSAVEd program. Therefore, if there is more than one copy of that program, be aware that each version is related to a different set of Branch and Mnemonic Tables and overlays.

Examples

>SAVE	*The current source program is output on LU 4.*
>SAVE 100,260	*Lines 100 through 260 of the current program are output to LU 4.*
>SAVE PROGA	*The current program is saved in file PROGA.*
>CSAVE PROGB	*Semi-compiles and saves program in file PROGB.*
>CSAVE	*Output semi-compiled form of current program to LU 4.*

9-3. MERGE

The MERGE command merges a source program or portion of a source program with a program in memory.

Format

MERGE [*limits*] [*filename* [:*security* [:*cartridge*]]]

Parameters

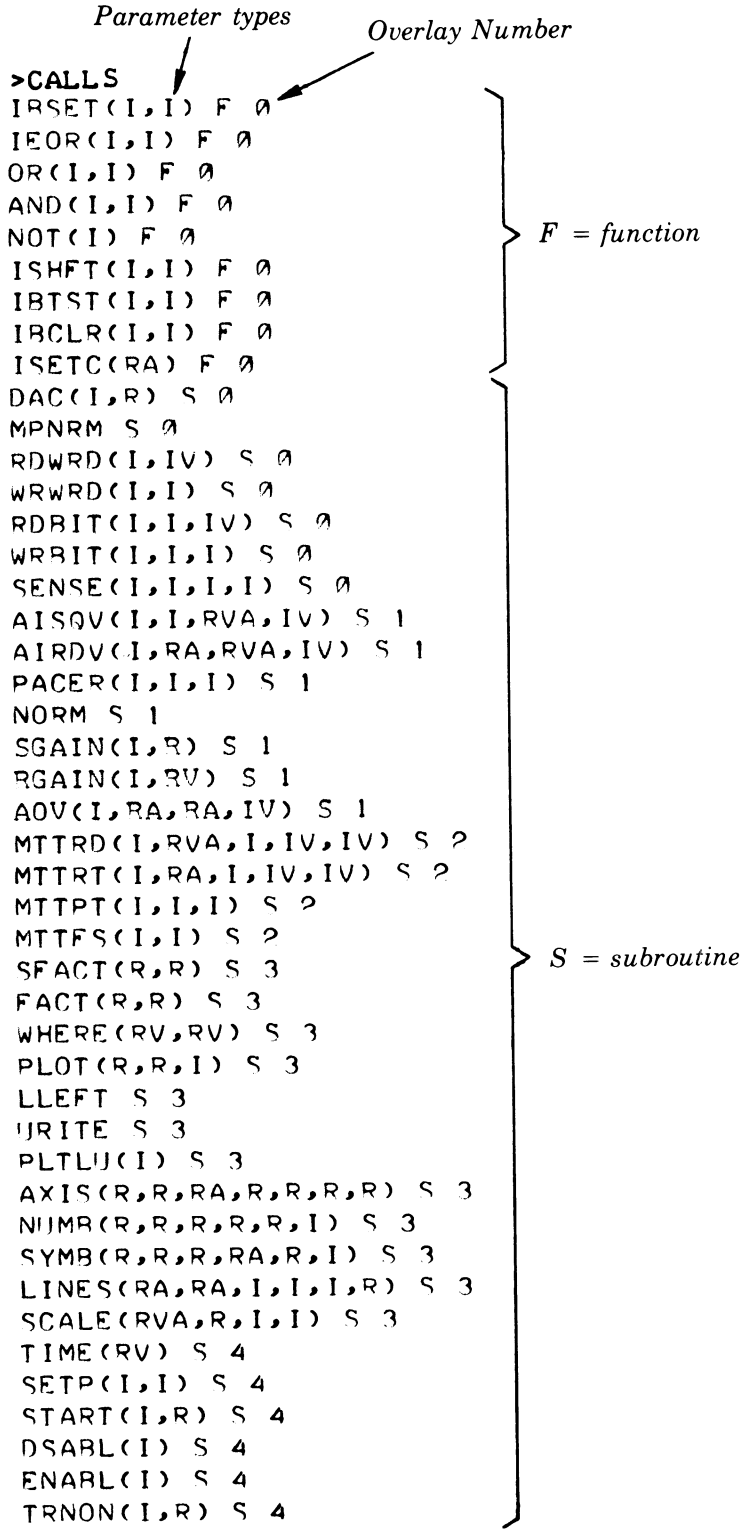
limits	beginning and ending line numbers of the program to be merged. Limits must be separated by a comma. If no limits are specified the entire program is merged.
filename	name of the program to be merged. Program name may be omitted, if program is to be loaded from the standard input device (LU 5).
security	optional security code. Must be supplied if program saved with security code.
cartridge	cartridge reference (label or LU number).

MERGE merges a BASIC source program with a program in memory. If any line numbers of the program being entered match those of the program in memory, the like numbered statements of the program in memory will not be replaced. MERGE is useful if you want to load a program with syntax errors. You can retype the statements which are in error and edit them, if necessary, and then merge in the rest of the program. The corrected statements will not be replaced by the ones in error.

Examples

>MERGE	*Merge source statements from LU 5.*
>MERGE 120,190	*Merge source statements numbered 120 through 190 from LU 5.*
>MERGE PROGA	*Merge source statements from file named PROGA.*

Example


Parameter types
Overlay Number
>CALLS
IBSET(I,I) F 0
IEOR(I,I) F 0
OR(I,I) F 0
AND(I,I) F 0
NOT(I) F 0
ISHFT(I,I) F 0
IBTST(I,I) F 0
IBCLR(I,I) F 0
ISETC(RA) F 0
DAC(I,R) S 0
MPNRM S 0
RDWRD(I,IV) S 0
WRWRD(I,I) S 0
RDBIT(I,I,IV) S 0
WRBIT(I,I,I) S 0
SENSE(I,I,I,I) S 0
AISQV(I,I,RVA,IV) S 1
AIRDV(I,RA,RVA,IV) S 1
PACER(I,I,I) S 1
NORM S 1
SGAIN(I,R) S 1
RGAIN(I,RV) S 1
AOV(I,RA,RA,IV) S 1
MTTRD(I,RVA,I,IV,IV) S 2
MTTRT(I,RA,I,IV,IV) S 2
MTTPT(I,I,I) S 2
MTTFS(I,I) S 2
SFACT(R,R) S 3
FACT(R,R) S 3
WHERE(RV,RV) S 3
PLOT(R,R,I) S 3
LLEFT S 3
WRITE S 3
PLTLU(I) S 3
AXIS(R,R,RA,R,R,R,R) S 3
NUMB(R,R,R,R,R,I) S 3
SYMB(R,R,R,RA,R,I) S 3
LINES(RA,RA,I,I,I,R) S 3
SCALE(RVA,R,I,I) S 3
TIME(RV) S 4
SETP(I,I) S 4
START(I,R) S 4
DSABL(I) S 4
ENABL(I) S 4
TRNON(I,R) S 4
F = function
S = subroutine

9-16. TABLES

The TABLES command specifies which Branch and Mnemonic tables to use. This command is necessary if you intend to use external subroutines written in other languages. The Branch and Mnemonic Tables are created when running the BASIC Table Generator described in Section 14.

Format

TABLES *branch filename* [:*SC*[:*CR*]], *mnemonic filename* [:*SC*[:*CR*]]

branch filename	is the name of the file, created by the RTE BASIC Table Generator,which contains subroutine branch information.
mnemonic filename	is the name of the file, created by the RTE BASIC Table Generator, which contains subroutine mnemonics.
SC	is an optional security code parameter.
CR	is an optional positive cartridge number or negative logical unit number.

The branch and mnemonic files must be unique to each copy of BASIC. If two BASIC programs try to access the same set of branch and mnemonic files, the error message "FILE CURRENTLY OPEN OR EXCLUSIVE OR LOCK REJECTED" is displayed.

Format

BREAK *statement number label[,statement number label,....statement number label]*
UNBREAK *statement number label[,statement number label,....statement number label]*
BREAK

Parameters

statement number label	execution suspends just before this statement. At least one number must be specified, as many as four may be specified separated by commas.

Once a breakpoint is reached, execution can only be resumed by entering a RESUME command. BREAK prompts for commands with two greater than (>) symbols.

To eliminate all breakpoints, enter the command UNBREAK prior to resuming program execution.

To eliminate specific breakpoints, enter the command UNBREAK and the line numbers of the breakpoints to be deleted.

To obtain a list of currently set breakpoints, enter BREAK with no parameters.

Legal commands that may be entered only during a break are: ABORT, RESUME, SET, and SHOW. Commands that may be entered during a break as well as other times include: BREAK/UNBREAK, CALLS, SIM/UNSIM, and TRACE/UNTRACE. If you enter any other command during a break, the system issues an error message. If you enter more than four statement number labels without using UNBREAK, an error message is printed.

Examples

```
>BREAK 30,70,90,100          Breakpoints are defined for lines 30,70,90, and
>RUN                         100.
<BREAK 30                    Breakpoint 30 is reached.
 >RESUME                     Resume execution by typing RESUME.
 BREAK 70                    Breakpoint 70 is reached.
 >UNBREAK 90                 Delete breakpoint 90.
 >RESUME                     Resume execution.
 BREAK 100                   Breakpoint 100 is reached.
 >
 BREAK                       Indicates breakpoints are set at statements 30, 90,
  30,0,90,100                and 100.
```

10-3. RESUME

The RESUME command terminates current debugging activity and resumes execution of the suspended program. This command may be entered only during a break period, or after a subroutine simulation suspension.

Format

RESUME
RES

There are no attendant parameters to this command.

The RESUME command restarts the program at the location printed when the break occurred. You may abbreviate the command to RES.

The random number sequence generated by the RND function (see Section V) is restarted upon typing RESUME. You may wish to avoid setting breakpoints around statements affected by random numbers.

Example

>BREAK 25,100	*Set breakpoints 25 and 100.*
*BREAK 25	*Breakpoint 25 is reached, program is suspended.*
>>RESUME	*Type RESUME command to resume execution of program.*
*BREAK 100	*Breakpoint 100 is reached.*

10-4. ABORT

The ABORT command terminates a suspended program and returns the BASIC Interpreter to a non-executing program state. ABORT may be entered only during a break period or following a subroutine simulation suspension.

Format

ABORT

There are no attendant parameters for this command.

When ABORT is entered, the break period is ended and the run terminated. You may now enter any command legal during normal BASIC operation, but cannot enter commands legal only during a break period. Break points and trace are unchanged.

Example

>BREAK 25,100	*Set breakpoints 25 and 100.*
*BREAK 25	*Breakpoint reached, execution suspended.*
>>ABORT	*Type ABORT to terminate program execution.*
BASIC READY	*BASIC is ready for next command.*
>	

SUBROUTINE TABLE GENERATION

SECTION XIV

In order to call external subroutines and functions written in BASIC, FORTRAN, or Assembly language, you must use the RTE Table Generator to define and generate two tables, the Branch and Mnemonic Tables, and create overlays which contain the actual subroutines.

When a specific subroutine is called, the module or overlay is loaded from the disc into a partition of memory. Remember that BASIC resides in another partition. Therefore, BASIC and the overlay reside in memory concurrently, at least initially. After control is passed to the subroutine, the BASIC Interpreter may be swapped out to the disc if necessary. All of the information that allows BASIC to access the correct overlay is contained in the Branch and Mnemonic Tables. Additional information is contained in the overlay which is used to pass control to the correct subroutine in the overlay.

Figure 14-1 illustrates the relationship of the BASIC Interpreter and a subroutine overlay in memory.

BASIC uses the Branch and Mnemonic Tables to transfer program execution from BASIC to the subroutine or function and back. It converts parameters and locates the subroutine overlay and the subroutine within the overlay. First, BASIC looks in the Branch and Mnemonic Tables to locate the proper overlay and to determine how parameters are to be passed. The parameters are stored in System Available Memory (SAM), and the overlay is scheduled. The operating system suspends BASIC and activates the overlay. The overlay obtains the parameters from SAM, stores them in the area behind the overlay, and transfers to the required subroutine. These steps are reversed when parameters and control are returned to BASIC. The complete process is detailed in Figure 14-1A.

