



Multi-User Real-Time BASIC

Reference Manual



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

PART NO. 92060-90016
PRODUCT NO. 92101A

Change 1 12/75
Printed in U.S.A. 10/75

LIST OF EFFECTIVE PAGES

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 Insert latest changed pages and destroy superseded pages.

Change 0 (Original) Oct 1975

Change 1 Dec. 1975

Title, ii

4-8

14-5 to 14-8

D-1 to D-32

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

HP Computer Museum

www.hpmuseum.net

For research and education purposes only.

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-II or RTE-III 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-II or RTE-III 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. If you must generate the BASIC system yourself, you should be familiar with the Batch-Spool Monitor Reference Manual which describes some File Manager, FMGR, commands you will need to use. It is recommended that you also read the RTE-III General Information Manual if you are using an RTE-III system. These manuals are shown in the documentation map which follows 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.

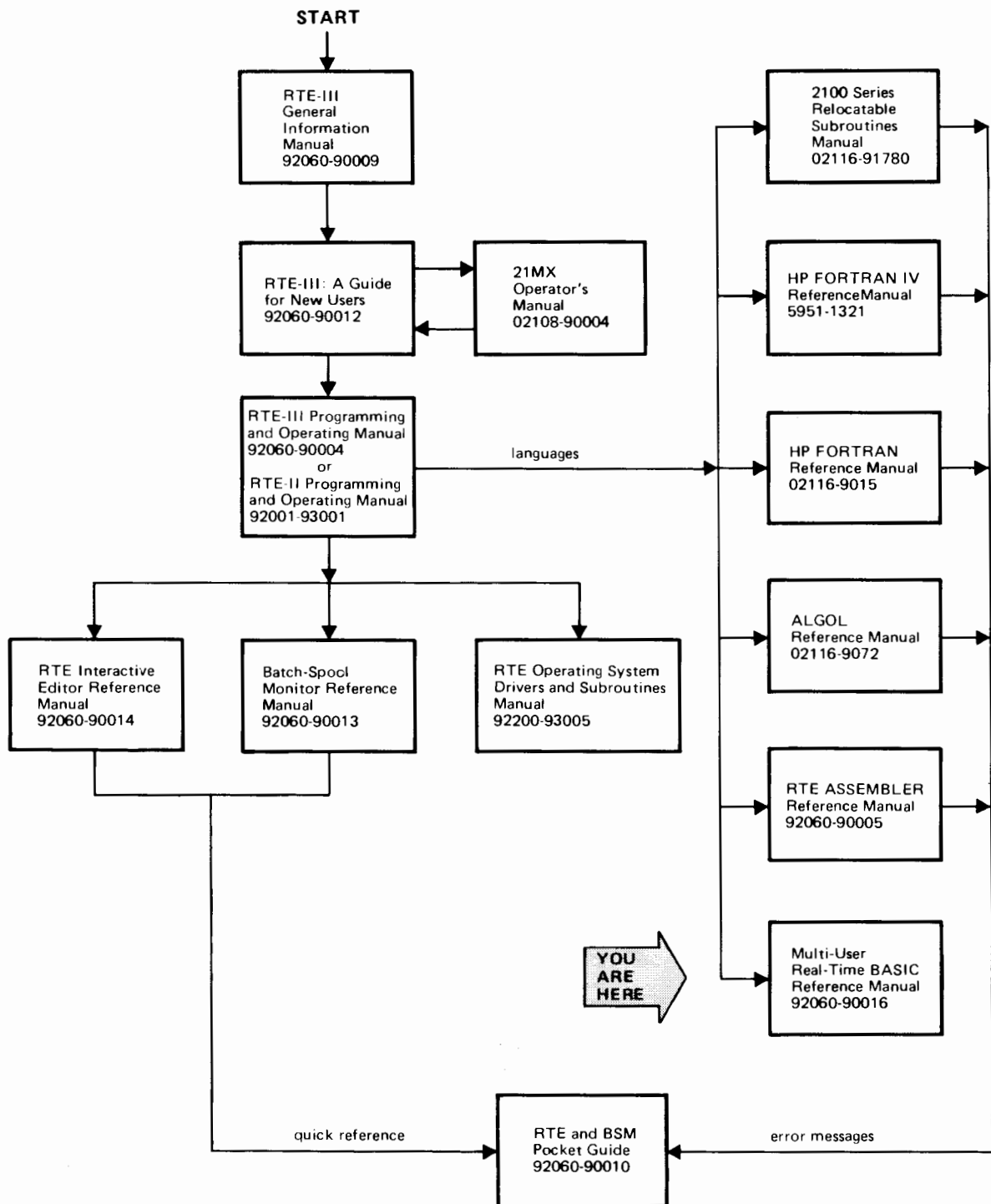
Section XVIII provides instructions on generating the Instrument Table tape which is required if the instrumentation subroutines are to be used.

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.

The components of Multi-User Real-Time BASIC can be ordered by the following part numbers:

Multi-User Real-Time BASIC	92101A
HP 7210 Plotter Driver (DVR 10)	72009-60001
HP 7210 Plotter Library	92409A
Instrument Table Generator	92413-16011

DOCUMENTATION MAP



CONTENTS

Section I	Page
INTRODUCTION	
Features	1-1
Conversational Programming	1-1
Multiple Peripheral Device I/O	1-1
Real-Time Task Scheduling	1-1
Program Debugging Aids	1-2
File Capabilities	1-2
Environment	1-2
Hardware	1-2
Software	1-3
Commands and Statements	1-5
Commands	1-5
Statements	1-5
BASIC Programs	1-6
Character Editing	1-7
Correction of Typing Errors	1-8
Logical Unit Numbers	1-8
Syntax Conventions	1-9

Section II	Page
EXPRESSIONS	
Constants	2-1
Numeric Constants	2-1
Floating-Point Numbers	2-2
Literal Strings	2-2
Variables	2-2
Functions	2-3
Operators	2-4
Evaluating Expressions	2-5

Section III	Page
STATEMENTS	
LET	3-1
REM	3-4
GOTO	3-4
END/STOP	3-5
FOR . . . NEXT	3-6
IF . . . THEN	3-8
PRINT	3-9
Numeric Output Formats	3-11
Integers	3-11
Fixed-Point Numbers	3-12
Floating-Point Numbers	3-12
TAB Function	3-13
READ/DATA/RESTORE	3-13
INPUT	3-14
PAUSE	3-14
WAIT	3-15
DIM	3-16
COM	3-16

Section IV	Page
STRINGS	
String	4-1
String Variable	4-2
Substring	4-2
Strings and Substrings	4-3
String DIM	4-4
String Assignment	4-5
String INPUT	4-5
Printing Strings	4-6
Reading Strings	4-7
String IF	4-8
LEN Function	4-8
Strings in DATA Statements	4-9
Printing Strings on Files	4-10
Reading Strings from Files	4-10

Section V	Page
FUNCTIONS	
System-Defined Functions	5-1
User-Defined Functions	5-2

Section VI	Page
SUBROUTINES	
GOSUB/RETURN	6-1
CHAIN	6-4
CALL	6-6
FAIL Error Option	6-9
The IERR Function	6-10
SERR	6-10
Parameter Conversion	6-10

Section VII	Page
FILES	
File Characteristics	7-1
CREATE and PURGE	7-2
FILES Statement	7-2
ASSIGN Statement	7-3
IF END# . . . THEN Statement	7-4
Restoring the Data Pointer	7-5
Serial File READ Statement	7-5
Reading a Record	7-6
Serial File PRINT Statement	7-7
Printing a Record	7-8
TYP Function	7-9
Modifying Records	7-9

Section VIII	Page
STARTING UP	
Scheduling BASIC	8-1
Using BASIC	8-2
Start Up Options	8-2

CONTENTS (continued)

Section IX	Page
OPERATOR COMMANDS	
LOAD	9-1
SAVE/CSAVE	9-3
MERGE	9-4
REPLACE	9-5
DELETE	9-5
CREATE	9-6
PURGE	9-7
RENAME	9-8
RESEQ	9-8
RUN	9-9
LOCK/UNLOCK	9-10
BYE	9-10
LIST	9-11
*BR,BASIC	9-11
CALLS	9-12

Section X	Page
DEBUGGING COMMANDS	
TRACE/UNTRACE	10-1
BREAK/UNBREAK	10-3
RESUME	10-3
ABORT	10-4
SIM/UNSIM	10-5
SHOW	10-5
SET	10-6

Section XI	Page
REAL-TIME TASK SCHEDULING	
Introduction	11-1
Methods of Initiating Tasks	11-1
Priorities	11-2
Response Time	11-3
The BASIC Scheduler	11-3
DSABL	11-5
ENABL	11-5
SETP	11-6
START	11-7
TIME	11-8
TRAP Statement	11-9
TRNON	11-11
TTYS	11-12
Program Example	11-12
Table Preparation	11-17
Error Messages	11-17

Section XII	Page
BIT MANIPULATION OPERATIONS	
Bit Manipulation Word Format	12-1
AND	12-1
IBCLR	12-2
IBSET (Bit Set)	12-3
IBTST (Bit-Test)	12-3
IEOR	12-4

NOT	12-5
OR	12-5
ISETC (Set to Octal)	12-6
ISHFT (Register Shift)	12-6
Branch and Mnemonic Table Preparation	12-7

Section XIII	Page
MAGNETIC TAPE I/O	
Magnetic Tape Operator Commands	13-1
Magnetic Tape Calls	13-2
MTTRT	13-2
MTTRD	13-2
MTTPT	13-3
MTTFS	13-4
Tape Manipulation Errors	13-4
Branch and Mnemonic Table Entries	13-5
Sample Program Using Magnetic Tape	13-5

Section XIV	Page
SUBROUTINE TABLE GENERATION	
RTETG	14-2
Scheduling RTETG	14-2
The First RTETG Command	14-3
Other RTETG Commands	14-3
RTETG Output Files	14-4
RTETG Commands Required for	
Library Subroutines	14-4
Running the Transfer File	14-6
Error Messages	14-6
Replacing a Subroutine	14-7

Section XV	Page
HP 2313/91000 DATA ACQUISITION SUBSYSTEM	
Measurement of Analog Input	15-1
Analog Output	15-1
HP 2313/91000 Subsystem Subroutines	15-1
AIRDV (Random Scan)	15-2
AISQV (Sequential Scan)	15-3
AOV (Digital to Analog Conversion)	15-4
NORM	15-5
PACER	15-6
RGAIN	15-7
SGAIN	15-8
Subsystem Errors	15-8
Table Preparation	15-9
Subsystem Concept	15-9
Card Configuration	15-10
Channel Numbering	15-11
Setting Gain	15-11

CONTENTS (continued)

Section XVI	Page
HP 6940 MULTIPROGRAMMER	
SUBSYSTEM	
HP 6940 Subsystem Subroutines	16-1
DAC	16-1
MPNRM	16-2
RDBIT	16-2
RDWRD (Read Channel)	16-3
SENSE	16-4
WRBIT	16-5
WRWRD (Write Channel)	16-6
Subsystem Errors	16-6
Table Preparation	16-7
Card Configuration	16-7
Expansion	16-8
Channel Numbering	16-8

Section XVII	Page
HP 7210 PLOTTER	
AXIS	17-1
FACT	17-2
LINES	17-2
LLEFT	17-3
NUMB	17-4
PLOT	17-4
PLTLU	17-5
SCALE	17-5
SFACT	17-6

SYMB	17-7
URITE	17-8
WHERE	17-8
Table Preparation	17-8

Section XVIII	Page
INSTRUMENT TABLE GENERATION	
Operating Instructions	18-1
HP 2313/91000 Configuration Phase	18-1
HP 6940 Configuration Phase	18-2
Loading the Tape	18-4
Error Messages	18-4

Appendix	Page
SUMMARY OF STATEMENTS, COMMANDS, AND SUBROUTINES	A-1
Statement Summary	A-1
Command Summary	A-3
Subroutine Summary	A-5
ERROR MESSAGES	B-1
ASCII CHARACTER SET	C-1
LOADING BASIC SOFTWARE	D-1
System Generation	D-1
Loading the Interpreter	D-2
System Considerations	D-2
Multiple Copies of BASIC	D-2
Summary of Steps Required to Generate a BASIC System	D-3



ILLUSTRATIONS

Title	Page
Typical System	1-3
RTE Memory Layout with BASIC	1-4
Preparing a FORTRAN Function for Use by BASIC Program	6-7
Preparing a FORTRAN Subroutine for Use by BASIC Program	6-8
FORTTRAN Subroutine to Convert String Parameter	6-11
Task State Definitions	11-4
Task Scheduling Program Example (Part 1)	11-13
Structure of Program Example in Figure 11-4	11-14
Task Scheduling Program Example (Part 2)	11-15
16-Bit Word	12-1

Title	Page
Record Positioning Example Using MTTPT	13-3
Tape Control Sample Program	13-5
BASIC and an Overlay in Memory	14-1
RTETG Commands for Library Subroutines	14-5
HP 2313 Subsystem Configuration	15-9
HP 6940 Subsystem Configuration	16-8
Channel Numbers for Additional 6940	16-9
Channel Numbers for Addition of a 6941 Extender	16-9
Plotter Control Sample Program #1	17-9
Plotter Control Sample Program #2	17-10
Plotter Control Sample Program #2 (Plot)	17-11
Dummy TRAP Module	D-1

TABLES

Title	Page
Statements	3-1
Operator Commands	9-2
Debugging Commands	10-1

Title	Page
RTETG Error Messages	14-6
Error Messages	18-4

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, ALGOL, 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-II or RTE-III 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 $1 + \text{INT}[(n + 1)/2]$ 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, ALGOL, and Assembly language programs can use the same 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.

1-7. ENVIRONMENT

1-8. HARDWARE

The BASIC Interpreter operates within the RTE-II or RTE-III hardware environment consisting of an HP 9600 Series Computer System. (Refer to the appropriate system Programming and Operating Manual for equipment configurations.)

For RTE-II, the BASIC Interpreter can operate within the minimum system on a 2100 or 21MX computer with 24K memory.

For RTE-III, the BASIC Interpreter can operate within the minimum system on a 21MX computer with 32K memory.

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.

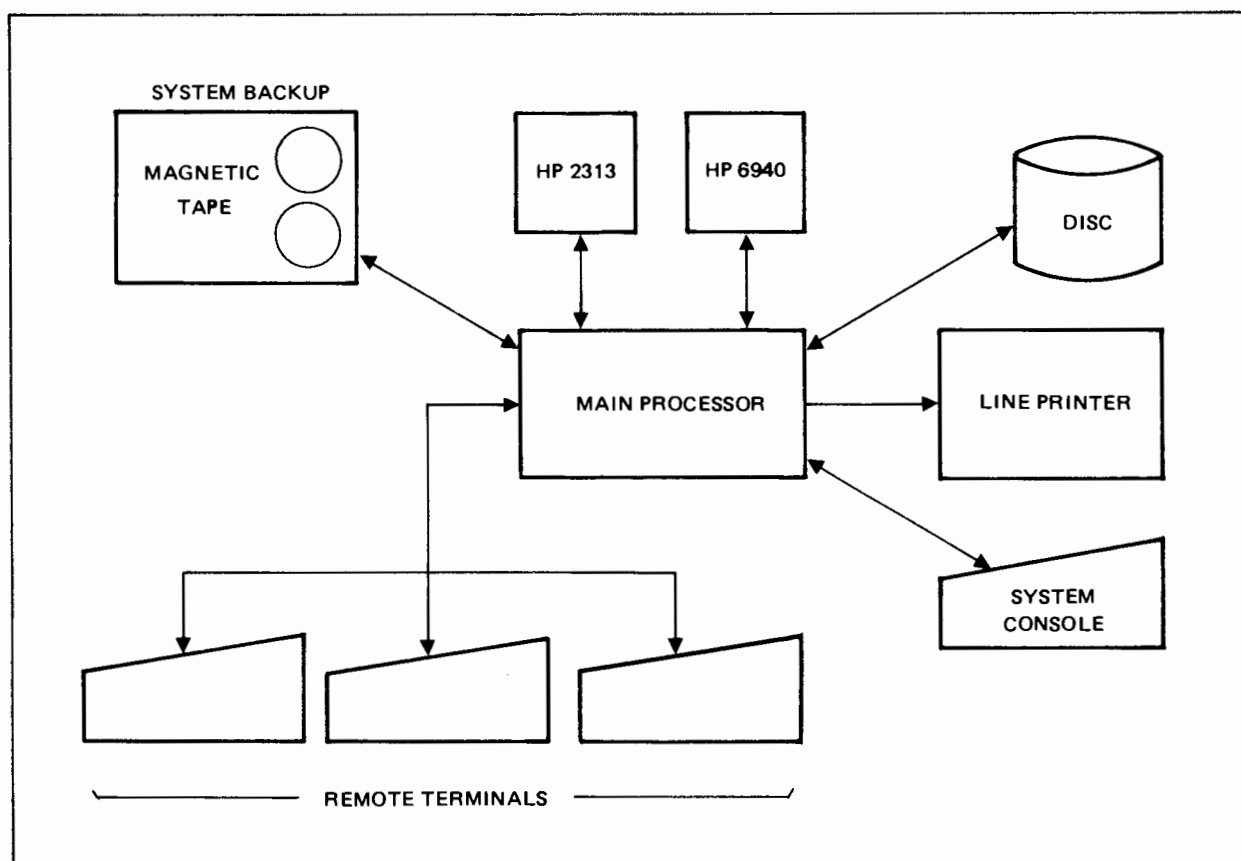


Figure 1-1. Typical System

1-9. SOFTWARE

The BASIC Interpreter is an option which runs under control of either of the following operating systems:

- Real-Time Executive II (RTE-II)
- Real-Time Executive III (RTE-III)

The BASIC Interpreter requires an RTE-II system with a background memory area of at least 8K words and the File Management Package. It requires an RTE-III system with a main memory partition of at least 8K words. At least 450 words of base page must be available.

Multi-User Real-Time BASIC is a self-contained segmented program operating in RTE-II background or an RTE-III partition. The subsystem consists of the following modules and components:

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

Introduction

The BASIC program consists of a main program and 8 disc resident segments. The purpose of each segment is:

- 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.

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

Figure 1-2 illustrates the layout of BASIC components in the RTE-II system memory. The layout is the same for RTE-III except two partitions are used instead of background and foreground.

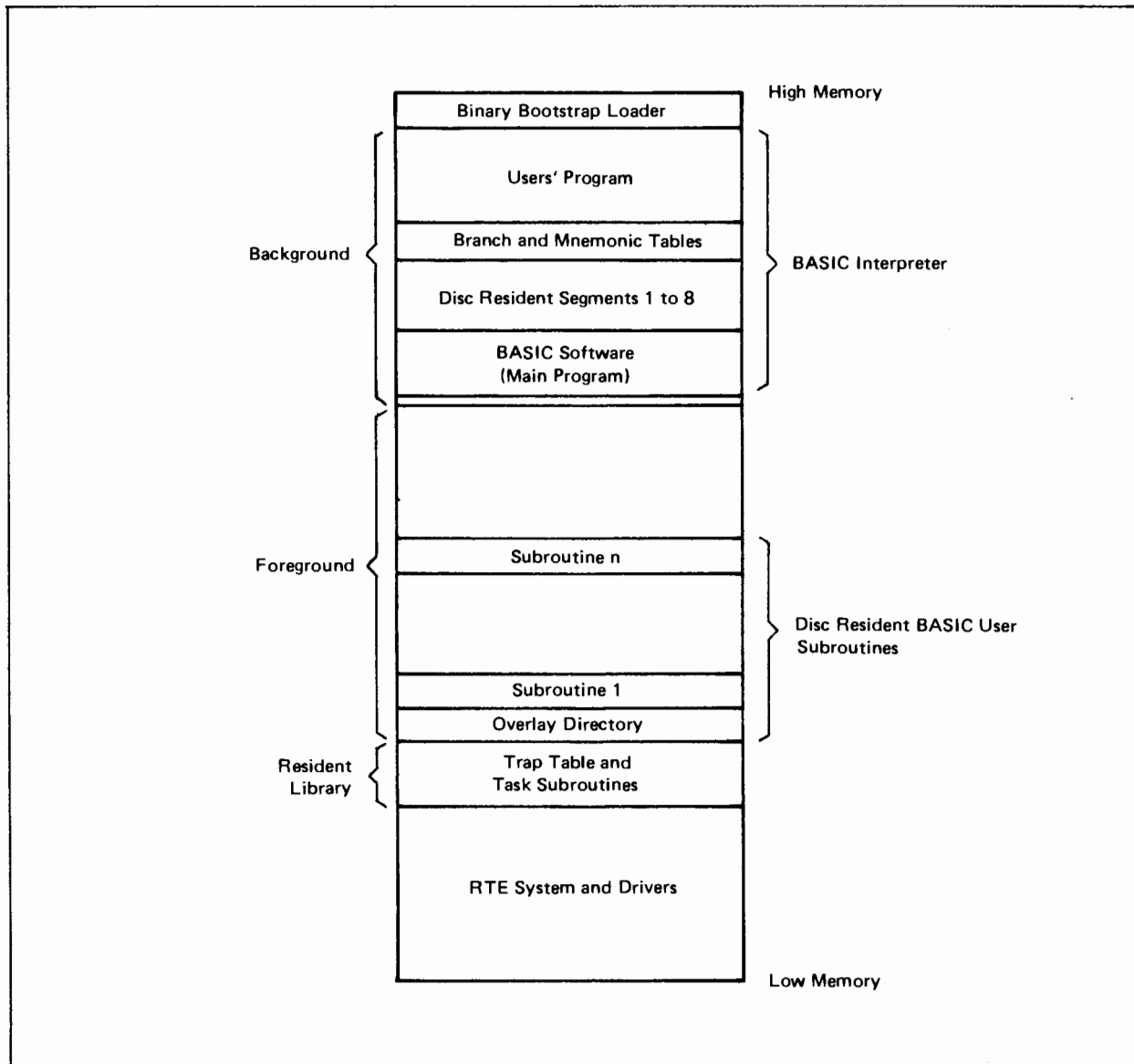


Figure 1-2. RTE Memory Layout with BASIC.

1-10. COMMANDS AND STATEMENTS

1-11. COMMANDS

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

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

Any BASIC command can be entered following the BASIC prompt character >. Each command is a single word that must be typed in its entirety with no embedded blanks. (DELETE and RESUME are exceptions, you may type DEL and RES.) If misspelled, the computer will return an error message. Some commands have parameters to further define command operation.

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

1-12. STATEMENTS

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

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

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

BASIC statements have a free format. This means that blanks are ignored. For instance, all these statements are equivalent.

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


1-13. BASIC PROGRAMS

Any statement or group of statements that can be executed constitutes a program.

A program may consist of only two statements.

This is an example of such a program.

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

100 is the statement number. PRINT is the key word or instruction that tells the Interpreter the kind of action to perform. In this case, it prints the result of the expression that follows. 35+5 is an arithmetic expression. It is evaluated by the Interpreter, and when the program is run, the result is printed. The END statement indicates the program is complete.

Usually a program contains more than one statement.

These four statements are a program:

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

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

This program runs exactly like the program above.

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

It is generally a good idea to number statements in increments of 10. This allows room to intersperse additional statements as needed. If you have too many statements to insert, you may renumber the statements with the RESEQ command.



1-14. CHARACTER EDITING

If you make an error while entering a statement, an error message is printed which indicates the nature of the error, and you can correct the error immediately. If you want to character edit the error, you must redisplay the incorrect statement by typing a P and then make the desired corrections using the control characters required.

Similarly if you find an inappropriate statement in an existing program, you can easily correct the statement. You can position to the line you want to edit by typing a slash (/) for each line, stepping line by line through the program until you find a line which requires editing. You can also use the LIST command to position to the line if you know the statement number. In this case, you do not have to redisplay the line with the P command.

If a syntax error is encountered when loading a program from a disc file, BASIC prints an error message and the offending line. The line is ready for character editing. After it is corrected, you can use the MERGE command to load the remainder of the program.

In each case, after the line you want to edit is displayed, you type a P and the appropriate control characters to change the line. Five special control characters are available for character editing and line correction. Four of the control characters are entered by first pressing the control key (CNTL) and holding it down while pressing the letter key I, R, C, or T. The fifth control character is a slash which is entered without using the control key.

Control Character	Use
I ^c	Inserts new characters into a line prior to the current position.
R ^c	Replaces characters in a line at the current position.
C ^c	Deletes specific characters from a line beginning at the current position.
T ^c	Truncates all remaining characters in a line including the current character.
/	Leaves a character unchanged or if typed instead of the P command, lists the next line in the program.
/n	Positions to program statement <i>n</i> and lists it.

The following is an example of character editing used to correct an entry error. (User entries are underlined for clarity.)

>10 A*10	<i>Incorrectly entered line.</i>
MISSING ASSIGNMENT OPERATOR IN LINE 10	<i>Error message.</i>
>P	<i>Request line be reprinted.</i>
10 A*10	<i>Line is reprinted.</i>
MISSING ASSIGNMENT OPERATOR IN LINE 10	
>P///X =	<i>Request edit. Insert X = after first three characters which are unchanged. (Space is counted as a character.) Line is reprinted showing corrections.</i>
10 X =A*10	

You can change the / control character to some other character by typing Xl in response to the BASIC prompt, >. l will be the new character replacing the slash. l can be any non-numeric character except: P, I^c, R^c, C^c, T^c.

Introduction

Here is an example of editing an incorrect line loaded from a disc file:

BASIC READY	<i>Command to load a program PROGA from a disc file.</i>
>LOAD PROGA	
UNDECIPHERABLE OPERAND IN LINE 200	<i>Error message and line number printed.</i>
200 LET A= -	<i>Line displayed.</i>
>P//////////6	<i>Type P and control characters to edit line.</i>
200 LET A= -6	<i>BASIC redisplay corrected line.</i>
>MERGE PROGA	<i>Command to merge the remainder of the program with the new line and previous ones.</i>

1-15. CORRECTION OF TYPING ERRORS

You may use the following keys to correct typing errors:

RUBOUT deletes the current line you are typing; on some terminals a DEL key is used instead.

BACKSPACE deletes a character.

If the terminal does not have a key labeled BACKSPACE, you may use Control H (H^c). Press the control key (CTRL), hold it down and press the H key. The backspace is printed as an underline or a back arrow ←. More than one character may be deleted by repeating BACKSPACE or H^c for each character you want to delete.

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

1-16. LOGICAL UNIT NUMBERS

Logical unit numbers, abbreviated LU in this manual, are decimal integers between 0 and 63 used to address I/O devices. Numbers 1 through 6 must always refer to the following devices:

- 1 - system console
- 2 - system disc
- 3 - auxiliary disc (optional)
- 4 - standard output unit
- 5 - standard input unit
- 6 - standard list unit

The standard devices may be:

- output - paper tape punch or magnetic tape
- input - paper tape reader, card reader, or terminal
- list - line printer or terminal.

The remaining logical unit numbers (7 - 63) may be assigned to any type device. Logical unit number 0 is not associated with a particular device but it used to essentially turn off an input or output statement.

1-17. SYNTAX CONVENTIONS

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

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

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

Some examples of expressions are:

$$(P + 5)/27$$

P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divided operator. Parentheses group those portions of the expression evaluated first.

If $P = 49$, it is an integer expression with the value 2.

$$(N - (R + 5)) - T$$

N, R, and T must all have been assigned values. + and - are the add and subtract operators. The innermost parentheses enclose the part evaluated first.

If $N=20$, $R=10$, and $T=5$, the value of the integer expression is zero.

2-1. CONSTANTS

A constant is either numeric or it is a literal string.

2-2. NUMERIC CONSTANTS

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

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

Examples of Integers:

1234
-70
0

Expressions

Examples of Fixed Point Numbers:

1234.
1234.56
-.0123

2-3. FLOATING-POINT NUMBERS

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

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

$1\text{E}+35 = 100000000000000000000000000000000000$
 $1\text{E}-35 = .000000000000000000000000000000000001$

Examples of Floating-Point Numbers:

1E+23	$= 1 \times 10^{23} = 1000000000000000000000000$
1.0E23	(same as above)
.001E26	(same as above)
1.02E+4	$= 1.02 \times 10^4 = 10200.$
1.02E-4	$= .000102$

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

2-4. LITERAL STRINGS

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

Examples of Literal Strings:

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

Blank spaces are significant within a string.

