BASIC/RT11
Language Reference Manual

Order No. DEC-11-LBACA-E-D

digital equipment corporation · maynard. massachusetts

First Printing, Sept. 1973
Revised, Dec. 1973
Revised, June, 1974
Revised, Oct. 1974
Revised, Oct. 1976

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1973, 1974, 1976 by Digital Equipment Corporation

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

7/77-15

CONTENTS

			Page
PREFACE			vii
CHAPTER 1		INTRODUCTION	
	1.1	LOADING AND RUNNING BASIC	1-1
CHAPTER 2		RT-11 BASIC ARITHMETIC	
	2.1	NUMBERS	2-1
	2.2	VARIABLES	2-2
	2.3	SUBSCRIPTED VARIABLES	2-2
	2.4	EXPRESSIONS	2-4
	2.5	ARITHMETIC OPERATIONS	2-4
	2.5.1	Priority of Arithmetic Operations	2-4
	2.5.2	Relational Operators	2-6
CHAPTER 3		RT-11 BASIC STRINGS	
	3.1	STRINGS	3-1
	3.2	STRING VARIABLES	3-1
	3.2.1	Subscripted String Variables	3-1
	3.3	STRING OPERATIONS	3-2
	3.3.1	Concatenation	3-2
	3.3.2	Relational Operations	3-2
CHAPTER 4		IMMEDIATE MODE OPERATIONS	
	4.1	USE OF IMMEDIATE MODE FOR STATEMENT EXECUTION	4-1
	4.2	PROGRAM DEBUGGING	4-1
	4.3	MULTIPLE STATEMENTS PER LINE	4-2
	4.4	RESTRICTIONS ON IMMEDIATE MODE	4-2
CHAPTER 5		RT-11 BASIC STATEMENTS	
	5.1	STATEMENT NUMBERS	5-1
	5.2	REMARK STATEMENT	5-1
	5.3	THE ASSIGNMENT STATEMENT - LET	5-2
	5.4	THE DIMENSION STATEMENT - DIM	5-3
	5.5	INPUT/OUTPUT STATEMENTS	5-4
	5.5.1	PRINT Statement	5-4
	5.5.1.1	Printing Variables	5-4
	5.5.1.2	Printing Strings	5-5
	5.5.1.3	Use of Comma and Semicolon ("," and ";")	5-6
	5.5.1.4	Selecting Output Device	5-7
	5.5.1.5	PRINT Statement - TAB Function	5-8
	5.5.2	INPUT Statement	5-8
	5.5.2.1	Selecting Input Devices	5-9
	5.5.3	DATA Statement	5-10
	5.5.4	READ Statement	5-10

Page

5.5.5 RESTORE Statement 5-11

5.6 RANDOMIZE Statement 5-12

5.7 PROGRAM CONTROL 5-13
5.7.1 GO TO Statement 5-13
5.7.2 IF THEN, IF GO TO and IF END Statements 5-14
5.7.3 FOR-NEXT Statements 5-15
5.7.4 GOSUB and RETURN Statements 5-18

5.8 PROGRAM TERMINATION 5-20
5.8.1 END Statement 5-20
5.8.2 STOP Statement 5-20
5.8.3 CHAIN Statement 5-20

5.9 FILE CONTROL 5-21
5.9.1 OPEN Statement 5-22
5.9.2 CLOSE Statement 5-25
5.9.3 OVERLAY Statement 5-26

CHAPTER 6 BASIC/RT-11 FUNCTIONS

6.1 ARITHMETIC FUNCTIONS 6-1
6.1.1 Sine and Cosine Functions, SIN(x) and COS(x) 6-2
6.1.2 Arctangent Function, ATN(x) 6-2
6.1.3 Square Root Function, SQR(x) 6-3
6.1.4 Exponential Function, EXP(x) 6-4
6.1.5 Logarithm Function, LOG(x) 6-4
6.1.6 Absolute Function, ABS(x) 6-6
6.1.7 Integer Function, INT(x) 6-6
6.1.8 Random Number Function, RND(x) 6-7
6.1.9 Sign Function, SGN(x) 6-8
6.1.10 Binary Function, BIN(x$) 6-9
6.1.11 Octal Function, OCT(x$) 6-9

6.2 USER DEFINED FUNCTIONS 6-10

6.3 STRING FUNCTIONS 6-15
6.3.1 User-Defined String Functions 6-16

CHAPTER 7 EDITING COMMANDS

7.1 SCRATCH COMMAND 7-2

7.2 OLD COMMAND 7-3

7.3 LIST/LISTNH COMMANDS 7-3

7.4 SAVE COMMAND 7-5

7.5 REPLACE COMMAND 7-5

7.6 RUN/RUNNH COMMANDS 7-6

7.7 CLEAR COMMAND 7-6

7.8 RENAME COMMAND 7-7

7.9 NEW COMMAND 7-7

		Page
CHAPTER 8	USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC	
8.1	CALL STATEMENT	8-1
8.2	SYSTEM FUNCTION TABLE	8-2
8.3	WRITING ASSEMBLY LANGUAGE ROUTINES	8-3
8.3.1	Sample User Functions	8-5
8.4	SYSTEM ROUTINES IN BASIC	8-7
8.5	REPRESENTATION OF NUMBERS IN BASIC	8-11
8.6	REPRESENTATION OF STRINGS IN BASIC	8-11
8.7	FORMAT OF TRANSLATED BASIC PROGRAM	8-12
8.7.1	Symbol Table Format	8-12
8.7.2	Translated Code	
8.8	BACKGROUND ASSEMBLY LANGUAGE ROUTINE	8-15
CHAPTER 9	ERROR MESSAGES	9-1
CHAPTER 10	DEMONSTRATION PROGRAMS	10-1
APPENDIX A	BOOTSTRAPPING THE RT-11 SYSTEM	A-1
APPENDIX B	ASCII CHARACTER SET	B-1
APPENDIC C	STATEMENTS, COMMANDS, FUNCTIONS	C-1
C.1	RT-11 BASIC STATEMENTS	C-1
C.2	COMMANDS	C-3
C.3	FUNCTIONS	C-4
APPENDIX D	GETARG, STORE, SSTORE LISTING	D-1
APPENDIX E	BASIC ERROR MESSAGES	E-1
APPENDIX F	ASSEMBLING AND LINKING BASIC	F-1
F.1	ASSEMBLING BASIC/RT11	F-1
F.1.1	Floating Point Math Package	F-2
F.2	LINKING BASIC/RT11	F-3
F.2.1	Linking BASIC/RT11 With User Functions	F-4
APPENDIX G	BASIC CORE MAP	G-1
INDEX		Index-1