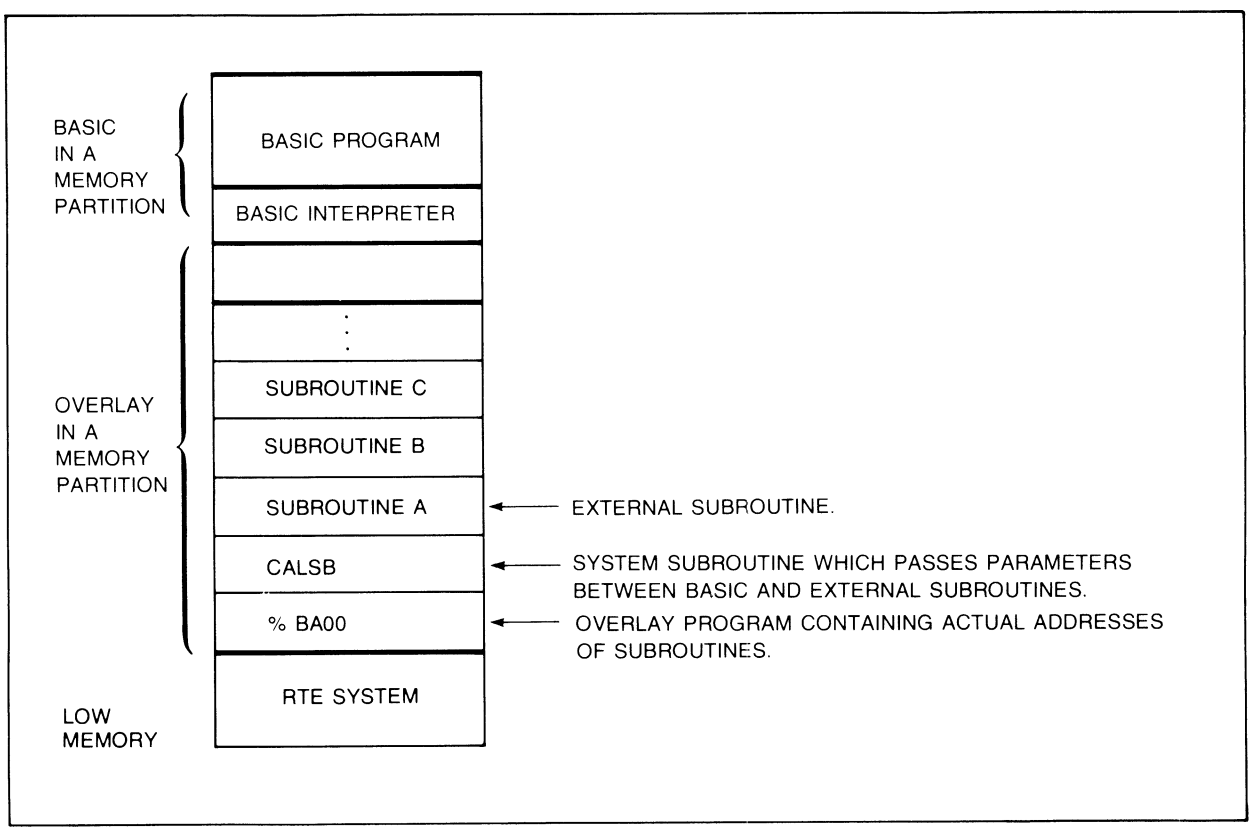
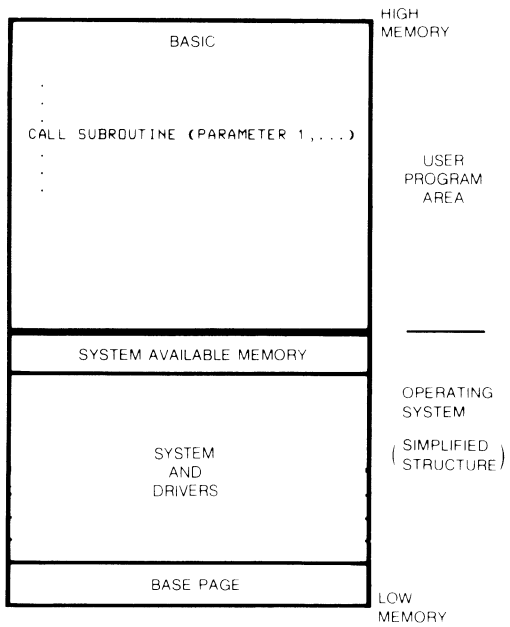
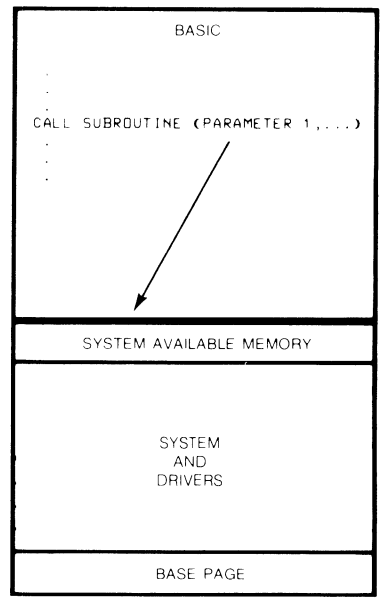


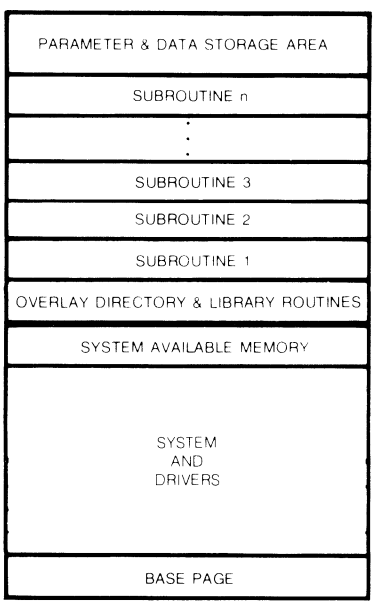
Figure 14-1. BASIC and an Overlay in Memory



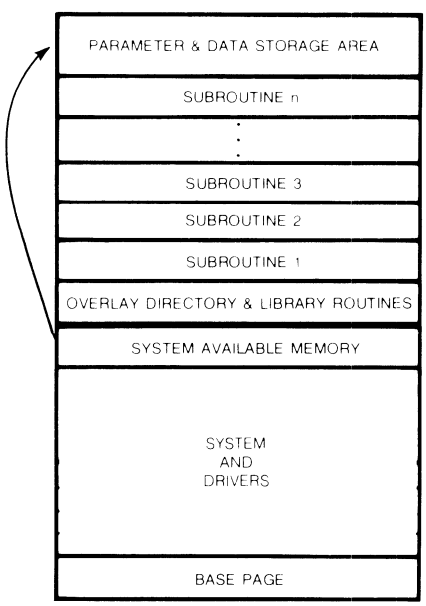
1 When a subroutine or FUNCTION subprogram is encountered in BASIC, the following procedure occurs.



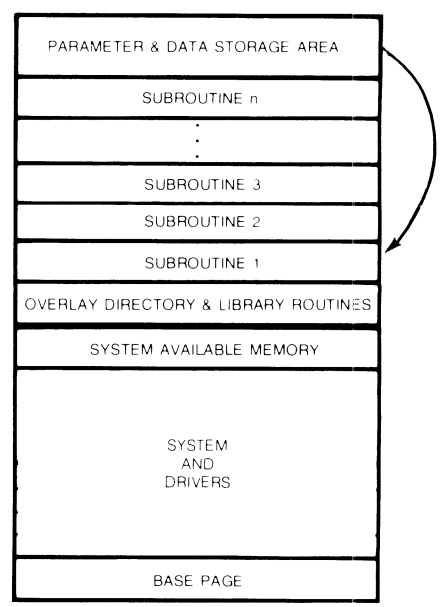
2 The subroutine call or FUNCTION subprogram is referenced in the Branch and Mnemonic Tables to determine which overlay is required, and what format the parameters are expected to be. The parameters are then stored in System Available Memory (S.A.M.) in the required format.



3. The overlay containing the subroutine or FUNCTION subprogram is brought into memory. BASIC may be swapped out to the disc.

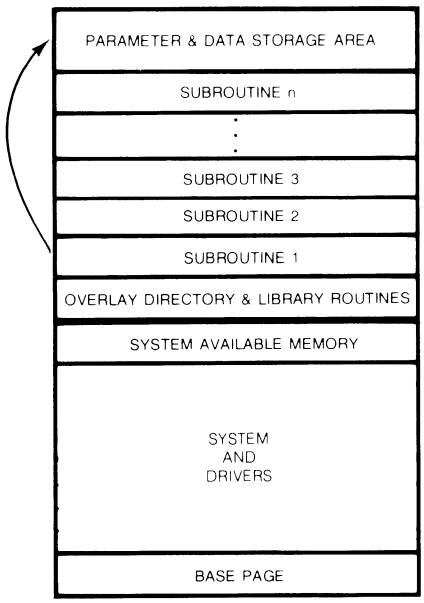


4 The parameters stored in S.A.M. are copied to a parameter storage area in the overlay.

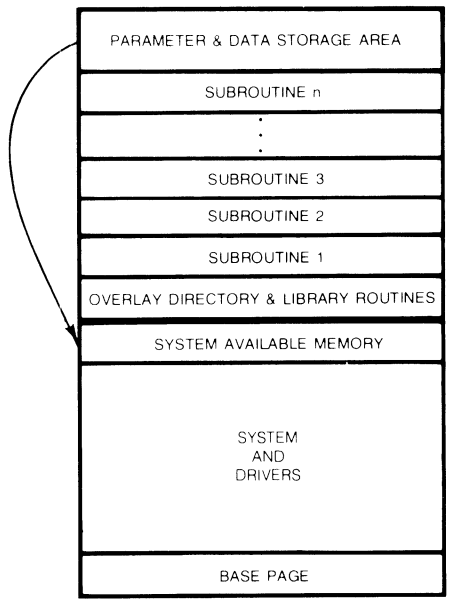


5 The parameters are handled by the subroutine.

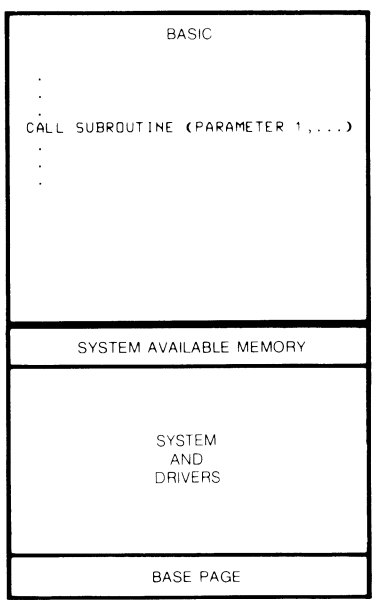
Figure 14-1A. The "Ten Steps" Performed When a Subroutine or Function Subprogram is Called by BASIC



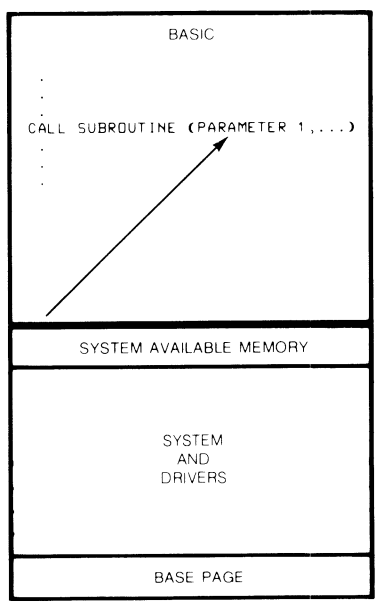
6. Any information to be returned to the BASIC program is stored back in the parameter storage area.



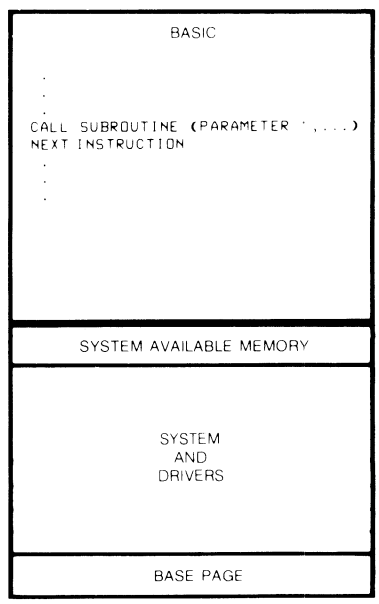
7. The parameters stored in the parameter storage area of the overlay are copied back into S.A.M.