2-5. VARIABLES

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

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 may be subscripted. 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)      A0(N,M)
P(1,1)    P5 (Q5,N/2)
X(N+1)    X9(10,10)
```

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

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

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

```
A$      P$
```

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

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

2-6. FUNCTIONS

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

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

Expressions

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 $+A$
-	Subtract (or if unary, negative)	$A - B$ or $-A$
*	Multiply	$A \times B$
/	Divide	$A \div B$
\uparrow or \wedge	Exponentiate	A^B

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

The relational operators are:

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

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

Logical or Boolean operators are:



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

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

Logical operators are evaluated as follows:

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

2-8. EVALUATING EXPRESSIONS

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

\uparrow or \wedge	(highest)
NOT	
* /	
+ -	
Relational (=, <, >, <=, >=, <>)	
AND	
OR	(lowest)

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

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

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

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

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

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

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

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

Expressions

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

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

STATEMENTS

SECTION

III

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

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

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

3-1. LET

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

Format

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

[LET] *variable* = *expression*

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

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

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

Table 3-1. Statements

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

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

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

Examples

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

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

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

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

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

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

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

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

3-2. REM

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

Format

REM *any characters*

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

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

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

Examples

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

3-3. GOTO

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

Format

GOTO *statement number label*

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

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

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

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

Examples

```
50 GOTO 100
60 GOTO A OF 100, 200, 300
```

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

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

```
>LIST
100 LET I=0
200 GOTO 600
300 PRINT I
400 REM THE VALUE OF I IS ZERO
500 LET I=I+1
600 GOTO I+1 OF 300,500,800
700 REM THE FINAL VALUE OF I IS 2
800 PRINT I
900 END
>RUN
0
2
```

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

3-4. END/STOP

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

Format

<p>END</p> <p>STOP</p>

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

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

Statements

Examples

```
200 IF A # 27.5 THEN 350
```

•

•

```
300 STOP
```

•

```
350 LET A = 27.5
```

•

•

```
500 IF B # A THEN 9999
```

•

•

```
9999 END
```

3-5. FOR . . . NEXT

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

Format

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

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

The NEXT statement terminates the loop:

NEXT *variable*

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

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

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

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

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

Examples

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

```
>LIST
 10 REM : RADIUS EXAMPLE
 20   FOR A=1 TO 5
 30   INPUT R
 40   PRINT "AREA OF CIRCLE WITH RADIUS ";R;" IS ";3.14159*R^2
 50   NEXT A
 60   END
>RUN
?1
AREA OF CIRCLE WITH RADIUS 1      IS 3.14159
?2
AREA OF CIRCLE WITH RADIUS 2      IS 12.5664
?4
AREA OF CIRCLE WITH RADIUS 4      IS 50.2654
?8
AREA OF CIRCLE WITH RADIUS 8      IS 201.062
?16
AREA OF CIRCLE WITH RADIUS 16     IS 804.247

BASIC READY
```

Statements

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

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

BASIC READY
>
```

3-6. IF . . . THEN

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

Format

<p>IF <i>expression</i> THEN <i>statement</i> <i>statement number label</i></p>

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

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

The word THEN is omitted from the statement in the above operations.

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

Examples

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

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

In the following example, if $X > 10$, the message in statement 40 is executed. Otherwise, the message in statement 60 is executed. Note that the relational operator is optional in logical evaluations.

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

3-7. PRINT

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

Format

PRINT [*print list*]

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

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

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

Statements

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

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

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

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

Examples

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

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

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

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

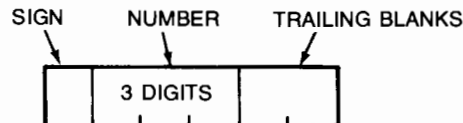
A*B =225      THE SUM OF A AND B IS 30
```

3-8. NUMERIC OUTPUT FORMATS

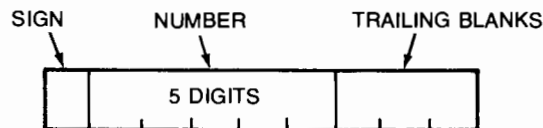
Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least one position to the right containing blanks. The width is always a multiple of three; the minimum width is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

Integers

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



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



Examples of integers:

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

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

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

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

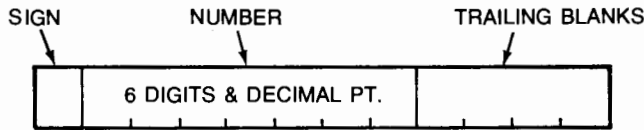
These integers are mixed in magnitude, but none are greater than 32767 or less than -32767:

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

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

Fixed-Point Numbers

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

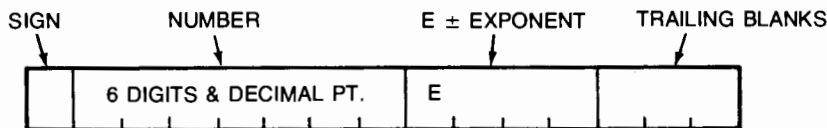


Examples of fixed-point numbers:

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

Floating-Point Numbers

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



Examples of floating-point numbers:

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

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

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

3-9. TAB FUNCTION

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

Format

TAB (*integer expression*)

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

3-10. READ/DATA/RESTORE

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

Format

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

Parameters

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

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

When a READ statement is executed, each variable is assigned a new value from the constant list in a DATA statement. RESTORE allows the first constant to be assigned again when READ is next executed or, if a label is specified, the first constant in the specified DATA statement.

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

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

Statements

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

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

Examples

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

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

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

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

3-11. INPUT

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

Format

INPUT *variable list*

Parameters

variable list list of variables separated by commas.

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

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

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

and

```
20 INPUT X
```

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

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

If you want to terminate the program and return control to the BASIC Interpreter, type Control Q (Q°).

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.

Format

```
COM variable list
```

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

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

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

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

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

3-14. PAUSE

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

Format

PAUSE [n]
Parameter
n optional parameter. If used, 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
```

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

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

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

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

4-1. STRING

A string is a set of 1 to 72 characters enclosed by quotation marks or the null string (no characters).

Typical Strings: "ABCDEFGHJKLMNOP"
"12345"
"BOB AND TOM"
"MARCH 13, 1970"

Null String: " "

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

Apostrophes and control characters are legal as string characters.

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

Strings

Strings are manipulated in string variables. For example:

$\begin{array}{ccc} 100 \text{ A\$} & = & \text{"THIS IS A STRING"} \\ \uparrow & & \uparrow \\ \textit{string} & & \textit{string} \\ \textit{variable} & & \end{array}$

200 B\$ = A\$(1,10)

↑ ↑

string *substring*

variable (*defined later*)

300 C\$ = ""

↑ ↑

string *null string*

variable

4-2. STRING VARIABLE

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

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

```
710 DIM A$(72),B$(20),C$(50)
```

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

100	DIM A\$(72)	<i>Sets physical length of A\$</i>
200	LET A\$="SAMPLE STRING"	<i>Logical length of A\$ is 13</i>
300	LET A\$="LONGER SAMPLE STRING"	<i>Logical length of A\$ is now 20</i>

4-3. SUBSTRING

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

A substring is defined by subscripts placed after the string variable. Characters within a string are numbered from the left starting with one. Subscripts must be positive, non-zero, and less than 73. Non-integer subscripts are rounded to the nearest integer.

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

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

prints the substring

BCDEF

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

```
300 PRINT Z$(3)
```

prints the substring

CDEFGH

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

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

Prints the substring

B

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

4-4. STRINGS AND SUBSTRINGS

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

```
100 A$ = B$(6,5) A$ now contains a null string.
```

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

Substrings can become strings. For example:

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


Strings

prints the string

```
CDE
```

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

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

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

prints the string

```
AB123FGH
```

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

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

See the STRING IF statement description in this section.

4-5. STRING DIM

Format

DIM string variable (number of characters in string)

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

The number of characters specified for a string in its DIM statement must be expressed as an integer from 1 to 72.

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

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

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

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

Example

```
35 DIM A$(72), B$(60)
40 DIM Z$(10)
45 DIM N$(2), R(5,5), P$(3)
```

4-6. STRING ASSIGNMENT

Format

$ \begin{array}{l} \text{[LET] } \begin{array}{l} \text{string variable} \\ \text{substring variable} \end{array} = \begin{array}{l} \text{"string literal"} \\ \text{string variable} \\ \text{substring variable} \end{array} \end{array} $

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

Strings contain a maximum of 72 characters, enclosed by quotation marks. String variables having more than 1 character must be mentioned in a DIM statement.

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

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

Example

```
200 LET A$ = "TEXT OF STRING"
210 B$ = "*** TEXT !!!"
220 LET C$ = A$(1,4)
230 D$ = B$(4)
240 F$(3,3)=N$
```

4-7. STRING INPUT

Format

$ \text{INPUT } \begin{array}{l} \text{string variable} \\ \text{substring variable} \end{array} \cdots $

Strings

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

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

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

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

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

Example

```
50 INPUT R$
55 INPUT A$,B$, C9, D10
60 INPUT A$(1,5)
65 INPUT B$(3)
```

4-8. PRINTING STRINGS

Format

<pre>PRINT <i>string variable</i> <i>substring variable</i> [<i>string variable</i> , <i>substring variable</i> , ...]</pre>

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

String and numeric values may be mixed in a PRINT statement (example statements 115 and 130 below).

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

WHAT IS

while statement 115 prints

YOUR NAME?END OF STRING 642

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

Example

```

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

```

4-9. READING STRINGS**Format**

<pre> READ <i>string variable</i> <i>substring variable</i> [<i>string variable</i> <i>substring variable</i> , ...] </pre>
--

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

A string variable (to be assigned more than 1 character) must be mentioned in a DIM statement before attempting to READ its value.

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

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

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

Example

```

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

```

4-10. STRING IF

Format

IF <i>string var. relational oper. string var.</i> THEN	<i>statement number label</i>
	<i>statement</i>

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

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

Characters are compared by their ASCII representation.

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

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

Strings may not use Boolean expressions.

Example

```

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

```

4-11. LEN FUNCTION

Format

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

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

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

Example

```

469 PRINT LEN(A$)
479 PRINT LEN(X$)
489 PRINT "TEXT"; LEN(A$); BS, C
499 IF LEN(P$) #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 <i>"string literal"</i> [, <i>string literal"</i> . . .]

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).

Strings up to 72 characters long may be stored in a DATA statement.

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:

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

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

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.

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 string variable
                                     substring variable 'substring variable' [, . . .]
```

The READ # statement reads string and substring values from a file.

String and numeric values may be mixed in a file and in a READ number statement; they must be separated by commas.

Example

```
710 READ #1,5; A$, B$  
715 READ #2; C$, A1, B2, X  
720 READ #3,6; C$(5),X$(4,7),Y$  
730 READ #N,P; C$, V$(2,7),R$(9)
```

If the file number is not equal to a file position number as defined in a FILES statement, input will be read from the logical unit of that number.

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 <i>e</i> 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 largest integer $\leq x$.
LEN(x\$)	Determines length (no. of characters) in character string identified by string variable x\$. See Section IV.
LOG(x)	Gives base 10 logarithm of variable or expression.
LN(x)	LN provides the logarithm of a positive expression to the base <i>e</i> .
OCT(x)	This function prints the octal equivalent of an integer value. The maximum possible range of the returned variable is 0-177777 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. x may have any value. A sequence of random numbers is repeatable if the initial value of x is negative and is followed by a positive argument. A random sequence can be achieved with positive arguments.

Functions

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 responses 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. |
| <i>expression</i> | provides a formula such as $X * X$ or $X \uparrow \text{TAN}(X)$. Whenever the function is called in the program, this formula will be evaluated. |

Example

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

When FNA (100) is called for in statement 30, the formula defined for FNA is evaluated to determine the value printed. Note that the results of the function may be used in computation:

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

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

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


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

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

6-1. GOSUB/RETURN

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

Format

GOSUB *statement number label*

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

RETURN

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

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

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

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

Subroutines

Examples

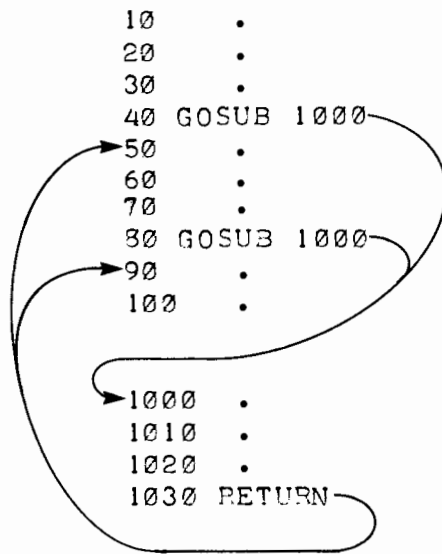
In the first example, line 20 contains a simple GOSUB statement; the subroutine is in lines 50 through 70, with RETURN in line 70.

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

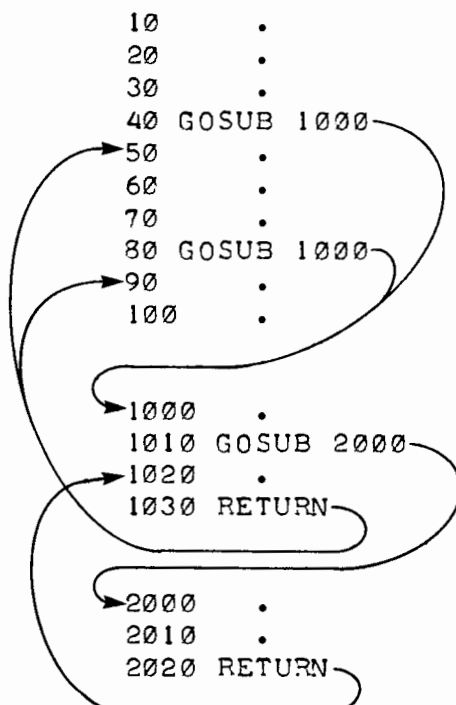
The GOSUB statement can follow the subroutine to which it transfers as in the example below.

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

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



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



Subroutines

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

This sequence shows logically nested GOSUB's:

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

The order in which this program is executed is:

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

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

6-2. CHAIN

The CHAIN statement terminates the current program and begins execution of another program, optionally starting at a specified statement number.

Format

```
CHAIN string variable
      string literal [, statement number label]
```

The *string variable* or *literal* is the name of a Real-Time BASIC program that is in your library. 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.

CHAIN calls the program identified by the string expression, and it replaces the current program. When the program called by CHAIN finishes execution, it terminates and does not automatically return to the calling program. The called program may call another program, including the original calling program, with the CHAIN statement.

Files do not remain open when one program chains to another.

Only variables declared in a COM statement are saved during a CHAIN operation. All variables and arrays of the current program that were not declared in COM are lost when the new program begins execution. All programs must contain the same number of file positions in the FILES statements so that common will be properly aligned.

If the programs are CSAVED, the time required to execute the CHAIN statement is reduced.

>LIST

```
10 REM..PROGRAM MAIN
20 LET X=200
30 LET A=X*3
40 PRINT "A= ";A
50 PRINT "LEAVE MAIN AND ENTER SUBA AT LINE 30"
60 CHAIN "SUBA",30
70 END
```

>LIST

```
10 REM..PROGRAM SUBA
20 PRINT "THIS STATEMENT IS NOT EXECUTED"
30 PRINT "ENTER SUBA - LINE 30"
40 LET B=125
50 LET C=B*2
60 PRINT "C= ";C
70 PRINT "END OF SUB A - TERMINATE HERE"
80 END
```

>SAVE SUBA

>RUN MAIN

A= 8.000000E+06

LEAVE MAIN AND ENTER SUBA AT LINE 30

ENTER SUBA - LINE 30

C= 15625

END OF SUB A - TERMINATE HERE

The main program, MAIN, calls program SUBA with a CHAIN command in line 50. Execution of SUBA begins in line 30, and execution terminates with the last line of SUBA. None of the variable values from MAIN are saved following execution of CHAIN.

6-3. CALL

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

Format

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

Parameters

<i>subroutine name</i>	name of the routine as entered into the system during system generation or when loaded on-line.
<i>parameter list</i>	list of variables or constants to be passed to the subroutine or variables into which the subroutine places information for the calling program. Spaces are not allowed between the subroutine name and the left parenthesis.
FAIL: <i>statement</i>	optional subroutine failure return. See paragraph 6-4.

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

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

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

A CALL statement is rejected by the interpreter unless the TABLES command has been given to specify the legal set of subroutine names.

Examples

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

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

```

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(R), OV=NN , INTG, ENT=NUM, FIL=FILEXX
0026  C
0027  C   WHERE:  R      INDICATES REAL PARAMETER (STRINGS ARE ALWAYS REAL)
0028  C           NN      INDICATES THE OVERLAY NUMBER
0029  C           FILEX    INDICATES THE FILE NAME OF THE RELOCATABLE FOR
0030  C                   FOR THIS FUNCTION.
0031  C
0032  C
0033  C
0034  C       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  C       NUM =I(2)/256
0042  C       RETURN
0043  C       END

```

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

```

0001  FTN,L,M
0002      SUBROUTINE CHR5(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 AS(10)
0015  C       20 AS="ABCDE"
0016  C       30 CHR5(65,AS)
0017  C       40 PRINT AS
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 FOLLOWS:
0029  C
0030  C   CHR5(I,RV), OV=NN, ENT=CHR5, FIL=FILEXX
0031  C
0032  C   WHERE:  I      INDICATES A INTEGER VARIABLE PASSED TO 'CHR5'
0033  C           RV      INDICATES A REAL VARIABLE (STRINGS ARE ALWAYS
0034  C                   SPECIFIED AS REAL) RETURNED FROM 'CHR5'
0035  C           FILEXX  INDICATES THE FILE NAME OF THE RELOCATABLE FOR
0036  C                   THIS SUBROUTINE.
0037  C
0038  C
0039  C
0040  C
0041  C
0042  C       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  C       J(2)=IAND(J(2),3776)
0050  C       J(2)=IOR(I*256,J(2))
0051  C       RETURN
0052  C       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.

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

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

These routines are described in Sections XI, XV, and XVI.

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

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

```

1000 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)
```

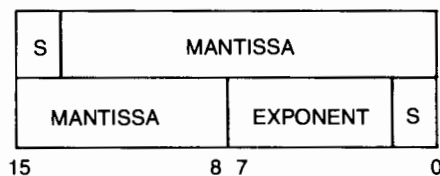
sets the ERRCD flag to the value of N. After execution of your subroutine you can examine the error code by using the IERR function. The value of I is unchanged.

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:

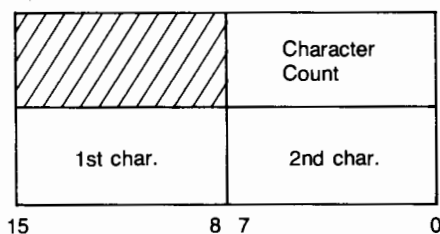
```
10 I= SERR (0)
```

6-7. PARAMETER CONVERSION

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



and for string is:



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

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

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

```

0001  FTN,L,M
0002      SUBROUTINE EXEC9(I)
0003  C
0004  C
0005  C
0006  C
0007  C THIS SUBROUTINE CALLS THE RTE 'EXEC' TO SCHEDULE A PROGRAM WITH WAIT
0008  C
0009  C   FOR EXAMPLE:
0010  C
0011  C       10 DIM A$(6)
0012  C       20 PRINT "INPUT PROGRAM NAME";
0013  C       30 INPUT A$
0014  C       40 CALL EXEC9(A$)
0015  C
0016  C
0017  C       >RUN
0018  C
0019  C       INPUT PROGRAM NAME?PROGA
0020  C
0021  C
0022  C
0023  C THE SUBROUTINE'S DESCRIPTION THAT MUST BE INPUT TO THE TABLE
0024  C GENERATOR TO CREATE THE PROPER ENTRY IN THE BRANCH AND MNEMONIC
0025  C TABLE IS AS FOLLOWS:
0026  C
0027  C   EXEC9(RA), OV=NN , INTG, LNT=EXEC9, FIL=EXEC9R
0028  C
0029  C       WHERE: RA      INDICATES REAL ARRAY PARAMETER( STRINGS
0030  C                   ARE ALWAYS SPECIFIED AS REAL )
0031  C                   NN      INDILATES THE OVERLAY NUMBER
0032  C                   EXEC9R   INDILATES THE FILE NAME OF THIS SUBROUTINE'S
0033  C                   RELOCATABLE.
0034  C
0035  C
0036  C   DIMENSION I(4)
0037  C
0038  C   CALL EXEC(9,I(2))
0039  C   RETURN
0040  C   END

```

BASIC program

FORTRAN Subroutine

Figure 6-3. FORTRAN Subroutine to Convert String Parameter

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 the Real-Time BASIC CREATE command if you are using it with BASIC programs.

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.

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 filename1 [, filename2, . . . filenamen [:security[:cartridge]]]
```

Parameters

<i>filename</i> _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.
<i>security</i>	optional security code which may be supplied with each filename.
<i>cartridge</i>	optional cartridge reference which may be supplied with each filename. (See Section IX for more information about security codes and cartridge reference parameters.)

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.

These numbers are used in the program to reference the files. For instance, the statement:

```
100 PRINT #2;A
```

would print the value of A onto the file named SCORE. This feature allows most programming to be done independently of files to be used. The FILES statement may be added any time before running the program.

Example

```
10 FILES MATH, SCORE, *, 0, NAMES
20 FILES GRP, FILE27, SAMPLE:JS:10
30 FILES MATH, DATA
```



7-4. ASSIGN STATEMENT

The ASSIGN statement is used to change the file referred to by a specified file number during the execution of a program.

Format

ASSIGN *filename* [:*security*[:*cartridge*]] , *file number* , *return variable*

Parameters

<i>filename</i>	the name of a file, (i.e., a literal string of up to six characters) enclosed in quotes, or a string variable containing a literal string.
<i>security</i>	optional security code. Must be supplied if file was created with a security code.
<i>cartridge</i>	optional cartridge reference (label or LU number).
<i>file number</i>	a number, variable, or expression whose value is between 1 and 16, indicating a file's position. The file number should not exceed the number of files declared in the FILES statement.
<i>return variable</i>	the value returned to this variable when the statement is executed. Values are summarized in Table 7-1.

When the ASSIGN statement is executed, the named file replaces the file previously referenced by the file number in the statement. Subsequent file references using this number will apply to the new file. Data written to the old file will be intact.

Example

```
20 ASSIGN A$, 3, B1
30 ASSIGN "NEWFL", S2, J
40 ASSIGN "$F2", 6, C
```

After each file is assigned you should test the value of B1, J, and C for error conditions as summarized in Table 7-1.

Each file that is assigned (opened) requires 144 words of user memory space.

Table 7-1. ASSIGN Return Variable Values

RETURN VARIABLE	MEANING
-1	Disc is inoperable.
-5	File number is out of range. It does not correspond to any of the positions in the FILES statement.
-6	Cartridge not found or file not found.
-7	Invalid security code.
-8	File currently open to a program that demands exclusive use of the file or to 7 programs (the maximum allowed).
-13	Cartridge is locked.

7-5. IF END # THEN STATEMENT

The IF END statement sets a flag for a specified file so that if and when an end-of-file condition occurs in reading and writing the file, control is transferred to a specified statement. If the flag is not set, an end-of-file condition causes the program to terminate.

Format

IF END # *file number* THEN *statement number label*

Parameters

file number a number, variable, or expression whose value is between 1 and 16, indicating a file's position.

statement number label the number of the statement to which control will transfer.

When an end-of-file condition occurs during execution of a READ # statement, the IF END statement transfers control to the statement specified. The IF END statement remains in effect until another IF END for the same file changes it, or until an ASSIGN statement containing the same file number is executed.

An end-of-file condition occurs when a file read operation encounters an end-of-file mark or the physical end of file, or when a file write operation encounters the physical end of the file.

Example

```
10 IF END #3 THEN 125
```

7-6. RESTORING THE DATA POINTER

A READ # statement may be used to reposition the data pointer to the start of a file. The statement can be used for any file accessed by the program.

Format

READ # *file number*, 1

Parameter

file number a number, variable, or expression whose value is between 1 and 16, indicating the file's position in the FILES statements.

When the statement is executed, the data pointer is set to point to the beginning of the first record. A serial read or print will begin at that position. Do not use PRINT #1, 1 to reset the pointer because this will delete the content of record 1.

Example

300 READ #1, 1 *The pointer for file position #1 is reset.*

7-7. SERIAL FILE READ STATEMENT

The READ # statement can be used to read items from a file into numeric or string variables. The first item read is the item following the current position of the data pointer.

Format

READ # *file number*
lu ; *variable list*

Parameters

file number a number, variable, or expression whose value is between 1 and 16, indicating a file's position. An expression is rounded to the nearest integer.

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*).

variable list a series of variables separated by commas. The rules governing this list are the same as those described for the READ statement described in Section III.

The READ # statement fills the variables in the list from the designated file beginning with the data item referenced by the data pointer and moving from record to record as necessary. A subsequent file read will start with the first data item not used by the prior read; record boundaries of portions of records not containing data are ignored.

Strings and numbers may be intermixed in the data list, but the types in the item list and data list must correspond in kind and order. The destination for a string value must be a string variable; the destination for a numeric value must be a numeric variable.

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

It is possible to check the type of the next data item with the TYP function described later in this section.

When an attempt is made to read beyond a logical or physical end-of-file, an end-of-file condition occurs. Unless an IF END statement transfers control to another statement in the program, the program terminates.

Example

```
400 READ #2; A, B(3), R(1,3)
```

7-8. READING A RECORD

The READ # statement may also be used to read items from a designated record into a numeric or string variable. This is called a direct read.

Format

READ # *file number, record number; variable list*

Parameters

<i>file number</i>	same as for READ #, paragraph 7-7.
<i>record number</i>	a number, variable, or expression indicating on which record the list is to be printed. Ignored if <i>file number</i> refers to LU number or type 0 file.
<i>variable list</i>	see paragraph 7-7.

Execution of a read record statement fills the variables in the variable list from the designated record only, starting at the beginning of the record. Encountering an end-of-record mark or the physical end-of-record will generate an end-of-file condition and terminate the program unless an IF END statement is provided.

The file pointer is positioned at the start of the record initially but may be left in the middle of the record following the read operation. In this case, the remainder of the record can be read by a subsequent serial read.

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

Example

```
75 READ #1,3; A, B1, C(I,J+2)
80 READ # 1,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 # ^{*file number*}_{*lu*} ; *print list* [, END]

Parameters

<i>file number</i>	a number, variable, or expression whose value is between 1 and 16, indicating a file's position in the FILES statement.
<i>lu</i>	if a file position corresponding to the <i>file number</i> 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 (<i>lu</i>).
<i>print list</i>	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.)

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.

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.

If a program terminates because of an error or you break program execution, files may not be completely updated.

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

Examples

20 FILES FILA	
30 PRINT #1; A,"STRING",A\$	<i>The value of A, the string "STRING", and A\$ are printed on FILA.</i>
20 FILES LINEP	<i>LINEP is a type 0 file corresponding to the line printer.</i>
.	
.	
60 PRINT #1; "SUMMARY"	<i>The string literal SUMMARY is printed on the line printer.</i>
100 FILES FILEA,FILB,FILC,0,FILD	
150 PRINT #4; A,B,C	<i>The variables A, B, C are punched on the paper tape punch, LU 4.</i>
200 PRINT#(I+J); 2,42,A,B,C,D(3,5),END	<i>The items are printed on the file in position I+J followed by an end-of-file mark.</i>

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

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

Example

```
200 PRINT #3,4; A,B,C,D
210 PRINT #J,9
```

Positions file pointer to record 9 on file named in position J.

7-11. TYP FUNCTION

The TYP function returns the type of the next data item for a particular file. This function is used in conjunction with file READ statements since the variables into which the data is read must be string if the item is string; numeric if the item is numeric.

Format

TYP (*file number*)

Parameter

file number a number, variable, or expression whose value is between 1 and 16, indicating a file's position.

The value returned by TYP depends on the type of the next data item in the file.

TYP(x)	MEANING
1	number
2	string
3	end-of-file
4	end-of-record

If the *file number* is greater than zero, the file is treated as a serial file. This means that the value 4 is never returned as end-of-record marks are skipped in a serial file.

If the *file number* is less than zero, the file is treated as a direct file and any of the values of TYP may be returned. Since the file number is based on the absolute value of the expression, an expression equal to -1 means that file number 1 is examined and treated as a direct file.

If the *file number* is zero, TYP returns a result based on the current position of the pointer to the DATA statements (see READ/DATA/RESTORE description, paragraph 3-10). The value 4 is never returned, and 3 means end-of-data.

7-12. MODIFYING RECORDS

If you print an item to a record, all items following that one are no longer in the record. It is necessary to read and save, either in memory or on another record, all items following the one you want to modify and to copy these items back on the record after printing the new item. You must also be sure that the item you are modifying is not replaced with data which requires more space in the record than is available. This can occur if you replace a string with a longer string.

If you are modifying an item in a serial file, you must use this same procedure but copy all items in the file which follow the item you are modifying.

Multi-User Real-Time BASIC is itself a program, and as such must be made available for use on your system. Procedures for loading BASIC are provided in Appendix D. Once BASIC is loaded and ready for use, you need simply schedule BASIC for operation as described below.

8-1. SCHEDULING BASIC

The command to start the BASIC Interpreter is:

```
:RUN,BASIC [,console[,list[,input[,output[,error list]]]]]
```

<i>console</i>	—is the logical unit number of the keyboard device you are using. Default is LU 1, the system console, or the <i>lu</i> of your MTM terminal.
<i>list</i>	—is the logical unit number of the list device you want to use. Default is LU 1, the system console, or the <i>lu</i> of your MTM terminal.
<i>input</i>	—is the logical unit number of the input device you want to use. Default is LU 5, the paper tape reader.
<i>output</i>	—is the logical unit number of the output device you want to use. Default is LU 4, the paper tape punch.
<i>error list</i>	—is the logical unit number of the error message logging device you want to use. Default is LU 1, the system console, or the <i>lu</i> of your MTM terminal.

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

BASIC may be scheduled from either the RTE Operating System, which prompts with an asterisk (*), or from the File Manager, which prompts with a colon(:).

The following is an example of the commands used to schedule BASIC starting with the Operating System and proceeding through the File Manager to BASIC.