PREFACE

This document describes the operating procedures for the BASIC/RT11 program and the features of the BASIC/RT11 language.

The user should be somewhat familiar with the standard BASIC language. If the user is totally unfamiliar with BASIC it is suggested that a BASIC primer be read prior to using this document. The BASIC language as it pertains to BASIC/RT11 is described in Chapters 5 and 6. Chapters 1, 2, 3 and 4 provide an introduction to BASIC/RT11 operating procedures, arithmetic and string operations. Editing commands, error messages and demonstration programs are covered in Chapters 7, 9 and 10.

The experienced BASIC user should pay particular attention to the description of operating procedures (Chapter 1) and the use of assembly language routines (Chapter 8) and the summary of statements, commands and functions (Appendix D).

NEW AND CHANGED INFORMATION

This revision of the manual incorporates the update DEC-11-LBACA-D-DN3, which removes the description of the laboratory and graphics extensions and corrects technical errors.

A description of the laboratory and graphics extensions which are available for use with BASIC/RT-11 can be found in the following manuals:

- BASIC-11 Lab Extensions User's Guide (DEC-11-LBEPA-A-D)
- BASIC-11 Graphics Extensions User's Guide (DEC-11-LBGEA-A-D)

CHAPTER 1

INTRODUCTION

BASIC/RT11 is a single-user, conversational programming language which uses simple English-type statements and familiar mathematical notations to perform an operation. BASIC is one of the simplest computer languages to learn and once learned has the facility of advanced techniques to perform more intricate manipulations or express a problem more efficiently.

BASIC/RT11 interfaces with the RT-11 Monitor to provide powerful sequential and random-access file capabilities and allows the user to save and retrieve programs from peripheral devices. BASIC/RT11 has provision for alphanumeric character string I/O and string variables (12K or larger systems) and allows user defined functions and assembly language subroutine calls from user BASIC programs.

1.1 LOADING AND RUNNING BASIC

BASIC is loaded under the control of the RT-11 monitor (Refer to the RT-11 System Reference Manual (DEC-11-ORUGA-A-D) for additional information on the RT-11 system), by typing:

```
R BASIC
```

and the RETURN key.

Through replies to the initial dialogue, BASIC allows selection of the functions to be loaded. Selectively loading functions maximizes space available for the user's program by removing unwanted functions from core.

When BASIC is first loaded with the R command, the dialogue described below is printed. This is once-only dialogue and does not occur again.

BASIC prints:

```
BASIC V01-05   (or current version)
*
```

and awaits specification on inclusion of the optional functions shown below. Refer to Chapter 6 for information on these functions. Depending on the response (carriage return, A, N or I) made to this message, all functions (carriage return or A) are included, none of the functions (N) are included or the functions are listed and may be individually selected for inclusion (I).

Selectively excluding functions can provide space for up to 20 or 30 additional user program lines.

Reply with one of the following codes:

Code	Explanation
A or carriage return	Loads all of the optional functions
N	Loads none of the optional functions
I	Allows the functions to be specified individually

If any character other than a carriage return, A, N, or I is typed, the message is repeated. If the reply is I, BASIC prints

```
Y-YES  N-NO

RND:
```

to allow specification of each function to be loaded as part of BASIC/RT11.

Reply with a Y or N for the RND function and each additional function as the names are printed. The optional functions are:

String BASIC	No String
RND	RND
ABS	ABS
SGN	SGN
BIN	BIN
OCT	OCT
TAB	
LEN	
ASC	
CHR$	
POS	
SEG$	
VAL	
TRM$	
STR$	

Each exclusion of a function provides room for between two and five additional program lines. Excluding the POS and SEG$ functions provides approximately ten additional lines each.

If a "user function" has been linked (Refer to Appendix F) into BASIC (to be referenced by a CALL statement) BASIC prints:

```
USER FNS LOADED
```

BASIC then prints the message

```
READY
```

and waits for a command or program line to be typed (refer to Chapter 4).

Typing CTRL/C at any time returns BASIC to the RT-11 Monitor. To continue BASIC after a CTRL/C return to the monitor, type the Monitor command REENTER (RE). BASIC will then print the READY message.

The program in core when the CTRL/C was executed is retained. Thus, user program execution may be terminated at any time without destroying the user program.

CHAPTER 2

RT-11 BASIC ARITHMETIC

2.1 NUMBERS

BASIC treats all numbers (real and integer) as decimal numbers--that is, it accepts any decimal number, and assumes a decimal point after an integer. The advantage of treating all numbers as decimal numbers is that any number or symbol can be used in any mathematical expression without regard to its type. Numbers used must be in the approximate range $10^{-38}<N<10^{+38}$.