8. Control is returned to BASIC. If BASIC was swapped out, it is brought back into memory.



9. The parameters stored in S.A.M. are copied back to the respective parameters in the BASIC program.



10. The next BASIC instruction line is executed.

Figure 14-1A. The "Ten Steps" Performed When a Subroutine or Function Subprogram is Called by BASIC (continued)

14-1. RTETG

The RTE Table Generator (RTETG) operates in batch mode. You can provide the description of a subroutine or function from a type 3 or 4 disc file or type 0 file only (for non-interactive devices only, such as cards, paper tape, or magnetic tape).

You specify:

- the name of the subroutine or function as it is called in your BASIC program,
- the type of each parameter (integer, real, or array) used by the routine,
- the entry point name of the routine if different from the name as used in your BASIC program,
- the name of the file containing the relocatable subroutine,
- overlay groupings.

RTETG processes this information and produces the Branch Table, the Mnemonic Table, relocatable overlay file, and a FMGR transfer file which is used to load the overlays. Each overlay is a relocatable program which allows execution to be transferred to the subroutine or function when it is called. When the overlay is loaded, it is linked to all routines defined as part of the overlay.

Each overlay is named and numbered as %Bxnn, which is the relocatable file name and the program name. x is the overlay identification letter from A-Z and nn is from 00-31. The maximum number of overlays handled at one time by RTETG and BASIC is 32. For example, the fourth overlay with ID letter A would be %BA04. A single overlay may be used for up to 64 subroutines. For efficiency you should arrange to have all subroutines and functions which are likely to be used consecutively in the same overlay. For example, the nine bit manipulation functions from %BASLB can be combined into one overlay. At the same time, do not lump everything into one overlay, or it will not fit into a memory partition.

As many as 26 pairs of tables and sets of overlays may be defined for each version of BASIC on a system. Each time you run BASIC, you specify which tables you are using. In this way, you can optimize efficiency by setting up tables which are appropriate for your program. Overlays and their corresponding Branch and Mnemonic Tables cannot be shared by another copy of BASIC.

Refer to Appendix D for information on loading RTETG.

14-2. SCHEDULING RTETG

When you schedule RTETG you must specify the name of a file where the commands defining your subroutines are to be found. The file may be a type 3 or 4 File Manager file or a type 0 file corresponding to a non-interactive input device such as the card reader or a terminal. You may optionally specify the logical unit number of the list device where commands will be printed.

The command to start RTETG is:

:RUN,RTETG,*filename* [:*security* [:*cartridge*]] [,*list*]

or

*RUN,RTETG,*filename* [:*security* [:*cartridge*]] [,*list*]

filename	name of a file. (Cannot specify a logical unit number.)
security	optional security code. Must be specified if file was created with a security code.
cartridge	optional cartridge reference (label or negative LU).
list	is the logical unit number of the list device. Default is the user's terminal.

The RTETG commands are described below.

14-3. THE FIRST RTETG COMMAND

The first command you use must specify the files to be created for the Branch Table, the Mnemonic Table, the transfer file, and an identification letter to identify the set of overlay to be produced. The command is:

brt [:*sc* [:*cr*]] ,*mnt* [:*sc* [:*cr*]] ,*trf* [:*sc* [:*cr*]] ,ID=*i* [,*prior* [,*sec* [,*cref*]]]

brt is the name to be given to the Branch Table file

sc is the security code to be assigned to the file it follows. The value may be an integer between −32767 and +32767 or two ASCII characters. Default equals 0, no security code. The first ASCII character may not be a number

cr is the cartridge reference number. A positive integer between 1 and 32767 is the label of the cartridge on which you want the file to reside. A negative number is the logical unit number of the cartridge. Default equals 0, use any cartridge.

mnt is the name to be given to the Mnemonic Table file.

trf is the name to be given to the transfer file.

ID=*i* indicates the upper case alphabetic character (A-Z) to identify your set of overlays since there may be more than one set of overlays defined in your RTE system. The ID= must always precede the *i* parameter.

prior is the priority assigned to the overlay program when it is scheduled. Default is 80.

sec is the security code to be used with the overlay relocatable files. The permissible values are the same as for the *sc* parameter.

cref is the cartridge reference number to be used with the overlay relocatable files. Default is 0, no cartridge reference. The permissible values are the same as for the *cr* parameter.

The three file names may be any legal File Manager file names. All files must be unique and not exist beforehand.

14-4. OTHER RTETG COMMANDS

For each subroutine or function you want to call from a BASIC program, you must use the following command. If a command takes more than one line, you must continue to enter parameters until an automatic end of line wraparound to the next line takes place. No carriage return is acceptable at the end of the first line.

name [(*p1,p2, . . . ,pn*)],OV=*nn* [,SZ=*mm*] [,SS] [INTG / REAL] [FP / FT / BP / BT] [,VL] [,ENT=*epoint*] [,FIL=*fname* [:*sc*[:*cr*]]]

name is the subroutine or function name which may be from 1 to 5 characters.

(p1,p2, . . . ,*pn*) is a list of 1 to 15 parameter types to be passed to the subroutine if it requires them. The list must be enclosed in parenthesis. If there are no parameters, do not specify any and do not put any parenthesis. *pn* equals:

[I / R] [V] [A]

(I=integer, R=real, V=value is returned from the subroutine, A=array variable) (1 < = *n* < = 15). For example, SUBX(I) or SUBZ(IV,RVA). To pass a string variable, RA or RVA should be specified. Note that the simple variable A is different from Array (A). Default is R for real. Refer to Section II for a description of variables.

From this point, the following parameters may be specified in any order.

OV=*nn* indicates the overlay in which the subroutine will reside. The OV = must always precede *nn*. (0 < = *nn* < = 31) overlays must be specified in ascending order.

SZ=*mm* indicates the number of pages required for the overlay. This parameter is used for all systems except RTE-II and need appear only once. If it appears more than once, the largest value supplied is used and a warning is printed. (1 < = *mm* < = 32). Specify at least one page more than the actual program size to allow enough room for parameter passing. The message ERROR NO MEMORY-1 IN LINE XXX will occur when there is a program call to an external subroutine and insufficient space is available to store the passed parameters. The SZ command referring to the flagged subroutine should be modified to provide for enough space in memory to store the parameters passed from the calling program.

SS indicates the overlay is to reference SSGA (not applicable to RTE-II).

INTG is used to indicate the function value is returned as an integer. If this parameter is used, the routine is automatically a function.

REAL is used to indicate the function value is returned as a floating point number. If this parameter is used, the routine is automatically a function.

FP, FT, BP, or BT specifies that the overlay is real-time Permanent, real-time Temporary, Background Permanent, or Background Temporary. The default is FP.

VL is used to indicate that the length of the subroutine call parameter list is variable. BASIC supports the calling of subroutines using a variable length parameter list only if the subroutine being called has been written to handle a variable length list. Remember that FORTRAN programs cannot handle a variable length list. BASIC always passes 15 parameter addresses to the subroutine. Unused parameters are zeroed out. Each parameter and its type must be specified in the RTETG command entry. BASIC then will allow execution of subroutine calling sequences with less than the maximum number of parameters.

For example, the RTETG command:
SUBX (RA,RA,I,I,IV), VL,OV=2

will allow BASIC to accept the call:
10 CALL SUBX (A,B,C)

but not the call:
10 CALL SUBX (A,B,C,D,E,F)

ENT=*epoint* specifies the 1 to 5 character subroutine entry point name. If this name is identical to the first five characters of name, it need not be specified. If it is different than the first 5 characters of name, it must be specified. ENT = must precede the entry point name. If greater than 5 characters, an error is issued.

FIL=*fname* specifies the name of a File Manager relocatable file where the routine resides. The FIL=must always precede the file name. If you do not supply a name when you run the transfer file, the disc resident libraries are searched for the routine. If the routine is not found, the Loader prompts for the name at the user's terminal when you load a program which uses the routine.

sc is the security code of the file named *fname*.

cr is the cartridge reference (label or LU) of the cartridge on which the file named *fname* resides.

Constant numbers, string literals, and expressions cannot be used as parameter values when calling a subroutine if the parameter has been defined as type V (returned from subroutine). Functions cannot be used in a parameter list.

14-5. RTETG OUTPUT FILES

RTETG creates two separate binary files for the Branch and Mnemonic Tables and stores them in type 7 File Manager files. See the TABLES command in Section IX.

The overlays are produced as separate standard relocatable files named %Bxnn and are stored in type 5 files. The third letter (x) of the program name and the entry point name vary according to the ID letter supplied with the command which defines the subroutine. The file name also corresponds to the entry point name. The last two characters of the name (*nn*), are digits which indicate the overlay number specified in the OV = command for each subroutine.

RTETG also creates a File Manager transfer file to load the overlays. This transfer file is a type 3 file. This transfer file is discussed in paragraph 14-7.