```
*RUN,FMGR
:RUN,BASIC
BASIC READY
>
```

There may be more than one copy of BASIC on your system. See Appendix D for more information about copies of BASIC.

8-2. USING BASIC

BASIC prompts for commands and program statements with the greater than symbol(>).The first command you should enter, if you plan to use subroutines in your program, is the TABLES command. This command declares the names of two tables used by BASIC, the Branch and Mnemonic Tables.

To specify the Branch and Mnemonic tables enter:

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 sub-routine branch information.

mnemonic filename is the name of the file, created by the RTE BASIC Table Generator, which contains sub-routine mnemonics.

SC is an optional security code parameter.

CR is an optional positive cartridge number or negative logical unit number.

Procedures for table generation are provided in Section XIV.

BASIC programs can be stored as File Manager files with the BASIC SAVE command and reloaded with the LOAD and RUN commands. The LOAD command does not execute the program, but merely makes it available for execution or editing. The RUN command runs the current program if you do not provide a file name with it, or it loads and runs the program in the file name provided. In BASIC the file name and program name are the same. Sections IX and X provide a complete description of the many operating commands available to you with Real-Time BASIC.

8-3. START UP OPTIONS

You may input commands to BASIC from the console keyboard or from an ASCII disc file previously prepared by using the RTE Interactive Editor, EDITR, or the File Manager, FMGR. If you input commands from a disc file, you may choose from two methods of initiating BASIC:

- you may start BASIC from the console as shown below, or
- you may schedule BASIC from another program. The program may be written in FORTRAN, ALGOL, Assembly language, or BASIC.

The ASCII command file can contain any legal BASIC commands. It may not, however, contain any BASIC statements. Following is an example of an ASCII command file:

>TABLES BRT,MNT	<i>Sets up Branch and Mnemonic Tables.</i>
>CREATE DATA1,20	<i>Creates a data file.</i>
>RUN PROGA	<i>Runs program PROGA.</i>
>LIST LP	<i>Lists program to type 0 file,LP, corresponding to line printer.</i>
>BYE	<i>Terminates Real-Time BASIC.</i>

To start BASIC with a command file from the console, input the following command to the File Manager:

```
:RU,BASIC,fl,na,me[,console,list]
```

fl,na,me is the name of the file containing the commands. Commas must be used to separate every two letters. The name is actually *fname*.

console is the logical unit number of the terminal you are using. Error messages will be output at the specified terminal. The default is LU 1, the system console.

list is the logical unit number of the list device you want to use. The default is LU 1, the system console.

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

Assembly language:

JSB EXEC	<i>Transfer control to RTE.</i>
DEF *+7	<i>Request code (.10 if schedule without wait).</i>
DEF .9	<i>Name of BASIC.</i>
DEF BASIC	
DEF P1	} <i>Parameters.</i>
DEF P2	
DEF P3	
DEF P4	
DEF P5	
.9 DEC 9	<i>Continue execution of program.</i>
BASIC ASC 3,BASIC	<i>Schedule with wait.</i>
P1 ASC 1,FN	<i>Name of BASIC Interpreter.</i>
P2 ASC 1,AM	<i>Command file name.</i>
P3 ASC 1,E	<i>Console logical unit number.</i>
P4 DEC 1	<i>List logical unit number.</i>
P5 DEC 6	

FORTRAN

DIMENSION NAME(3)	<i>Store name of BASIC in integer array name.</i>
DATA NAME/2HBA,2H51,2HC /	
CALL EXEC(9,NAME,2HFN,2HAM,2HE ,1,6)	<i>Transfer control to RTE.</i>
	<i>FNAME is the command file name. LU 1 is the console, LU 6 the list device.</i>

For additional information on scheduling BASIC from programs see the RTE-II or RTE-III Programming and Operating Manual and the Batch-Spool Monitor Reference Manual.

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.



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

<i>limits</i>	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.
<i>filename</i>	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.
<i>security</i>	optional security code. Must use if program saved with security code.
<i>cartridge</i>	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.

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 VII.)
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.

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

<i>limits</i>	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.
<i>filename</i>	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 <i>filename</i> is specified, the program will be output to LU 4 or the LU number specified as the output parameter in the RUN,BASIC command.
<i>security</i>	optional security code.
<i>cartridge</i>	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.

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

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

Examples

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

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

<i>limits</i>	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.
<i>filename</i>	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).
<i>security</i>	optional security code. Must be supplied if program saved with security code.
<i>cartridge</i>	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.

9-4. REPLACE

The REPLACE command enables you to replace a source or semi-compiled program on disc with the program (or part of the program) currently in memory. The entire file on disc is replaced in either case.

Format

```
REPLACE [limits] [filename [:security [:cartridge]]]
```

Parameters

<i>limits</i>	beginning and ending line numbers of the program in memory you want to store in the file. Limits must be separated by a comma. If no limits are specified, the entire program is stored.
<i>filename</i>	name of the program on disc which will be replaced. If no name is specified, the command executes like the SAVE command with output to the LU number specified as the output parameter in the RUN,BASIC command.
<i>security</i>	optional security code.
<i>cartridge</i>	optional cartridge reference (label or LU).

This command replaces an entire program stored on disc with a program or part of a program currently in memory. The name of this disc program is not changed by the operation.

Once the operation is completed, the program may be loaded from disc to memory and executed.

Examples

>REPLACE	<i>Program is output to a logical unit.</i>
>REPLACE 250,350 PROGS	<i>Lines 250 through 350 of the program in memory replace all of the PROGS file.</i>

9-5. DELETE

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

Format

```
DELETE [limits]
```

Parameter

<i>limits</i>	beginning and ending line numbers of the portion of the program to be deleted. Limits must be separated by a comma. If limits are not specified, the entire program is removed.
---------------	---

Operator Commands

DELETE effectively erases the program currently in memory. Versions of the same program stored on disc are not affected by the DELETE operation. The limits parameter allows you to delete specific portions of a program in memory.

The DELETE command is the only one which may be abbreviated; you may use DEL when entering this command.

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

Examples

>DELETE	<i>Deletes the current program from memory.</i>
>DELETE 50,425	<i>Deletes lines 50 through 425 of the current program.</i>
>DEL	<i>Deletes the current program from memory.</i>
>45	<i>Deletes line 45.</i>

9-6. CREATE

The CREATE command enables you to create a data file (type 1) on a specified disc cartridge.

Format

CREATE *filename* [:*security* [:*cartridge*]] [,*file length*]

Parameters

<i>filename</i>	name of file to be created. Must conform to the same naming conventions as imposed on programs, e.g. no more than six characters in length, and so forth.
<i>security</i>	optional security code to assign to file being created to prevent unauthorized access to it.
<i>cartridge</i>	optional cartridge reference (label or LU number) specifying on which cartridge the file is to be created.
<i>file length</i>	number of 128 word records in file. Restricted only by the amount of disc available on the specified cartridge or all cartridges if none is specified. The default file length is one 128 word record.

CREATE initializes a file with an end-of-file mark, that is, as an empty file containing only an end-of-file indicator. If there is not room on the disc for the file, an error message is printed. You can terminate BASIC with BYE and then use the File Manager PK (pack) command to pack the disc and/or purge unused files to recover contiguous space on the disc. See the Batch-Spool Monitor Reference Manual for information about the PK command.

Examples

```
>CREATE FILEA,10
```

Creates a file named FILEA containing 10 records.

```
>CREATE FL101:386,20
```

Creates a file named FL101 of length 20 with security code 386.

9-7. PURGE

The PURGE command enables you to remove a program or data file stored on disc.

Format

```
PURGE filename [:security [:cartridge]]
```

Parameters

<i>filename</i>	name of the file to be purged.
<i>security</i>	security code of file to be purged. Must be supplied if the file was created with a security code.
<i>cartridge</i>	optional cartridge reference (label or LU number).

This command removes the version of a program (or data file) stored on disc. Only the entire program can be purged. A version of the program currently in memory is not affected by the PURGE command.

Once purged, a program is no longer available for loading and/or execution. Purging a file does not make more space available on the disc. To do so you must use the File Manager PK (pack) command. See the Batch-Spool Monitor Reference Manual for information about this command.

Examples

```
>PURGE FL101:386
```

Purges file named FL101 with security code 386.

```
>PURGE PROGA
```

Purges the file containing PROGA.

```
>PURGE FILEA
```

Purges the data file named FILEA.

9-8. RENAME

The RENAME command enables you to establish a new name for a program or data file.

Format

RENAME *old filename* [:*security* [:*cartridge*]] ,*new filename*

Parameters

<i>old filename</i>	name of the file to be changed.
<i>security</i>	optional security code of file to be changed. Must be supplied if the file was created with a security code.
<i>cartridge</i>	optional cartridge reference (label or LU number).
<i>new filename</i>	name to be assigned to the file.

This command removes the name of a file on disc and replaces it with a new name. Once the RENAME operation is completed, the program may be loaded and executed by its newly assigned name only. The old name can no longer be used for the program or data file, but may be assigned to another file.

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

Example

>RENAME PROGA,PROGZ

The file named PROGA is now named PROGZ.

9-9. RESEQ

The RESEQ command renumbers a range of statement numbers in specified increments.

Format

RESEQ [*new initial number* [,*increment* [,*first old number* [,*last old number*]]]]

Parameters

<i>new initial number</i>	number to be assigned to the first statement. Default = 10.
<i>increment</i>	interval between new statement numbers. Default = 10.
<i>first old number</i>	number of first statement to be renumbered. If omitted, renumbering begins with the first statement.
<i>last old number</i>	number of last statement to be renumbered. If omitted, renumbering terminates with the last program statement.

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.

Examples

>RESEQ	<i>Renumber entire program in increments of 10.</i>
>RESEQ 20,5,15,123	<i>Statements 15 through 123 are renumbered in increments of 5 beginning with 20.</i>
>RESEQ 110,5	<i>The entire program is renumbered in increments of 5 beginning with the number 110.</i>

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

<i>limits</i>	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.
<i>filename</i>	name of the program (filename) to be loaded and executed. Name is specified only if the program is not currently in memory.
<i>security</i>	optional security code. Must be used if program was saved with security code.
<i>cartridge</i>	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	<i>Executes the program currently in memory.</i>
>RUN PROGA	<i>Loads PROGA if not in memory. Executes the program in either case.</i>
>RUN 50,75,PROGB	<i>Loads and executes statements 50 through 75 in PROGB.</i>

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	<i>Locks the line printer, LU 6.</i>
>UNLOCK	<i>Unlocks all locked devices.</i>

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.

9-13. LIST

The LIST command enables you to copy a program or portion of a program onto a specified device.

Format

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



Parameters

<i>limits</i>	beginning and ending line numbers of the portion of the program to be listed. Limits must be separated by a comma. If no limits are specified, the entire program currently in memory is listed.
<i>filename</i>	name of the type 0 file corresponding to the device on which the program is to be listed. Default is LU 1 or the LU specified as the list parameter in the RUN,BASIC command.
<i>security</i>	optional security code.
<i>cartridge</i>	optional cartridge reference (label or LU number).

The LIST command enables you to list the program in memory at your terminal or onto a specified device. The program is listed in ascending sequence according to line numbers. You may request a partial list by specifying line number limits. FOR. .NEXT statements are indented 2 spaces each for easy identification of program loops.

Examples

>LIST	<i>Lists the current program in memory.</i>
>LIST 120,180,FILEB	<i>Lists statements 120 through 180 from program in FILEB.</i>
>LIST FILEB	<i>Lists entire program in FILEB.</i>

9-14. *BR,BASIC

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

- When a program is executing and you wish to interrupt it, press any key at your terminal.
- An asterisk (*) or *lu* > is printed.
- Enter BR,BASIC or BR and the name of the copy of BASIC you are using.

Control is returned to the Interpreter at this point and the message OPERATOR TERMINATION IN LINE xxx is printed on the console.

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

Examples

*BR,BASIC

Break the BASIC program.

13>BR,BASCA

Break a copy of the BASIC program named BASCA.

9-15. CALLS

The CALLS command prints a list of all the mnemonics in the current Mnemonics Table including the name, form (subroutine/function), type of parameters, and overlay number of each subroutine or function.

Format

CALLS

There are no attendant parameters to this command.

Example

Parameter types
Overlay Number

```

>CALLS
IRSET(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
URITE S 3
PLTLIJ(I) S 3
AXIS(R,R,RA,R,R,R,R) S 3
NIJMR(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

DEBUGGING COMMANDS

SECTION**X**

Multi-User Real-Time BASIC provides commands that enable you to debug a program while it is running. The path of execution through a program can be traced; program execution can be interrupted at critical points; and variables can be displayed and modified. In addition, subroutine calls can be displayed and simulated.

Real-Time BASIC commands used to perform the above tasks are defined in this section. Table 10-1 lists the commands, briefly states their functions, and delineates applicable use limitations.

Table 10-1. Debugging Commands

COMMAND	FUNCTION	USAGE LIMITS
TRACE	Turns on tracing of program or selected statements in the program.	Normal command mode or during a break.
UNTRACE	Turns off trace facility.	None.
BREAK	Causes program execution to be interrupted at from 1 to 4 specified break points.	None.
UNBREAK	Turns off break points set by break command.	None.
RESUME	Resumes program execution following a break point or a call simulation.	During break only.
ABORT	Aborts program execution following a break point.	During break only.
SIM	Simulates a subroutine call: causes a program to list a subroutine call when encountered, and suspends program execution.	None.
UNSIM	Turns off simulation facility.	None.
SHOW	Displays values of variables specified. Follows BREAK or SIM.	During break only.
SET	Permits entry of new variable content. Follows BREAK or SIM.	During break only.

10-1. TRACE/UNTRACE

The TRACE command is used to turn on the BASIC facility to trace a program's execution from statement to statement. The UNTRACE command is used to turn the trace facility off.

Format

TRACE [limits]

UNTRACE

Parameter

limits beginning and ending line numbers of the statements within the program to be traced. If omitted, the entire program is traced. If one statement is to be traced, enter the line number of the statement.

The RUN command must be given to initiate the trace operation.

Once tracing begins, a message is printed for each traced statement executed. This message consists of an asterisk, the word TRACE, and the line number of the statement. You can follow the activity of the program by noting the statements listed as they are executed. If you use the TRACE command a second time, the new limits replace the existing ones.

The UNTRACE command may be given at any time a BASIC program is not executing.

Examples

```
>TRACE  
>RUN
```

Request that all of the current program statements be traced.

```
>TRACE 955  
>RUN
```

Request that statement 955 be traced.

```
>TRACE 75,200  
>RUN  
*TRACE 75  
*TRACE 80
```

Statements 75 through 200 are traced.

```
.  
.  
.
```

10-2. BREAK/UNBREAK

The BREAK command enables you to specify points within your program where program execution is to be suspended. A breakpoint can be a line number or range of line numbers. The UNBREAK command is used to eliminate scheduled breaks.

Format

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

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.

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	<i>Breakpoints are defined for lines 30,70,90, and 100.</i>
>RUN	
*BREAK 30	<i>Breakpoint 30 is reached.</i>
>>RESUME	<i>Resume execution by typing RESUME.</i>
*BREAK 70	<i>Breakpoint 70 is reached.</i>
>>UNBREAK 90	<i>Delete breakpoint 90.</i>
>>RESUME	<i>Resume execution.</i>
*BREAK 100	<i>Breakpoint 100 is reached.</i>
>>	

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.

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.

>

10-5. SIM/UNSIM

The SIM command simulates a subroutine call. This command is identical to a breakpoint in as much as program execution is suspended just prior to the execution of the statement. The UNSIM command is used to turn off the simulation.

Format

SIM
UNSIM

There are no attendant parameters to these commands.

SIM is entered prior to program execution (or resumption). When a subroutine call is encountered, BASIC lists the subroutine call and suspends the program. The suspension message printed contains an asterisk, the statement number, the word CALL, the subroutine name, and pertinent variables. To illustrate:

```
* 125 CALL AISQV (A,B(1),4,N)
```

You may display or modify variables at this point by using the SHOW and SET commands. To continue execution of the program, enter the RESUME command. Execution continues at the statement following the subroutine call.

To eliminate any further subroutine call simulation, enter UNSIM prior to resuming program execution.

To resume execution type RESUME; to terminate the program type ABORT.

Example

>SIM	<i>Request subroutine simulation.</i>
>RUN	<i>Execute the program.</i>
*125 CALL AISQV (A,B(1),4,N)	<i>BASIC prints information and suspends when</i>
>>SHOW B(3)	<i>subroutine is executed. Request value of B(3) var-</i>
	<i>iable.</i>
3.5	<i>BASIC prints the value.</i>
>>UNSIM	<i>Turn off subroutine simulation.</i>
>>RESUME	<i>Resume program execution.</i>

10-6. SHOW

The SHOW command prints the values of the items specified. This command may be used only during a break period or subroutine call simulation.

Format

SHOW *variable list*

Parameters

variable list numeric or string variables separated by commas. Expressions are not allowed.

Example

<code>*BREAK 100</code>	<i>Set breakpoint 100.</i>
<code>>>SHOW X,Y,Z(10),A\$</code>	<i>Request values of variables in list.</i>
<code>1.3 2.7 3.35544E+2 ABC\$</code>	<i>BASIC prints values: X=1.3, Y=2.7, etc.</i>
<code>>>RESUME</code>	<i>Resume program execution by typing RESUME.</i>

10-7. SET

The SET command allows you to set any variable to a constant value. This command may be used only during a break period or following a subroutine call simulation.

Format

Set *variable* = *constant* [,*variable*=*constant*,. . .*variable*=*constant*]

Parameters

variable numeric or string variable to be set to a value.

constant new value to be given to the variable. Note: Expressions are not allowed.

More than one variable may be set at a time. The equals sign must be used to equate the variable to a constant.

Example

<code>>RUN</code>	<i>Run the current program in memory.</i>
<code>*BREAK 100</code>	<i>Breakpoint 100 is reached.</i>
<code>>>SHOW A(20),B\$(2,5)</code>	<i>Examine variables.</i>
<code>1.357 ABCD</code>	<i>BASIC prints values of variables.</i>
<code>>>SET A(20)=-1.357</code>	<i>Modify the value of variable A(20).</i>
<code>>>SET B\$(2,5)=WXYZ</code>	<i>Modify the value of B\$(2,5).</i>
<code>>>RESUME</code>	<i>Resume program execution.</i>

REAL-TIME TASK SCHEDULING

SECTION

XI

11-1. INTRODUCTION

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

A task is defined as a group of BASIC statements initiated by a call to one of the BASIC scheduling subroutines (e.g. START, TRAP, etc.) and terminated by a RETURN statement. Each task is uniquely identified by the line number of the first statement in the task. If a task is initiated by executing line 2000 of the BASIC program, that task will be represented in all scheduling calls as 2000. No blanks are allowed in the name or between the subroutine name and the left parenthesis preceding the parameter list.

11-2. METHODS OF INITIATING TASKS

A task can be initiated in three different ways:

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

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

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

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

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

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

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

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

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

>RUN

TASK 2000 AT 10 SECOND INTERVALS

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

To terminate this example press any key on the system console or your terminal. The RTE system responds by printing an * or *lu* >. Type BR, BASIC (or BR, and the name of the copy of BASIC you are using) and press RETURN. The program will be aborted and Real-Time BASIC returns to command mode.

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

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

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

```
100 TRAP 5 GOSUB 2000
```

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

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

11-3. PRIORITIES

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

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

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

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

11-4. RESPONSE TIME

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

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

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

11-5. THE BASIC SCHEDULER

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

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

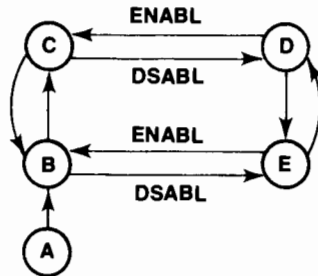
The BASIC Scheduler considers a task to be in one of five states at all times. For purposes of discussion, these states will be labeled A, B, C, D, and E. Refer to Figure 11-1.

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

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

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

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



State A — Undefined

A statement referencing the task has not yet been executed.

State B — Dormant

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

State C — Pending

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

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

State D — Pending/Disabled

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

State E — Dormant/Disabled

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

Figure 11-1. Task State Definitions

11-6. DSABL

The DSABL routine disables a specified task.

Format

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

Parameter

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

FAIL: *statement* optional error return statement.

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

Example

```
350 CALL DSABL(2000)           Disables task 2000.
```

11-7. ENABL

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

Format

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

Parameter

statement number label first statement number of task to be enabled. If 0, all tasks are enabled.

FAIL: *statement* optional error return statement.

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

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

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

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

11-8. SETP

The SETP routine sets the priority of a task.

Format

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

Parameters

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

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

Example

500 CALL SETP(1500,45)	<i>Task 1500 is given priority 45.</i>
------------------------	--

11-9. START

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

Format

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

Parameters

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

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

Example

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

Task 3300 will be scheduled in 35 seconds.



11-10. TIME

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

Format

CALL TIME(*time*)

Parameter

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

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

>LIST

```

10  LET L=17
20  CALL START(100,6)
29  WAIT (100)
30  GOTO 29
100  CALL START(100,5)
110  CALL TIME(T1)
120  LET S1=INT(T1/60)
130  LET S=INT(T1-S1*60)
140  LET H=INT(S1/60)
150  LET M=INT(S1-H*60)
160  PRINT #L;H;TAB(2);":";M;TAB(5);":";S
170  RETURN
180  END

```

>RUN

```

9      :49      :3
9      :49      :8
9      :49      :13
9      :49      :18
9      :49      :23
9      :49      :28

```

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

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

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

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

```
CALL TTYS(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.

Examples

```
100 TRAP 3 GOSUB 1000
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(5,1,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.

11-12. TRNON

The TRNON routine executes a task at a specified time.

Format

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

Parameters

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

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

Example

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

11-13. TTYS

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

Format

```
CALL TTYS(lu,trapn)
```

Parameters

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

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

Example

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

Associate trap number 5 and task 100.

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

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

11-14. PROGRAM EXAMPLE

Figure 11-3 represents the structure of the working program shown in figure 11-4. The routine shown in figure 11-2 generates the data required by the program in figure 11-4.

Line 999 in figure 11-4 defines an idle-loop. In order to get some idea of how much time is spent in looping, a counter can be installed as part of the loop. Different variables are used for each task since one task may destroy the contents of variables used by another task. When a variable is to be shared by more than one task, you must analyze the implications of interrupts breaking into the statement sequence.

```

4 LET L4=4
5 LET L6=6
10 LET C1=20
20 LET C2=1
30 LET C3=10
40 LET C4=3
50 LET C5=5
60 LET C6=.1
70 LET M1=36
100 FOR I=1 TO 100
110 LET X=C6*I
120 GOSUB 1000
130 NEXT I
140 PRINT# L4;999
150 STOP
1000 LET Y=C1*COS(C2*X)+C3*SIN(C4*X)+C5-2*C5*RND(X)
1005 PRINT# L4;Y
1010 LET P=Y+M1
1020 IF P<M1 GOTO 1200
1030 IF P>M1 GOTO 1300
1040 PRINT# L6;TAB(M1);"X"
1050 RETURN
1200 PRINT# L6;TAB(P);"X";
1210 PRINT# L6;TAB(M1);"I"
1220 RETURN
1300 PRINT# L6;TAB(M1);"I";
1310 PRINT# L6;TAB(P);"X"
1390 RETURN
1400 END

```

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

27.8551	10.6984
25.2468	9.84983
26.9359	4.74023
32.7251	6.6409
32.4648	8.21229
26.0227	5.04531
23.1503	7.74416
23.0191	10.4607
12.6957	13.2719
12.1831	9.58012
8.38185	19.0826
3.45362	18.99
-7.24108	24.6351
-1.18006	22.219
-5.77463	29.8881
-7.25732	29.5723
-10.3784	28.426
-8.12697	21.0262
-15.2137	19.6844
-12.8658	16.6032
-6.25166	14.049
-5.85369	16.0063
-8.8297	6.51914
-4.75087	4.82018
-7.68372	.951558
-6.94039	-6.74977
-12.7432	-7.43395
-13.3699	-9.90004
-16.9722	-13.6216
-18.1321	-15.5273
-20.8142	-11.5115
-17.0846	-14.503
-22.8928	-9.28221
-24.363	-8.73984
-23.5851	-8.01055
-26.8436	-2.87477
-21.9709	-2.86945
-28.5017	-4.23474
-22.5021	-9.50069
-14.2186	-8.39849
-10.5363	-16.9804
-10.8942	-14.0062
-6.624686	-18.1544
2.30019	-19.4267
6.92991	-23.0569
8.10668	-26.4572
12.2503	-26.6669
13.2217	-28.2566
15.5333	-23.6533
10.9137	999
8.84168	

Figure 11-2. Task Scheduling Program Example (Part 1)

	ASSIGN PERIPHERALS	1-20	
	INITIALIZATION OF TASKS	100-900	
	IDLE LOOP	999	
RETURN	READING & PLOTTING DATA	1000-1320	TASK 1
RETURN	PRINTING CALCULATION RESULTS ONTO TELETYPE	2000-2150	2
RETURN	GIVE TIME AND SUMMARY CALCULATIONS WHEN KEYED	3000-3090	3
RETURN	GIVE MESSAGE AT 8 AM	4000-4300	4
RETURN	GIVE MESSAGE AT 12 NOON	5000-5100	5
RETURN	GIVE MESSAGE AT 5 PM	6000-6100	6
RETURN	GIVE MESSAGE AT 00:05 AM	7000-7030	7
RETURN	BAD DATA PROCESSOR	8000-8120	8
RETURN	CONVERT SECONDS TO HHMMSS	9000-9060	9

Figure 11-3. Structure of Program Example in Figure 11-4.

<pre> 1 LET L1=1 2 LET L2=11 3 LET L3=11 4 LET L4=4 5 LET L5=5 6 LET L6=6 10 LET L0=11 20 LET D=100 100 REM***** SET UP TASK 1000 **** 110 LET D1=1 120 LET A1=0 140 LET N1=0 160 SETP(1000,50) 170 START(1000,D1) 200 REM***** SET UP TASK 2000 **** 210 LET S2=0 220 LET N2=0 230 LET D2=10 240 SETP(2000,70) 250 START(2000,D2) 300 REM*****SET UP TASK 3000 **** 310 SETP(3000,5) 320 TRAP 1 GOSUB 3000 330 TTYS(11,1) 400 REM*****SET UP TASK 4000 **** 420 SETP(4000,99) 430 TRNON(4000,80000) 500 REM*****SET UP TASK 5000 **** 510 SETP(5000,60) 520 TRNON(5000,120000) 600 REM*****SET UP TASK 6000 **** 610 SETP(6000,1) 620 TRNON(6000,170000) 700 REM*****SET UP TASK 7000 **** 710 TRNON(7000,105) 900 REM*****IDLE LOOP ***** 999 GOTO 999 1000 REM 1002 REM ** TASK 1000 ** READ DATA AND PLOT IT* 1004 REM 1010 START(1000,D1) 1020 READ# L5;X1 1030 IF X1>36 OR X1<-36 GOTO 8000 1040 LET A1=A1+X1 1050 LET N1=N1+1 1060 LET P1=36+X1 1070 IF P1>36 GOTO 1200 1080 IF P1<36 GOTO 1300 1090 PRINT# L6;TAB(36);"Y" 1100 RETURN 1200 PRINT# L6;TAB(36);"I"; 1210 PRINT #L6;TAB(P1);"X" 1220 RETURN 1300 PRINT# L6;TAB(P1);"Y"; 1310 PRINT# L6;TAB(36);"I" 1320 RETURN 2000 REM 2002 REM ** TASK 2000 ** DATA COMPRESSION ROUTINE* 2004 REM 2010 START(2000,D2) 2015 GOSUB 9000 </pre>	<p><i>Initialize variables which assign outputs to specific devices.</i></p> <p><i>Lines 100 through 900 schedule tasks 1000 through 7000.</i></p> <p><i>Set priority = 50.</i></p> <p><i>Tell Scheduler to initiate task 1000 D1 seconds from now.</i></p> <p><i>Since D1=1, task 1000 executes 1 second after statement 170 executes. This allows time for all other tasks to be initiated.</i></p> <p><i>Execute task 2000 in 10 seconds (D2=10.)</i></p> <p><i>Set priority to 5.</i></p> <p><i>Associate trap 1 with task 3000.</i></p> <p><i>Associate LU 11 with trap 1. Pressing any key on LU 11 interrupts BASIC and starts task 3000.</i></p> <p><i>Start task 4000 at 8 a.m.</i></p> <p><i>Start task 5000 at 12 noon.</i></p> <p><i>Start task 6000 at 5 p.m.</i></p> <p><i>Start task 7000 at 1 min. 5 sec. after midnight.</i></p> <p><i>Forces program to continue executing while waiting for tasks to be scheduled. Control returns here if no task is scheduled.</i></p> <p><i>Note that task 1000 and 2000 will try to execute simultaneously. Task 1000 has a higher priority than task 2000 so will be scheduled first or may interrupt task 1000 which must wait.</i></p> <p><i>The tasks defined in lines 1000 to 9060 are identical to subroutines.</i></p>
---	--

Figure 11-4. Task Scheduling Program Example (Part 2)

```

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

```

Figure 11-4. Task Scheduling Program Example (Part 2) (Continued)

```

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

```

Figure 11-4. Task Scheduling Program Example (Part 2) (Continued)

11-15. TABLE PREPARATION

In order to use the task scheduling subroutines, you must add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in Section XIV. A list of the RTETG commands required to add the subroutines described in this section is given there.