In addition to integer and real formats, a third format is recognized and accepted by BASIC. This format is called exponential or E-type notation, and in this format, a number is expressed as a decimal number times some power of 10. The form is:

```
xxEn
```

where E represents "times 10 to the power of"; thus the number is read: "xx times 10 to the power of n". For example:

$$23.4E2 = 23.4*10^2 = 2340$$

Data may be input in any one or all three of these forms. Results of computations are output as decimals if they are within the range .01<n<999999; otherwise, they are output in E format. Numbers are stored up to 24 bits of significance. If a number with more than 24 bits is entered, it is rounded and stored as 24 bits. BASIC handles six significant digits in normal operation and prints 6 decimal digits as illustrated below:

Value Typed In	Value Output By BASIC
.01	.01
.0099	9.90000E-03
999999	999999
1000000	1.00000E+06

BASIC automatically suppresses the printing of leading and trailing zeros in integer and decimal numbers, and, as can be seen from the preceding examples, formats all exponential numbers in the form:

```
(sign) x.xxxxxE(+ or -)n
```

where x represents the number carried to six decimal places, E stands for "times 10 to the power of", and n represents the exponential value. For example:

```
-3.47021E+08 is equal to -347,021,000
 7.26000E-04 is equal to .000726
```

Floating point format is used when storing and calculating most numbers.

However, if the number entered is an integer, it is handled as an integer unless the operation being performed requires that it be

changed to floating point. Multiply and divide operations require this transformation but addition and subtraction of integer quantities less than 2^{15} in magnitude is done with the corresponding single machine instruction. Thus, maintaining numbers in (or converting numbers to) integer form may significantly increase the speed of arithmetic expression evaluation.

NOTE

> Because core size limitations prohibit the storage of infinite binary numbers, some numbers cannot be expressed exactly in BASIC/RT. Accuracy is approximately 5-1/2 digits, and errors in the 6th digit can occur. For example, .999998 as a result of some functions may be equal to 1. Discrepancies of this type are magnified when such a number is used in mathematical operations.

2.2 VARIABLES

A variable in BASIC is an algebraic symbol representing a number, and is formed by a single letter or a letter optionally followed by a single digit. For example:

Acceptable Variables	Unacceptable Variables
I	2C - a digit cannot begin a variable.
B3	AB - two or more letters cannot form a variable.
X	11 - numbers alone cannot form a variable.

Subscripted and string variables are described in later sections. The user may assign values to variables either by indicating the values in a LET statement, or by inputting the values as data in an INPUT statement or by a READ statement; these operations are discussed in Chapter 5.

The value assigned to a variable does not change until the next time a statement is encountered that contains a new value for that variable. All variables are set equal to zero (0) before program execution. It is only necessary to assign a value to a variable when an initial value other than zero is required. However, good programming practice would be to set variables equal to 0 wherever necessary. This ensures that later changes or additions will not misinterpret values.

2.3 SUBSCRIPTED VARIABLES

In addition to the simple variables described in section 2.2, BASIC allows the use of subscripted variables. Subscripted variables provide additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. In BASIC, variables are allowed one or two subscripts.

The name of a subscripted variable is any acceptable BASIC variable name followed by one or two integer expressions (within the range 0-32767) in parentheses. For example, a list might be described as A(I) where I goes from 0 to 5 as shown below:

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

This allows reference to each of the six elements in the list, and can be considered a one-dimensional algebraic matrix as follows:

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

A two-dimensional matrix B(I,J) can be defined in a similar manner:

B(0,0),B(0,1),B(0,2),...,B(0,J),...,B(I,J)

and graphically illustrated as follows:

B(0,0)	B(0,1)	B(0,2)	B(0,3)	B(0,J)
B(1,0)	B(1,1)	B(1,2)	B(1,3)	B(1,J)
B(2,0)	B(2,1)	B(2,2)	B(2,3)	B(2,J)
B(3,0)	B(3,1)	B(3,2)	B(3,3)	B(3,J)
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
B(I,0)	B(I,1)	B(I,2)	B(I,3)	B(I,J)

Subscripts used with subscripted variables throughout a program can be explicitly stated or be any legal expression. If the value of the expression is non-integer, the value is truncated so that only the subscript is an integer.

It is possible to use the same variable name as both a subscripted and unsubscripted variable. Both A and A(I) are valid variables and can be used in the same program. The variable A has no relationship to any element of the matrix A(I). However, BASIC will not accept the same variable name as both a singly and a doubly subscripted variable name in the same program.

Use of subscripted variables requires a dimension (DIM) statement to define the maximum number of elements in a matrix. ("Matrix" is the general term used in this manual to describe all elements of a

subscripted variable.) The DIM statement is discussed in paragraph 5.4.

If a subscripted variable is used without appearing in a DIM statement, it is assumed to be dimensioned to length 10 in each dimension (that is, having eleven elements in each dimension, 0 through 10). However, all matrices should be correctly dimensioned in a program.

2.4 EXPRESSIONS

An expression is a group of symbols which can be evaluated by BASIC. Expressions are composed of numbers, variables, functions, or a combination of the preceding separated by arithmetic or relational operators.

The following are examples of expressions acceptable to BASIC:

Arithmetic Expressions

```
4
A7*(B↑2+1)
```

Not all kinds of expressions can be used in all statements, as is explained in the sections describing the individual statements.

2.5 ARITHMETIC OPERATIONS

BASIC performs addition, subtraction, multiplication, division and exponentiation. Formulas to be evaluated are represented in a format similar to standard mathmetical notation. The five operators used in writing most formulas are:

Symbol Operator	Example	Meaning
+	A + B	Add B to A
-	A - B	Subtract B from A
*	A * B	Multiply A by B
/	A / B	Divide A by B
↑	A ↑ B	Exponentiation (Raise A to the Bth power)

Unary plus and minus are also allowed, e.g., the - in the -A+B or the + in +X-Y. Unary plus is ignored. Unary minus is treated as explained below.

2.5.1 Priority of Arithmetic Operations

When more than one operation is to be performed in a single formula, as is most often the case, rules are observed as to the precedence of the operators.