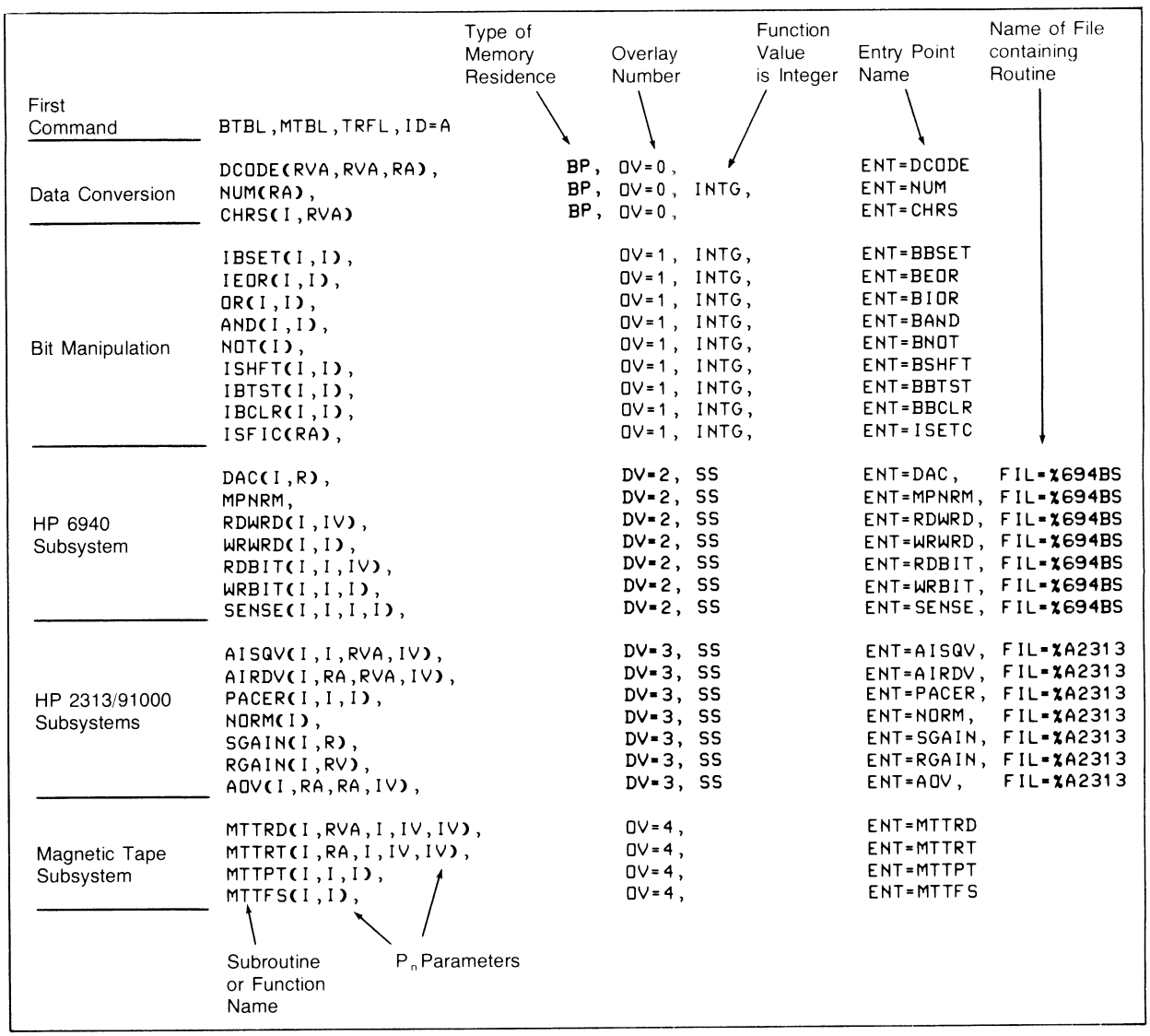


Figure 14-2. RTETG Commands for Library Subroutines

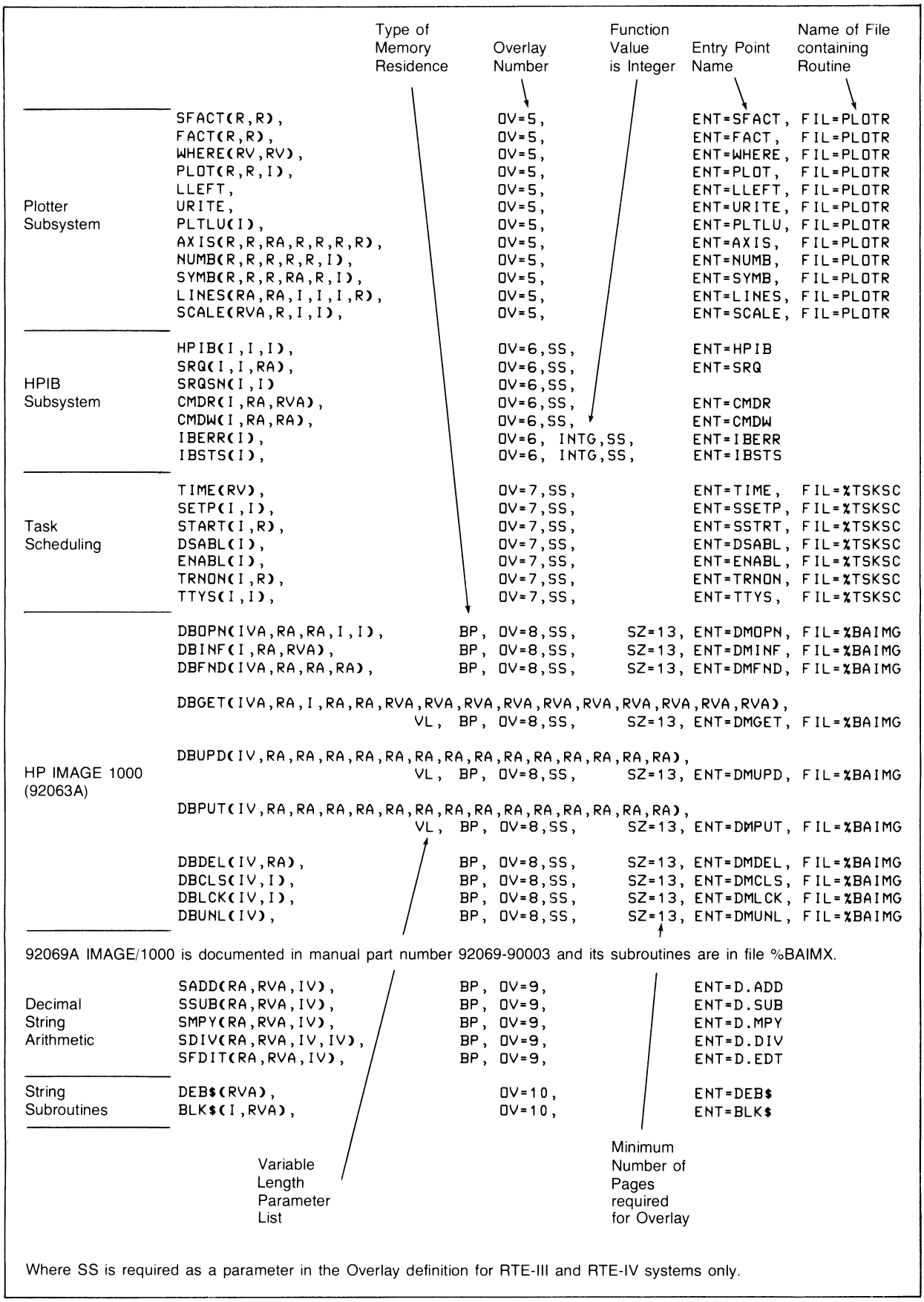


Figure 14-2. RTETG Commands for Library Subroutines (Continued)

14-6. RTETG COMMANDS REQUIRED FOR LIBRARY SUBROUTINES

In order to use the subroutines and functions described in various sections of this manual, you must enter the subroutine and function names in the Branch and Mnemonic Tables. You may select from figure 14-2 the commands for the particular subsystems and subroutines you want to use. Always refer to the appropriate subsystem manual for complete information regarding the subroutines. The commands contain extra spaces for readability. All commands must be entered as one record, terminated with a carriage return.

The first command specifies a Branch Table named BTBL, a Mnemonic Table named MTBL, a Transfer File named TRFL, and an A to identify the group of overlays, from %BA00–%BA10.

The BASIC Interpreter does not check the Branch and Mnemonic Tables before it parses a line to isolate reserved words. A user function (not a subroutine) included in the Branch and Mnemonic Table that has the letter combination AND or OR in its function name will not be recognized, e.g., SANDR(I), CORE(I,I). These examples would be interpreted as S AND R(I), C OR E(I,I). Therefore, any functions which have these letter combinations in their name should be changed.

14-7. LOADING OVERLAYS

After running RTETG, you are now ready to load the overlays. There must be one ID segment available for each overlay. The transfer file is a type 3 source file named and created in the first RTETG command. The transfer file can be used to load the overlays. The loader list device is the same list device specified in the :RUN,RTETG command, Section 14-2. To run the transfer file:

```
*RUN,FMGR       Schedule FMGR.
:TR,trf         Type the TRANSFER command with the transfer file name.
:EXIT           When FMGR returns a prompt, exit.
$END FMGR
```

Under RTE-IV, IVB, and 6/VM it is recommended to use the transfer file as a guide to on-line loading rather than actually running it. This is because certain conditions are assumed when running the transfer file:

1. The loader list device is the same list device specified in the :RUN,RTETG command.
2. The Subroutine Library, %BASLB, has already been generated into the disc resident library.
3. The loader is executed without automatic renaming (i.e., :RU,LOADR:IH).
4. If program type is defaulted in RTETG, the transfer file loads the overlay as a real-time permanent program. Under Session Monitor, the user capability level must be 60 to load programs permanently with the loader.
5. All overlay programs are loaded at one time with the transfer file.

In RTE-IVB and 6/VM if you on-line load an overlay yourself and save the program as a type 6 file, you must save the type 6 file on a different disc LU than the overlay relocatable file. In RTE-6/VM, an overlay must not be loaded as extended background, also you cannot use the multi-level loader.

Below is an example of a RTETG command file, the transfer file and how an overlay was actually loaded under RTE-IVB.

```
BTBL:AJ:V1, MTBL:AJ:V1, TRFL:AJ:V1, ID=A
BAMY(RVA,RVA,RA),     OV=0,SZ=6,BP,       ENT=BAMY,FIL=%BAMY
NUM(RA),              OV=0,SZ=4,INTG,     ENT=NUM
CHRS(I,RVA),          OV=0,SZ=4,          ENT=CHRS
IBSET(I,I),           OV=1,SZ=4,INTG,     ENT=BBSET
IEOR(I,I),            OV=1,SZ=4,INTG,     ENT=BEOR
OR(I,I),              OV=1,SZ=4,INTG,     ENT=BIOR
AND(I,I),             OV=1,SZ=4,INTG,     ENT=BAND
NOT(I),               OV=1,SZ=4,INTG,     ENT=BNOT
ISHFT(I,I),           OV=1,SZ=4,INTG,     ENT=BSHFT

:LI,TRFL:AJ:V1
TRFL   T=00003 IS ON CR V1   USING 00010 BLKS R=0000

0001  :PU,##.RTG
0002  : ST, %BA00 :       0:       0,##.RTG,BR
0003  : DU, %BAMY :       0:       0,##.RTG,BR,2,99
0004  :RU,LOADR:IH,,##.RTG,  1  , BG  , PE  ,, 6
0005  :PU,##.RTG
0006  : ST, %BA01 :       0:       0,##.RTG,BR
0007  :RU,LOADR:IH,,##.RTG,  1  , RT, PE  ,, 4
0008  :PU,##.RTG
0009  ::
```

To load %BA00:

```
:RU,LOADR
[/LOADR:  LIB,%BASLB  ]     if not in Disc Resident Library
 /LOADR:  LIB,%BAMY         subroutine BAMY is in %BAMY
 /LOADR:  REL,%BA00         load first overlay
 /LOADR:  END
:SP,%BA00::-3               save overlay program on LU 3
```

To load %BA01:

```
:RU,LOADR
[/LOADR:  LIB,%BASLB  ]     if not in Disc Resident Library
 /LOADR:  REL,%BA01         load second overlay
 /LOADR:  END
:SP,%BA01::-3               save overlay program on LU 3
```

The subroutines in %BA01 are in the BASIC Subroutine Library, %BASLB.

14-8. ERROR MESSAGES

Table 14-1 contains a summary of the error messages you may encounter while running RTETG. Each error is prefixed by * ERROR *.

Table 14-1. RTETG Error Messages

MESSAGE	EXPLANATION AND ACTION
BAD ENTRY POINT NAME	A name contains illegal characters. See the RTE-IV Assembler Reference Manual for naming rules.
BAD FILE NAME	A name contains illegal characters. See Section 7 for naming rules.
BAD SECURITY CODE	RTETG cannot access the command file. Create it without a security code or use the correct one.
CANNOT READ COMMAND FILE	If the command file is from a type 0 file, make sure that it is set up for reading.
CARTRIDGE LOCKED	Make sure the disc is not being packed before running RTETG.
CARTRIDGE NOT FOUND	Cartridge specified in first command line is not mounted in system.
COMMAND FILE NOT FOUND	Check to make sure the command file name is spelled correctly.
DIRECTORY FULL	Directory track is full. Purge some files and pack the disc or re-initialize the disc specifying one more directory track.
DISC DOWN	Possible disc malfunction. Try to turn disc back on or check out disc problem.
DUPLICATE FILE NAME	One of the files to be created is named the same as an existing file. Use a different name for the file or specify another cartridge.
ENTRY POINT NAME TOO LARGE	The entry point name exceeds 5 characters.
EOF OR SOF ERROR	Trying to read past the end-of-file mark. If using a logical unit for input make sure that you are at the beginning of the data.
FILE OPEN	The command file is open and RTETG cannot access it. Use the File Manager DL command to determine which program has it open. Decide whether to wait until later or to abort the program which is using it if the operator has abandoned the run without terminating the program.
ILLEGAL FORMAT	Syntax error in RTETG command. Check for missing commas, bad characters, misspelled keywords.
ILLEGAL PARAMETER SPECIFICATION	There is an error in one of the parameter specifications supplied within parenthesis following the subroutine name.
MUST BE NUMERIC LIST DEVICE	The list device specified in the run string must be a logical unit and not a file.
MUST SPECIFY A COMMAND FILE OR LU	RTETG is not interactive.
NAME TOO LARGE	The file name exceeds 6 characters or the subroutine name exceeds 6 characters.

Table 14-1. RTETG Error Messages (Continued)

MEANING	EXPLANATION AND ACTION
NOT ENOUGH PARAMETERS	You have not supplied all of the required parameters with the RTETG command.
NOT ENOUGH ROOM ON CARTRIDGE	Cartridge specified is full. Purge and pack the cartridge or specify another one.
OVERLAY OUT OF NUMERIC ORDER	Overlay numbers must be specified in ascending order.
TABLE OVERFLOW	You have exeeded the limit of 300 subroutine specifications.
TOO MANY PARAMETERS	You have supplied too many parameters with the RTETG command.

14-9. REPLACING A SUBROUTINE

If a subroutine you have entered in the Branch and Mnemonic Tables does not work properly, you may need to replace the subroutine or regenerate the tables.

If the problem is in the subroutine itself, the operation is a simple one. First correct the programming error in the subroutine and purge the old overlay program as shown below. Then reload the overlay. Now you can again run BASIC with the corrected subroutine.