11-16. ERROR MESSAGES

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

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

In all cases the ERRCD flag is set to the error number. You may use the FAIL: option and IERR to interrogate the flag. See paragraphs 6-4 and 6-5.

BIT MANIPULATION OPERATIONS

SECTION

XII

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

12-1. BIT MANIPULATION WORD FORMAT

Each word within the computer can be thought of as a 16-bit shift register. The bits are numbered from right to left as shown in Figure 12-1. Each set of three bits is weighted 4-2-1. For example, a bit pattern of 101 has an octal value of 5.

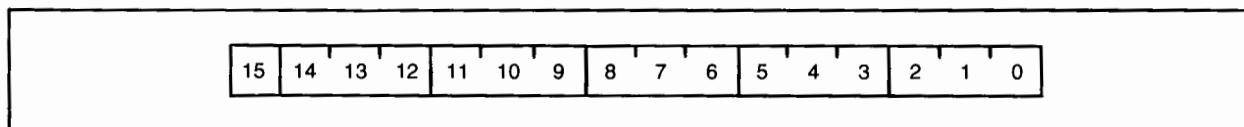


Figure 12-1. 16-Bit Word

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

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

12-2. AND

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

Format

$A = \text{AND} (arg1, arg2)$

Parameters

arg1 first value to be ANDed.

arg2 second value to be ANDed.

result A returned value of the operation.

Bit Manipulation Operations

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

<i>arg1</i>	0011
<i>arg2</i>	0101
<i>result</i>	0001

Example

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

ISETC function described in paragraph 12-9.

Results

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

12-3. IBCLR (Bit Clear)

This function sets selected bit positions to zero. The position to be set to zero is determined by the number given in the *bit posit* parameter where the bits are referenced right to left starting with zero. Refer to Figure 12-1 for the bit positions. Note that only one bit can be "zeroed" at a time.

Format

A = IBCLR (*value*, *bit posit*)

Parameters

value starting value.

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

result A returned value.

Example

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

Results

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

Bit position 8 is cleared (set to zero).

12-4. IBSET (Bit Set)

This function sets selected bit positions to "1". The position to be set to one is determined by the number given in the *bit posit* parameter where the bits are referenced right to left starting with zero. Refer to Figure 12-1 for the bit positions. Note that only one bit can be set to "1" at a time.

Format

$A = \text{IBSET}(\text{value}, \text{bit posit})$

Parameters

value starting value.

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

result A returned value.

Example

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

Results

```
X  =452  =100-101-010
      ↓
Z  =652  =110-101-010
```

Bit position 7 is set to 1.

12-5. IBTST (Bit Test)

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

Format

$A = \text{IBTST}(\text{value}, \text{test posit})$

Parameters

value value to be tested.

test posit the position of the bit is to be tested. The least significant bit is zero. If *test posit* is greater than 15 or less than zero, *result* is set to zero.

result A value of the tested bit (either a "1" or "0").

Example

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

X = the value to be tested.

15 = the left-most bit, or bit 15.

Z = a "1" or "0" depending on the value of bit 15 in word X.

12-6. IEOB

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

Format

A = IEOB (arg1, arg2)

Parameters

arg1 first value to be Exclusive OR'ed.

arg2 second value to be Exclusive OR'ed.

result A returned value of the operation.

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

arg1	0011
arg2	0101
result	0110

Example

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

Results

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

12-7. NOT

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

Format

$A = \text{NOT } (value)$

Parameters

value value to be complemented.

result A returned value of the operation.



Example

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

Results

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

12-8. OR

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

Format

$A = \text{OR } (arg1, arg2)$

Parameters

arg1 first value to be Inclusive OR'ed.

arg2 second value to be Inclusive OR'ed.

result A returned value of the operation.

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

<i>arg1</i>	0011
<i>arg2</i>	0101
<i>result</i>	0111

Example

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

Results

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

12-9. ISETC (Set to Octal)

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

Format

$$A = \text{ISETC} \left(\begin{array}{l} \text{"octal numb"} \\ \text{string variable} \end{array} \right)$$

Parameters

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

Example

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

Results

X is printed as 27765.

12-10. ISHFT (Register Shift)

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

Format

A = ISHFT (*value*, *shift numb*)

Parameters

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

Example

```

10 X = ISETC("276")           Shifts all bits two positions left.
20 Z = ISHFT(X,2)
30 PRINT OCT(Z)
40 END

```

Results

```

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

```

12-11. BRANCH AND MNEMONIC TABLE PREPARATION

In order to use the routines described in this section, you must enter the names of the functions and other information in the Branch and Mnemonic Tables. The procedure for doing this is described in Section XIV with a list of the precise commands required.

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

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

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

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

13-1. MAGNETIC TAPE OPERATOR COMMANDS

There are several operator commands available that allow you to manipulate the tape drive from the system console. The command format consists of a control word and file name.

Format

Command	Purpose
REWIND <i>mag tape unit filename</i>	Rewind the magnetic tape all the way back to the beginning.
WEOF <i>mag tape unit filename</i>	Write an end-of-file mark on the tape.
SKIPF <i>mag tape unit filename</i>	Skip to the end of the current file and stop at the beginning of the next file on the magnetic tape.
BACKF <i>mag tape unit filename</i>	Backspace past the previous file mark and stop.
Parameter	
<i>mag tape unit filename</i>	name of a type 0 file created by File Manager CR command and corresponding to a magnetic tape unit. For example: :CR,MT,8,BO creates a file named MT corresponding to LU 8. See the Batch-Spool Monitor Reference manual for more information.

13-2. MAGNETIC TAPE CALLS

Descriptions of all magnetic tape calls available in Real-Time BASIC are provided in the remainder of this section. Figure 13-2 contains a sample program which uses the routines described here.

13-3. MTTRT

The MTTRT routine writes a record onto a tape.

Format

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

Parameters

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

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

Example

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

13-4. MTTRD

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

Format

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

Parameters

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

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

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

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

Example

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

13-5. MTTPT

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

Format

```
CALL MTTPT(lu,fspace,rspace)
```

Parameters

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

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

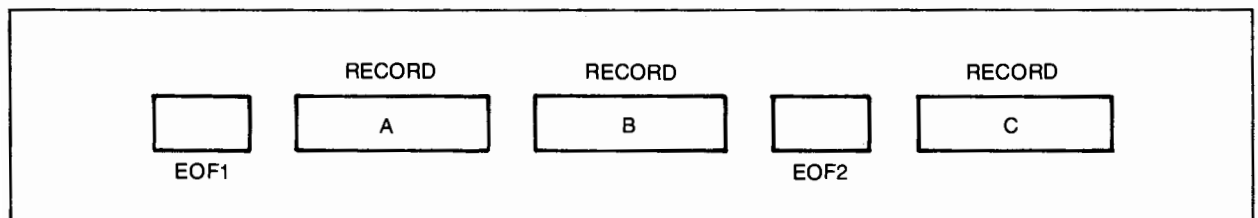


Figure 13-1. Record Positioning Example Using MTTPT.

Examples (Refer to figure 13-1)

1. File position is immediately after record C, and *fspace* = -1 (backspace 1 file). File position moves to immediately after record B before EOF2.
2. File position is immediately after record C; *fspace* = -2 (backspace 2 files) and *rspace* = 1 (forward space 1 record). File position moves to immediately after EOF1 before record A.
3. File position is immediately after record C, and *rspace* = -3 (backspace 3 records). File position moves between record A and B.

13-6. MTTFS

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

Format

CALL MTTFS(*lu,func*)

Parameters

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

Examples

435 CALL MTTFS(12,1)	<i>WRITE an EOF on LU 12.</i>
860 CALL MTTFS(11,2)	<i>REWIND LU 11. Control returns to program immediately.</i>
755 CALL MTTFS(11,3)	<i>REWIND LU 11 and make tape device unavailable.</i>
950 CALL MTTFS(10,0)	<i>Erase 4 inches of tape on LU 10.</i>

13-7. TAPE MANIPULATION ERRORS

The following message is given when tape errors occur:

ERROR MAGTP-*n* IN LINE *nnnn*

If *n* equals:

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

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

13-8. BRANCH AND MNEMONIC TABLE ENTRIES

During system generation, the routines described in this section must be entered in the Branch and Mnemonic Tables if the magnetic tape drive calls are to be used. The procedure for doing this is described in Section XIV.

13-9. Sample Program Using Magnetic Tape

Figure 13-2 contains a sample program which demonstrates some of the previously described routines. The tape is rewound, file records are written on it, it is rewound again and then read until an end-of-file is encountered. The tape is then backspaced twice to allow the last data record to be reread.

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

Figure 13-2. Tape Control Sample Program

SUBROUTINE TABLE GENERATION

SECTION

XIV

In order to call external subroutines and functions written in BASIC, FORTRAN, ALGOL, 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 foreground partition of memory. Remember that BASIC resides in a background 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.

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.

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

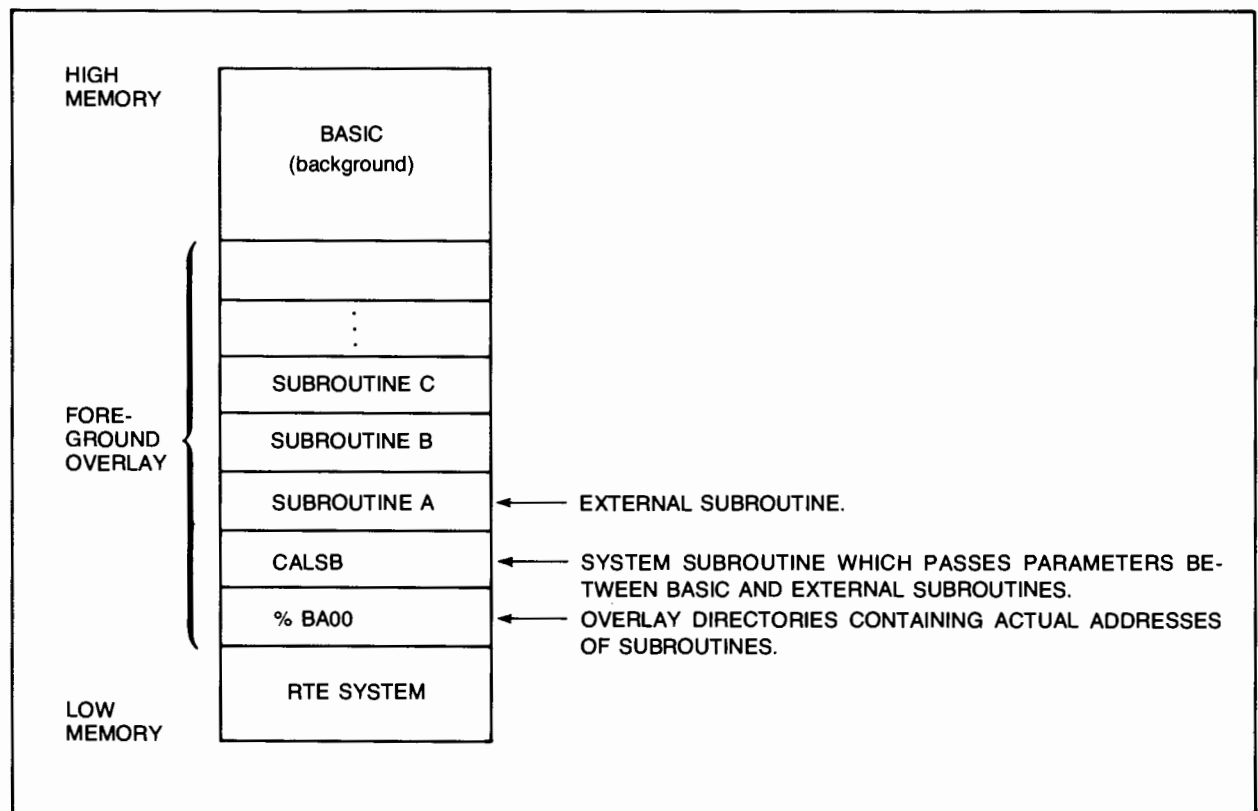


Figure 14-1. BASIC and an Overlay in Memory

14-1. RTETG

The RTE Table Generator (RTETG) operates interactively or in batch mode. You can provide the description of a subroutine or function from a type 3 or 4 disc file, on cards, paper tape, magnetic tape, or interactively at the system console or a remote terminal. 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, overlay directories, and a FMGR transfer file which is used to load the overlays. Each overlay directory 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.

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. The maximum number of overlay directories (and therefore, overlays) is 32.

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.

RTETG is a segmented program and requires a minimum of 5K of system background area.

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 an 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,fi,ln,am [,list]
```

or

```
*RUN,RTETG,fi,ln,am [, list]
```

fi,*ln*,*am* are pairs of characters which define the input file name. They must be separated by commas.

list is the logical unit number of the list device. Default is LU 6.

The RTETG commands are described below. If you are using a terminal to enter the commands, there will be no prompt character. Each command is entered as a record and terminated with a carriage return. A Control D indicates the end of all commands.

14-3. THE FIRST RTETG COMMAND

The first command you use must specify the files to be used for the Branch Table, the Mnemonic Table, the transfer file, and an identification letter to identify the group of overlay directories 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 Overlay Directories 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 directory when it is scheduled. Default is 80.

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

cref is the cartridge reference number to be used with the overlay directories. 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.

14-4. OTHER RTETG COMMANDS

For each subroutine or function you want to call from a BASIC program, you must use the following command:

name[(p1,p2, . . . , pn)],OV=nn[,SZ=mm] $\begin{bmatrix} \text{INTG} \\ \text{REAL} \end{bmatrix}$ [,ENT=epoint][,FIL=fname[:sc[:cr]]]

name is the subroutine or function name which may be from 1 to 7 characters.

(p1,p2, . . . , pn) is a list of the parameter types to be passed to the subroutine if it requires them. The list must be enclosed in parenthesis. *pn* equals:

$\begin{bmatrix} \text{I} \\ \text{R} \end{bmatrix} [\text{V}] [\text{A}]$

(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).

Subroutine Table Generation

OV= <i>nn</i>	indicates the overlay in which the subroutine will reside. The OV= must always precede <i>nn</i> . ($0 \leq nn \leq 31$)
SZ= <i>mm</i>	indicates the number of pages required for the overlay. This parameter is used only for RTE-III systems and need appear only once. If it appears more than once, the largest value supplied is used and a warning is printed. ($1 \leq mm \leq 32$)
INTG	is used if you are defining a function 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.
ENT= <i>epoint</i>	specifies the 1 to 5 character subroutine entry point name. If this name is identical to <i>name</i> it need not be specified. ENT= must precede the entry point name.
FIL= <i>fname</i>	specifies the name of a File Manager 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 library is searched for the routine. If the name is not found the Loader prompts for the name at the system console when you load a program which uses the routine.
<i>sc</i>	is the security code of the file named <i>fname</i> .
<i>cr</i>	is the cartridge reference (label or LU) of the cartridge on which the file named <i>fname</i> 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).

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.

The Overlay Directories are produced as separate standard relocatable programs named % BX*nn* and are stored in type 5 files. The third letter 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 corresponds to the entry point name. *nn*, the last two characters of the name, are digits which indicate the overlay number.

14-6. RTETG COMMANDS REQUIRED FOR LIBRARY SUBROUTINES

In order to use the subroutines described in various sections of this manual, you must enter the subroutine 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. The commands contain extra spaces for readability.

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 set of Overlay Directories.



First Command	Overlay Number	Function Value is Integer	Entry Point Name	Name of File Containing Routine
BTBL, MTHL, TRFL, IL=A				
Bit Manipulation	IBSET(I,I),	OV=0,	INTG,	ENT=IBSET
	IEOR(I,I),	OV=0,	INTG,	ENT=IEOR
	OR(I,I),	OV=0,	INTG,	ENT=BIOR
	AND(I,I),	OV=0,	INTG,	ENT=IBAND
	NOT(I),	OV=0,	INTG,	ENT=IBNOT
	ISHFT(I,I),	OV=0,	INTG,	ENT=ISHFT
	IBTST(I,I),	OV=0,	INTG,	ENT=IBTST
	IBCLR(I,I),	OV=0,	INTG,	ENT=IBCLR
	ISETC(RA),	OV=0,	INTG,	ENT=ISETC
	DAC(I,R),	OV=0,		ENT=DAC, FIL=A6940
6940 Subsystem	MPNRM,	OV=0,	ENT=MPNRM,	FIL=A6940
	RDWRD(I,IV),	OV=0,	ENT=RDWRD,	FIL=A6940
	WRWRD(I,I),	OV=0,	ENT=WRWRD,	FIL=A6940
	RDBIT(I,I,IV),	OV=0,	ENT=RDBIT,	FIL=A6940
	WRBIT(I,I,I),	OV=0,	ENT=WRBIT,	FIL=A6940
	SENSE(I,I,I,I),	OV=0,	ENT=SENSE,	FIL=A6940
2313/91000 Subsystems	AISQV(I,I,RVA,IV),	OV=1,	ENT=AISQV,	FIL=A2313
	AIRDV(I,RA,RVA,IV),	OV=1,	ENT=AIRDV,	FIL=A2313
	PACER(I,I,I),	OV=1,	ENT=PACER,	FIL=A2313
	NORM,	OV=1,	ENT=NORM,	FIL=A2313
	SGAIN(I,R),	OV=1,	ENT=SGAIN,	FIL=A2313
	RGAIN(I,R),	OV=1,	ENT=RGAIN,	FIL=A2313
Magnetic Tape	AOV(I,RA,RA,IV),	OV=1,	ENT=AOV,	FIL=A2313
	MTTRD(I,RVA,I,IV,IV),	OV=2,	ENT=MTTRD	
	MTTRT(I,RA,I,IV,IV),	OV=2,	ENT=MTTRT	
	MTTPT(I,I,I),	OV=2,	ENT=MTTPT	
	MTTFS(I,I),	OV=2,	ENT=MTTFS	
	SFACT(R,R),	OV=3,	ENT=SFACT,	FIL=PLOTR
Plotter	FACT(R,R),	OV=3,	ENT=FACT,	FIL=PLOTR
	WHERE(RV,RV),	OV=3,	ENT=WHERE,	FIL=PLOTR
	PLOT(R,R,I),	OV=3,	ENT=PLOT,	FIL=PLOTR
	LLEFT,	OV=3,	ENT=LLEFT,	FIL=PLOTR
	URITE,	OV=3,	ENT=URITE,	FIL=PLOTR
	PLTLU(I),	OV=3,	ENT=PLTLU,	FIL=PLOTR
	AXIS(R,R,RA,R,R,R,R),	OV=3,	ENT=AXIS,	FIL=PLOTR
	NUMB(R,R,R,R,R,I),	OV=3,	ENT=NUMB,	FIL=PLOTR
	SYMB(R,R,R,RA,R,I),	OV=3,	ENT=SYMB,	FIL=PLOTR
	LINES(RA,RA,I,I,I,R),	OV=3,	ENT=LINES,	FIL=PLOTR
Task Scheduling	SCALE(RVA,R,I,I),	OV=3,	ENT=SCALE,	FIL=PLOTR
	TIME(RV),	OV=4,	ENT=TIME,	FIL=SCHEDR
	SETP(I,I),	OV=4,	ENT=SSETP,	FIL=SCHEDR
	START(I,R),	OV=4,	ENT=SSTRT,	FIL=SCHEDR
	DSABL(I),	OV=4,	ENT=DSABL,	FIL=SCHEDR
	ENABL(I),	OV=4,	ENT=ENABL,	FIL=SCHEDR
	TRNON(I,R),	OV=4,	ENT=TRNON,	FIL=SCHEDR
	TTYS(I,I),	OV=4,	ENT=TTYS,	FIL=SCHEDR

Figure 14-2. RTETG Commands for Library Subroutines

14-7. RUNNING THE TRANSFER FILE

The transfer file is produced as a type 3 source file. After you have run RTETG, you must use the File Manager TR command to execute the transfer file and load the overlays. There must be one ID segment available for each overlay. To run the transfer file:

```
*RUN,FMGR
:TR,trf
:EXIT
$END FMGR
```

Schedule FMGR.
Type the TRANSFER command and the name of the transfer file. After FMGR prompts for another command, give the EXIT command.

When you have loaded the BASIC Interpreter as described in Appendix D, BASIC will be ready for execution.

The transfer file normally produces overlays which execute in the foreground of an RTE-II system. You might need to have an overlay execute in the background, for example when there is not enough foreground memory to load the overlay (the 7210 plotter routines require 2K words of memory and could easily bring about this condition).

To force the overlays to execute in the background, edit the transfer file produced by RTETG, and modify the necessary overlay LOADR request from:

```
:RU,LOADR,99,6,7,0000
```

to:

```
:RU,LOADR,99,6,9,0000
```

Then run the transfer file. The overlay will be executed in the background with BASIC (BASIC will swap to the disc while the overlay is executing). Note that the execution time of a program using background overlays is much slower than one using foreground overlays.

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 *.

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. This can occur if a specification in the Branch and Mnemonic Tables is erroneous or if there are programming errors in the subroutine.

If there is an error in the tables 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,%BA00
```

Purge all overlay relocatable file names. % BAnn, where A is the ID letter, nn is the overlay number.

```
:PU,%BAnn
```

Table 14-1. RTETG Error Messages

MESSAGE	MEANING
TOO MANY PARAMETERS	You have supplied too many parameters with the RTETG command.
NAME TOO LARGE	The file name exceeds 6 characters or the subroutine name exceeds 6 characters.
NOT ENOUGH PARAMETERS	You have not supplied all of the required parameters with the RTETG command.
ILLEGAL PARAMETER SPECIFICATION	There is an error in one of the parameter specifications supplied within parenthesis following the subroutine name.
ENTRY POINT NAME TOO LARGE	The entry point name exceeds 5 characters.
ILLEGAL FORMAT	Syntax error in RTETG command. Check for missing commas, bad characters, misspelled keywords.
TABLE OVERFLOW	You have exceeded the limit of 300 subroutine specifications.
BAD ENTRY POINT NAME	A name contains illegal characters. See the RTE Assembler Reference Manual for naming rules.
BAD FILE NAME	A name contains illegal characters. See paragraph 7-1 for file naming rules.
DISK DOWN	Contact the person responsible for system maintenance.
DISK OR DIRECTORY FULL	Try packing the disc with the File Manager PK command and then run RTETG again. If that does not work, use a different cartridge or purge some files.
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.
CARTRIDGE LOCKED	Make sure the disc is not being packed before running RTETG.
BAD SECURITY CODE	RTETG cannot access the command file. Create it without a security code.
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.
COMMAND FILE NOT FOUND	Check to make sure the command file name is spelled correctly.
CANNOT READ COMMAND FILE	If the command file is type 0, make sure that it is set up for reading and that the logical unit number is correct.

Subroutine Table Generation

Next you must remove the overlay programs by using the Loader.

```
:RU,LOADR , , , 4  
PNAME?%BA00
```

Remove all absolute overlay programs

```
:RU,LOADR , , , 4  
PNAME?%BAnn
```

After purging all files and programs simply correct the RTETG input file and run the Table Generator again.

If the problem is in the subroutine itself, the operation is considerably simpler. First, correct the programming error in the subroutine and purge the old overlay absolute programs from the RTE system with the Loader as shown above. Then run the transfer file to reload all the overlays.

```
:TR,transfer filename
```

Now you can again run BASIC and use your new subroutines.

Table 14-1. RTETG Error Messages (Continued)

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.
CARTRIDGE LOCKED	Make sure the disc is not being packed before running RTETG.
BAD SECURITY CODE	RTETG cannot access the command file. Create it without a security code.
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.
COMMAND FILE NOT FOUND	Check to make sure the command file name is spelled correctly.
CANNOT READ COMMAND FILE	If the command file is type 0, make sure that it is set up for reading and that the logical unit number is correct.

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. This can occur if a specification in the Branch and Mnemonic Tables is erroneous or if there are programming errors in the subroutine.

If there is an error in the tables 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,%BA00
.
.
:PU,%BA $nn$ 
```

Purge all overlay relocatable file names. %BA nn , where A is the ID letter, nn is the overlay number.

Next you must remove the overlay programs by using the Loader.

```
:RU,LOADR , , , 4
PNAME?%BA00
.
.
:RU,LOADR , , , 4
PNAME?%BA $nn$ 
```

Remove all absolute overlay programs

After purging all files and programs simply correct the RTETG input file and run the Table Generator again.

If the problem is in the subroutine itself, the operation is considerably simpler. First, correct the programming error in the subroutine and purge the old overlay absolute programs from the RTE system with the Loader as shown above. Then run the transfer file to reload all the overlays.