In any given mathematical formula, BASIC performs the arithmetic operations in the following order of evaluation:

1. Parentheses receive top priority. Any expression within parentheses is evaluated before an unparenthesized expression.

2. In the absence of parentheses, the order of priority is:

   a. Unary minus

   b. Exponentiation (proceeds from left to right).

   c. Multiplication and Division (of equal priority).

   d. Addition and Subtraction (of equal priority).

3. If either 1 or 2 above does not clearly designate the order of priority, then the evaluation of expressions proceeds from left to right.

The expression A↑B↑C is evaluated from left to right as follows:

1. A↑B = step 1

2. (result of step 1)↑C = answer

The expression A/B*C is also evaluated from left to right since multiplication and division are of equal priority:

1. A/B = step 1

2. (result of step 1)*C = answer

The expression A+B*C↑D is evaluated as:

1. C↑D = step 1

2. (result of step 1)*B = step 2

3. (result of step 2)+A = answer

Parentheses may be nested, or enclosed by a second set (or more) of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated.

In the following example:

```
A=7*((B↑2+4)/X)
```

The order of priority is:

1. B↑2 = step 1

2. (result of step 1)+4 = step 2

3. (result of step 2)/X = step 3

4. (result of step 3)*7 = A

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example:

```
A*B↑2/7+B/C*D↑2
((A*B↑2)/7)+((B/C)*D↑2)
```

Both of these formulas are executed in the same way, but the second is easier to understand.

Spaces may be used in a similar manner. Since the BASIC interpreter ignores spaces (except when enclosed in quotation marks), the two statements:

```
10 LET B = D↑2 + 1
10LETB=D↑2+1
```

are identical, but spaces in the first statement provide ease in reading. When the statement is subsequently listed, extra spaces are ignored.

2.5.2 Relational Operators

Relational operators allow comparison of two values and are used to compare arithmetic expressions or strings in an IF...THEN statement. The relational operators are:

Mathematical Symbol	BASIC Symbol	Example	Meaning
=	=	A=B	A is equal to B.
<	<	A<B	A is less than B.
≤	<= or =<	A<=B	A is less than or equal to B.
>	>	A>B	A is greater than B.
≥	>= or =>	A>=B	A is greater than or equal to B.
≠	< > or> <	A><B	A is not equal to B.

The symbols =<, =>, >< are accepted by BASIC but are converted to <=, >= and <> and are shown in that form in a listing.

CHAPTER 3

RT-11 BASIC STRINGS

3.1 STRINGS

The previous chapters describe the manipulation of numerical information only; however, BASIC also processes information in the form of character strings. A string, in this context, is a sequence of characters treated as a unit. A string can be composed of alphabetic, numeric, or special characters. (A character string may contain letters, numbers, spaces, or any combination of characters.) A character string can be 255 characters long. However, the LINE FEED key cannot be used to type a string on two or more terminal lines.

3.2 STRING VARIABLES

Any variable name followed by a dollar sign ($) character indicates a string variable. For example:

```
A$
C7$
```

are simple string variables and can be used, for example, as follows:

```
LET A$="HELLO"
PRINT A$
```

Note that the string variable A$ is separate and distinct from the variable A.

3.2.1 Subscripted String Variables

Any list or matrix variable name followed by the $ character denotes the string form of that variable. For example:

```
V$(n)          M2$(n)
C$(m,n)        G1$(m,n)
```

where m and n indicate the position of the matrix element within the whole.

The same name can be used as a numeric variable and as a string variable in the same program with the restriction that a one-dimensional and a two-dimensional matrix cannot have the same name in the same program. For example:

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

can all be used in the same program, but

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

cannot.

String lists and matrices are defined with the DIM statement (paragraph 5.4), as are numerical lists and matrices.

In BASIC without strings, string variables are illegal.

3.3 STRING OPERATIONS

3.3.1 Concatenation

Concatenation puts one string after another without any intervening characters. It is specified by an ampersand (&) and works only with strings. The maximum length of a concatenated string is 255 characters.

For example:

```
10 READ A$, B$, C$
20 DATA "11","33","22"
30 PRINT A$&C$&B$
40 END
RUNNH
112233
```

3.3.2 Relational Operations

When applied to string operands, the relational operators indicate alphabetic sequence. The comparison is done, character by character, left to right, on the ASCII value. For example:

```
55 IF A$<B$ THEN 100
```

When line 55 is executed, the first characters of each string (A$ and B$) are compared; if they are the same, then the second characters of each string are compared and so on until the characters differ. If the character in A$ is less than the character in B$ then execution continues at line 100. Otherwise, execution continues at the next statement in sequence. Essentially the strings are compared for alphabetic order. Table 3-1 contains a list of the relational operators and their string interpretations.

In any string comparison, trailing blanks are ignored (i.e., "ABC" is equivalent to "ABC ").

Table 3-1

Relational Operators Used With
String Variables

Operator	Example	Meaning
=	A$ = B$	The strings A$ and B$ are alphabetically equal.
<	A$ < B$	The string A$ alphabetically precedes B$.
>	A$>B$	The string A$ alphabetically follows B$.
<= or =	A$ < = B$	The string A$ is equivalent to or precedes B$ in alphabetical sequence.
>= or =>	A$ > = B$	The string A$ is equivalent to or follows B$ in alphabetical sequence.
<> or ><	A$ <> B$	The strings A$ and B$ are not alphabetically equal.

CHAPTER 4

IMMEDIATE MODE OPERATIONS

4.1 USE OF IMMEDIATE MODE FOR STATEMENT EXECUTION

It is not necessary to write a complete program to use BASIC. Most of the statements discussed in this manual can be included in a program for later execution or given on-line as commands, which are immediately executed by the BASIC processor. This latter facility makes BASIC an extremely powerful calculator.