If there is an error in the specifications of the tables themselves, you must generate them again. To do so, you must first purge all files created by the Table Generator. For example:

```
:PU,mnemonic table filename
:PU,branch table filename
:PU,transfer filename
:PU,%BX00:::5          Purge all overlay relocatable file names, %Bxnn, where x is the ID letter
                       and nn is the overlay number.
:PU,%Bxnn:::5
```

Next you must remove the overlay programs. Refer to the appropriate RTE reference manuals for detailed information about removing programs. The following example can be used as a guideline when removing programs loaded permanently with the LOADR PE opcode.

```
RTE-II, III                    RTE-IV,IVB,6/VM
:RU,LOADR,,,4                  :RU,LOADR,,,,PU
PNAME?%Bx00                    /LOADR: PNAME?%Bx00
      :                              :
:RU,LOADR,,,4                  :RU,LOADR,,,,PU
PNAME?%Bxnn                    /LOADR: PNAME?%Bxnn
```

If overlay programs were saved with the FMGR SP command, they should be removed.

```
RTE-IV,IVB,6/VM
:PU,%Bx00:::6
      :
:PU,%Bxnn:::6
```

After purging all files and programs, correct the RTETG input file, run the Table Generator, and load the overlay programs.

Integer format examples:

```
DDD
4D
2DDD        equivalent
2D2D

2DX3D
SDDD
S4D
DX3DS
```

Integer output examples:

Format Specification	Value	Format of Output
4D	1234	1234
S4D	1234	+1234
4DS	1234	1234+
5D	1234	1234
4D	1234.8	1235
DXDDD	1234	1 234
S10D	1234	+1234
DSDDD	1234	1+234
5D	-1234	-1234
4D	1234.2	1234

Fixed-point format examples:

```
DDD.DDD
DDD.3D
3D.3D       equivalent
3D.DDD

S3D.3D
DXDXDX.DDXD
XD6X4D.8D
DDSDD.3D
```

Fixed-point output examples:

Format Specification	Value	Format of Output
3D.4D	465.465	465.4650
4D.2D	465.465	465.47
4D.3D	-465.465	-465.465
SDD2D.D	465.465	+465.5
S2D.4D	.465	+0.4650
S.4D	.465	+.4650
D.4D	-.465	-.4650
2D.4D	-.465	-0.4650

Floating point format examples:

```
SD.5DE
DDD.DDDXEX
SD.8DXE
S6DE
S6D.E
S6D.XE
S6D.DDDE
```

Floating point output examples:

Format Specification	Value	Format of Output
SDXE	4.82716×10^{21}	+5 E+21
DDDD.DDE	SAME	4827.16E+18
S5DX.X5DEX	SAME	+48 . 27159E +20
SD.5DE	SAME	+4.82716E+21
S.10DE3X	SAME	+.4827159382E +22

19-6. CARRIAGE CONTROL

One of the following optional carriage control characters may appear as the first non-blank character of a format string:

- \+ suppressing the line feed
- – suppressing the carriage return
- # suppressing both the carriage return and linefeed

If supplied, the carriage control character must be followed by a comma and at least one slash, format specification, or group. The specified action occurs at the completion of the PRINT USING statement. If the carriage control character is not supplied, the default action is a carriage return and a linefeed at the completion of the PRINT USING statement.

When going to a type 0 file for the line printer, a blank or carriage control character must be inserted after the first " in the format string character because control characters are in the first column (e.g., IMAGE " ONE LINE",5A) for line printers.

Examples:

```
10 PRINT USING "#,DDD,2X,AA";A,A$
20 IMAGE -,2A,3X,4D
```

19-7. LITERAL STRING

A literal string (any combination of characters not including a quote mark and enclosed by quote marks) can be included as a format specification in an IMAGE statement and is printed as it appears.

Example:

```
400 IMAGE "TOTAL = ",X,S3D.2D
```

19-8. DELIMITERS

A delimiter serves to set off the format specifications and can be either a comma or a slash. A comma serves no purpose other than to delimit format specifications. A slash can delimit format specifications and, in addition, generates a carriage return, line feed sequence.

Example:

```
500 IMAGE -,3A/3D,3D
```

ERROR MESSAGES

APPENDIX B

Four types of errors may cause error messages: command errors, statement syntax errors, compiling errors, and execution errors resulting from program execution. The error messages are self-explanatory.

COMMAND ERRORS

Command error messages are printed following the command that caused the error.

```
BAD OR MISSING FILE SIZE
BAD SEQUENCE NUMBER
BREAKPOINT ALREADY SET
CAN'T EDIT COMMANDS
DEL OR SAVE PROGRAM
DUPLICATE FILE NAME
ILLEGAL FILE TYPE
ILLEGAL TABLE ORDER
INCOMPATIBLE 'CSAVE' FILE
INVALID COMMAND
INVALID FILE NAME
INVALID LIMITS
INVALID LOGICAL UNIT NUMBER
INVALID LU
INVALID SECURITY CODE
INVALID STATEMENT NUMBER
LU LOCKED OR NO RN AVAILABLE
MORE THAN 4 BREAKPOINTS
NO CALLS DEFINED
NOT ENOUGH ROOM FOR MNEMONIC TABLE
NO TYPE 0 FILE FOUND
PROGRAM FILE IS NOT ON LOGICAL UNIT 2 OR 3
PROGRAM FILE WAS NOT SET UP ON CURRENT SYSTEM
PROGRAM SCHEDULE ERROR
READ FROM WRITE DEVICE OR VICE-VERSA
REQ'D ID SEGMENT NOT FOUND OR NONE AVAILABLE
SEQUENCE NUMBER OVERFLOW/OVERLAP
```

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, or type P to have the erroneous statement reprinted for character editing.

```
ARRAY TOO LARGE
BAD OR MISSING FILE REFERENCE
CHARACTERS AFTER STATEMENT END
ILLEGAL EXPONENT
ILLEGAL OR MISSING INTEGER
```

```
ILLEGAL PARAMETER
ILLEGAL READ OR INPUT VARIABLE
ILLEGAL STRING RELATIONAL OPERATOR
ILLEGAL STRING VARIABLE
MISSING ASSIGNMENT OPERATOR
MISSING LEFT PARENTHESIS
MISSING OR BAD ARRAY VARIABLE
MISSING OR BAD FUNCTION NAME
MISSING OR BAD LIST DELIMITER
MISSING OR BAD SIMPLE VARIABLE
MISSING OR BAD TRAP NUMBER
MISSING OR ILLEGAL DATA ITEM
MISSING OR ILLEGAL 'OF'
MISSING OR ILLEGAL 'STEP'
MISSING OR ILLEGAL SUBROUTINE
MISSING OR ILLEGAL SUBSCRIPT
MISSING OR ILLEGAL 'THEN'
MISSING OR ILLEGAL 'TO'
MISSING RIGHT PARENTHESIS
NO CLOSING QUOTE
NOT A FORTRAN FUNCTION
NOT A SUBROUTINE CALL
OUT OF STORAGE
PARAMETER NOT STRING
SIGN WITHOUT NUMBER
STRING LONGER THAN 255 CHARACTERS
STRING NOT PERMITTED
TOO MANY PARAMETERS
UNDECIPHERABLE OPERAND
WRONG NUMBER OF CHARACTERS
```

COMPILING 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, compilation 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 *n*.

```
ARRAY TOO LARGE
BAD FILES STATEMENT
COM STATEMENT OUT OF ORDER
DIMENSIONS NOT COMPATIBLE
FUNCTION DEFINED TWICE
INVALID FILE NAME
INVALID LU
INVALID SECURITY CODE
LAST STATEMENT NOT END
MISSING SEGMENTS
NEXT WITHOUT MATCHING FOR
OUT OF STORAGE
SYMBOL TABLE OVERFLOW
TOO MANY FILES
UNDEFINED STATEMENT REFERENCED
UNMATCHED FOR
VARIABLE DIMENSIONED TWICE
```

EXECUTION ERRORS

These errors are detected during program execution and printed as they occur; the run terminates.

The line number where the error occurred is appended to all run error messages in the form: IN LINE *n*, where *n* is the line number of the statement that caused the error.

```
BAD CHARACTER AFTER REPLICATOR
BAD DATA
BAD EXPONENT
BAD FLOATING POINT SPECIFICATION
END-OF-FILE/END-OF-RECORD
EXP OUT OF RANGE
FILE NOT OPEN
GOSUBS NESTED 20 DEEP
ILLEGAL CHARACTER IN FORMAT
ILLEGAL FILE TYPE
ILLEGAL FORMAT FOR NUMBER
ILLEGAL FORMAT FOR STRING
ILLEGAL OR MISSING DELIMITER
ILLEGAL TRAP/SEQ NUMBER
MISSING FORMAT SPECIFICATION
MISSING LEFT PARENTHESIS
MISSING REPLICATOR
MISSING RIGHT PARENTHESIS
MULTIPLE DECIMAL POINTS
NEGATIVE NUMBER TO REAL POWER
NEGATIVE STRING LENGTH
NO CLOSING QUOTE
NON-CONTIGUOUS STRING
NON-EXISTENT FILE REFERENCED
OUT OF DATA
OUT OF RANGE IN FUNCTION
OUT OF STORAGE
OVERLAY NOT FOUND
PRINT USING IS NOT ALLOWED TO A FILE
REFERENCED STATEMENT NOT DATA
REPLICATOR TOO LARGE
REPLICATOR ZERO
RETURN WITH NO PRIOR GOSUB
SCHEDULED BUT DELETED TASK
STATEMENT REFERENCED NOT IMAGE STMT
STRING OVERFLOW
SUBROUTINE OR FUNCTION TERMINATED ABNORMALLY
SUBSCRIPT OUT OF BOUNDS
TOO MANY PARENTHESIS LEVELS
```

OTHER ERRORS

Errors may also occur when you are accessing files. Appropriate error messages are printed and the line number is printed if the error occurs during program execution.

If you are operating under RTE-II (92001B), RTE-III (92060B), or RTE-IV (92067A), the error message "COMMAND FILE NOT FOUND" may also mean that the disc cartridge specified was not found or that there was not enough room on the cartridge.

When an IONR (I/O Not Ready) message is issued while using default logical units, you must satisfy the I/O request and make the device ready. For example, if a LOAD command is done from LU 5 (usually the right terminal CTU) and you forgot to insert your mini-cartridge, you will get an IONR error message. Simply insert the mini-cartridge and "UP,eqt" to continue.

In RTE-IVB session environment, if the message FILE MANAGER ERROR is displayed, consult your System Manager. This is a result of any FMGR error with an error code value greater than 41. If an I/O request is made to a logical unit not defined under your session, BASIC issues an error message and allows you to continue.

LOADING BASIC SOFTWARE

APPENDIX D

SYSTEM GENERATION

Here is a summary of the items that should be loaded at system generation time:

Resident Library	#92101-12002	Always required (unless replaced by a dummy module at load time).
Subroutine Library	#92101-12003	Optional.
Instrument Tables	No part number.	Required for specific instruments.
ALARM Program	#92413-16007	Required for Event Sense.

The BASIC Interpreter and Table Generator must not be loaded during RTE system generation. Refer to the Summary of Steps section in this appendix.

If you intend to utilize TRAP and task scheduling statements (see Section XI), the BASIC Resident Library Module (#92101-12002) must be loaded at the time of system generation. This must be done to satisfy references to external subroutines when the BASIC Interpreter is loaded. During the Parameter Input Phase of RTE-III, IV, IVB, or 6/VM system generation, the following program types must be changed:

TRAP,30	**Put into SSGA*
TTYEV,17	**Memory Resident Program Using SSGA*

If you do not require TRAP and task scheduling statement execution, you do not need to include the BASIC Resident Library in your system generation process. However, to satisfy the external subroutine references in the BASIC Interpreter, you do not need to provide a dummy TRAP module when you load the BASIC software. If you choose to do this, BASIC will operate properly, but TRAP statements and all time scheduling statements will not function. Further, if you use the dummy TRAP module, the execution speed of BASIC is improved because the task scheduling table is not scanned after execution of each scheduling statement. The listing in Figure D-1 shows a dummy TRAP module or you can use %DTRAP (#92101-16035) provided with your BASIC software.

```
0001                    ASMB,R,L,Q
0002  00000                 NAM TRAP
0003*
0004* DUMMY TRAP MODULE
0005* FOR BASIC INTERPRETER
0006* MUST BE LOADED IF TRAP AND TASK SCHEDULING IS NOT DESIRED
0007* AND IF %BAMLB IS NOT GENERATED INTO THE SYSTEM
0008*
0009                    ENT TRAP
0010  00000 000000  TRAP  NOP
0011  00001 000000R       ISZ TRAP
0012  00002 000000R       JMP TRAP,I
0013                    END TRAP
**  NO ERRORS *TOTAL **RTE ASMB 92067-16011**
```

Figure D-1. Dummy TRAP Module.