```
:TR,transfer filename
```

Now you can again run BASIC and use your new subroutines.

HP 2313/91000 DATA ACQUISITION SUBSYSTEM

SECTION

XV

This section contains all the Real-Time BASIC calls used to communicate with the HP 2313/91000 Data Acquisition Subsystem. Information about generating a system containing the instrument (subsystem) subroutines is provided at the end of the section.

15-1. MEASUREMENT OF ANALOG INPUT

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

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

15-2. ANALOG OUTPUT

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

15-3. HP 2313 SUBSYSTEM SUBROUTINES

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

CALL name(parameters as required)

Each call is listed in alphabetical order by name.

15-4. AIRDV (Random Scan)

The AIRDV subroutine reads analog input in a random manner.

Format

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

Parameters

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

Example

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

```
10 DIM V(75),A(75)           Reads 75 voltages into the V array.
20 LET N=75
30 CALL AIRDV(N,A(1),V(1),E)
```

15-5. AISQV (Sequential Scan)

The AISQV routine reads analog input sequentially.

Format

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

Parameters

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

Example

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

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

15-6. AOV (Digital to Analog Conversion)

The AOV routine converts digital information to analog voltage output.

Format

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

Parameters

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

Example

```
300 CALL AOV(25,C(1),V(1),E)  Convert data from channels in the C array to
                                analog voltages and store in V array.
```

15-7. NORM

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

Format

CALL NORM [(unit)]

Parameter

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

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

Example

```
200 LET Z = 2
210 CALL NORM(Z)
```

*Set subsystem unit equal to 2.
Normalize the HP 2313 Subsystem.*

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

15-8. PACER

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

Format

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

Parameters

rate basic pacer rate in microseconds. Basic rate must be between 0 and 255 inclusive.

mult rate multiplier. This parameter specifies one of eight ranges (0 - 7) which is a power of ten multiplier applied to the basic pace rate.

For example:

rate = 25 (25 microseconds)

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

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

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

unit HP 2313 unit number. This number is defined as part of the instrument table. See Paragraph 16-10. If *unit* is to be supplied, the PACER sub-routine must be modified in the Branch and Mnemonic Tables.

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

Example

```

100 REM NORMALIZE SYSTEM          Perform a paced reading on a single channel
110 CALL NORM                     with a period of 1 millisecond.
200 REM RATE = 1*10*3 MICROSECONDS
210 CALL PACER(1,3,0)
300 REM TAKE 20 READINGS FROM CHANNEL 5
310 CALL AISQV(20,-5,V(1),E)
400 REM TURN OFF PACER
410 CALL PACER(0,0,0)

```

15-9. RGAIN

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

Format

CALL RGAIN(*chan,gain*)

Parameters

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

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

Example

```

400 CALL RGAIN(55,X)             Determine present gain on channel 55 and store
                                   it in X.

```

15-10. SGAIN

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

Format

CALL SGAIN(*chan,gain*)

Parameters

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

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

Example

500 CALL SGAIN(CH,12.5) *The channel CH is set to 12.5.*

15-11. SUBSYSTEM ERRORS

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

ERROR *name-numb* IN LINE *xx*

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

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

HLMPX/LLMPX

ADC-1 Driver timeout

ADC-2 Parameter value outside defined range

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

DAC

AOV-1 Driver timeout

AOV-2 Addressed channel undefined.

15-12. TABLE PREPARATION

In order to use the HP 2313/91000 Subsystem subroutines you must provide information about the subsystem which is incorporated in the Instrument Table. The method for doing this is described in Section XVIII.

You must also add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in Section XIV. A list of the RTETG commands required to add these subroutines is given there.

15-13. SUBSYSTEM CONCEPT

Figure 15-1 illustrates the concept of the HP 2313 Subsystem configuration. A subsystem is defined as the equipment occupying a computer I/O slot. Note that in the figure, the HP 2313 is one subsystem (including all three boxes) having one I/O slot number, and the HP 91000 DAS Card is another subsystem occupying an I/O slot number. The Real-Time BASIC software treats the HP 91000 exactly like an HP 2313 Subsystem. The only difference is that there are no low-level multiplexers or Digital-to-Analog cards in the HP 91000.

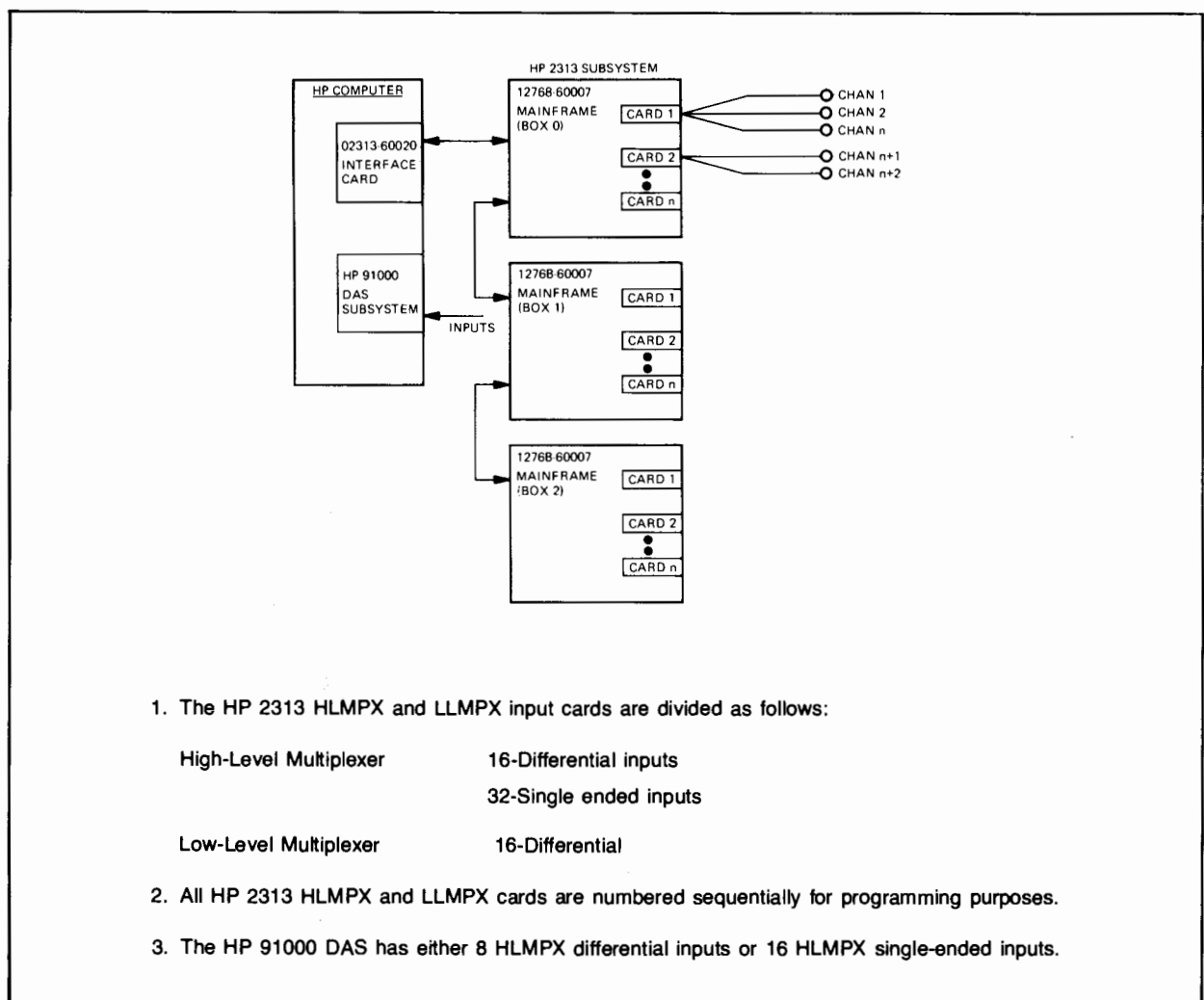


Figure 15-1. HP 2313 Subsystem Configuration

15-14. CARD CONFIGURATION

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

Box 0 Configuration

slot 0 - System Pacer (optional)
slot 1 - ADC
slot 2 - Sample and Hold Amplifier
slot 3 - } Working Cards
slot 11 - }

The working cards must be in the following order:

- a. All high-level multiplexers with single-ended inputs.
- b. All high-level multiplexers with differential inputs.
- c. All low-level multiplexers.
- d. All dual DACs.

If the subsystem contains a Last Address Detector (LAD) card, it must follow the last multiplexer or DAC card.

15-15. CHANNEL NUMBERING

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

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

15-16. SETTING GAIN

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

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

HP 6940 MULTIPROGRAMMER SUBSYSTEM

SECTION

XVI

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

16-1. HP 6940 SUBSYSTEM SUBROUTINES

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

CALL *name(parameters as required)*

Each call is listed in alphabetical order by name.

16-2. DAC

The DAC subroutine converts digital information to analog.

Format

CALL DAC(*chan,value*)

Parameters

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

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

Example

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

Outputs 10 volts on channel 1.

16-3. MPNRM

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

Format

CALL MPNRM

16-4. RDBIT

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

Format

CALL RDBIT(*chan,nbit,bit*)

Parameters

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

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

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

Examples

```
100 CALL RDBIT(-1,3,0)
```

Performs digital input with wait from channel 1 bit 3.

```
200 CALL RDBIT(1,2,B)
```

Reads a bit from channel 1, bit position 2 into location B without wait.

16-5. RDWRD (Read Channel)

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

Format

CALL RDWRD(*chan*,*word*)

**Parameters**

chan number of the channel being read.

word location into which the results of the information from the channel will be placed.

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

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

Examples

100 CALL RDWRD(1,W)

Reads a complete word of information from channel 1 and places the information in location W.

200 CALL RDWRD(-1,W)

Digital input is performed with the wait flag set.

16-6. SENSE

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

Format

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

Parameters

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

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

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

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

The SENSE command is designed for the HP 69434 Event Sense Card only. If the SENSE command is addressed to any other card, the following message is given:

```
ERROR A6940-2 IN LINE nnnn
```

where *nnnn* is the line number.

The HP 69434A card is operated in the "not equal" mode (jumper W3 in position D on the card). In this mode, an event is detected when the external data is "not equal" to the reference bit (parameter *bit* in the SENSE call).

Example

Time the duration of the "on" (1) condition of bit 4 on channel 2.

300 TRAP 5 GOSUB 500	<i>Define TRAP 5.</i>
310 TRAP 6 GOSUB 600	<i>Define TRAP 6.</i>
320 SENSE(2, 4, 1, 5)	<i>Sense when bit 4 of channel 2 changes from 0 to 1.</i>
.	
.	<i>Execute TRAP 5.</i>
500 CALL TIME(X)	<i>Record time when bit changes. Then sense when</i>
510 SENSE(2, 4, 0, 6)	<i>turns "off" (1 to 0) and execute TRAP 6.</i>
520 RETURN	
600 CALL TIME(Y)	<i>Record time when turns off.</i>
610 LET Z = Y - X	<i>Compute interval between on and off.</i>
620 PRINT "TIME INTERVAL = ";Z	<i>Print interval.</i>
630 SENSE(2, 4, 1, 5)	<i>Sense "on" again.</i>
640 RETURN	

16-7. WRBIT

The WRBIT routine writes a bit onto a channel.

Format

CALL WRBIT(chan,nbit,bit)

Parameters

<i>chan</i>	number of the channel.
<i>nbit</i>	bit position starting from the right with bit 0.
<i>bit</i>	state of <i>nbit</i> , either a 0 or 1.

WRBIT writes a single bit on the specified channel of the HP 6940. The bit can turn a light on or off, close or open a relay, or perform any other discrete function.

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

Examples

100 CALL WRBIT(1,3,0)	<i>Write a 0 bit onto channel 1 position 3 without wait.</i>
200 CALL WRBIT(-1,3,0)	<i>Perform digital output with wait.</i>

16-8. WRWRD (Write Channel)

The WRWRD routine writes the contents of a word onto a channel.

Format

```
CALL WRWRD(chan,word)
```

Parameters

chan number of the channel.

word location from which the information is written.

WRWRD writes on the channel specified, the contents of the word stated by the parameter *word*. This provides a 12-bit parallel digital output which corresponds to the 12 least significant bits (right-most bits) of *word*. Depending upon the associated hardware, lights will be operated, relays closed, etc.

Digital output can be performed with or without wait for the output card flag. The flag is set by making the channel number negative in the WRWRD call.

Examples

```
100 CALL WRWRD(1,W)
```

Writes the lower 12 bits of information from the location W onto channel 1 without wait.

```
200 CALL WRWRD(-1,W)
```

Performs the same output with wait.

16-9. SUBSYSTEM ERRORS

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

ERROR *name-numb* IN LINE *xx*

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

The following is a list of HP 6940 errors.

INPUT/OUTPUT CALLS

A6940-1 Driver timeout.

A6940-2 Parameter value outside defined range.

DAC CALL

DAC-1 Driver timeout.

DAC-2 Parameter value outside defined range.

16-10. TABLE PREPARATION

In order to use the HP 6940 subroutines you must provide information about the subsystem which is incorporated in the Instrument Table. The method for doing this is described in Section XVIII.

You must also add the names of the subroutines you want to use to the Branch and Mnemonic Tables. Generation of these tables is described in Section XIV. A list of the RTETG commands required to add the HP 6940 subroutines is given there.

16-11. CARD CONFIGURATION

The configuration of the various types of cards in an HP 6940 Subsystem must follow the conventions given below. If expansion is anticipated, configuration can include blank slots to accommodate future cards. The blank slots then have channel numbers assigned to them, but cannot be accessed until the hardware is added.

Figure 16-1 illustrates the concept of the HP 6940 Subsystem configuration. The cards must be installed in the following order:

- a. All HP 69434 Event Sense cards operating in the "not equal" mode. A maximum of 15 cards is allowed for each 6940. Event Sense cards are not allowed in the HP 6941.
- b. All digital output or input/output cards.
 - HP 69435 - Pulse counter
 - HP 69330 - Relay output
 - HP 69331 - TTL output
 - HP 69333 - Relay output with readback
 - HP 69335 - Stepping motor
 - HP 69600 - Timer
- c. All digital input only cards.
 - HP 69430 - Isolated digital input
 - HP 69431 - Digital input
- d. All analog output cards.
 - HP 69321 - Voltage DAC
 - HP 69370 - Current DAC

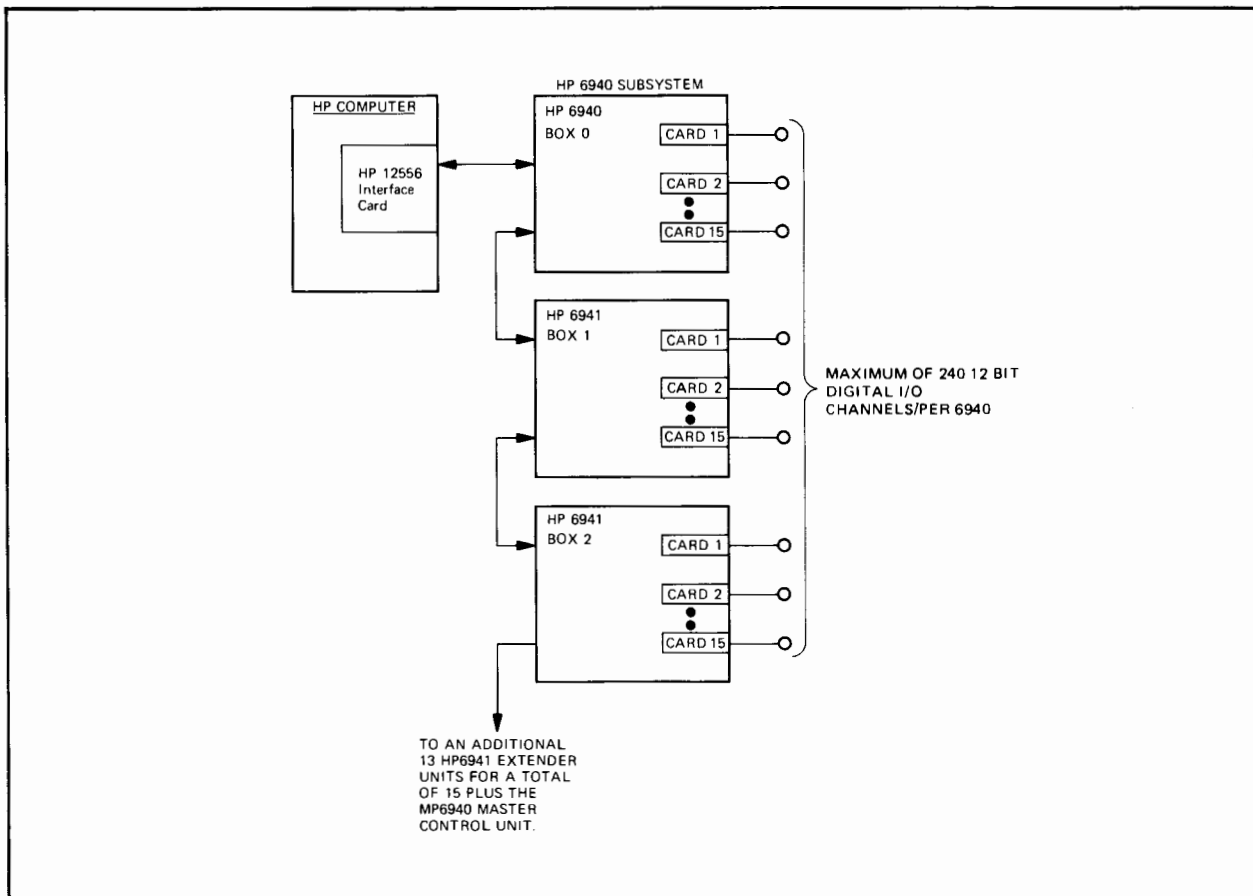


Figure 16-1. HP 6940 Subsystem Configuration

16-12. EXPANSION

The system may be expanded by adding additional HP 6940 Subsystems or by adding HP 6941 Extenders to existing subsystems. The order of cards described in paragraph 16-11 applies to each HP 6940 added. The card order also applies to the slots beginning with slot 401 in the 6940 through slot 414 in a HP 6941, if one is added. Remember that Event Sense cards are not allowed in the HP 6941.

16-13. CHANNEL NUMBERING

The channels are numbered sequentially starting after the last HP 2313 channel number. If there is no HP 2313, the first HP 6940 channel is 1.

Figures 16-2 and 16-3 illustrate two channel numbering schemes and the effect of expanding the system in one of the two ways described above. Assume that the system shown in the examples has 96 HP 2313 channels.

	CHANNEL NUMBERS	6940/6941 SLOT NUMBERS	CARD TYPE
1st 6940	97	400	Event Sense
	98	401	Event Sense
	99	402	Digital I/O
	.	.	.
	108	411	Digital I/O
	109	412	Voltage DAC
	110	413	Current DAC
	111	414	Current DAC
2nd 6940	112	400	Event Sense
	113	401	Digital I/O
	.	.	.
	.	.	.
	126	414	Digital I/O

Figure 16-2. Channel Numbers for Additional 6940

	CHANNEL NUMBERS	6940/6941 SLOT NUMBERS	CARD TYPE
1st 6940	97	400	Event Sense
	98	401	Event Sense
	99	402	Event Sense
	100	403	Blank cards for
	101	404	future Event Sense
	102	405	expansion
	103	406	Digital I/O
	.	.	.
	.	.	.
	111	413	Digital I/O
	112	414	Blank (future Digital I/O)
6941	113	400	Voltage DAC
	114	401	Current DAC

Figure 16-3. Channel Numbers for Addition of a 6941 Extender

The plotter draws precise lines on a paper surface that is a maximum size of 15 by 10 inches. The lines are drawn by a pen on the plotter which moves from a desired point to some other desired point in either a pen-up (no line) or pen-down position. To facilitate the drawing of these lines, several routines have been written for your use with BASIC programs. These routines interface with the standard RTE Driver DVR10. The routines can be separated into two categories:

- those that draw characters (alphabetic characters and special symbols), and
- those that draw lines.

Figure 17-5 at the end of this section contains a program which uses most of these routines, and figure 17-6 contains a program which uses some of the special routines (e.g. scale, axis, lines).

17-1. AXIS

The AXIS routine plots one axis (horizontal or vertical) of a graph with a specified axis label, a specified length, and a specified value at each inch marker.

Format

```
CALL AXIS(x,y,"label",length,angle,min val, inc val)
```

Parameters

<i>x</i>	horizontal coordinate.
<i>y</i>	vertical coordinate.
<i>label</i>	axis label (Enclose in parentheses as a literal string.)
<i>length</i>	length of axis in inches.
<i>angle</i>	angle of axis in degrees. 0° is horizontal, angle increases in counter-clockwise direction.
<i>min val</i>	minimum value of axis (may be calculated with SCALE routine).
<i>inc val</i>	incremental value (may be calculated with SCALE routine).

Parameter *length* can be positive to place label counterclockwise to axis (as in y axis) and negative to place clockwise (as in x axis).

SCALE must be called before AXIS if points on the graph are scaled by SCALE. AXIS calls the SYMB routine to plot the labels 0.14 inches high.

Numbers are printed with .07 inch character height. Since X and Y always refer to the origin of the axis, leave at least .5 inch for axis label and numbers.

Example

```
100 CALL AXIS(0,0,"PSI",10,90,1,1)
```

Plots the Y axis with the label "PSI" on the counterclockwise side, 10 inches long at 90 degrees.

17-2. FACT

The FACT (factor) routine sets the ratio between the horizontal and vertical axis.

Format

```
CALL FACT(x,y)
```

Parameters

<i>x</i>	horizontal axis
<i>y</i>	vertical axis

FACT and SFACT are interrelated routines. Once SFACT is used to establish the graph limits (paper size), FACT does not need to be called unless you want to change the scale relationship.

Initially the horizontal and vertical axis in FACT are automatically set to 1 (e.g. the magnitude of 2 equals two inches in both directions). If you wish to have something plotted at half size, and SFACT has been set to 15 X 10, set FACT (.3333,.5) in the program. *x* and *y* are multiplied by 1000 plotter increments per inch when the new scaling factor is established.

The following equations give the actual translation from the X,Y in the plot call to the value actually put out by the plotter.

$$X_{out} = 1000.0 * X_{in} * X_{fact}$$

$$Y_{out} = 1000.0 * Y_{in} * Y_{fact}$$

X_{in} is X in PLOT call. X_{fact} is X in FACT call (set to 1 originally). Y_{in} is Y in PLOT call. Y_{fact} is Y in FACT call (set to 1 originally).

17-3. LINES

The LINES routine plots a line and/or symbols through successive data points in arrays previously scaled by the SCALE routine.

Format

```
CALL LINES(x(1),y(1),numb,scale,cntrl,symb)
```

Parameters

<i>x</i>	array scaled for the abscissa.
<i>y</i>	array scaled for the ordinate.
<i>numb</i>	number of points to be plotted.
<i>scale</i>	integer that specifies the point to be scaled. 1 = every point. 2 = every second point. <i>n</i> = every <i>n</i> th point.
<i>cntrl</i>	control value: 0 = line plot only. 1 = symbol at every point with line. -1 = symbol at every point. 2 = line and symbol at every second point. -2 = symbol at every second point. <i>n</i> = line and symbol at every <i>n</i> th point. - <i>n</i> = symbol at every <i>n</i> th point.
<i>symb</i>	number of centered symbol to be plotted (see SYMB routine).

Since the LINES routine requires the adjusted minimum and delta values produced by the SCALE routine, SCALE must be called before LINE for each graph.

Before calling this routine, the pen should be moved to the point 0,0 on the graph, and the origin should be set at that point with LLEFT.

17-4. LLEFT

The LLEFT routine lifts the pen and moves it to the lower-left corner.

Format

```
CALL LLEFT
```

When LLEFT is called, the internal *x* and *y* references are set to zero. This provides a defined reference for starting a plot. The pen is left in the up position at the completion of this call. This means that a call to PLOT must be made to put the pen down.

17-5. NUMB

The NUMB routine plots a number, with or without the decimal point, at a specified height, location, and angle.

Format

```
CALL NUMB(x,y,height,numb,angle,digits)
```

Parameters

<i>x</i>	horizontal coordinate of lower left corner of number.
<i>y</i>	vertical coordinate.
<i>height</i>	height of number in inches.
<i>numb</i>	number to be plotted.
<i>angle</i>	angle at which the number is to be plotted.
<i>digits</i>	number of digits.
	0 = print the decimal point of an integer.
	-1 = suppress the decimal point.
	<i>n</i> = number of digits to print to the right of the decimal point (maximum of seven).

Example

```
100 CALL NUMB(5.32,8.79,0.1,0.0,-1)
110 CALL NUMB(6.3,8.79,0.1,J,0.0,-1)
120 CALL NUMB(7.16,8.79,0.1,K,0.0,-1)
```

Plot three numbers .1 inches high, with decimal point suppressed, at 8.79 inches above 0,0 and 5.32,6.3, and 7.16 inches to the right of 0,0.

17-6. PLOT

The PLOT routine moves the pen from an origin to a destination.

Format

```
CALL PLOT(x,y,pram)
```

Parameters

<i>x</i>	horizontal coordinate in inches.
<i>y</i>	vertical coordinate in inches.
<i>pram</i>	constant or variable name set equal to one of the following:
	-2 = move with pen down; consider the point where the pen stops (x,y) as the new origin.
	-3 = move with the pen up; consider the point where pen stops as new origin.
	+2 = move with the pen down; origin unchanged.
	+3 = move with the pen up; origin unchanged.

If the LLEFT routine is called first to establish the lower-left corner, the pen is left up. The PLOT routine must be called as follows to put the pen down:

```
CALL PLOT(0,0,2)
```

The parameters of the routine determine whether the origin is the original reference of the lower-left corner of the paper or the last known position from the previous plot.

Example

Plot a rectangle 8.5" by 11" starting at the origin (assuming the pen starts at the origin and that scaling has been set at 15 by 10 inches by calling LLEFT and SFACT(15,10)).

100 CALL PLOT(0,0,2)	<i>Sets the pen down.</i>
110 CALL PLOT(11.,0.,2)	<i>Moves the pen from X,Y = 0,0 to X,Y = 11,0.</i>
120 CALL PLOT(11.,8.5,2)	<i>Moves the pen from X,Y = 11,0 to X,Y = 11,8.5.</i>
130 CALL PLOT(0.,8.5,2)	<i>Moves the pen from X,Y = 11,8.5 to X,Y = 0,8.5.</i>
140 CALL PLOT(0.,0.,2)	<i>Moves the pen from X,Y = 0,8.5 to the origin.</i>

17-7. PLTLU

The PLTLU routine defines the logical unit number of the plotter for all the plotter calls.

Format

```
CALL PLTLU(lu)
```

Parameter

lu logical unit number of the plotter.



The PLTLU routine must be the first routine called to establish the logical unit number.

17-8. SCALE

The SCALE routine scales an array of numbers to fit a specified size graph; the values generated are used by the LINES and AXIS routines.

Format

```
CALL SCALE(array(1),length,num pts,scale)
```

Parameters

<i>array</i>	an array of values.
<i>length</i>	length of axis in inches.
<i>num pts</i>	number of points to be plotted.
<i>scale</i>	integer specifying the points to be scaled:
	1 = every point.
	2 = every second point.
	<i>n</i> = every <i>n</i> th point.

Separate calls are required for X and Y axis.

The adjusted minimum value is a number less than or equal to the minimum data value. The adjusted delta value is the result of subtracting the minimum data value from the maximum data value, divided by the length of the axis and adjusted to provide one-inch increments that will cover the data. The adjusted scale values are used by the LINES and AXIS routines.

The adjusted values are stored following the array. The minimum value of Y is stored in $Y(np*k+1)$ where np is the number of points to be plotted; the delta value is stored in $Y(np*k+2)$. Therefore, the array must be dimensioned $(k+2)$ locations larger than $(np*k)$, which is the number of locations necessary for data points. Normally, k equals 1, so an array Z of ten data points would be dimensioned as Z(12).

Example

```
10 DIM X(52),Y(52)           Dimension X and Y arrays.
.
.
100 CALL SCALE(X(1),6.5,50,1) Scale every point in a 50 point array, fitting X
110 CALL SCALE(Y(1),10.0,50,1) values on a 6.5 inch X axis and Y values on a 10
                                inch Y axis.
```

17-9. SFACT

The SFACT routine sets or adjusts the plotter for the particular size paper being used.

Format

CALL SFACT(*width,height*)

Parameters

width width scale for horizontal movements.

height height scale for vertical movements.

SFACT and FACT are interrelated routines. Once SFACT is used to establish the graph limits (paper size) FACT does not need to be called unless you wish to change the scale size. Initially the width and height parameters in SFACT are automatically set to 10 inches by 10 inches.

SFACT should be used as follows:

- a. Place the paper on the plotter bed.
- b. Manually adjust the pen and carriage travel to the lower left and then upper right corners of the paper (image area if desired).
- c. Call SFACT with the parameters set to the paper size that was set up in step b.

After the plotter is adjusted and SFACT has been called with the proper dimensions, the horizontal and vertical dimensions in other calls are set for a one-to-one relationship.

Example

```

10 CALL PLTLU(13)
20 CALL LLEFT
25 CALL SFACT(15,10)
30 CALL PLOT(0,0,2)
40 CALL PLOT(5,5,2)

```

The pen moves five inches horizontally and five inches vertically for a paper size of 15 by 10 inches.

17-10. SYMB

The SYMB routine is used when characters are to be plotted.

Format