BASIC distinguishes between lines entered for later execution and those entered for immediate execution solely on the presence (or absence) of a line number. Statements which begin with line numbers are stored; statements without line numbers are executed immediately upon being entered to the system. Thus the line:

```
10 PRINT "THIS IS A PDP-11"
```

produces no action at the console upon entry, while the statement:

```
PRINT "THIS IS A PDP-11"
```

causes the immediate output:

```
THIS IS A PDP-11
```

4.2 PROGRAM DEBUGGING

Immediate mode operation is especially useful in two areas: program debugging and the performance of simple calculations in situations which do not occur with sufficient frequency or with sufficient complications to justify writing a program.

In order to facilitate debugging a program, STOP statements can be liberally placed throughout the program. Each STOP statement causes the program to halt, at which time the various data values can be examined and perhaps changed in immediate mode. The

```
GO TO xxxxx
```

command is used to continue program execution (where xxxxx is the number of the next program line to be executed). The values assigned to variables when the RUN command was executed remain intact until a Scratch, Clear, or another RUN Command is executed.

When using immediate mode, nearly all the standard statements can be used to generate or print results. If the STOP occurs in the middle of a FOR loop, modifications cannot be made to the section of the program which precedes the FOR.

If CTRL/C is used to halt program execution, the GO TO command can be used to continue execution but since CTRL/C does not print the number of the line where execution stopped, it is difficult to know where to resume the program.

4.3 MULTIPLE STATEMENTS PER LINE

Multiple statements can be used on a single line in immediate mode. For example:

```
A=1 \PRINT A
1
```

On a LT33 or LT35 terminal, type a SHIFT/L to produce the backslash character.

Program loops are allowed in immediate mode; thus a table of square roots can be produced as follows:

```
FOR I=1 TO 10\PRINT I,SQR(I)\NEXT I
 1            1
 2            1.41421
 3            1.73205
 4            2
 5            2.23607
 6            2.44949
 7            2.64575
 8            2.82843
 9            3
 10           3.16228
```

4.4 RESTRICTIONS ON IMMEDIATE MODE

Certain commands, while not illegal, make no logical sense when used in immediate mode. Commands in this category are DEF, DIM, DATA and RANDOMIZE.

The INPUT statement is illegal in immediate mode and its use results in the ?ILN error message.

Also, since user functions are not defined until the program is executed, function references in immediate mode cause an error unless the program containing the definition was previously executed.

Thus the following dialogue might result if a function was defined in a user program and then referenced in immediate mode.

```
10 DEF FNA(X) = X↑2 + 2*X\REM SAVED STATEMENT
PRINT FNA(1)\REM IMMEDIATE MODE

?UFN

READY
```

but if the sequence of statements is:

```
RUNNH
READY
```

```
PRINT FNA(1)
3
```

the immediate mode statement is executed.

If output files are opened in immediate mode, a CLOSE command must be issued or the last block of data may not be written.

Note that virtual files can be edited by selectively modifying values in immediate mode. For example,

```
OPEN "FILE" AS FILE VF1(1000)
VF1(137)=12.6
PRINT VF1(212)
13.1
CLOSE
```

CHAPTER 5

RT-11 BASIC STATEMENTS

A user program is composed of lines of statements containing instructions to BASIC. Each line of the program begins with a line number that identifies that line as a statement and indicates the order of statement execution. Each statement starts with an English word specifying the type of operation to be performed. Statement lines are terminated with the RETURN key which is non-printing.

5.1 STATEMENT NUMBERS

A 1-5 digit statement number is placed at the beginning of each line in a BASIC program. BASIC executes the statements in a program in numerically consecutive order regardless of the order in which they were typed. Statement numbers must be within the range 1 to 65532. When first writing a program, it is advisable to number lines in increments of five or ten to allow insertion of forgotten or additional lines when debugging the program. If there are no available lines for insertion of statements, the user program can be resequenced. (Refer to Chapter 10, program #4 for a resequence example.)

All BASIC statements and computations must be written on a single line; they cannot be continued onto a following line. However, more than one statement may be written on a single line when each statement after the first is preceded by a backslash. For example:

```
10 INPUT A,B,C
```

is a single statement line, whereas

```
20 LET X=11 \PRINT X,Y,Z\ IF X=A THEN 10
```

is a multiple statement line containing three statements: LET, PRINT, and IF. Most statements may be used anywhere in a multiple statement line; exceptions are noted in the discussion of each statement. Only the first statement on a line can (and must) have a line number. It should be remembered that program control cannot be transferred to a statement within a line, but only to the first statement of a line.

Typing a statement number with no statement after it causes the previous statement with the same number to be deleted.

5.2 REMARK STATEMENT

It is often desirable to insert notes and messages within a user program. Such data as the name and purpose of the program, how to use it, how certain parts of the program work, and expected results at various points are useful things to have present in the program for ready reference by anyone using that program.

The REMARK or REM statement is used to insert remarks or comments into a program without these comments affecting execution. Remarks do, however, use core area which may be needed by an exceptionally long program.

The REMARK statement must be preceded by a line number except when the REMARK statement is used in a multiple statement line, where it can only be the last statement. The message itself can contain any printing character on the keyboard. BASIC completely ignores anything on a line following the letters REM. (The line number of a REM statement can be used in a GO TO or GOSUB statement, see sections 5.7.1 and 5.7.4, as the destination of a jump in the program execution.) Typical REM statements are shown below:

```
10 REM- THIS PROGRAM COMPUTES THE
11 REM- ROOTS OF A QUADRATIC EQUATION
```

5.3 THE ASSIGNMENT STATEMENT - LET

The LET statement assigns the value of the expression to the specified variable. The general format of the LET statement is:

```
LET variable = expression
```

where variable is a numeric or string variable and expression is an arithmetic or string expression. All items in the statement must be either string or numeric; they cannot be mixed. The word LET is optional.