If the HP 6940 or HP 2313 Subsystem is used, the Instrument Tables produced by the Instrument Table Generator (see Section XVIII) must be input during system generation and changed to a type 30 (put into SSGA) module (RTE-III, IV, IVB, and 6/VM). If the HP 6940 Event Sense interrupts are required, then the ALARM program (#92413-16007) must also be input at that time.

If the ALARM program is used, the following entries should be made at generation. During the parameter Input Phase, change the program type:

ALARM,30 **Put into SSGA*

In the Interrupt Table:

sc,PRG,ALARM *where sc is the subchannel of the 6940*

If you plan to schedule tasks from an auxiliary terminal, you must also include an entry in the Interrupt Table as follows:

sc,PRG,TTYEV *where sc is the subchannel of the auxiliary terminal.*

You will get an undefined external message if you generate in the BASIC memory resident library and you do not generate in HP 6940 or HP-IB. You can ignore this message if you do not intend to use these subsystems.

NOTE

If your system has a 2313 and a 6940 *and* you are using both the ISA FORTRAN Extension Package and the BASIC 2313 and 6940 subroutines, the following generation error will be reported:

ERROR05	# GET!	DUPLICATE ENTRY POINT
ERR08	# GET!	DUPLICATE PROGRAM NAME

Ignore this error message: both # GET! routines are identical.

The BASIC Subroutine Library (#92101-12003) should be loaded during system generation but may be input and stored as a File Manager file. The advantage of loading it at system generation time is that the Loader automatically searches this library for undefined references. The Subroutine Library contains HP supplied software for using the HP 2313, HP 6940, HP 7970, BASIC callable task scheduling routines, and bit manipulation routines. The Resident Library contains all pertinent routines for handling traps, time scheduled tasks, and searching the trap and task tables. The Resident Library must be loaded during system generation (see above) to be able to use TRAP and time scheduling tasks.

LOADING BASIC AND RTETG UNDER RTE-II, III

When the RTE System is operational with all modules loaded as required, you may load the BASIC Interpreter (#92101-12001) by using the File Manager and On-line Loader. BASIC must not be generated during RTE system generation. BASIC can only be added to the system by the On-line Loader.

Here are the required File Manager commands for RTE-II/III systems:

:LG,10	*Assign 10 LG tracks. (Load and GO)*
:MR,*lu*	*Input BASIC relocatable tape part 1 of 3.*
:MR,*lu*	*lu is the logical unit number of the input device (paper tape reader or mini-cartridge).*
:MR,*lu*	*Repeat for parts 2 and 3 of the tape.*
[:MR,%DTRAP]	*Dummy Trap module must be specified here if %BAMLB is not loaded at system generation.*

```
:SA,LG,%BASIC          Modules must be merged together
:RU,LOADR,99,1,28,1,2  Loader command for RTE-II. If you are using RTE-III substitute
                       this command:
                       :RU,LOADR,99,1,28,NN001,2 where NN is the number of pages
                       required for BASIC (at least 10).
:RU,BASIC              Start the BASIC Interpreter.
```

The Branch and Mnemonic Table Generator, RTETG, can be loaded by using the same procedure as described above using %BATGN (#92101-16008) and %BATG3 (#92101-16024).

To load under RTE-II/III systems:

```
:LG,
:MR,lu                 Logical unit of input device containing %BATGN
:MR,lu                 %BATG3 (#92101-16024)
:SA,LG,%RTETG
:RU,LOADR,99,1,28,1,2
```

LOADING BASIC and RTETG under RTE-IV, IVB, and 6/VM.

BASIC itself must not be generated into the RTE system. To load under RTE-IV, IVB or 6/VM, use the loader command file #BASIC (#92101-17001) or the following commands:

```
:RU,LOADR
LOADR: OP,LB           Large background program.
/LOADR: SZ,xx          xx is the partition size required. xx must be > 11.
/LOADR: LIB,%DTRAP     Load the dummy TRAP module.
/LOADR: REL,%BAIN1
/LOADR: REL,%BAIN2
/LOADR: REL,%BBUFF     Mnemonic Table buffer must be relocated
/LOADR: REL,%BAIN3     in between %BAIN2 and %BAIN3.
/LOADR: END            Terminate the load.
```

In RTE-6/VM, do not load BASIC or RTETG as extended background and do not use MLLDR, the Multi-Level Loader.

Save BASIC and its eight segments with FMGR commands:

```
:SP,BASIC
:SP,BASC1
:SP,BASC2
:SP,BASC3
:SP,BASC4
:SP,BASC5
:SP,BASC6
:SP,BASC7
:SP,BASC8
```

%BBUFF is a module (#92101-16034) containing 500 words of buffer area to add room for the Mnemonic Table required for external subroutines. Each subroutine needs 5 words from %BBUFF; up to 100 subroutines can be accessed by BASIC. The external subroutines are located in overlay programs generated via RTETG (see Section XIV). If you wish to use more subroutines with BASIC, you can create your own %BBUFF to load with the BASIC Interpreter. &BBUFF (#92101-18034) is provided for your convenience to alter according to your needs. If you always will use less than 80 subroutines, %BBUFF is not necessary in loading BASIC.

The Branch and Mnemonic Table Generator, RTETG, is loaded under RTE-IV, IVB, and 6/VM with the loader command file #RTETG (#92101-17002) or the following commands:

```
:RU,LOADR
/LOADR: OP,LB
/LOADR: SZ,xx          override partition size, xx > 11
/LOADR: REL,%BATGN     #92101-16008
/LOADR: REL,%BATG4     #92101-16023
/LOADR: END
```

Save RTETG and its segments with FMGR commands:

```
:SP,RTETG
:SP,TG00S
:SP,TG01S
:SP,TG02S
```

SET UP FILES FOR LOADING OVERLAYS

Next, use the File Manager to load the files used for task scheduling and communication between the 2313 and 6940. The names and part numbers of these files are:

Task Scheduler (#92101-16013)
A2313 (#29102-60016)
A6940 (#29102-16003)

To load the files, use the following File Manager commands:

:ST,*lu*,SCHREDR,BR (to load the task scheduler file)
:ST,*lu*,A2313,BR (to load the A2313 file)
:ST,*lu*,A6940,BR (to load the A6940 file)

Note that *lu* represents the logical unit number of the input unit, the paper tape reader.

SYSTEM CONSIDERATIONS

You must properly prepare your RTE and BASIC system to obtain multiple terminal operation of BASIC using the background swapping capabilities of RTE. The steps required are itemized below:

- The RTE Operating System must be generated to allow background swapping.
- Provide at least 11K background (or partition) area and at least 2K foreground (or partition) area.
- Include at least as many keyboard/printer devices during system generation as are required for BASIC terminals.
- Provide at least 6 ID segments and 11 background or short ID segments for the BASIC Interpreter, RTETG, and a maximum of 4 overlays.
- Include any instrument drivers at system generation time that might be called by an instrument device subroutine. Also include the Instrument Table generated with the procedure described in Section XVIII.
- Prepare the system for use with the Multi-Terminal Monitor or Session Monitor. All requirements and instructions are included in the appropriate RTE Programming and Operating Manual and generation reference manuals.

- Communication between BASIC and overlays (refer to Section XIV) is done by means of class I/O. One class number per overlay is recommended.

- System Available Memory (SAM) is needed for parameter passing for external subroutines in overlays. The area required can be calculated by analyzing the subroutine with the largest amount of data:

  SAM = data +1 − (3*n) words

  where data is the total number of words for variables and arrays; n is the number of parameters.

  For example, CALL SUB (A(100),I,B) requires:

  ```
  200 words for real array A
    1 word for integer I
    2 words for real B
   10 words for parameters (1+3*3)

  213 words total for SAM
  ```

  If you cannaot determine the exact requirements for SAM, a good rule of thumb is 2K words for SAM.

MULTIPLE COPIES OF BASIC

A copy of BASIC is automatically created for each user under RTE-IVB and 6/VM Session Monitor. BASIC is renamed to BASxx, where xx is the system LU of your terminal. To execute BASIC, type :RU,BASIC. To abort or break your BASIC, type the system command OF,BASxx or BR,BASxx.

To prepare multiple copies of BASIC for use with MTM (Multi-Terminal Monitor) terminals, you must use the following File Manager commands:

:SP,BASIC	*Save a copy of the BASIC program in a file.*
:RN,BASIC,BASIX	*Rename the file BASIX.*
:SP,BASIC	*Save another copy of the BASIC program.*
:RN,BASIC,BASIK	*Name the file BASIK.*
:SP,BASIC	*Save another copy of the BASIC program.*
:RN,BASIC,BASIQ	*Name the file BASIQ.*

There are now four versions of BASIC: one permanent program loaded previously as a permanent program, BASIX, BASIK, and BASIQ (copies of the program saved in files which can be loaded by typing the File Manager RUN command). The RUN command creates a temporary ID segment for the copy of BASIC that you are using.

There are other ways of creating multiple access to the BASIC program including making multiple ID segments pointing to the same BASIC program file, but the advantage of doing it this way is that you only have to use the above command sequence once. When you load the RTe system again, the files will still be available and you can run the various versions of BASIC by typing the File Manager RUN command.

Each terminal in the system must be enabled with the following File Manager command:

:CN,*lu*,20B	*lu* is the logical unit number of the terminal.

When the terminals are enabled and multiple copies of the program have been made, you can press any key on the terminal keyboard and the system prompts with:

lu >	*lu* is the logical unit number of the terminal you are using.

You now have access to the operating system and can execute the BASIC Interpreter by using the RTE RUN command:

lu >RUN,BASIC	*Run BASIC with RTE run command.*
>BASIC READY	*BASIC indicates it is ready.*
>TABLES *branch table,mnemonic table*	*If you are using subroutines, provide the table names.*

You can run whichever version of BASIC you want as long as no one else is using it.

SUMMARY OF STEPS REQUIRED TO GENERATE A BASIC SYSTEM

The remainder of this appendix contains a sample RTE-II system generation. These are the steps you must follow to prepare your system:

1. Use the Instrument Table Generator to produce the HP 2313 and/or HP 6940 Instrument Tables if you intend to use the subroutines related to these instruments. For RTE-III, IV, IVB, and 6/VM systems the Instrument Tables should have their type changed to program type 30.

2. Prepare answers for RTE system generation (see the RTE Programming and Operating Manual for detailed instructions). Make sure the Interrupt Table includes ALARM (for 6940) and TTYEV (for auxiliary teleprinter) if you intend to use them.

3. Generate the RTE system. Be sure to include the BASIC Resident Library (92101-12002) and BASIC Subroutine Library (92101-12003) and the Instrument Table from step 1. For RTE-III, IV, IVB, and 6/VM systems, the TRAP, TTYEV and ALARM modules must have their type changed at system generation during the Parameter Input phase. This is done by using the following commands:

TRAP,30	*Put into SSGA*
TTYEV,17	*Memory Resident program with SSGA*
ALARM,30	*Put into SSGA*

4. Load the new RTE system. Initialize File Manager (FMGR) with the :IN command. (See the Batch-Spool Monitor Reference Manual for detailed instructions.)

5. Load BASIC (92101-12001) and create as many copies as necessary.

6. Load the Branch and Mnemonic Table Generator, RTETG (92101-16008); and the RTE-II/III Transfer File Builder, %BATG3 (92101-16024), or RTE-IV, IVB, 6/VM Transfer File Builder %BATG4 (92101-16023).

7. Load these files if needed: Task Scheduler, A2313, A6940 — use the File Manager.

8. Prepare the Branch and Mnemonic Table Generator input (see Section XIV). Run RTETG.

9. Run the transfer file which RTETG creates.

10. Run BASIC.

INDEX

Note that "*" indicates a changed page.