```

CALL SYMB(x,y,size,"char",integer,theta,pram)

```

Parameters

<i>x</i>	horizontal starting coordinate.
<i>y</i>	vertical starting coordinate.
<i>size</i>	character height in inches.
<i>"char"</i>	actual characters to be plotted enclosed in quotes (string literal).
<i>string variable</i>	string variable containing characters to be printed.
<i>integer</i>	any single integer 0 through 25 (without quotes) representing a special character.
<i>theta</i>	number of degrees counterclockwise that the line of plotting is to be rotated.
<i>pram</i>	constant or variable name set equal to one of the following: <ul style="list-style-type: none"> 1 = plot the characters between the quotes or in the string variable. -1 = draw the special character and leave the pen up. -2 = draw the special character and leave the pen down.

The SYMB routine is used to write characters on the paper if *pram* = +1. If *pram* = -1 or -2, the special character represented by the ASCII number is drawn. If the pen is left down, moving the pen to the place where the next character is to be drawn causes the special characters to be connected like points along a graph. If the pen is left up, the points are drawn without intervening connecting lines. When drawing graphs, it is recommended that symbols that are centered be used. The coordinates specified by you refer to the lower left corner of an imaginary box enclosing the character to be drawn. If you wish to continue to call PLOT and have the set of ASCII characters be contiguous, place the value 999 as the *x* and *y* coordinates. Be aware that the *x* and *y* coordinates from one subroutine are not transmitted to the other subroutines. You can use WHERE to supply missing information when you transfer from the PLOT to the SYMB routine, etc.

The following are the ASCII characters which may be enclosed within quotes:

A through Z and 0 through 9.

@ [] \ ↑ ← ! " # \$ % & ' () * , - . / 0 : ; < = > ? "space"

These symbols are centered and have their ASCII reference number immediately above the symbol:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
☐	⊙	△	+	×	◇	⊕	⊗	⌞	⌟	⌘	⌙	⌚	⌛	⌜	⌝	⌞	⌟	⌠	⌡	⌢	⌣	⌤	⌥	⌦	⌧	⌨

17-11. URITE

The URITE routine lifts the pen and moves it to the upper right corner of the paper to facilitate unloading the paper at the end of a plot. This routine does not affect any of the coordinates.

Format

CALL URITE

17-12. WHERE

The WHERE routine indicates the current position of the plotter pen.

Format

CALL WHERE(x,y)

Parameters

x variable in which the horizontal coordinate is returned.

y variable in which the vertical coordinate is returned.

17-13. TABLE PREPARATION

In order to use the plotter subroutines, you must add the name of the routines to the Branch and Mnemonic Tables. The procedure for doing this is described in Section XIV. The required RTETG commands are provided there.

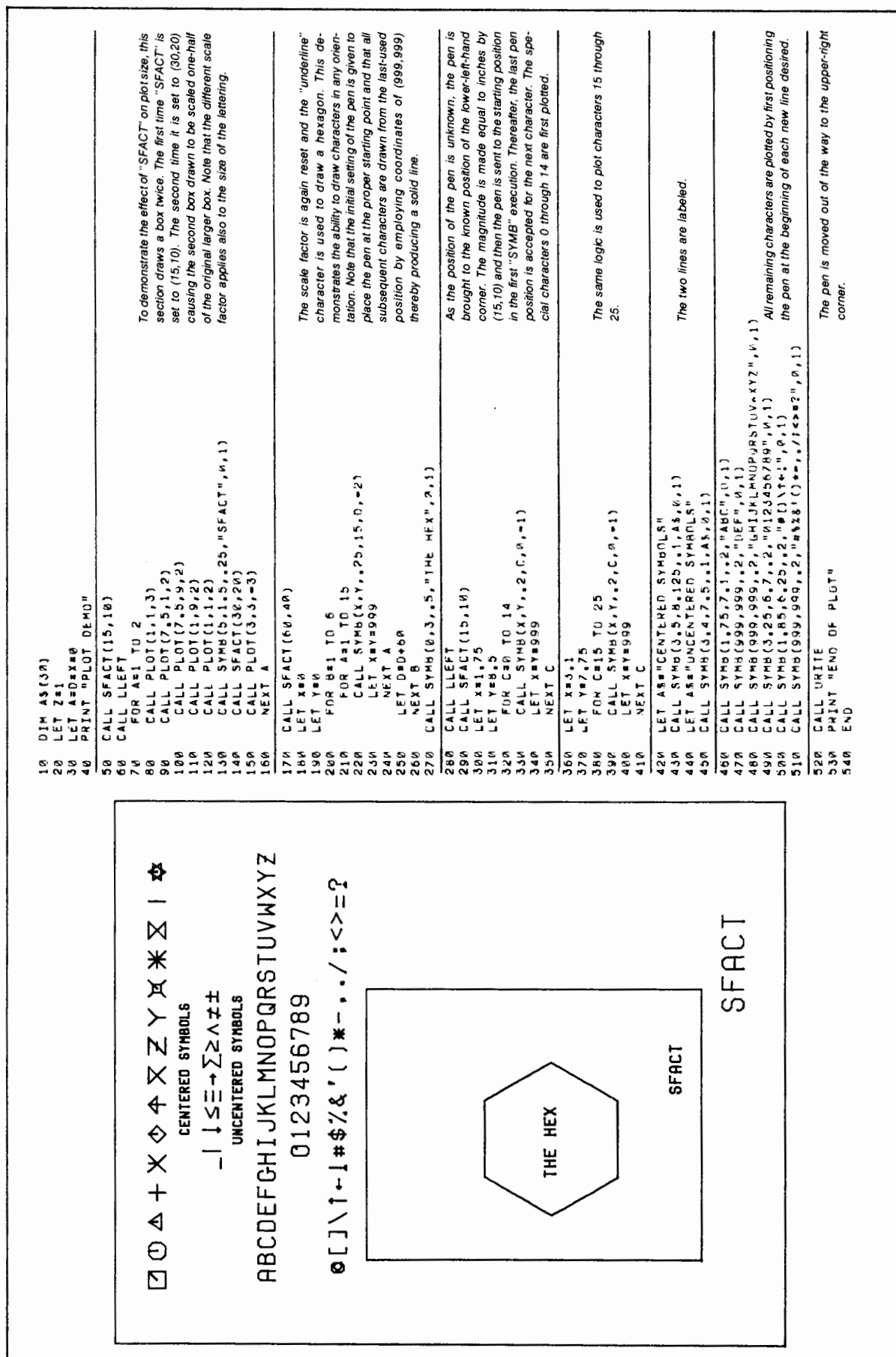


Figure 17-1. Plotter Control Sample Program #1

```

10 REM
20 REM 7210 PLOTTER EXAMPLE
30 REM
40 LET N=36
50 CALL PLTLU(18)
60 CALL LLEFT
70 CALL SFACT(15,10)
80 CALL PLOT(.5,0,-3)
90 REM
100 REM PLOT AXES FOR X AND Y
110 REM
120 CALL AXIS(0,5,"RADIANS",-14,0,0,1)
130 CALL AXIS(0,1,"VOLTAGE",8,90,-.4,.1)
140 LET F=0
150 LET F=F+1
160 REM
170 REM PLOT CURVES
180 REM
190 DIM X(142),Y(142)
200 IF F=2 CALL PLOT(0,1,-3)
210 FOR I=0 TO 14 STEP .1
220 LET X(I*10+1)=I
230 IF F=1 LET Y(I*10+1)=SIN(I)*4
240 IF F=2 LET Y(I*10+1)=COS(I)*4
250 NEXT I
260 REM
270 REM SCALE ARRAY
280 REM
290 CALL SCALE(X(1),14,140,1)
300 CALL SCALE(Y(1),10,140,1)
310 REM
320 REM PLOT THE ARRAY
330 REM
340 CALL LINES(X(1),Y(1),140,1,1,F)
350 IF F<2 GOTO 150
400 REM
410 REM PRINT PLOT NUMBER
420 REM
430 CALL SYMB(12.2,8.5,.15,"PLOT #",0,1)
440 CALL NUMB(999,999,.15,N,0,-1)
500 REM
510 REM PRINT LEGEND
520 REM
530 CALL SYMB(12.2,1.5,.2,"LEGEND",0,1)
540 CALL PLOT(12.2,1.4,3)
550 CALL PLOT(13.3,1.4,2)
560 CALL SYMB(12.2,1.1,.15,1,0,-1)
570 CALL SYMB(999,999,.15," SINE WAVE",0,1)
580 CALL SYMB(12.2,.6,.15,2,0,-1)
590 CALL SYMB(999,999,.15," COSINE WAVE",0,1)
600 CALL URITE
610 STOP
9999 END

```

Figure 17-2. Plotter Control Sample Program #2

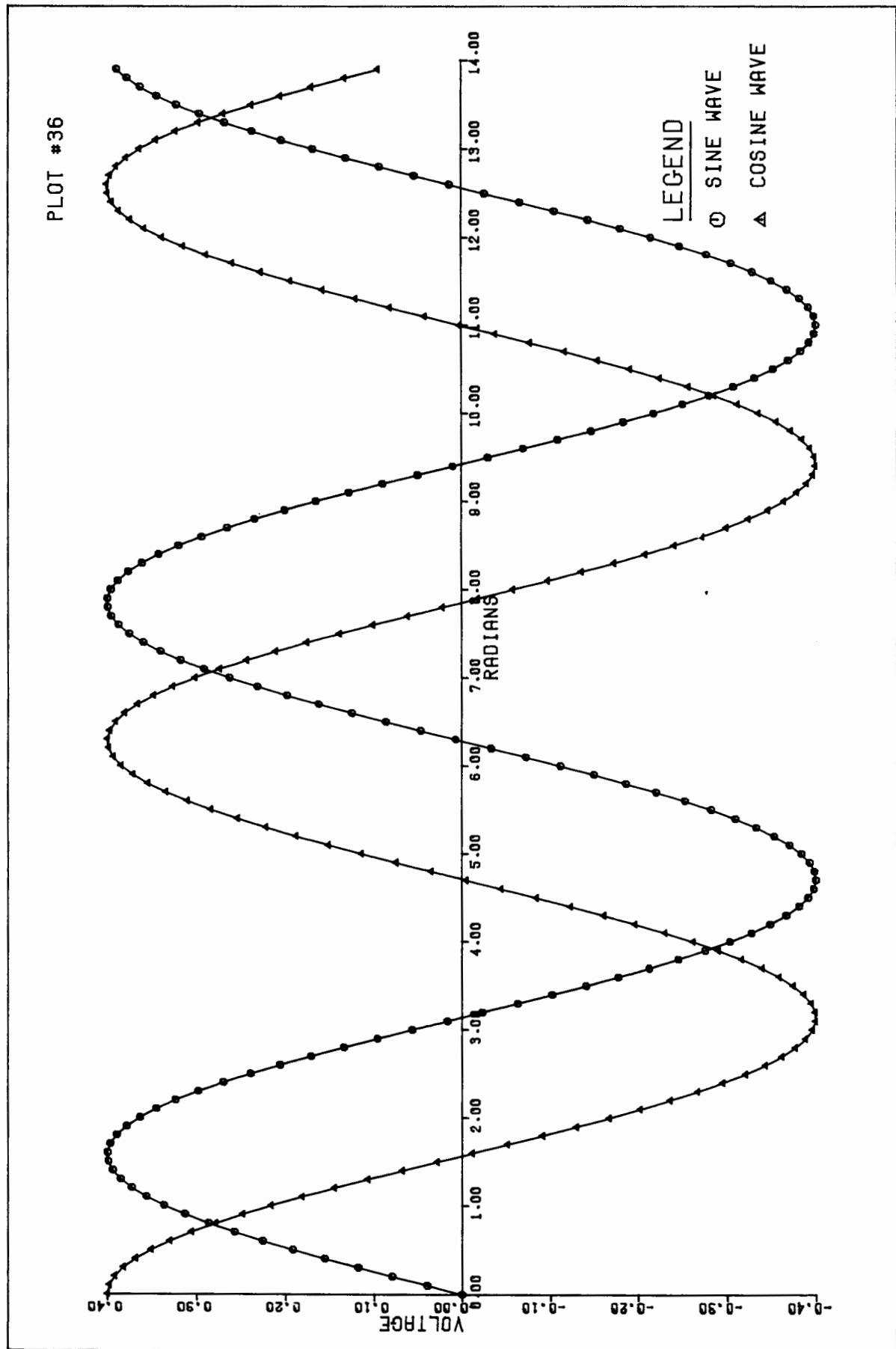


Figure 17-3. Plotter Control Sample Program #2 (Plot)

INSTRUMENT TABLE GENERATION

SECTION

XVIII

In order for the HP 2313 and HP 6940 to be configured into the RTE system, it is necessary to generate an Instrument Table tape which is loaded during RTE system generation. This tape contains all of the information concerning the 2313 and 6940 subsystems which is required by the RTE Operating System. You generate the Instrument Table tape with the Instrument Table Generator program. (See Preface for part number.)

Before generating the Instrument Table tape, be sure that the HP 2313 and HP 6940 cards are in the proper order as described in Sections XV and XVI.

18-1. OPERATING INSTRUCTIONS

The Table Generator is an absolute program that uses the HP System Input/Output (SIO) drivers and is loaded into memory with the Basic Binary Loader (BBL). To execute the program do the following:

- a. Apply power to all equipment. Set the system teleprinter to LINE.
- b. Load the Table Generator tape and configured SIO Drivers into memory using the BBL.
- c. Go to the starting address of the program, location 2, and set the switch register as follows:
 - If the teleprinter punch is to be used, set bit 15 to 1. The following information halts will occur:
HLT07 (turn on punch)
HLT70 (turn punch off).
 - If the teleprinter punch is not used, set bit 15 to 0 (clear the switch register).
- d. Push RUN. The Table Generator begins with the HP 2313 Configuration Phase. See below.

After each question is printed by the system, respond with the required answer followed by carriage return and linefeed. The Table Generator prompts with a hyphen. For clarity, it has been omitted from the following text. Responses are shaded and are only examples; actual responses should be given as appropriate to the particular system you are generating.

If an error is made and discovered before RETURN is pressed, type the RUBOUT key then RETURN. Otherwise restart at step c above.

18-2. HP 2313/91000 CONFIGURATION PHASE

This part of the generation produces information about the HP 2313/91000 Subsystems. For specific information on the configuration of those subsystems see Section XV.

After being loaded into memory and execution started, prints:

```
*2313 AND 6940 TABLE GENERATOR*  
*TURN ON PUNCH*
```


If the punch is already on, or after it is turned on, some leader will be punched.

# OF 2313'S? 1	The Table Generator requests the number of HP 2313 Subsystems and 91000 Subsystems. The number is equal to the total of both types of subsystems. If there are no subsystems of either type, type a 0 (zero) and the Table Generator skips to the HP 6940 Configuration Phase. If there is more than one subsystem, subroutines NORM and PACER must have the unit number parameter added in the Branch and Mnemonic Tables. (See Section XIV.)
SUBSYSTEM #01 CONFIGURATION	One is the unit number used in the subsystem calls referencing this subsystem.
LU? 7	The Table Generator requests the logical unit number of the subsystem.
#HL - SE? 32	Specify the number of high-level single-ended input channels. Each HP 2313 card has 32 channels, so multiply the number of HLMPX single-ended cards times 32 for the total number of channels. If the subsystem is a HP 91000, answer either 16 or 0. If 16, the next question must be answered 0 for the HP 91000.
#HL - DIF? 16	Specify the number of high-level differential input channels. Each 2313 card has 16 channels, so multiply the number of HLMPX differential cards times 16 for the total number of channels. If the subsystem is a HP 91000, answer either 8 or 0.
#LL,GAIN? 8,12.5 8,100 0,0	Specify the number of low-level channels in a group and the gain of that group. For the HP 91000 Subsystem, answer 0,0. The allowable gains are: 12.5, 25, 50, 100, 125, 250, 500, 1000. The sample responses specify that there are 8 channels in one group, all with a gain of 12.5, and 8 channels in a second group, all with a gain of 100. A negative gain specifies that each single channel group has the specified gain (e.g., 8,-12.5 indicates that 8 single channel groups each have a 12.5 gain.) The prompt character (hyphen) is repeated until a 0,0 response is given.
#DAC'S? 2	Specify the number of Digital-to-Analog Converters. For the HP 91000, answer 0. Each HP 2313 card has 2 DAC'S. If the system has at least one DAC, the AOV subroutine must be entered in the Branch and Mnemonic Tables.

If more than one 2313 Subsystem is being used, the prompts above (beginning with SUBSYSTEM #*n* CONFIGURATION) will be repeated for each subsystem until you respond with a zero to the first prompt.

18-3. HP 6940 CONFIGURATION PHASE

Refer to Section XVI for HP 6940 configuration information. If the answer to the number of 2313's question was zero or the last HP 2313 subsystem has been described, the Table Generator begins this part of the generation.

OF 6940'S?

1

The Table Generator requests the number of subsystems. If there are none, enter a zero and the Table Generator will skip to the Constant Configuration Phase. Otherwise it prints the next prompt.

OF CHANNELS IN 2313'S?

64

Enter the total number of multiplexer channels in all the HP 2313 Subsystems you configured in the preceding part of the table generation.

SUBSYSTEM #01
CONFIGURATION

I/O SLOT?

12

Enter the computer I/O slot for this HP 6940 Subsystem.

LU?

8

Enter the logical unit number of this subsystem.

#EVENT?

2

Enter the number of event sense cards.

DIG INPUT FOR PRESET
CNTR W/INT?

0

Enter zero. This question does not apply to Multi-User Real-Time BASIC.

DIGITAL I/O?

9

Enter the number of digital input/output cards.

DIGITAL INPUT?

0

Enter the number of digital input-only cards.

VOLTAGE DACS?

2

Enter the number of voltage digital-to-analog converter cards.

CURRENT DACS?

2

Enter the number of current digital-to-analog converter cards.

TIMERS?

0

Enter zero. This question does not apply to Multi-User Real-Time BASIC.

COUNTERS?

0

Enter zero. Does not apply to Real-Time BASIC.

If more than one 6940 Subsystem is being used, the prompts above (beginning with the message SUBSYSTEM #n CONFIGURATION) will be repeated for each subsystem. After the last subsystem has been described, the Table Generator prints:

*ENTER INSTRUMENT
CONFIGURATION
CONSTANTS*

/E

Enter symbols and constants representing any additional instrument subsystems. Terminate the list with "/E".

At this point, the Instrument Table tape will be punched out. Then the Table Generator prints:

TABLE GENERATION COMPLETE

Tear off the tape, label it, and save it for loading during RTE system generation.

18-4. LOADING THE TAPE

You must load the Instrument Table tape into the RTE system during system generation along with the BASIC Subroutine Libraries that contain the BASIC software. This method of loading allows the Loader to automatically search and load the table.

18-5. ERROR MESSAGES

Table 18-1 contains a summary of the error messages you may encounter while running the Table Generator.

Table 18-1. Error Messages

MESSAGE	MEANING	CORRECTIVE ACTION
ILLEGAL CONTROL DIRECTIVE	Incorrect format.	Type again in correct format.
ILLEGAL RESPONSE	Incorrect input.	Type again with correct response.

SUMMARY OF STATEMENTS, COMMANDS, AND SUBROUTINES

APPENDIX

A

STATEMENT SUMMARY



This summary of Real-Time BASIC statements provides the statement names in alphabetical order with a brief description and a reference to the paragraph or paragraphs containing a complete statement description.

STATEMENT	DESCRIPTION	REFERENCE
ASSIGN	Dynamically assigns a file name to a file number and opens the file.	7-4
CALL	Calls for execution of an external subroutine, optionally passing parameters to the subroutine.	6-3
CHAIN	Terminates the current program and calls for execution of the Real-Time BASIC program named in the CHAIN statement. Variables are shared between programs if named in COM statement.	6-2
COM	Declares a common block to contain specified variables used in common by more than one program. Effective when one program calls another with CHAIN.	3-13
DATA	Provides data to be read by READ statements.	3-10 4-12
DEF	Defines a function.	5-2
DIM	Reserves storage for arrays and sets upper bounds on the number of elements.	3-12
	DIM also reserves storage for strings and sets their maximum character length.	4-5
END	Terminates execution of the current program. Last statement in the program must be END.	3-4
FILES	Allocates file numbers to file names or reserves file numbers for later assignment with ASSIGN. FILES is declarative and unlike ASSIGN, is not executed.	7-3
FOR. . .NEXT	Allows repetition of a group of statements between FOR and NEXT. The number of repetitions is determined by the initial and final values of a FOR variable, and by an optional step specification.	3-5
GOTO	Transfers control to a specified statement label.	3-3

STATEMENT	DESCRIPTION	REFERENCE
GOTO. . .OF	Multibranch GOTO transfers control to one of a list of statement labels depending on the value of an integer expression.	3-3
GOSUB	Causes execution of a subroutine beginning at a specified statement label. Following a RETURN statement in the subroutine, control returns to the statement following GOSUB.	6-1
GOSUB. . .OF	Multibranch GOSUB executes one of a list of subroutines depending on the value of an integer expression.	6-1
IF END #. . .THEN	Specifies action to be taken when an end-of-file condition occurs.	7-5
IF. . .THEN	Evaluates a conditional expression and specifies action to be taken if condition is true. The condition is a numeric or string expression. The action may transfer to a statement label or may be a single executable statement.	3-6 4-10
INPUT	Requests user input to one or more variables by printing a ? and accepts string or numeric data from the terminal.	3-11 4-7
LET	Introduces assignment statement that assigns one or more values to a variable or array element. The word LET may be omitted.	3-1 4-6
NEXT	Terminates a loop introduced by a FOR statement. Specifies a variable that must match the FOR variable.	3-5
PAUSE	Stops program execution without terminating the program.	3-14
PRINT	Prints the contents of a list of numeric or string expressions on the list device.	3-7 4-8
PRINT #	Outputs the contents of a list of numeric or string expressions to the specified file.	4-13 7-9, 10
PRINT #lu	Outputs contents of a list of numeric or string variables or expressions to the specified lu (logical unit number). May be used without FILES statement.	4-13 7-9, 10
READ	Assigns constants and string literals from one or more DATA statements to the variables specified in READ. Treats contents of all DATA statements as a single data list.	3-10 4-9
READ #	Reads one or more items from a file into specified variables.	4-14 7-6, 7, 8
READ #lu	Reads one or more items from a lu (logical unit) into specified variables.	4-14 7-7, 8

STATEMENT	DESCRIPTION	REFERENCE
REM	Introduces remarks and comments in the program listing.	3-2
RESTORE	Resets the data pointer to the beginning of the program or to the first DATA statement following a specified label.	3-10
RETURN	Returns control from a GOSUB subroutine to the statement following the last GOSUB.	6-1
STOP	Terminates execution of the run.	3-4
TRAP	Associates a trap number with a task.	11-11
WAIT	Causes an executing program to stop for a specified number of milliseconds.	3-15

COMMAND SUMMARY

Each command is listed by name in alphabetical order followed by a brief description and a reference to the paragraph or paragraphs containing a complete description of the command.

COMMAND	DESCRIPTION	REFERENCE
ABORT	Legal only in break period; terminates the suspended program and returns control to the BASIC Interpreter where all commands are legal.	10-4
BACKF	Backspaces past previous file mark.	13-1
*BR,BASIC	Breaks execution of a program.	9-14
BREAK	Specifies breakpoints where execution of program will be interrupted to enter debugging commands.	10-2
BYE	Terminates execution of BASIC Interpreter.	9-12
CALLS	Lists all of the mnemonics in the current Mnemonics Table.	10-8
CREATE	Creates a Real-Time BASIC formatted file with a specified length; optionally on a specified cartridge with a security code.	9-6
CSAVE	Stores a program in semi-compiled format as a disc file.	9-2
DELETE or DEL	Deletes one or a range of more than one statement from current program.	9-5
LIST	Lists the contents of the current program at the terminal or on a specified type 0 file.	9-13
LOAD	Loads all or a portion of a source or semi-compiled program.	9-1

Summary of Statements, Commands, and Subroutines

STATEMENT	DESCRIPTION	REFERENCE
LOCK	Locks a peripheral device to your program.	9-11
MERGE	Merges a source program with a program in memory.	9-3
PURGE	Deletes the specified data or program file from the system.	9-7
RENAME	Renames a program or data file.	9-8
REPLACE	Replaces a source program on a disc file.	9-4
RESEQ	Renumbers any group of statements in the current program, optionally from a new first line number with a specified increment. By default, renumbering starts at 10 with increments of 10.	9-9
RESUME or RES	Resumes normal BASIC operation following breakpoint or a call simulation.	10-3
REWIND	Rewinds magnetic tape on specified device.	13-1
RUN	Executes the current program or gets and executes a specified program file.	9-10
SAVE	Saves the current program as a program disc file.	9-2
SET	Legal only in break period, sets any program variable to a constant value.	10-7
SHOW	Legal only in break period; lists the values of the specified variables.	10-6
SIM	Simulates a subroutine call.	10-5
SKIPF	Skips to end of current file.	13-1
TABLES	Specifies names of Branch and Mnemonic Tables.	8-2
TRACE	Traces variables and array values, and the execution of statements and segmented programs.	10-1
UNBREAK	Turns off any or all breakpoints specified with the BREAK command.	10-2
UNLOCK	Unlocks a locked device.	9-11
UNSIM	Turns off simulation facility.	10-5
UNTRACE	Turns off tracing facility.	10-1
WEOF	Writes an EOF mark on magnetic tape.	13-1

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
DAC	Converts digital value to analog output. (6940)	16-2
DSABL	Disables a specified task.	11-6
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
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

Summary of Statements, Commands, and Subroutines

SUBROUTINE	DESCRIPTION	REFERENCE
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
SCALE	Scales an array of numbers to fit a specified graph size.	17-8
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
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

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.

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.

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*.

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.

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.

ASCII CHARACTER SET

APPENDIX

C

Graphic	Decimal Value	Comments	Graphic	Decimal Value	Comments
	0	Null	*	42	Asterisk
	1	Start of heading	+	43	Plus
	2	Start of text	,	44	Comma
	3	End of text	-	45	Hyphen (Minus)
	4	End of transmission	.	46	Period (Decimal)
	5	Enquiry	/	47	Slant
	6	Acknowledge	0	48	Zero
	7	Bell	1	49	One
	8	Backspace	2	50	Two
	9	Horizontal tabulation	3	51	Three
	10	Line feed	4	52	Four
	11	Vertical tabulation	5	53	Five
	12	Form feed	6	54	Six
	13	Carriage return	7	55	Seven
	14	Shift out	8	56	Eight
	15	Shift in	9	57	Nine
	16	Data link escape	:	58	Colon
	17	Device control 1	;	59	Semicolon
	18	Device control 2	<	60	Less than
	19	Device control 3	=	61	Equals
	20	Device control 4	>	62	Greater than
	21	Negative acknowledge	?	63	Question mark
	22	Synchronous idle	@	64	Commercial at
	23	End of transmission block	A	65	Uppercase A
	24	Cancel	B	66	Uppercase B
	25	End of medium	C	67	Uppercase C
	26	Substitute	D	68	Uppercase D
	27	Escape	E	69	Uppercase E
	28	File separator	F	70	Uppercase F
	29	Group separator	G	71	Uppercase G
	30	Record separator	H	72	Uppercase H
	31	Unit separator	I	73	Uppercase I
	32	Space	J	74	Uppercase J
!	33	Exclamation point	K	75	Uppercase K
"	34	Quotation mark	L	76	Uppercase L
#	35	Number sign	M	77	Uppercase M
\$	36	Dollar sign	N	78	Uppercase N
%	37	Percent sign	O	79	Uppercase O
&	38	Ampersand	P	80	Uppercase P
'	39	Apostrophe	Q	81	Uppercase Q
(40	Opening parenthesis	R	82	Uppercase R
)	41	Closing parenthesis	S	83	Uppercase S

ASCII Character Set

Graphic	Decimal Value	Comments	Graphic	Decimal Value	Comments
T	84	Uppercase T	j	106	Lowercase j
U	85	Uppercase U	k	107	Lowercase k
V	86	Uppercase V	l	108	Lowercase l
W	87	Uppercase W	m	109	Lowercase m
X	88	Uppercase X	n	110	Lowercase n
Y	89	Uppercase Y	o	111	Lowercase o
Z	90	Uppercase Z	p	112	Lowercase p
[91	Opening bracket	q	113	Lowercase q
\	92	Reverse slant	r	114	Lowercase r
]	93	Closing bracket	s	115	Lowercase s
^	94	Circumflex	t	116	Lowercase t
_	95	Underscore	u	117	Lowercase u
`	96	Grave accent	v	118	Lowercase v
a	97	Lowercase a	w	119	Lowercase w
b	98	Lowercase b	x	120	Lowercase x
c	99	Lowercase c	y	121	Lowercase y
d	100	Lowercase d	z	122	Lowercase z
e	101	Lowercase e	{	123	Opening (left) brace
f	102	Lowercase f		124	Vertical line
g	103	Lowercase g	}	125	Closing (right) brace
h	104	Lowercase h	~	126	Tilde
i	105	Lowercase i		127	Delete

LOADING BASIC SOFTWARE

APPENDIX

D

SYSTEM GENERATION

One module of BASIC, the BASIC Resident Library (#92101-12002) must be loaded at the time the system is generated. If this is not done, undefined references occur when the BASIC Interpreter is loaded. A dummy module may be supplied, however, when BASIC is loaded to satisfy these undefined references. If this is done, the TRAP statement and all time scheduling statements will not function but BASIC will operate properly. Figure D-1 contains the dummy TRAP module.

```
PAGE 0002 001

0001          ASMB,R,L,C
0002 00000          NAM TRAP,7
0003*
0004* DUMMY TRAP MODULE
0005*
0006          ENT TRAP
0007 00000 000000 TRAP NOP
0008 00001 036000R      ISZ TRAP
0009 00002 126000R      JMP TRAP,I
      TRAP 00007 00006 00008 00009 00010
```

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. 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, a special entry in the Interrupt Table is required as follows:

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.

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 Resident Library contains all pertinent routines for handling traps, time scheduled tasks, and searching the trap and task tables. 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.

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 sub-routines, the following generation error will be reported:

```
ERR05      #GET!      DUPLICATE ENTRY POINT
ERR08      #GET!      DUPLICATE PROGRAM NAME
```

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

Loading Basic Software

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.

LOADING THE INTERPRETER

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. Here are the required File Manager commands:

:LG,10	<i>Assign 10 LG tracks. (Load and GO)</i>
:MR, <i>lu</i>	<i>Input BASIC relocatable tape part 1 of 3.</i>
:MR, <i>lu</i>	<i><i>lu</i> is the logical unit number of the input device (paper tape reader).</i>
:MR, <i>lu</i>	<i>Repeat for parts 2 and 3 of the tape.</i>
:RU,LOADR,99,1,28,1,2	<i>Loader command for RTE-II. If you are using RTE-III substitute this command:</i>
	<i>:RU,LOADR,99,1,28,NN001,2 where NN is the number of pages required for BASIC (at least 8).</i>
:RU,BASIC	<i>Start the BASIC Interpreter.</i>

The Branch and Mnemonic Table Generator, RTETG, can be loaded by using the same commands as described above, except the tape to be loaded is part number 92101-16008.

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*,SCHEDR,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 8K background (or partition) area and at least 1.5K 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 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. All requirements and instructions are included in the RTE Programming and Operating Manual.

MULTIPLE COPIES OF BASIC

To prepare multiple copies of BASIC for use on the Multi-User Real-Time BASIC terminals you must use the following File Manager commands:

<code>:SP,BASIC</code>	<i>Save a copy of the BASIC program in a file.</i>
<code>:RN,BASIC,BASIX</code>	<i>Rename the file BASIX.</i>
<code>:SP,BASIC</code>	<i>Save another copy of the BASIC program.</i>
<code>:RN,BASIC,BASIK</code>	<i>Name the file BASIK.</i>
<code>:SP,BASIC</code>	<i>Save another copy of the BASIC program.</i>
<code>:RN,BASIC,BASIQ</code>	<i>Name the file BASIQ.</i>

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:

<code>:CN,lu,20B</code>	<i>lu is the logical unit number of the terminal.</i>
-------------------------	---

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:

<code>lu ></code>	<i>lu is the logical unit number of the terminal you are using.</i>
----------------------	---

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

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

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.
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.
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).
7. Load these files: 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.