The LET statement does not indicate algebraic equality, but performs calculations within the expression (if any) and assigns the value to the variable.

The meaning of the equal (=) sign should be clarified. In algebraic notation, the formula X=X+1 is meaningless. However, in BASIC (and most computer languages), the equal sign designates replacement rather than equality. Thus, this formula is actually translated: "add one to the current value of X and store the new result back in the same variable X". Whatever value has previously been assigned to X will be combined with the value 1. An expression such as A=B+C instructs the computer to add the values of B and C and store the result in a third variable A. The variable A is not being evaluated in terms of any previously assigned value, but only in terms of B and C. Therefore, if A has been assigned any value prior to its use in this statement, the old value is lost; it is instead replaced by the value B+C.

The LET statement can also be used to set a value in a virtual memory file element as follows:

```
LET VFn(i)=expression
```

Examples:

LET X=2	Assigns the value 2 to the variable X.
LET X=X+1+Y	Adds 1 to the current value of X then adds the value of Y to that result and assigns that value to X.

```
LET B$="STRING"
```

Assigns the characters "STRING" to the string variable B$.

5.4 THE DIMENSION STATEMENT - DIM

The DIMension statement reserves space for lists and tables used by the program. The DIM statement is of the form:

```
DIM variable(n), variable(n,m), variable$(n), variable$(n,m)
```

where variables specified are indicated with their maximum subscript value(n) or values(n,m).

For example:

```
10 DIM X(5), Y(4,2), A(10,10)
12 DIM I4(100), A$(25)
```

Only integer constants (such as 5 or 5070) can be used in DIM statements to define the size of a matrix. Variables cannot be used to specify the bounds of arrays. Any number of matrices can be defined in a single DIM statement as long as their representations are separated by commas.

The first element of every matrix is automatically assumed to have a subscript of zero. Dimensioning A(6,10) sets up room for a matrix with 7 rows and 11 columns. This zero element is illustrated in the following program:

```
10 REM - MATRIX CHECK PROGRAM
20 DIM A(6,10)
30 FOR I=0 TO 6
40 LET A(I,0) = I
50 FOR J=0 TO 10
60 LET A(0,J) = J
70 PRINT A(I,J);
80 NEXT J \ PRINT\NEXT I
90 END

RUNNH
 0  1  2  3  4  5  6  7  8  9  10
 1  0  0  0  0  0  0  0  0  0  0
 2  0  0  0  0  0  0  0  0  0  0
 3  0  0  0  0  0  0  0  0  0  0
 4  0  0  0  0  0  0  0  0  0  0
 5  0  0  0  0  0  0  0  0  0  0
 6  0  0  0  0  0  0  0  0  0  0

READY
```

Notice that a variable has a value of zero until it is assigned another value.

Whenever an array is dimensioned (m,n), the matrix is allocated m+1 by n+1 elements. Core space can be conserved by using the 0th element of the matrix.

For example, DIM A(5,9) dimensions a 6 x 10 matrix which would then be referenced beginning with the A(0,0) element.

The size and number of matrices which can be defined depend upon the amount of storage space available.

A DIM statement can be placed anywhere in a multiple statement line and can appear anywhere in the program. A matrix can only be dimensioned once. DIM statements need not appear prior to the first reference to an array, although DIM statements are generally among the first statements of a program to allow them to be easily found if any alterations are later required.

All arrays specified in DIM statements are allocated space when the RUN command is executed.

5.5 INPUT/OUTPUT STATEMENTS

Input/Output (I/O) statements, such as PRINT, INPUT, and READ, bring data into and output results or data from a program during execution.

5.5.1 PRINT Statement

The PRINT statement is used to output data to the terminal. The general format of the PRINT statement is:

```
PRINT list
```

The list is optional and can contain expressions, text strings, or both. Elements of the list must be separated by appropriate delimiters (space, comma, semicolon).

When used without the list, the PRINT statement:

```
25 PRINT
```

causes a blank line to be output on the terminal (a carriage return/line feed operation is performed).

5.5.1.1 Printing Variables

PRINT statements can be used to perform calculations and print results. Any expression within the list is evaluated before a value is printed. For example,

```
10 LET A=1\ LET B=2\ LET C=3+A
20 PRINT
30 PRINT A+B+C
RUNNH
 7

READY
```

All numbers are printed with a preceding sign (minus for negative and space for positive) and a following blank space.

The PRINT statement can be used anywhere in a multiple statement line. For example:

```
10 A=1\ PRINT A\ A=A+5\ PRINT\ PRINT A
```

prints the following on the terminal when executed:

```
 1

 6

READY
```

Notice that the terminal performs a carriage return/line feed at the end of each PRINT statement. Thus the first PRINT statement outputs a 1 and a carriage return/line feed; the second PRINT statement, the blank line; and the third PRINT statement, a 6 and another carriage return/line feed.

5.5.1.2 Printing Strings

The PRINT statement can be used to print a message or string of characters, either alone or together with the evaluation and printing of numeric values. Characters are indicated for printing by enclosing them in single or double quotation marks (therefore each type of quotation mark can only be printed if surrounded by the other type of quotation mark). For example:

```
10 PRINT "TIME'S UP"
20 PRINT '"NEVERMORE"'
RUNNH
TIME'S UP
"NEVERMORE"

READY
```

As another example, consider the following line:

```
40 PRINT "AVERAGE GRADE IS";X
```

which prints the following (where X is equal to 83.4):

```
AVERAGE GRADE IS 83.4
```

When a character string is printed, only the characters between the quotes appear; no leading or trailing spaces are added. Leading and trailing spaces can be added within the quotation marks using the keyboard space bar; spaces appear in the printout exactly as they are typed within the quotation marks.