UPDATE	DESCRIPTION
4 (Contd)	B. Write in "Update 4" to the following pages and make these corrections:

page 4-3, under the following PRINT statement example, add:

300 PRINT Z$[3]
or
300 PRINT Z$[3,0]

page 9-8, the RESEQ command description of first old number should read:

number of first statement **in the program** to be renumbered. . .

page 12-1, last sentence of top paragraph should read:

These functions may be incorporated in the BASIC system at **table** generation time by placing the proper name, entry point and parameter conversion in the Branch and Mnemonic table.

page A-2, the reference page number for the PAUSE statement is 3-19.

page A-3, the reference page number for the LOAD command is 9-2.

The following are prior updates merged together. Pages superseded by the current update are not included.

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, a system-defined function (see Section V) 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 × B
/	Divide	A ÷ 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 ≤ B
>=	Greater than or equal to	A ≥ B
<> or #	Not equal	A ≠ 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.

COM is an array which is placed in a known fixed location in memory. Upon completion of the first program and the loading of the second program, the first location in the COM area is aligned with the first 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 D(1),C(2),B(4)
Position in Memory	First Program	Second Program
xxx1	A(1)	D(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.

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, 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}) 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.

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.

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
```

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 statement number, the word DATA, quotation marks, 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 ON FILES

Format

PRINT # *filenumber* , *record number* ; { *string variable* / *substring variable* / *"string literal"* } [, . . .]

The PRINT # statement prints string or substring variables or string literals on a file.

String and numeric variables may be mixed in a single file or record within a file (example statement 360 below).

The formula for determining the number of words required for storage of a string on a file is:

$$1 + \frac{\textit{number of characters in string}}{2}$$ if the number of characters is even;

$$1 + \frac{\textit{number of characters in string} + 1}{2}$$ if the number of characters is odd.

If the file number is not equal to a file position as defined in a FILES statement, the output will go to the logical unit of the same number. When printing to the line printer using PRINT #6, BASIC inserts a leading space in column 1. The PRINT USING (Section 3-7a) or PRINT to a type 0 file should be used to control the lineprinter.

Example

```
350 PRINT #5; "THIS IS A STRING."
355 PRINT #8; C$, B$, X$, Y$, D$
360 PRINT #7,3; X$, P$, "TEXT", 27.5,R7
365 PRINT #N,R; P$, N, A(5,5), "TEXT"
```

4-14. READING STRINGS FROM FILES

Format

READ # *file number* [, *record number*] ; { *string variable* / *substring variable* } { *string variable* / *substring variable* } [, . . .]

```
0001  FTN,L,M
0002        INTEGER FUNCTION NUM(I)
0003  C
0004  C
0005  C
0006  C
0007  C THIS FUNCTION RETURNS THE NUMERIC VALUE OF THE FIRST CHARACTER
0008  C OF THE STRING EXPRESSION ACCORDING TO THE STANDARD CHARACTER CODE.
0009  C
0010  C   FOR EXAMPLE:
0011  C
0012  C       10 PRINT NUM("A")
0013  C       20 END
0014  C
0015  C       >RUN
0016  C
0017  C       65
0018  C
0019  C
0020  C
0021  C THE FUNCTION'S DESCRIPTION THAT MUST BE INPUT TO THE TABLE
0022  C GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
0023  C TABLE IS AS FOLLOWS:
0024  C
0025  C     NUM(RA), OV=NN, INTG, ENT=NUM, FIL=FILxx
0026  C
0027  C   WHERE: RA   INDICATES REAL PARAMETER (STRINGS ARE ALWAYS REAL)
0028  C          NN   INDICATES THE OVERLAY NUMBER
0029  C          FILX INDICATES THE FILE NAME OF THE RELOCATABLE FOR
0030  C               THIS FUNCTION.
0031  C
0032  C
0033  C
0034        DIMENSION I(2)
0035  C
0036  C  RIGHT JUSTIFY CHARACTER BY DIVIDING
0037  C
0038  C  RIGHT HALF OF THE FIRST WORD OF A STRING IS THE CHARACTER COUNT
0039  C  AND MUST NOT BE DISTURBED.
0040  C
0041        NUM =I(2)/256
0042        RETURN
0043        END
```

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

```
0001  FTN,L,M
0002        SUBROUTINE CHRS(I,J)
0003  C
0004  C
0005  C
0006  C
0007  C  THIS SUBROUTINE CAUSES THE NUMERIC VALUE OF THE FIRST PARAMETER
0008  C  TO REPLACE THE FIRST CHARACTER OF THE SECOND PARAMETER WHICH
0009  C  IS A STRING VARIABLE.
0010  C
0011  C
0012  C    FOR EXAMPLE:
0013  C
0014  C       10 DIM A$(10)
0015  C       20 AS="YBCDE"
0016  C       30 CHRS(65,A$)
0017  C       40 PRINT A$
0018  C       50 END
0019  C
0020  C       >RUN
0021  C
0022  C       ABCDE
0023  C
0024  C
0025  C
0026  C  THE FUNCTION DESCRIPTION THAT MUST BE INPUT TO THE TABLE
0027  C  GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
0028  C  TABLE IS AS FOLLOW:
0029  C
0030  C       CHRS(I,RVA), END=CHRS
0031  C
0032  C       WHERE: I    INDICATES AN ITEGER VARIABLE PASSED TO 'CHRS'
0033  C              RVA  INDICATES A REAL VARIABLE (STRINGS ARE ALWAYS
0034  C                   SPECIFIED AS REAL) RETURNED FROM 'CHRS'.
0035  C
0036  C
0037  C
0038  C
0039  C
0040  C
0041  C
0042        DIMENSION J(2)
0043  C
0044  C  PLACE CHARACTER IN FIRST CHARACTER POSITION OF STRING 'J'
0045  C
0046  C  THE RIGHT HALF OF THE FIRST WORD OF A STRING IS THE CHARACTER
0047  C  COUNT AND MUST NOT BE DISTURBED.
0048  C
0049        J(2)=IAND(J(2),377B)
0050        J(2)=IOR(I*256,J(2))
0051        RETURN
0052        END
```

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

6-4. THE FAIL ERROR OPTION

Some of the externally defined subroutines supplied with the Real-Time 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.

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

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

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

You can use the FAIL ERROR option in subroutines you write for yourself, as well as in the HP subroutines listed above. In your own subroutines you also have the option of processing errors without printing out the error messages. There is no way of avoiding the error message when you are using the HP subroutines, because these routines call an internal error message routine.

After execution of a CALL statement, in either an HP subroutine or your own, the interpreter checks the value of ERRCD. If this value is non-zero, and you have not included the FAIL ERROR option, subroutine CALSB prints the error message:

SUB. OR FUNCT. TERMINATED ABNORMALLY IN LINE *xxx*

and the program aborts. Subroutine CALSB is the parameter-passing linkage between BASIC overlays and the subroutine you write in BASIC language. The loader attaches CALSB to your BASIC routine.

If you do not, in your subroutine, set a non-zero value in ERRCD, there will be no error message and no abort. If, on the other hand, you do set ERRCD non-zero, you will get the above error message, and the program will abort. If you set ERRCD non-zero, and also include the FAIL ERROR option in your CALL statement, the program will not abort and you can additionally interrogate ERRCD in your program:

```
10 CALL subroutine FAIL: GO TO 100
     .
     .
     .
100 I = IERR(0)
110 PRINT I
```

In statement 100, IERR is the function that interrogates ERRCD. The PRINT statement allows you to check what may have caused the error in the execution of your routine.

The following is a sample subroutine you might use to set a value in ERRCD:

```
ASMB,R,L
       NAM PASS,7          Subroutine to set ERRCD
       ENT PASS
       EXT .ENTR,ERRCD
ICODE  BSS 1
PASS   NOP
       JSB .ENTR
       DEF ICODE
       LDA ICODE,I
       STA ERRCD
       JMP PASS,I
       END
```

Alternatively, you can use subroutine ERROR to print a BASIC-type error message:

```
ASMB,R,L
       NAM PASS,7          Subroutine to set ERRCD
       ENT PASS
       EXT .ENTR,ERRCD,ERROR
ICODE  BSS 1
PASS   NOP
       JSB .ENTR
       DEF ICODE
       LDA ICODE,I
       STA ERRCD
       JSB ERROR
       DEF *+3
       DEF ERRCD
       DEF MESS
       JMP PASS,I
MESS   DEC 4
       ASC 2,TEST
       END
```

Using the ERROR subroutine prints an error message of the form:

ERROR TEST–*number* IN LINE *xxx*

where *number* is the value of ERRCD. The following is a FORTRAN subroutine that uses PASS, and a sample BASIC program to call it:

```
FTN4,L
      SUBROUTINE TRYIT (IFLAG,ICODE)
C IF IFLAG #0, THEN SET ERRCD TO ICODE
      IF (IFLAG.EQ.0) GO TO 900
      CALL PASS (ICODE)
900   RETURN
      END
```

```
10 PRINT "INPUT A,N (A#0: TAKE ERR EXIT/N=ERRCD)";
20 INPUT A,N
30 CALL TRYIT (A,N) FAIL: GO TO 50
40 STOP
50 PRINT "ERRCD IS NOW ";IERR(0)
60 END
```

Notice that ERRCD is examined in BASIC with the function IERR, which has a single dummy parameter, not used but necessary. Also notice that ERRCD may be set in the BASIC program by the function SERR, which also has a single dummy parameter.

6-5. 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
•
•
•
9000 IF IERR(X) = 1 GOTO 9100
9010 IF IERR(X) = 2 GOTO 9200
```

Specify task 2000 to be executed at 12:45 and 15 sec. If error, go to 9000.

If error is 1, go to 9100.

If error is 2, go to 9200.

6-6. 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)
```

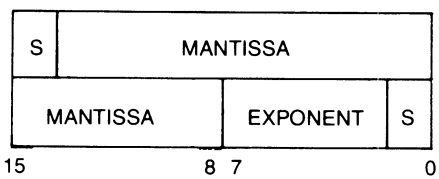
sets the ERRCD flag to the value of N. After execution of your subroutine you can examine the error code by using the IERR function. The value of I is unchanged.

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:

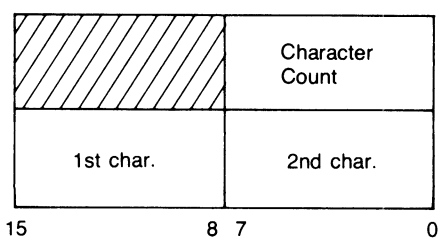
```
10 I= SERR (0)
```

6-7. PARAMETER CONVERSION

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



and for string is:



Any parameter may be omitted, but all parameters following it must also be omitted.

RESEQ can not be used to change the order of statements in a program. When negative statement numbers are used in TRAP statements, RESEQ will resequence but it cannot put back the negative sign. A warning message is issued. Remember to edit those TRAP statements which originally had negative statement numbers.

Examples

```
>RESEQ                    Renumber entire program in increments of 10.
>RESEQ 20,5,15,123        Statements 15 through 123 are renumbered in
                          increments of 5 beginning with 20.
>RESEQ 110,5              The entire program is renumbered in incre-
                          ments of 5 beginning with the number 110.
```

9-10. RUN

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

Format

RUN [*limits*] [*filename* [:*security* [:*cartridge*]]

Parameters

limits	beginning and ending line numbers of the portion of the program to be executed. Limits must be separated by a comma. If limits are not specified, the entire program is executed.
filename	name of the program (filename) to be loaded and executed. Name is specified only if the program is not currently in memory.
security	optional security code. Must be used if program was saved with security code.
cartridge	optional cartridge reference (label or LU number).

This command loads and executes a program or portion of a program. It can be used for a program already in memory or a program stored on disc, obviating the LOAD command.

Examples

```
>RUN                      Executes the program currently in memory.
>RUN PROGA                Loads PROGA if not in memory. Executes the
                          program in either case.
>RUN 50,75,PROGB          Loads and executes statements 50 through 75 in
                          PROGB.
```

9-11. LOCK/UNLOCK

The LOCK command allows you to use a peripheral device exclusive of all other system users. The UNLOCK command returns a device to common availability.

Format

LOCK *lu*
UNLOCK [*lu*]

Parameter

lu	logical unit number of the peripheral device to be locked or unlocked. *lu* is optional with the UNLOCK command. If not specified all locked devices are unlocked.

The LOCK command locks a specified peripheral device to your version of BASIC, so that no other program can access that device while the lock is in effect. UNLOCK relinquishes your control of the device.

BASIC automatically unlocks all peripherals at termination of the BASIC Interpreter.

If you attempt to use a device locked by another user, an appropriate error message is generated.

Examples

```
>LOCK 6          Locks the line printer, LU 6.
>UNLOCK          Unlocks all locked devices.
```

9-12. BYE

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

Format

BYE

BASIC is terminated immediately upon entry of the BYE command. Control is returned to the RTE Operating System or the program that scheduled BASIC (i.e. FMGR or some other calling program). You should always use BYE (not *OFF,BASIC,1) to terminate your BASIC session so that your files will be properly closed.

11-11. 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, 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 11-1), 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.

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.

After RESEQing TRAP statements with negative statements numbers, remember to put back the negative sign on those statements. The RESEQ command resequences properly but it can not put back the negative signs.

Examples

```
110 TRAP 3 GOSUB 2000
```

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.