```

* 2313 AND 6940 TABLE GENERATOR * 2313 and 6940 Instrument Table Generation

* TURN ON PUNCH *
# OF 2313'S?
-
2

SUBSYSTEM #01 CONFIGURATION

LU?
-
12

# HL - SE?
-
32

# HL - DIF?
-
16

# LL, GAIN?
-
8,12.5
-
8,100
-
0,0

# DACS?
-
2

SUBSYSTEM #02 CONFIGURATION

LU?
-
11

# HL - SE?
-
32

# HL - DIF?
-
16

# LL, GAIN?
-
8,1000
-
8,50
-
0,0

```

Generate Instrument Table for two HP 2313 subsystems and two HP 6940 subsystems.

Required:

1. logical unit numbers,
2. type and number of cards,
3. gains of multiplexer cards,
4. I/O slots for HP 6940's.

Terminal interaction.

Loading Basic Software

DACS?

-

2

OF 6940'S?

-

2

OF CHANNELS IN 2313'S?

-

128

SUBSYSTEM #01 CONFIGURATION

I/O SLOT?

-

12

LU?

-

7

EVENT?

-

2

DIG INPUT FOR PRESET CNTR W/INT?

-

0

DIGITAL I/O?

-

2

DIGITAL INPUT?

-

0

VOLTAGE DACS?

-

2

CURRENT DACS?

-

2

TIMERS?

-

0

COUNTERS?

-

0

```
SUBSYSTEM #02 CONFIGURATION

I/O SLOT?
-
14

LU?
-
13

# EVENT?
-
2

# DIG INPUT FOR PRESET CNTR W/INT?
-
0

# DIGITAL I/O?
-
2

# DIGITAL INPUT?
-
0

# VOLTAGE DACS?
-
2

# CURRENT DACS?
-
2

# TIMERS?
-
0

# COUNTERS?
-
0

* CONFIGURATION CONSTANT PHASE

* ENTER INSTRUMENT CONFIGURATION CONSTANTS
-
/E

* TABLE GENERATION COMPLETE *
```

Loading Basic Software

RTE-II System Generation

MH DISC CHNL?

21

TRKS, FIRST TRK ON SUBCHNL:

0?

203,0

1?

203,0

2?

203,0

3?

203,0

4?

203,0

5?

203,0

6?

/E

128 WORD SECTORS/TRACK?

48

SYSTEM SUBCHNL?

1

SCRATCH SUBCHNL?

1

AUX DISC (YES OR NO OR # TRKS)?

NO

START SCRATCH?

100

TBG CHNL?

13

PRIV. INT. CARD ADDR?

0

FG SWAPPING?

YE

BG SWAPPING?

YE

FG CORE LOCK?

YE

BG CORE LOCK?

YE

SWAP DELAY?

50

LWA MEM?

77677

D-8 Change 1

PRGM INPT?
MT

LIBR INPT?
PT

PRAM INPT?
TY

INITIALIZE SUBCHNL:
0?

NO

2?

NO

3?

NO

4?

NO

5?

NO

PUNCH BOOT?
YES

PUNCH BOOT?
NO

*EOT

NO UNDEF EXTS

PARAMETERS

D,RTR,1,1
WMZAT,2,1
TATLG,2,98
MEM,1,9000
SYSON,82
ASMB,3,95
FTN4,3,95
XREF,3,96
LOADR,3,97
EDITR,3,50
CNTRL,2,10
DSCMD,3,90
COUMP,3,90
DDUMP,3,90
AUTOR,3,1
PRMPT,2
RSPNS,2
/E

Loading Basic Software

CHANGE ENTS?

```
.MPY,RP,100200
.OIV,RP,100400
.ULD,RP,104200
.DST,RP,104400
/E
```

OF BLANK ID SEGMENTS?

10

BASIC requires a minimum of 2 ID segments, plus one for each overlay.

OF BLANK BG SEG. ID SEGMENTS?

22

BASIC requires 11 background ID segments.

FWA BP LINKAGE?

30

SYSTEM

SCSYS(0099)02000 01777 92001-16012 REV.C 750305

DISPA(0099)02000 03162 92001-16012 750326

RTIME(0099)03163 03734 92001-16012 750305

SASCM(0099)03735 04014 92001-16012 741120

RTIOC(0099)04075 07716 92002-16022 741125

SALC (0099)07746 10153 92001-16012 741120

EXEC (0099)10157 11713 92001-16012 750305

STRRN(0099)11722 12065 92001-16012 750326

SCHED(0099)12131 15202 92002-16012 750225

DVP43(0099)15265 15612 92001-16004 REV.B 741028

SYSLB(0099)15613 15612 92001-16005 REV.B 741120

SBALB(0099)15613 15612 92002-16006 REV.C 750312

DVR00(0099)15626 16717

Include all required subsystem drivers.

DVR12(0099)16743 17600

DVR10(0099)17620 20123 7210 PLOTTER DVR,72009-60001 A SCHULTZ

DVA72(0099)20133 22040 09611-16005 REV.A 750617

DVR62(0099)22056 23116 29009-60001 REV. C 75156 -TLD-

DVR23(0099)23117 23762 92202-16001 REV. A

DVR31(0099)23764 25154

F40.C(0099)25157 25156

F2F.B(0099)25157 25156

BP LINKAGE 00171

** OF I/O CLASSES?

10

** OF LU MAPPINGS?

10

** OF RESOURCE NUMBERS?

10

BUFFER LIMITS (LOW, HIGH)?

100,400

* EQUIPMENT TABLE ENTRY

EQT 01?

21,DVR31,D

EQT 02?

15,DVR00,B

EQT 03?

23,DVR23,D,B,T=9999

EQT 04?

17,DVR02,B,T=50

EQT 05?

14,DVA72

EQT 06?

20,DVR12,B,T=100

EQT 07?

12,DVA72

EQT 08?

16,DVR01,T=50

EQT 09?

25,DVR00,B,T=5000

EQT 10?

26,DVR00,B,T=5000

EQT 11?

27,DVR00,B,T=5000

EQT 12?

30,DVR00,B,T=5000

Loading Basic Software

EQT 13?
31,DVR10,B,T=9999

EQT 14?
04,DVP43

EQT 15?
10,DVR62

EQT 16?
11,DVR62

EQT 17?
/E

* DEVICE REFERENCE TABLE

1 = EQT #?
2

2 = EQT #?
1,1

3 = EQT #?
0

4 = EQT #?
4,4

5 = EQT #?
8

6 = EQT #?
6

7 = EQT #?
7,0

8 = EQT #?
3

9 = EQT #?
8

10 = EQT #?
1

11 = EQT #?
16,0

12 = EQT #?
15,1

13 = EQT #?
5,1

14 = EQT #?
9

15 = EQT #?
2

16 = EQT #?
11

17 = EQT #?
12

18 = EQT #?
13

19 = EQT #?
14

20 = EQT #?
/E

* INTERRUPT TABLE

4,ENT,\$POWR

10,EQT,15

11,EQT,16

12,PRG,ALARM

14,PRG,ALARM

15,PRG,PRMPT

16,EQT,8

17,EQT,4

20,EQT,6

21,EQT,1

22,EQT,1

23,EQT,3

24,EQT,3

25,PRG,PRMPT

26,EQT,10

27,PRG,TTYEV

These entries allow interrupts to be serviced by the event sense facility of the 6940 in BASIC.

This entry allows interrupts to be serviced by the auxiliary TTY in I/O slot 27.

Loading Basic Software

30,PKG,PRMPT

31,EOT,13

/E

BP LINKAGE 00177

LIB ADDRS 30633

CHANGE LIB ADDRS?

0

LIBRARY

This is the TRAP and scheduling module used by BASIC (it is required).

TRAP	030633	31650	PRE-RELEASE	
PRTN	31651	31753	92001-16005	741120
86940	31754	32121		

BP LINKAGE 00201

FG COMMON 00000

CHANGE FG COMMON?

0

FG RES ADD 32122

CHANGE FG RES ADD?

0

FG RESIDENTS

HP 6940 and auxiliary TTY modules referenced in the Interrupt Table.

D.RTR(0001)	32127	34052	92002-16007	750421
P.PAS	34076	34124	92002-16006	740801
ALARM(0002)	34125	34522	92413-16007	07MAY75
TTYEV(0002)	34523	34532	29102-60013	
MEM (9000)	34533	34627	VERSION 2	NOV, 1973 -TS

BP LINKAGE 00207

FG DSC ADD 34630

CHANGE FG DSC ADD?

0

FG DISC RESIDENTS

PRMPT(0010)	34630	34740	92001-16003	REV,B	741216
EQLU	34741	35016	92001-16005		741120

RSPNS(0010)	34630	34776	92001-16003 REV.B	741002
EOLU	34777	35054	92001-16005	741120
MESSS	35055	35147	92001-16005	741120
.ENTR	35150	35237		

TATLG(0098)34630 35143

CNTRL(0010)	34630	34757
.ENTR	34760	35047
RMPAR	35050	35072
GETAD	35073	35110

WHZAT(0001)	34654	36305	RTE-II ONLY J.F.B.	741021
TMVAL	36323	36342	92001-16005	741120
.ENTR	36343	36432		

SYSON(0090)34630 34664 10 JUL 74 EJW

BP LINKAGE 00213

BP LINKAGE 00213
 CHANGE BP LINKAGE?
 600

SYS AVMEM 36433
 CHANGE SYS AVMEM?
 46000

BG BOUNDRY 46000
 CHANGE BG BOUNDRY?
 50000

BG COMMON 00000
 CHANGE BG COMMON ?
 0

BG RES ADD 50000
 CHANGE BG RES ADD?
 0

BG RESIDENTS

(NONE)

BG DSC ADD 50000
 CHANGE BG DSC ADD?
 0

BG DISC RESIDENTS

LOADR(0097)50000 60661 92001-16002 REV.B 741230

Loading Basic Software

AUTOR(0001)	50000	50405	
TNVAL	50406	50425	92001-16005 741120
FMTIO	50426	51656	
FRMTR	51671	54430	
DBLE	54512	54546	
SNGL	54547	54614	
.XPAK	54615	55011	
.XCOM	55012	55062	
.XFER	55063	55126	
CLRIO	55127	55131	
FLIB	55132	55234	
.ENTR	55235	55324	
.FLUN	55325	55345	
.PACK	55346	55461	
IAND	55462	55471	
PAUSE	55472	55635	
.ZRLB	55636	55676	
.OPSX	55677	55736	
FMGR (0090)	50000	50752	92002-16008 REV.C 750312
FM.CM	50754	52602	
.DRCT	52700	52706	92001-16005 741120
IFBRK	52707	52730	92001-16005 741120
OPEN	52731	53116	92002-16006 741205
CLOSE	53117	53225	92002-16006 740801
SDPEN	53226	53434	92002-16006 740801
RWND\$	53435	53545	92002-16006 740801
R/WS	53546	53701	92002-16006 740801
.ENTR	53702	53771	
RMPAR	53772	54014	
GETAD	54016	54033	
FMGR0(0099)	54034	54041	92002-16008 740801
PK..	54042	55465	
COR.A	55466	55501	92001-16005 741120
CR..	55503	56606	
READF	56735	57462	92002-16006 740801
REID	57463	57565	92001-16005 741120
RWPDF	57566	57647	92002-16006 740801
NAM..	57650	57744	92002-16006 740801
P.PAS	57745	57773	92002-16006 740801
RWSUB	57774	60245	92002-16006 750422
LOCK.	60247	60316	
FM.UT	60317	61441	
CREA.	61442	61513	
CREAT	61514	61771	92002-16006 741022
.XFER	61772	62035	
FMGR1(0099)	54034	54147	92302-16008 741223
.PARS	54150	55171	
C.TAB	55172	55322	92002-16008 740801
CA..	55323	55515	92002-16008 741119
REA.C	55516	55600	
EE..	55601	55642	
TR..	55644	56065	
MR..	56102	56236	
SE..	56237	56423	

IF..	56424	56634		
AB..	56635	57027		
DP..	57030	57067		
INPRS	57070	57164	92001-16005	741119
READF	57165	57712	92002-16006	740801
REIO	57716	60020	92001-16005	741120
POSNT	60022	60256	92002-16006	740801
P.PAS	60257	60305	92002-16006	740801
RWSUB	60306	60557	92002-16006	750422
WRLG	60560	60575	92002-16006	740801
CK.SM	60576	60711		
.XFER	60712	60755		
%WRIT	60756	61456		
.OPSY	61457	61516		
FMGR2(0099)	54034	54045	92002-16008	740801
IN.IT	54046	54773		
IN..	55001	56652		
MC..	56757	57266		
RC..	57267	57454		
PU..	57455	57677		
PURGE	57700	57776	92002-16006	740801
NAM..	57777	60073	92002-16006	740801
J.PUT	60076	60122	92002-16006	740801
IPUT	60123	60143	92002-16006	740801
FID.	60144	60263		
MSC.	60264	60320		
LOCK.	60321	60370		
FM.UT	60371	61513		
CNT.	61514	61734		
FCONT	61740	62031	92002-16006	740801
.XFER	62037	62102		
FMGR3(0099)	54034	54041	92002-16008	740801
LI..	54042	55501		
DL..	55503	56722		
READF	57034	57561	92002-16006	740801
REIO	57562	57664	92001-16005	741120
LOCF	57674	60062	92002-16006	750416
P.PAS	60106	60134	92002-16006	740801
RWSUB	60135	60406	92002-16006	750422
MSC.	60407	60443		
FM.UT	60444	61566		
.XFER	61567	61632		
FMGR4(0099)	54034	54050	92002-16008	740801
ST.DU	54051	55305		
CO..	55306	56010		
F.UTM	56015	56163		
OPMES	56164	56364		
CREAT	56365	56642	92002-16006	741022
READF	56643	57370	92002-16006	740801
REIO	57371	57473	92001-16005	741120
RWNDF	57474	57555	92002-16006	740801
LOCF	57556	57744	92002-16006	750416
NAM..	57750	60044	92002-16008	740801
P.PAS	60047	60075	92002-16006	740801

Loading Basic Software

RWSUB	60076	60347	92002-16006	750422
FM.UT	60350	61472		
CREA.	61473	61544		
CK.SM	61545	61660		
.XFER	61661	61724		
FMGR5(0099)	54034	54043	92002-16008	740801
??..	54044	56157	92002-16008	740801
RU..	56160	56620		
RP..	56621	57762		
TL..	57763	60002		
READF	60004	60531	92002-16006	740801
REIO	60532	60634	92001-16005	741120
LOCF	60635	61023	92002-16006	750416
P.PAS	61024	61052	92002-16006	740801
RWSUB	61053	61324	92002-16006	750422
J.PUT	61325	61351	92002-16006	740801
IPUT	61352	61372	92002-16006	740801
ID.A	61373	61462		
BUMP.	61463	61521	92002-16006	741025
SET.T	61522	61550	92002-16006	740801
TL.	61551	61600	92002-16006	741025
ST.TM	61601	61635	92002-16006	741223
.XFER	61636	61701		
FMGR6(0099)	54034	54044	92002-16008	740801
CN..	54045	54105		
JU..	54106	55203		
RNRQ	55204	55431	92001-16005	741120
\$ALRN	55432	55537	92001-16005	741106
KCVT	55540	55552	92001-16005	741120
EO..	55555	56400		
MESSS	56446	56540	92001-16005	741120
QF..	56541	56634		
LG..	56635	57001		
PARSE	57002	57021	92001-16005	741120
INPRS	57022	57116	92001-16005	741119
NAMF	57117	57270	92002-16006	740801
READF	57274	60021	92002-16006	740801
REIO	60022	60124	92001-16005	741120
POST	60125	60153	92002-16006	740801
NAM..	60154	60250	92002-16006	740801
P.PAS	60251	60277	92002-16006	740801
RWSUB	60300	60551	92002-16006	750422
SPOPN	60552	60622	92002-16006	741025
SET.T	60623	60651	92002-16006	740801
ST.TM	60652	60706	92002-16006	741223
B.FLG	60707	60755	92002-16006	741118
LULU.	60756	61051	92002-16006	740801
RANGE	61052	61075	92002-16006	740801
ONOFF	61076	61441	92002-16006	750128
EX.TM	61442	61630	92002-16006	741008
IPUT	61631	61651	92002-16006	740801
FREE.	61652	61721	92002-16006	740801
LU.CL	61722	61763	92002-16006	740801
AVAIL	61765	62057	92002-16006	741231
.XFER	62066	62131		

FMGR7(0099)	54034	54043	92002-16008	740801
CL..	54044	54325		
NX.JB	54326	55173		
RNRQ	55174	55421	92001-16005	741120
SALRN	55422	55527	92001-16005	741106
LU..	55531	56604		
KCVT	56674	56706	92001-16005	741120
CS..	56707	57104		
READF	57105	57632	92002-16006	740801
REIO	57633	57735	92001-16005	741120
FSTAT	57736	57762	92002-16006	740801
POST	57764	60012	92002-16006	740801
P.PAS	60017	60045	92002-16006	740801
RWSUB	60046	60317	92002-16006	750422
3POPN	60320	60370	92002-16006	741025
B.FLG	60371	60437	92002-16006	741118
LULU.	60440	60533	92002-16006	740801
RANGE	60534	60557	92002-16006	740801
AVAIL	60560	60652	92002-16006	741231
.XFER	60653	60716		

FMGR8(0099)	54034	54042	92002-16008	740801
SA..	54043	55015		
SP..	55016	55635		
MS..	55636	56131		
READF	56151	56676	92002-16006	740801
REIO	56677	57001	92001-16005	741120
RWNDF	57002	57063	92002-16006	740801
LUCF	57064	57252	92002-16006	750416
P.PAS	57253	57301	92002-16006	740801
RWSUB	57302	57553	92002-16006	750422
IPUT	57554	57574	92002-16006	740801
CREA.	57575	57646		
CREAT	57662	60137	92002-16006	741022
NAM..	60152	60246	92002-16006	740801
CK.SM	60247	60362		
ID.A	60363	60452		
WRISS	60453	60511	92002-16006	740801
READ.	60512	60536	92002-16006	740801
.XFER	60537	60602		
XWRIS	60603	61072		
SREAD	61073	61630		
.OPSY	61631	61670		

EDITR(0050)	50000	54454	92002-16010	REV.C	750505
REIO	54455	54557	92001-16005	741120	
CREAT	54560	55035	92002-16006	741022	
OPEN	55036	55223	92002-16006	741205	
READF	55224	55751	92002-16006	740801	
CLOSE	55754	56062	92002-16006	740801	
NAM..	56072	56166	92002-16006	740801	
\$OPEN	56167	56375	92002-16006	740801	
P.PAS	56376	56424	92002-16006	740801	
RWSUB	56425	56676	92002-16006	750422	
RWNDS	56677	57007	92002-16006	740801	
R/WS	57010	57143	92002-16006	740801	

Loading Basic Software

.XFER	57144	57207	
.ENTR	57210	57277	
RMPAR	57300	57322	
GETAD	57323	57340	
ASMB (0095)50000	55577	92060-16022	REV,A 750420
ASMB0(0099)55600	56406	92060-16023	REV,A 750420
ASMB1(0099)55600	57630	92060-16024	REV,A 750420
ASMB2(0099)55600	60076	92060-16025	REV,A 750420
ASMB3(0099)55600	56671	92060-16026	REV,A 750602
ASMB4(0099)55600	57331	92060-16027	REV,A 750420
FTN4 (0095)50000	63573		
F4.0 (0099)63574	71444		
F4.1 (0099)63574	67335		
F4.2 (0099)63574	71072		
F4.3 (0099)63574	71072		
XREF (0096)50000	56166	92060-16028	REV,A 750420
.OPSY	56167	56226	
CDUMP(0090)50000	51046		
ICAR	51047	51105	
IGET	51106	51120	
FMTIO	51123	52353	
FRMTR	52442	55201	
DBLE	55351	55405	
SNGL	55406	55453	
.XPAK	55454	55650	
.XCDM	55651	55721	
.XFER	55722	55765	
CLRIO	55766	55770	
FLIB	55771	56073	
.ENTR	56076	56165	
.FLUN	56166	56206	
.PACK	56207	56322	
.IAND	56323	56332	
RMPAR	56333	56355	
GETAD	56356	56373	
.ZRLB	56374	56434	
.OPSY	56435	56474	
DDUMP(0090)50000	50717		
ICAR	50720	50756	
FMTIO	50762	52212	
FRMTR	52254	55013	
DBLE	55153	55207	
SNGL	55210	55255	

.XPAK	55256	55452		
.XCOM	55453	55523		
.XFER	55524	55567		
CLRIO	55570	55572		
FLIB	55573	55675		
.ENTR	55676	55765		
.FLUN	55767	56007		
.PACK	56011	56124		
IAND	56125	56134		
RMPAR	56135	56157		
GETAD	56160	56175		
.ZRLB	56176	56236		
.OPSY	56237	56276		
DSCMD(0090) 50000 51467				
IPUT	51470	51510	92002-16006	740801
IGET	51511	51523		
FMTIO	51524	52754		
FRMTR	53101	55640		
DBLE	56000	56034		
SNGL	56035	56102		
.XPAK	56103	56277		
.XCOM	56300	56350		
.XFER	56351	56414		
CLRIO	56415	56417		
FLIB	56420	56522		
.ENTR	56523	56612		
.FLUN	56613	56633		
.PACK	56634	56747		
.GOTO	56750	56772		
IAND	56773	57002		
RMPAR	57003	57025		
GETAD	57026	57043		
.ZRLB	57044	57104		
.OPSY	57105	57144		
DSAVE(0090) 50001 64300				
CLRIO	64302	64304		
IAND	64305	64314		
RMPAR	64315	64337		
GETAD	64340	64355		
PAUSE	64356	64521		
.OPSY	64522	64561		
.TAPE	64562	64574		
DRSTR(0090) 50001 64163				
CLRIO	64165	64167		
IAND	64170	64177		
RMPAR	64200	64222		
GETAD	64223	64240		
PAUSE	64241	64404		
.OPSY	64405	64444		
.TAPE	64445	64457		

BP LINKAGE 01275

SYSTEM STORED ON DISC

SYS SIZE: 26 TRKS, 011 SECS(10)

RTE-II System Generation is finished.

Loading Basic Software

SET TIME

FMGR 002

:IN,JS,-2,2,BASIC,100

FMGR-006

:

Load the RTE system.

Initialize File Manager with security code, disc labels, and first tracks used for files.

The FMGR-006 message is printed if there is not a WELCOME file on the system.

COM	50000	50362					
BASIC	50363	52666	92101-16002	750724			Load BASIC on-line after RTE system is loaded.
DEBUG	52667	52671					
SPDUM	52672	52675	92101-16003	750724			

REIO	52676	53000	92001-16005	741120			
.XFER	53001	53044					
.ENTR	53045	53134					
FLIB	53135	53237					
READF	53240	53765	92002-16006	740801			
RMPAR	53766	54010					
P.PAS	54011	54037	92002-16006	740801			
R/W\$	54040	54173	92002-16006	740801			
RWSUB	54174	54445	92002-16006	750422			
RWND\$	54446	54556	92002-16006	740801			
..FCM	54557	54571					
.FLUN	54572	54612					
..DLC	54613	54627					
.PACK	54630	54743					
GETAD	54744	54761					
.ZRLB	54762	55022					
.OPSY	55023	55062					

This is the BASIC load map.

BASC1	55063	60460	92101-16004	750724			
MNEM	60461	60551	92101-16005	750724			

BASC2	55063	61412	92101-16006	750724			
CLOSE	61413	61521	92002-16006	740801			
OPEN	61522	61707	92002-16006	741205			
\$OPEN	61710	62116	92002-16006	740801			
MNEM	62117	62207	92101-16005	750724			

BASC3	55063	56343	92101-16006	750724			
OPEN	56344	56531	92002-16006	741205			
\$OPEN	56532	56740	92002-16006	740801			
CLOSE	56741	57047	92002-16006	740801			

BASC4	55063	62500	92101-16006	750812
BRTBL	62501	62546	92101-16007	750724

AINI	62547	62637		
ALOGT	62640	62650		
ALOG	62651	62770		

SQRT	62771	63117		
ATAN	63120	63255		
EXP	63256	63420		
SICOS	63421	63564		
TAN	63565	63730		
ABS	63731	63740		
PWR2	63741	63770		
CHEBY	63771	64067		
MANT	64070	64103		

BASC5	55063	57201	92101-16008	750724
-------	-------	-------	-------------	--------

LURQ	57202	57526	92001-16005	741120
CLOSE	57527	57635	92002-16006	740801
OPEN	57636	60023	92002-16006	741205
NAMF	60024	60175	92002-16006	740801
LOCF	60176	60364	92002-16006	750416
PURGE	60365	60463	92002-16006	740801
CREAT	60464	60741	92002-16006	741022
\$OPEN	60742	61150	92002-16006	740801
FCONT	61151	61242	92002-16006	740801
\$ALRN	61243	61350	92001-16005	741106
NAM..	61351	61445	92002-16006	740801

BASC6	55063	57502	92101-16008	750724
-------	-------	-------	-------------	--------

LURQ	57503	60027	92001-16005	741120
CLOSE	60030	60136	92002-16006	740801
OPEN	60137	60324	92002-16006	741205
LOCF	60325	60513	92002-16006	750416
POSNT	60514	60750	92002-16006	740801
\$OPEN	60751	61157	92002-16006	740801
\$ALRN	61160	61265	92001-16005	741106

BASC7	55063	60156	92101-16008	750724
-------	-------	-------	-------------	--------

CLOSE	60157	60265	92002-16006	740801
OPEN	60266	60453	92002-16006	741205
\$OPEN	60454	60662	92002-16006	740801
MNEM	60663	60753	92101-16005	750724

Loading Basic Software

BASC8	55063	55770	92101-16008	750724
CLOSE	55771	56077	92002-16006	740801
OPEN	56100	56265	92002-16006	741205
SOPEN	56266	56474	92002-16006	740801
RTETG	50000	55142		

T.LNK	55143	55170	92101-16009	750407	Load RTETG (BASIC Table Generator) on-line.
-------	-------	-------	-------------	--------	--

.XFER	55171	55234		
.ENTR	55235	55324		
READF	55325	56052	92002-16006	740801
CLOSE	56053	56161	92002-16006	740801
OPEN	56162	56347	92002-16006	741205
RWNDF	56350	56431	92002-16006	740801
PURGE	56432	56530	92002-16006	740801
RMPAR	56531	56553		
CREAT	56554	57031	92002-16006	741022
POST	57032	57060	92002-16006	740801
R/WS	57061	57214	92002-16006	740801
P.PAS	57215	57243	92002-16006	740801
SOPEN	57244	57452	92002-16006	740801
RWSUB	57453	57724	92002-16006	750422
RWNDS	57725	60035	92002-16006	740801
GETAD	60036	60053		
REIO	60054	60156	92001-16005	741120
NAM..	60157	60253	92002-16006	740801

This is load map of RTETG.

TG008	60254	60266	92101-16009	750404
T.BTE	60267	62034	92101-16009	750818
.DRCT	62035	62043	92001-16005	741120

TG018	60254	60266	92101-16009	750407
T.MES	60267	60661	92101-16009	750407
T.OVL	60662	61541	92101-16009	750507
.DRCT	61542	61550	92001-16005	741120
KCVT	61551	61563	92001-16005	741120

TG028	60254	60261	92101-16009	750407
T.TRF	60262	61534		
.DRCT	61535	61543	92001-16005	741120
KCVT	61544	61556	92001-16005	741120
CNUMD	61557	61576	92001-16005	741120
IABS	61577	61610		

```

:RU,RTETG,CO,MN,D
$END RTETG
:TR,TRFL
:LG,10
:MR,%BA00 :      0:      0
:MR,A6940 :      0:      0
:RU,LOADR,99,6,7, 0000
:LG,10
:MR,%BA01 :      0:      0
:MR,A2313 :      0:      0
:RU,LOADR,99,6,7, 0000
:LG,10
:MR,%BA02 :      0:      0
:RU,LOADR,99,6,7, 0000
:LG,10
:MR,%BA03 :      0:      0
:MR,PLOTR :      0:      0
:RU,LOADR,99,6,7, 0000
:LG,10
:MR,%BA04 :      0:      0
:MR,SCHEDR:      0:      0
:RU,LOADR,99,6,7, 0000
::
:EX
/LOADR:%BA00 READY
$END FMGR
/LOADR:$END
/LOADR:%BA01 READY
/LOADR:$END
/LOADR:%BA02 READY
/LOADR:$END
/LOADR:%BA03 READY
/LOADR:$END
/LOADR:%BA04 READY
/LOADR:$END

```

After creation of the RTETG input file, the Table Generator can be executed and the output includes:

1. Branch and Mnemonic Files,
2. Transfer File used to produce overlays,
3. Overlay relocatable files.

After running the Table Generator, RTETG, run the Transfer File using the File Manager. This produces overlays %BA00 through %BA04.

Loading Basic Software