When a comma separates a text string from another PRINT list item, the item is printed at the beginning of the next available print zone (refer to paragraph 5.5.1.3). Semicolons separating text strings from other items are ignored. Thus, the previous example could be expressed as:

```
40 PRINT "AVERAGE GRADE IS" X
```

and the same printout would result. A comma or semicolon appearing as the last item of a PRINT list always suppresses the carriage return/line feed operation.

BASIC does an automatic carriage return/line feed if a string is printing past column 72.

Although string variables are illegal in the BASIC without strings, literal strings may be used in a PRINT statement.

5.5.1.3 Use of Comma and Semicolon ("," and ";")

BASIC considers the terminal printer to be divided into five zones of fourteen columns each. When an item in a PRINT statement is followed by a comma, the next value to be printed appears in the next available print zone. For example:

```
10 LET A=3\ LET B=2
20 PRINT A,B,A+B,A*B,A-B,B-A
```

When the preceding lines are executed, the following is printed:

```
 3             2             5             6             1
-1
```

Notice that the sixth element in the PRINT list is printed as the first entry on a new line, since the five print zones of the 72-character line were already used.

Two commas together in a PRINT statement cause a print zone to be skipped. For example:

```
10 LET A=1\ LET B=2
20 PRINT A,B,,A+B
RUNNH
 1             2                           3

READY
```

If the last item in a PRINT statement is followed by a comma, no carriage return/line feed is output, and the next value to be printed (by a later PRINT statement) appears in the next available print zone. For example:

```
10 A=1\B=2\C=3
20 PRINT A,\PRINT B\ PRINT C
RUNNH
 1             2
 3
```

```
READY
```

If a tighter packing of printed values is desired, the semicolon character can be used in place of the comma. A semicolon causes no further spaces to be output. A comma causes the print head to move at least one space to the next print zone or possibly perform a carriage return/line feed. The following example shows the effects of the semicolon and comma.

```
10 LET A=1\ B=2\ C=3
20 PRINT A;B;C;
30 PRINT A+1;B+1;C+1
40 PRINT A,B,C
99 END
RUNNH
 1  2  3  2  3  4
 1             2              3

READY
```

The following example demonstrates the use of the formatting characters , and ; with text strings:

```
110 LET X=119050\G=87\A=85.44\N=26
120 PRINT "NO."X,"GRADE ="G;"AVE. ="A;
130 PRINT "NO. IN CLASS ="N
900 END
RUNNH
NO. 119050     GRADE = 87 AVE. = 85.44 NO. IN CLASS = 26

READY
```

5.5.1.4 Selecting Output Device

The PRINT statement can also be used to select a particular output file. The form of the statement is:

```
PRINT #expression:expression list
```

where expression has the value 0 to 7. If the value of the expression is 0, output is to the terminal; otherwise, the output is to the sequential file which was opened as logical unit (expression). (See section 5.9.1, OPEN statement.) Output is formatted exactly as if done by the PRINT statement. The colon (:) is required when variables follow the expression.

If a file written by the PRINT statement is to be later read by the INPUT statement then the necessary separating commas must be specified (within quotation marks) in a PRINT statement with more than one item in the PRINT list.

Examples:

```
10 OPEN "LP:" FOR OUTPUT AS FILE #2
20 OPEN "DT0: DATA" FOR OUTPUT AS FILE #7
30 PRINT #0: "OUTPUT TO TERMINAL"
40 PRINT #2: "OUTPUT TO LINE PRINTER"
50 PRINT #7: 10,",",20,",",30
```

NOTE

If the line printer is not on line when a BASIC program is attempting to output to it, BASIC will wait for the line printer to be put on line and will then start or continue its output.

5.5.1.5 PRINT Statement - TAB Function

The TAB function is used in a PRINT statement to space to the specified column on the output device. The columns on the output devices are numbered 0 to 71.

The form of the command is:

```
PRINT TAB(x);
```

where (x) is the column number in the range 0-255. If x exceeds 71, however, consecutive subtractions of 72 are done until the number of spaces to be output is less than or equal to 71. If the column number specified is greater than 255 or negative, the error message ?ARG is printed. If (x) is non-integer, only the integer portion of the number is used.

If the column number (x) specified is less than or equal to the current column number, printing starts at the current position.

The PRINT TAB(x) statement can be used with any output device which can be specified in a PRINT statement (refer to paragraph 5.5.1.4).

Examples:

PRINT #0: TAB(5);	Spaces to column 5 of the terminal paper and prints next output beginning at column 5. If ; is missing, the output of the next PRINT statement executed begins at the left margin of the next line.
PRINT #2: TAB(80);	Outputs 8 spaces on the line printer assuming #2 previously opened.

5.5.2 INPUT Statement

The INPUT statement is used when data is to be input from the terminal keyboard or a file during program execution. The form of the statement is:

```
INPUT list
```

where list is a list of variable names separated by commas. Refer to paragraph 5.5.2.1 for data input from files.

When an INPUT statement is executed, BASIC prints a question mark (?) on the terminal and waits for data to be input.

BASIC inputs the next number from the input stream, saves the value as a numeric value. Numbers input on the same line must be separated by commas. If the data is alphabetic, BASIC inputs all characters up to a carriage return.

For example:

```
10 INPUT A,B,C
```

causes BASIC to pause during execution, print a question mark, and wait for input of three numeric values separated by commas. The values are input to the computer by typing the RETURN key.

If too few values are entered, BASIC prints another ? to indicate that more data is needed and waits for the additional data to be entered. If too many values are typed, the excess data on that line is ignored. The strings entered in response to the INPUT statement cannot be continued on another line since string input is terminated by the RETURN key.

When there are several values to be entered via the INPUT statement, it is helpful to print a message explaining the data needed.