```
100 TRAP 3 GOSUB 1000
110 TRAP 4 GOSUB 2000
120 TRAP 3 GOSUB 2000
```

Statement 120 causes error SCHED-3 since task 2000 is associated with two trap numbers.

```
5   TRAP 7 GOSUB 170
10  SENSE(5,4,1,7)
•
•
•
160 GOTO 160
170 CALL WRBIT(2,4,1)
180 PRINT "RELAY CLOSED"
190 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.

```
40 TRAP 7 GOSUB 960
50 SENSE(5,I,J,7)
60 CALL WRBIT(2,4,0)
•
•
•
960 PRINT "RELAY CLOSED"
970 TRAP 7 GOSUB 1000
980 CALL WRBIT(2,4,1)
990 RETURN
1000 CALL WRBIT(2,4,0)
1010 PRINT "RELAY OPEN"
1020 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.

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	REFERENCE
AIRDV	Reads HP 2313 analog input in a random manner.	15-4
AISQV	Reads 2313 analog input sequentially.	15-5
AOV	Converts digital values to analog output. (2313)	15-6
AXIS	Plots an axis of a graph.	17-1
BLK$	Initializes a string to a specified number of blanks and resets the logical length of the string.	E-4
CHRS	Places the ASCII character of a specified decimal value into a string.	6-8 & E-6
DAC	Converts digital value to analog output. (6940)	16-2
DBCLS	Closes IMAGE data base files.	*
DBDEL	Deletes existing data records from IMAGE data sets.	*
DBFND	Locates the beginning of an IMAGE data chain in preparation for access to entries in the chain.	*
DBGET	Reads data items from IMAGE data sets.	*
DBINF	Provides information about the organization and components of the IMAGE data base being accessed.	*
DBLCK	Locks an IMAGE data base temporarily to provide exclusive access.	*
DBOPN	Initiates access to IMAGE data bases and defines the user's mode of access.	*
DBPUT	Adds new data records to IMAGE data sets.	*
DBUNL	Unlocks a data base previously locked by a call to DBLCK.	*
DBUPD	Modifies the values of data items in existing IMAGE data records.	*
DEB$	Deletes leading and trailing blanks from a string.	E-5
DSABL	Disables a specified task.	11-6

*Refer to HP IMAGE/1000 Data Base Management System Reference Manual, part no. 92063-90001.

SUBROUTINE	DESCRIPTION	REFERENCE
ENABL	Enables a specified task, permits scheduling of a previously disabled task.	11-7
FACT	Sets the ratio between the horizontal and vertical axis of a graph.	17-2
LINES	Plots a line and/or symbols through successive data points in arrays.	17-3
LLEFT	Lifts the plotter pen and moves it to the lower-left corner.	17-4
MPNRM	Clears the event sense mode and erases the channel/bit to trap number correspondence. (6940)	16-3
MTTFS	Writes an end-of-file and rewinds the magnetic tape.	13-6
MTTPT	Positions a magnetic tape forward or backward a certain number of files and/or records.	13-5
MTTRD	Reads a data record from magnetic tape into an array.	13-4
MTTRT	Writes a record onto a magnetic tape.	13-3
NORM	Normalizes the 2313 Subsystem, resets it to a home or known state.	15-7
NUM	Converts the first character of an ASCII string to its decimal value.	6-7 & E-6
NUMB	Plots a number, with or without decimal point, at a specified height, location, and angle.	17-5
PACER	Sets the pace rate of the HP 2313 Subsystem.	15-8
PLOT	Moves the plotter pen from an origin to a destination.	17-6
PLTLU	Defines the logical unit number of the plotter for all plotter calls.	17-7
RDBIT	Reads (or checks the state) of a specified bit on a channel. (6940)	16-4
RDWRD	Reads the contents of a channel into a word. (6940)	16-5
RGAIN	Reads the gain on a particular channel. (2313)	15-9
SADD	Adds one decimal substring to a second decimal substring.	*
SCALE	Scales an array of numbers to fit a specified graph size.	17-8
SDIV	Divides one decimal substring into a second decimal substring.	*

*Refer to HP Decimal String Arithmetic Routines Manual, part no. 02100-90140.

SUBROUTINE	DESCRIPTION	REFERENCE
SEDIT	Edits data in one decimal substring using an edit mask within a second decimal substring.	*
SENSE	Sets up link between event sense and a specified trap. Senses a change in the bit pattern. (6940)	16-6
SETP	Sets the priority of a task.	11-8
SFACT	Sets or adjusts the plotter for the particular size paper being used.	17-9
SGAIN	Sets the gain for all channels in a group. (2313)	15-10
SMPY	Multiplies one decimal substring by a second decimal substring.	*
SSUB	Subtracts one decimal substring from a second decimal substring.	*
START	Schedules a task for processing after a specified delay.	11-9
SYMB	Writes characters on a plot.	17-10
TIME	Returns the time according to the system real-time clock.	11-10
TRNON	Executes a task at a specified time.	11-12
TTYS	Sets up a link between a trap number and a teleprinter logical unit number.	11-13
URITE	Moves the plotter pen to the upper right so the paper can be removed.	17-11
WHERE	Indicates the current position of the plotter pen.	17-12
WRBIT	Writes a bit onto a channel. (6940)	16-7
WRWRD	Writes the contents of a word onto a channel. (6940)	16-8

*Refer to HP Decimal String Arithmetic Routines Manual, part no. 02100-90140.

BASIC/HP-IB DATA CONVERSION

APPENDIX E

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. These subroutines are included in the BASIC Subroutine Library, %BASLB.

DATA CONVERSION REQUESTS — DCODE

BASIC subroutine DCODE converts binary numbers into ASCII-coded strings or vice versa.

```
DCODE(V1,A$,F$)
DCODE(B$,V2,F$)
```

where

V1 = binary variable to be converted.
A$ = string to contain ASCII-coded result. (Note that the string must be predefined as to size and content, as discussed later.)
B$ = ASCII-coded string to be converted.
V2 = variable to contain binary result.
F$ = format statement by which conversion will occur.

The format specification must be contained within parentheses and may be either of the following:

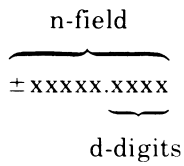
Fn.d = floating point form: n is the number of characters including sign and decimal point; d is number of digits following the decimal point.

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:



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	$$$$$$$

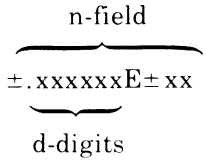
A special use of the F-format is the production of integer strings. The method consists of defining an F-format as Fn.0, where n is the exact number of integer digits to be produced. For example, 123.0 would result in an integer string when the F-format is specified as F3.0. (See example 2 below.)

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:



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

BLK$

BASIC subroutine BLK$ initializes a string to a specified number of blanks and resets the logical length of the string to the number of blanks.

Format

CALL BLK$ *(number of blanks, string)*

Parameters

number of blanks	The string is initialized to the number of blanks specified. The range is from 1 to 255 and must not exceed the maximum dimensioned length of the string.
string	a valid dimensioned string array.

BLK$ assumes that the string is a valid string and does not verify the fact.

Examples

```
DIM D$(10)
  .
  .
  .
BLK$(10,D$)
```

Subroutine BLK$ sets the logical length of string D$ to 10 in the first word and will initialize the succeeding 10 characters in the string to blanks.

As a second example, suppose you make the following call to BLK$:

```
DIM F$(10)
  .
  .
  .
BLK$(5,F$)
```

Subroutine BLK$ sets the logical length of string F$ to 5 in the first word and will initialize the succeeding five characters to blanks.

If you specify a first parameter for BLK$ less than 1 or greater than 255, the subroutine produces the following error message:

```
ILLEGAL FIRST PARAMETER-BLK$
```

DEB$

BASIC subroutine DEB$ deletes leading and trailing blanks from a string.

Format

CALL DEB$ *(string)*

Parameters

string a valid dimensioned string array.

DEB$ assumes that the string is a valid string and does not verify. A valid string must contain at least one non-blank ASCII character.

Examples

```
10 DIM B$(10)
20 LET B$="   ABCD "
30 DEB$(B$)
40 PRINT B$
50 END
```

Statement 40 will print the string without leading and trailing blanks as follows:

```
ABCD
```

DEB$ has changed the logical length from 10 to 4 and the string has been shifted to the left to delete leading blanks.

If there are blanks embedded in the string, DEB$ does not delete them:

```
10 DIM B$(10)
20 LET B$=" AB C D "
30 DEB$(B$)
40 PRINT B$
50 END
```

After the call to **DEB$**, string **B$** prints as follows:

```
AB C D
```

Subroutine **DEB$** changes the logical length of the string from 10 to 7 and shifts the string to the left, deleting leading and trailing blanks, but the embedded blanks remain in the string.

There are two error messages that DEB$ can print on the user terminal. The first occurs if the string passed to the subroutine has not been initialized and therefore has a length of zero:

```
STRING NOT INITIALIZED-DEB$
```

The second occurs if the string contains all blanks:

```
STRING ALL BLANKS-DEB$
```

Both messages are for information only. The subroutine takes no action.

NUM and CHRS

Many devices communicate with the RTE Operating System using ASCII characters. Sometimes, these characters are non-printing or pure binary numbers. BASIC needs a means to convert these ASCII characters to decimal values or visa versa. This is handled by function NUM and subroutine CHRS.

The NUM function takes the first character of an ASCII string and converts the character to its decimal value (See Appendix C for the decimal to ASCII equivalence.)

Format

NUM *(string)*

Parameters

string a string array, substring array, or string literal.

The string array must be defined before the function NUM is used.

Examples

The conversion of the first character of a string array to its decimal value would be:

```
  5 DIM A$[10]
 10 LET A$="ABCDE"
 20 PRINT NUM(A$)
 30 END

>RUN

65                    the decimal value of the ASCII character A.
```

A substring example would be:

```
  5 DIM A$[10]
 10 LET A$="ABCDE"
 20 PRINT NUM(A$(2,2))
 30 END

>RUN

66                    the decimal value of the second element in A$, an ASCII B.
```

The conversion of a literal string is:

```
 10 PRINT NUM("F")
 20 END

>RUN

70                    the decimal value of the literal string F.
```

The subroutine CHRS takes the decimal value of an ASCII character and places the ASCII character in the specified location of a string variable. The string must be defined prior to the call to the subroutine CHRS.

Format

CALL CHRS *(decimal value, string)*

Parameters

decimal value	a valid specified decimal value of an ASCII character.
string	a string array or substring array in which the ASCII character is to be placed. If a string array is specified, the ASCII character is placed as the first element of the array. Array subscripts can be specified for any element except for the first element (i.e., do not use A$[1,1]; use A$).

The string must be defined prior to calling CHRS.

Examples

The control information for setting the margin on the 9871A line printer is 6 blanks followed by an escape character (decimal value 27) and an M. The 9871A line printer is LU9 in this case and when the print is issued to the line printer (file LP), the control sequence is executed as demonstrated by the listing of the program.

```
  10 DIM A$[10]
  20 LET A$[1,6]="      "
  30 CALL CHRS(27,A$[6,6])
  40 LET A$[7,7]="M"
  50 PRINT #9;A$
  60 END

>RUN

>LIST LP

      10 DIM A$[10]
      20 LET A$[1,6]="      "
      30 CALL CHRS(27,A$[6,6])
      40 LET A$[7,7]="M"
      50 PRINT #9;A$
      60 END
```