```

XBA00  34641 34662
A6940  34663 35411
DAC    35412 35561

```

29102-16003A 05JUN75

These are the load maps
of the overlays:

```

.ENTR  35562 35651
CALSB  35652 36374
BITCR  36375 36563
#GETI  36564 36656
ISETC  36657 36745
RMPAR  36746 36770
GETAD  36771 37006

```

92101-16012 750923
29102-60007 REV. A
92413-16005 REV A 24APR75
92101-16011 750724

%BA00

ENTRY POINTS

```

*EXEC  10157
*$LIBR 10367
*$LIBX 11050
*PRTN  31660
*A6940  31763
*XBA00  34641
*CALS0  35652
*BRSET  36462
*BEOR  36407
*BIOR  36417
*BAND  36377
*BNOT  36426
*BSHFT 36436
*BBTST 36501
*BBCLR 36523
*ISETC 36660
*DAC   35414
*MPNRM 35265
*RDWRD 34665
*WRWRD 34723
*ROBIT 34702
*WRBIT 34737
*SENSE 34737
*.ENTR 35571
*#GETI 36564
*ERROR 36171
*.ENTP 35562
*ERRCD 36317
*LUERR 36302
*DESPT 36306
*RMPAR 36746
*GETAD 36771
*ADRES 37005

```

%BA01	34641	34651	
A2313	34652	35624	29102-80016B 05JUN75
ADV	35625	36000	

%BA01

.ENTR	36001	36070	
..ADC	36071	36114	
CALSB	36115	36637	92101-16012 750923
..FCM	36640	36652	
..DLC	36653	36667	
RMPAR	36670	36712	
GETAD	36713	36730	

ENTRY POINTS

*EXEC	10157
*\$LIBR	10367
*\$LIBX	11050
*PRTN	31660
*.DST	104400
*%BA01	34641
*CALS8	36115
*AISQV	35332
*AIRDV	35025
*PACER	35514
*NORM	34753
*SGAIN	34677
*RGAIN	34654
*ADV	35631
*..ADC	36071
*..FCM	36640
*.ENTR	36010
*ERROR	36434
*..DAC	36111
*.ENTP	36001
*ERRCD	36562
*LUERR	36545
*DESPT	36551
*RMPAR	36670
*..DLC	36653
*GETAD	36713
*ADRES	36727

Loading Basic Software

XBA02 34641 34646

%BA02

MTTDR 34647 35122

CALSB 35123 35645

.ENTR 35646 35735

PAUSE 35736 36101

RMPAR 36102 36124

.OPSY 36125 36164

GETAD 36165 36202

A-92101-16014-1 REV. A
92101-16012 750923

ENTRY POINTS

*EXEC 10157

*SLIBR 10367

*SLIBX 11050

*PRTN 31660

*XBA02 34641

*CALSB 35123

*MTTRD 34662

*MTTRT 34647

*MTTPT 34755

*MTTFS 35050

*.ENTR 35655

*ERROR 35442

*.STOP 35774

*ERRCD 35570

*LUERR 35553

*DESPT 35557

*RMPAR 36102

*.ENTP 35646

*.PAUS 35736

*.OPSY 36125

*GETAD 36165

*ADRES 36201

XBA03 34641 34656
 PLOT 34657 35207
 SYMB 35210 36464
 AXIS 36465 37765
 LINES 37766 40503
 SCALE 40504 41360
 NUMB 41361 42211

%BA03

92409-80001 REV. C
 92409-80002 REV. C

.ENTR 42212 42301
 FLIB 42302 42404
 .FCM 42405 42417
 CALSB 42420 43142
 ERR0 43143 43215
 ALOG 43216 43335
 .DLC 43336 43352
 SIC0S 43353 43516
 ABS 43517 43526
 IABS 43527 43540
 PWR2 43541 43570
 .RTOR 43571 43662
 .IENT 43663 43723
 CHEBY 43724 44022
 MANT 44023 44036
 .ZRLB 44037 44077
 .RTOI 44100 44233
 RMPAR 44234 44256
 .OPSY 44257 44316
 .FLUN 44317 44337
 EXP 44340 44502
 GETAD 44503 44520

92101-16012 750923

ENTRY POINTS

*EXEC 10157
 *SLIBR 10367
 *SLIBX 11050
 *PRTN 31660
 *.MPY 100200
 *.DIV 100400
 *.DL0 104200
 *.DST 104400
 *PPTLU 31651
 *DFAC 31654
 *CFAC 31652
 *XPEN 31656
 *YPEN 31657
 *XBA03 34641
 *CALS0 42420
 *SFACT 35110
 *FACT 34703
 *WHERE 34661
 *PLOT 34731
 *LLEFT 35051
 *URITE 35074
 *PLTLU 35035
 *AXIS 36474

Loading Basic Software

```

*NUMB 41367
*SYMB 35216
*LINE3 37774
*SCALE 40510
*.ENTR 42221
*FLOAT 42345
*IFIX 42336
*.FDV 42327
*.FMP 42320
*.FAO 42302
*SIN 43364
*COS 43353
*ERR0 43143
*.FSB 42311
*..FCM 42405
*.RTOR 43571
*ALOG 43216
*ABS 43517
*IABS 43527
*.RTOI 44100
*.ENTP 42212
*.ZRLB 44037
*..DLC 43336
*ERRCD 43065
*LUERR 43050
*DESPT 43054
*ERROR 42737
*RMPAR 44234
*.UPSY 44257
*LN 43216
*.FLUN 44317
*.MANT 44023
*.CHER 43724
*.IENT 43663
*.PWR2 43541
*EXP 44340
*GETAD 44503
*ADRES 44517

```

```

XBA04 34641 34651
SCHO 34652 35427 92101-16013 750724

.ENTR 35430 35517
CALSB 35520 36242 92101-16012 750923
..FCM 36243 36255
..DLC 36256 36272
RMPAR 36273 36315
GETAD 36316 36333

```

%BA04

ENTRY POINTS

*EXEC	10157
*\$LIBR	10367
*\$LIBX	11050
*PRTN	31660
*TRAP#	31646
*TIME	31145
*TSNXT	31345
*TSEND	31343
*TSTBL	31262
*TSTIM	31346
*TSCNT	31350
*TSPTR	31344
*FINDS	31374
*TRMAK	31471
*TRDEL	31444
*TRTBL	31573
*TRPTR	31635
*TRNXT	31636
*TRFLG	31637
*TRMSK	31644
*TRPNO	31643
*SEKNO	31641
*PRIND	31642
*%BA04	34641
*CALSB	35520
*SSETP	34742
*SSTRY	35174
*DSABL	35073
*ENABL	35033
*TKNON	35157
*TTYS	35001
*ERROR	36037
*.ENTR	35437
*..FCM	36243
*.ENTP	35430
*ERRCO	36165
*LUERR	36150
*DESPT	36154
*RMPAR	36273
*..DLC	36256
*GETAD	36316
*ADRES	36332

BASIC READY

>TABLES BTBL,MTBL

>CALLS

<<NAME(PARAMETER DESCRIPTION) TYPE-OVERLAY#>>

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

MTTTR(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

URITE 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

BASIC is ready for use with a complete set of subroutines.

Use the CALLS command to list them.

Now you may use BASIC as it is described in this manual.

- ABORT command, 10-4
- ABS function, 5-1
- ADC, HP 2313, 15-10
- AIRDV (random scan) routine, 15-2
- AISQV (sequential scan) routine, 15-3
- ALARM program, D-1
- alignment, common area, 3-17
- analog input
 - measurement, 15-1
 - read randomly, 15-2
- analog output, 15-1
- AND function, 12-1
- AOC-1, -2, -3 errors, 15-8
- AOV (digital to analog conversion) routine, 15-4
 - Instrument Table generation, 18-2
- AOV-1, -2 errors, 15-8
- argument, function, 5-1
- arithmetic operators, 2-4
- array
 - definition, 2-3
 - initialization, 3-16
 - maximum, 3-16
 - size declaration, 3-16
- ASCII character set, C-1
- assignment operator, 3-1
- ASSIGN statement, 7-3
- associate trap number with task, 11-9
- ATN function, 5-1
- auxiliary teleprinter interrupt, 11-12
- AXIS routine, 17-1
- A6940-1, -2 errors, 16-6

- BACKF command, 13-1
- background memory area, 1-3
- BACKSPACE key, 1-8
- Basic Binary Loader (BBL), 18-1
- BASIC
 - command file (ASCII) 8-2
 - components layout, 1-4
 - copies, D-2
 - initiate from another program, 8-3
 - load map, D-21
 - ordering information, iv
 - prompt, 8-2
 - Resident Library, D-1
 - Scheduler, 11-3
 - software, loading, D-1
 - software part nos., iv
 - statement formats, 1-5
 - Subroutine Library, D-1
 - subsystem modules, 1-3
 - system generation, D-1
- bit clear, function, 12-2
- bit manipulation, 12-1
- bit set function, 12-3
- bit test function, 12-3
- Boolean operators, 2-5
- Branch and Mnemonic Tables
 - declare, 8-2
 - generation, 14-1
- BR,BASIC, 9-11
 - example, 11-2
- break BASIC program, 9-11
- BREAK command, 10-3
- break points, 10-3
- BYE command, 9-10

- C^c
 - call, function, 2-3
- CALL statement, 6-6
- CALLS command, 9-12
 - sample, complete system subroutines, D-31
- calls, magnetic tape, 13-2
- card configuration, HP 6940, 16-7
- Cartridge Directory, 7-2
- cartridge reference, 7-2
- CHAIN statement, 6-4
- channel numbering
 - HP 2313, 15-11
 - HP 6940, 16-8
- character editing, 1-7
- clear event sense mode, 16-2
- CN command, File Manager, D-3
- CNTL (control) key, 1-7
- column number, 2-3
- commands, avoid with real-time processing, 11-3
- commands from disc file, 8-2
- commands, introduction, 1-5
- commands, legal during break, 10-3
- command summary, A-3
- common area, 3-16
 - alignment, 3-17
- COM statement, 3-16
 - chain to program, 6-5
- compare strings, 4-8
- conditional transfer, 3-8
- consecutive task initiation, 11-2
- constant, 2-1
- control characters, editing, 1-7
- control D (RTETG program), 14-2
- control H, 1-8
- control Q, 3-15, 3-17
 - break program execution, 9-11
- conventions, file creation, 7-1
- conversational programming, 1-1
- convert digital to analog, 15-4, 16-1
- convert parameters, FORTRAN subroutine, 6-12
- copies of BASIC, D-2
- copy BASIC program to a device, 9-11
- correction of typing errors, 1-8

- COS function, 5-1
- CR (create) command, File Manager, 7-2
- CREATE command, BASIC, 7-2, 9-6
- CSAVE command, 9-3
 - chained program, 6-5
 - file type, 7-2
- current digital to analog conversion card, 18-3
- D^C(RTETG program), 14-2
- DAC (digital to analog conversion)
 - cards, 15-10
 - Instrument Table generation, 18-2
 - routine, HP 6940, 16-1
- DAC-1, -2 errors, 16-6
- data list, 3-13
 - pointer, 3-14
- DATA statement, 3-13
 - strings, 4-9
- debugging activity, terminate, 10-3
- debugging commands, 10-1
- default devices, running BASIC with, 8-1
- DEF FNx statement, 5-2
- DELETE (DEL) command, 1-5, 9-5
- DEL key, 1-8
- destination string, 4-5
- digital input only cards (Instrument Table generation), 18-3
- digital input/output cards, 18-3
- digital to analog, 15-4
 - conversion, 16-1
- DIM (dimension) statement, 2-3, 3-16
- documentation map, v
- DSABL routine, 11-5
- dummy TRAP module, D-1
- DVR 10 plotter driver, 17-1
 - part number, iv
- editing control characters, 1-7
- eliminate break points, 10-3
- ENABL routine, 11-5
- end-of-file condition, 7-4
 - magnetic tape, 13-1
- end-of-record mark, 7-6
- END statement, 3-5
- environment, BASIC, 1-2
- erase channel/bit to trap no. correspondence, 16-2
- ERRCD flag, 6-9, 6-10
 - ENABL routine, 11-5
 - with tasks, 11-9, 11-17
- ERROR A6940-2 IN LINE, 16-4
- ERROR MAGTP-n IN LINE, 13-4
- error messages
 - Instrument Table tape generation, 18-4
 - RTETG, 14-6
 - summary of formats, B-1
 - task, 11-17
 - 2313/91000 subsystem, 15-8
 - 6940 subsystem, 16-6
- evaluating expressions, 2-5
- event interrupt, 11-3
- event sense
 - cards, 18-3
 - interrupt (ALARM program), D-1
 - mode, clear, 16-2
 - routine, HP 6940, 16-4
- exclusive or function, 12-4
- execute a BASIC program, 9-9
- execute task at specified time, 11-11
- EXP function, 5-1
- expansion, HP 6940, 16-8
- expressions, 2-1, 2-4
- external event interrupt, 11-2
- external subroutine, parameter conversion, 6-11
- FACT (factor) routine, 17-2
- FAIL error option, 6-9
- features, 1-1
- fixed point numbers, 2-1
 - field width, 3-12
 - print format, 3-12
- floating point number, 2-2
 - print format, 3-12
- FOR. . .NEXT statement, 3-6
- formal parameters, 2-3
- format
 - integer, 3-11
 - real type data, 6-10
 - string data, 6-10
- FORTTRAN function, prepare and use with BASIC, 6-7
- FORTTRAN subroutine
 - convert string parameter, 6-12
 - prepare and use with BASIC, 6-8
- file
 - capabilities, 1-2
 - characteristics, 7-1
 - conventions, 7-1
 - create data (type 1), 9-6
 - label, 7-2
 - length, 1-2
 - magnetic tape, 13-1
 - organization, 1-2
 - security code, 7-1
 - types, 7-2
 - type 1, 1-2
- File Management Package, 1-3
- FILES statement, 7-2
 - chained programs, 6-5
- free format, BASIC, 1-5
- functions, 2-3, 5-1
 - FORTTRAN with BASIC, 6-7
- gain
 - allowable for low-level channels, 18-2
 - set all channels, 15-8
 - setting, HP 2313/91000, 15-11
 - setting, request, 15-7
- GO command, 3-17
- GOSUB statement, 6-1
- GOTO statement, 3-4

- H^c, 1-8
- hardware, 1-2
- hierarchy of operators, 2-5
- highest numbered statement, 3-5
- high-level input, 15-1
- HLMPX (high-level multiplexer), 15-11
 - differential cards, 18-2
 - single ended cards, 18-2
- HLT07,70, 18-1
- home or known state, reset, 15-5
- HP 2313/91000 data acquisition subsystem, 15-1
 - card configuration, 15-10
 - channel numbering, 15-11
 - concept, 15-9
 - configuration, 15-9, 18-1
 - errors, 15-8
 - Instrument Table, 18-1
 - normalize, 15-5
 - setting gain, 15-11
- HP 6940 subsystem, 16-1
 - card configuration, 16-7
 - channel numbering, 16-8
 - clear event sense, 16-2
 - configuration, 16-8, 18-2
 - errors, 16-6
 - expansion, 16-8
 - Instrument Table generation, 18-1
- HP 7210 Plotter, 17-1
 - sample program, 17-9
 - symbols (ASCII reference number), 17-8
- HP 9600 computer system, 1-2
- HP 91000 (Instrument Table generation), 18-2
- I^c, 1-7
- IBCLR (bit clear) function, 12-2
- IBSET (bit set) function, 12-3
- IBTST (bit test) function, 12-3
- idle-loop, 11-2
- ID segment, 14-6
 - prepare at system generation, D-2
- IF END #. THEN statement, 7-4
- IF. . THEN statement, 3-8
 - strings, 4-8
- IEOR (exclusive or) function, 12-4
- IERR function, 5-1, 6-10
- inclusive or function, 12-5
- initialize array, 3-16
- initiate BASIC from another program, 8-3
- initiating tasks, 11-1
- input data, 3-13
- INPUT statement, 3-14
 - string, 4-5
- instrument drivers, prepare at system generation, D-2
- Instrument Table generation
 - errors, 18-4
 - input at system generation, D-1
 - loading tape, 18-4
 - operating instructions, 18-1
 - part numbers, iv
 - sample, D-4
- integer expression, 2-1
- integer format, 3-11
- interrupt BASIC program or listing, 9-11
- interrupt, maximum time to service, 11-3
- Interrupt Table system generation, D-13
- INT function, 5-1
- I/O slot, computer, 15-9, 18-3
- ISETC function (set to octal), 12-6
- ISHFT function (register shift), 12-6
- item type, 3-13
- jumper W3, 16-4
- label, file, 7-2
- Last Address Detector (LAD) card, 15-10
- layout, BASIC components, 1-4
- legal commands during break, 10-3
- LEN function, 4-8, 5-1
- length, string data item, 1-2
 - logical, 4-2
 - physical, 4-2
- LET statement, 3-1
 - strings, 4-5
- library subroutines, RTETG commands, 14-5
- line number, 3-1
- LINES routine, 17-2
- LIST command, 1-7, 9-11
- list subroutine names, 9-12
- literal strings, 2-2
- LLEFT routine, 17-3
- LLMPX (low-level multiplexer), 15-11, 18-2
- LN function, 5-1
- load and execute BASIC program, 9-9
- LOAD command, 9-1
- load map, BASIC, D-21
- loading BASIC software, D-1, D-2
- loading Instrument Table tape, 18-4
- loading RTE system, sample, D-21
- loading RTETG, D-2
- LOCK command, 9-10
- LOG function, 5-1
- logical expression, order of evaluation, 2-6
- logical length of string, 4-2, 4-8
- logical operation, 2-4
- logical operators, 2-5
- logical unit number, 1-8
 - plotter, 17-5
- looping statements, 3-6
- lower-case alphabetic character, 4-1
- low-level input, 15-1
- magnetic tape I/O, 13-1
 - errors, 13-4
 - operator commands, 13-1
 - position, 13-3
 - read or write record, 13-2
 - sample program, 13-5
 - subroutine calls, 13-2
 - write EOF, 13-4
- main memory partition, 1-3
- mathematical operation, 2-4
- maximum array size, 3-16
- maximum number of tasks per program, 11-2
- measurement of analog input, 15-1

- memory, BASIC and overlays, 14-1
- MERGE command, 9-4
- methods of initiating tasks, 11-1
- minimum width numeric field, 3-11
- modifying records, 7-9
- MPNRM routine, 16-2
- MTTFS routine, 13-4
- MTTPT routine, 13-3
- MTTRD routine, 13-2
- MTTRT routine, 13-2
- multi-branch GOSUB statement, 6-1
- multi-branch GOTO statement, 3-5
- multiple peripheral device I/O, 1-1
- multiple terminal operation (MTM), D-2

- nested loops, 3-7
- nesting subroutines, 6-1, 6-4
- NEXT statement, 3-6
- normalize 2313/91000 subsystem, 15-5
- NORM routine, 15-5
 - Instrument Table, 18-2
- NOT function, 12-5
- null string, 2-2, 4-1
 - creating a, 4-4
- NUMB routine, 17-4
- numeric constant, 2-1
- numeric field, minimum width, 3-11
- numeric output formats, 3-11
- numeric variables, 2-3

- OCT function, 5-1
- ordering BASIC, iv
- OR function, 12-5
- operator commands, 9-1, 9-2
- operators, 2-4
 - hierarchy of, 2-5
- options, start up BASIC, 8-2
- output line format, PRINT statement, 3-9
- Overlay Directories, 14-2
- overlay load maps, D-25
- overlay, subroutine, 14-1

- P command, 1-7
- pace rate, set HP 2313, 15-6
- PACER routine, 15-6
 - Instrument Table, 18-2
- parameter conversion, BASIC/other languages, 6-10
- part numbers, BASIC software, iv
- PAUSE statement, 3-17
- peripheral devices required, 1-2
- PK (pack) command, 9-6, 9-7
- PLOT routine, 17-4
- plotter
 - see HP 7210 plotter
- Plotter Library part number, iv
- PLTLU routine, 17-5
- pointer, data list, 3-14
- pointer, file data, 7-1
- position magnetic tape, 13-3
- print format
 - fixed point numbers, 3-12
 - floating point numbers, 3-12

- PRINT statement, 3-9
 - string, 4-6
- PRINT # statement
 - print a record, 7-8
 - serial file, 7-7
 - strings, 4-10
- priorities, task, 11-2, 11-3, 11-6
- program, BASIC, 1-6
- program debugging aids, 1-2
- program delay, 3-18
- program filename, BASIC, 8-2
- program name, BASIC, 8-2
- prompt character, BASIC, 1-5
- PURGE command, 9-7

- Q^c, 3-15, 3-17
 - break program execution, 9-11
- question mark prompt, 3-15
- quotation marks in string, 4-1

- R^c, 1-7
- RDBIT routine, 16-2
- RDWRD (read channel) routine, 16-3
- read analog input
 - randomly, 15-2
 - sequentially, 15-3
- read channel, HP 6940, 16-3
- read magnetic tape record, 13-2
- READ statement (data list), 3-13
 - strings, 4-7
- READ # statement
 - read a record, 7-6
 - restore data pointer, 7-5
 - serial file read, 7-5
 - strings, 4-10
- real-time and event task scheduling, 1-1
- real-time, definition of, 11-1
- Real-Time Executive II (or III), 1-3
- real-time task scheduling, 11-1
- real type data format, 6-10
- record, file, 7-1
 - magnetic tape, 13-1
- register shift function, 12-6
- relational operators, 2-4, 3-1
- relay contact, HP 6940, 16-2
- relocatable program (overlay directory), 14-2
- remarks, insertion of, 3-4
- REM statement, 3-4
- RENAME command, 9-8
- REPLACE command, 9-5
- replacing a subroutine, 14-7
- reposition file data pointer, 7-5
- request TIME, real-time clock, 11-8
- reschedule task, 11-2
- RES command, 1-5
- RESEQ command, 9-8
- Resident Library, BASIC, D-1
- response time, tasks, 11-3
- restore file data pointer, 7-5
- RESTORE statement, 3-13
- RESUME command, 10-3
- RETURN key, 1-5

- RETURN statement, 6-1
- return variables, ASSIGN statement, 7-4
- REWIND command, 13-1
- RGAIN routine, 15-7
- RND function, 5-1
- row number, 2-3
- RTE memory layout with BASIC, 1-4
- RTETG (Table Generator program), 14-2
 - commands, 14-3, 14-4
 - error messages, 14-6
 - loading, D-2
 - load map, D-23
 - output files, 14-4
 - scheduling, 14-2
- RTE-II system generation sample, D-7
- RUBOUT key, 1-8
- RUN command, 9-9, 10-2, 3-14
- running the transfer file, 14-6

- sample and hold amplifier, 15-10
- sample BASIC system generation, D-4
- sample program, magnetic tape, 13-5
- sample program, plotter, 17-9
- SAVE command, 7-2, 9-3
- SCALE routine, 17-5
- schedule disabled task, 11-5
- schedule task after specified delay, 11-7
- Scheduler, BASIC, 11-3
- scheduling BASIC, 8-1
- SCHED-2, -3, -4, -5, -6 error, 11-7
 - SCHED-3, 11-9
- security code, file, 7-1
- semi-compiled program, 9-3
- SENSE routine, HP 6940, 16-4
 - used with trap, 11-9
- serial file read statement, 7-5
- serial file print statement, 7-7
- serial storage devices, 7-1
- SERR function, 5-2, 6-10
- SET command, 10-6
- SETP routine, 11-6
- set
 - gain, HP 2313/91000, 15-11
 - to octal function, 12-6
 - task priority, 11-6
 - variable to constant, 10-6
- SFACT routine, 17-6
- SGAIN routine, 15-8
- SGN function, 5-2
- SHOW command, 10-5
- SIM command (simulate subroutine call), 10-5
- SIN function, 5-2
- SIO drivers, 18-1
- size of array, declare, 3-16
- SKIPF command, 13-1
- slash (/) control character, 1-7
- software (BASIC), loading, D-1
- source string, 4-5
- spacial purpose characters, 4-5
- specify break points, 10-3
- SQR function, 5-2
- standard devices, 1-8

- starting up BASIC Interpreter, 8-1
- START routine, 11-7
 - example, 11-1
- statements, 1-5, 3-1
 - numbers, 1-5
 - summary, A-1
- states, task, 11-3
- stop program execution, 3-17
- STOP statement, 3-5
- strings, 4-1
 - and substrings, 4-3
 - array, 2-3
 - assignment, 4-5
 - constant, 3-10
 - data format, 6-10
 - data item length, 1-2
 - DIM statement, 4-4
 - IF statement, 4-8
 - in DATA statement, 4-9
 - INPUT, 4-5
 - length, 4-9
 - PRINT, 4-6
 - PRINT #, 4-10
 - READ, 4-7
 - READ #, 4-10
 - variable, 4-2
- subroutine, 6-1
 - call simulation, 10-5
 - FORTTRAN, prepare and use with BASIC, 6-8
 - HP 2313 subsystem, 15-1
 - Library, BASIC, D-1
 - magnetic tape, 13-2
 - names, list of external, 9-12
 - replacing, 14-7
 - RTETG commands for library, 14-5
 - summary, A-5
 - table generator, 14-1
- subscripts, 2-3, 3-3
 - string, 4-2
- substring, 4-2
- subsystem configuration, HP 6940, 16-8
- summary of steps to generate BASIC system, D-3
- SUSPEND command, RTE, 11-3
- suspended program, terminate, 10-4
- suspension message, 10-5
- SWR function, 5-2
- symbols, plotter (ASCII reference no.), 17-8
- SYMB routine, 17-7
- syntax conventions, manual, 1-9
- system defined functions, 5-1
- system generation, D-1
 - RTE-II sample, D-7
- system input/output drivers, 18-1
- system pacer, HP 2313, 15-10

- TAB function, 3-13, 5-2
- Table Generator, instrument, 18-1
- Table Generator program, 14-1
 - see RTETG
- TABLES command (relation to CALL statement), 6-6
- TAN function, 5-2

task

- associate trap no., 11-9
- definition, 11-1
- disable specified, 11-5
- execute at specified time, 11-11
- priority, 11-2
- schedule after delay, 11-7
- schedule disabled, 11-5
- scheduling program, example, 11-13
- set priority, 11-6
- Scheduler vs. Interpreter, 11-3
- states, 11-3, 11-4
- teleprinter, allow auxiliary interrupt, 11-12
- terminals, enable multiple, D-3
- terminal input, 3-14
- terminate
 - BASIC Interpreter, 9-10
 - debugging activity, 10-3
 - program, 3-15
 - subroutine simulation, 10-5
 - suspended program, 10-4
- time delay, WAIT, 3-18
- TIME routine, 11-8
- TIM function, 5-2
- TM command, RTE, 11-1
- TRACE command, 10-2
- transfer file, File Manager, 14-2
 - running, 14-6
- TRAP
 - module, system generation, D-13
 - statement, 11-9, D-1
 - Table module, 1-3
- TRAP-1 error, 11-10
- TR command (transfer), File Manager, 14-6
- TRNON routine, 11-1
- TTY routine, 11-12
 - used with trap, 11-9
- two question marks, 3-15
- TYP function, 5-2, 7-9
- types, file, 7-2
 - type 0, FILES statement, 7-6
- type, item, 3-13
- typical system, 1-3

unary operators, 2-4

- UNBREAK command, 10-3
- UNLOCK command, 9-10
- UNSIM command, 10-5
- UNTRACE command, 10-2
- URITE routine, 17-8
- user-defined functions, 5-2
- user-written subroutines, 1-3
 - using BASIC, 8-2

value, function, 5-1

- variables, 2-2
- voltage digital to analog converter card, 18-3

WAIT statement, 3-18

- WEOF command, 13-1
- WHERE routine, 17-8
- words required to store string on file, 4-10
- work format, bit manipulation, 12-1
- WRBIT routine, 16-5
- write bit on HP 6940 channel, 16-5
- write EOF to tape, 13-4
- write record, magnetic tape, 13-2
- write word on HP 6940 channel, 16-6
- WRWRD routine, 16-6
- W3, jumper, 16-4

X command, editing, 1-7

%BXnn, overlays, 14-4

- 16-bit word format, 12-1
- 2100 and 21MX computer, 1-2
- 2313 subsystem, 15-1
 - see HP 2313
- 6940 subsystem
 - see HP 6940 subsystem
- 7210
 - see HP 7210 Plotter
- 91000 subsystem, 15-1
 - DAS card, 15-9
 - see HP 2313/91000



Sales and service from 172 offices in 65 countries.
11000 Wolfe Road, Cupertino, California 95014

PART NO. 92060-90016

Printed in U.S.A. 10/75