For example:

```
10 PRINT "YOUR AGE IS ";
20 INPUT A
30 PRINT "SOC.  SEC.  #";
40 INPUT B
```

5.5.2.1 Selecting Input Devices

The INPUT statement also allows the selection of a particular input device. The form of the statement is:

```
INPUT #expression:list
```

where expression has the value 0 to 7. If the value is equal to 0, the terminal is the input device. If the value is not 0, input is read from the sequential file with the logical unit number expression (assigned by the OPEN statement). If the value is not within the range 0 - 7 or was not specified in an OPEN statement, the error message ?DCE (Device Channel Error) results. A question mark is not output when this form of the INPUT statement is used.

Excess data on an input line is ignored. If the data is insufficient to fill the list, BASIC looks for more data on the next line.

The colon (:) is required when variables follow the expression.

Examples:

```
OPEN "PR:" FOR INPUT AS FILE #1
INPUT #1:A,B
```

This statement causes BASIC/RT11 to input data from the high speed paper tape reader and store the data in variables A and B.

```
INPUT #0:  X,Y,Z
```

Input data from the terminal and store in variables X, Y and Z. Logical unit 0 defaults to the terminal.

5.5.3 DATA Statement

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

```
DATA data list
```

where the data list contains the numbers or strings to be assigned to the variables listed in a READ statement. Individual items in the data list are separated by commas; strings must be enclosed in quotation marks.

For example,

```
150 DATA 4,7.2,3
170 DATA 1.34E-3, 3.17311,"ABC"
```

The location of DATA statements is arbitrary as long as they appear in the correct order; however, it is good practice to collect all DATA statements near the end of the program for fast reference when checking a program.

When the RUN command is executed, BASIC searches for the first DATA statement and saves a pointer to its location. Each time a READ statement is encountered in the program, the next value in the data statement is assigned to a variable. If there are no more values in that DATA statement, BASIC looks for the next DATA statement. If control is transferred to a DATA statement, the statement is ignored.

5.5.4 READ Statement

A READ statement assigns the next available element in a DATA statement to the first variable in its list. Then it assigns the next available element in a DATA statement to the next variable in its list until all variables have been satisfied. The elements in the DATA statement must be in the correct order by type; if a string element is found where a number element is expected, or vice versa, the error message ?NSM is output. The READ statement is of the form:

```
READ variable list
```

The items in the variable list may be simple variable names or string variable names or subscripted variables and are separated by commas. For example,

```
READ A1,A2,B$,B1,C(3,5),D$(1)
```

Since data must be read before it can be used in a program, READ statements generally occur near the beginning of the program. A READ statement can be placed anywhere in a multiple statement line.

If an element in a data list is neither a number nor a string enclosed in quotes, the message ?BDR is printed. All subsequent READ's cause the ?OOD message. If there is no data available in the data table for the READ to store, the message ?OOD is printed.

Items in the data list in excess of those needed by the program's READ statements are ignored.

5.5.5 RESTORE Statement

The RESTORE statement resets the DATA list or specified sequential file (previously opened for input) to the beginning. RESTORE is of the form:

```
RESTORE #n
```

where n is a digit in the range 1 to 7. If #n is omitted, the DATA list is reset to its start. When a digit is specified, the appropriate input sequential file is repositioned to its start. (Refer to Section 5.9 for types of files.)

Examples:

```
30 RESTORE
```

causes the next READ statement following line 30 to begin reading data from the first DATA statement in the program, regardless of where the last value was found.

```
100 RESTORE #2
```

repositions the input sequential file associated with logical unit #2 to the beginning.

A further example of the use of RESTORE follows:

```
15 READ B,C,D
.
.
.
55 RESTORE
60 READ E,F,G
.
.
.
80 DATA 6,3,4,7,9,2
.
.
100 END
```

The READ statements in lines 15 and 60 both read the first three data values provided in line 80. (If the RESTORE statement had not been inserted before line 60, then the second READ would pick up data in line 80 starting with the fourth value.)

Since the values are being read as though for the first time, the same variable names may be used the second time through the data, if desired. To skip unwanted values, replacement, or dummy, variables may be inserted. For example:

```
1 REM - PROGRAM TO ILLUSTRATE USE OF RESTORE
20 READ N
25 PRINT "VALUES OF X ARE:"
30 FOR I=1 TO N
40 READ X
50 PRINT X,
60 NEXT I
70 RESTORE
185 PRINT
190 PRINT "SECOND LIST OF X VALUES"
200 PRINT "FOLLOWING RESTORE STATEMENT:"
210 FOR I=1 TO N
220 READ X
230 PRINT X,
240 NEXT I
250 DATA 4,1,2
251 DATA 3,4
300 END


RUNNH
VALUES OF X ARE:
 1             2             3             4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
 4             1             2             3
READY
```

The second time the data values are read, the first X picks up the value originally assigned to N in line 20, and as a result, BASIC prints:

```
4             1             2             3
```

To circumvent this, a dummy variable could be inserted to pick up and store the first value. This variable would not be represented in the PRINT statement, so the output would be the same each time through the list.

5.6 RANDOMIZE Statement

The RANDOMIZE statement causes the random number generator to calculate different random numbers every time the program is run. When executed, RANDOMIZE causes the RND function (explained in Chapter 6) to choose a random starting value to produce random results. The RANDOMIZE statement is written as

```
RANDOMIZE
```

RANDOMIZE may be placed anywhere in the program. It is good practice to completely debug a program before inserting the RANDOMIZE statement.

The following program demonstrates the use of the RANDOMIZE statement.

```
10 REM - RANDOM NUMBERS USING RANDOMIZE.
15 RANDOMIZE
```

```
25 PRINT "RANDOMIZED NUMBERS:"
30 FOR I = 1 TO 4
40 PRINT RND(0),
50 NEXT I
60 END

RUNNH
RANDOMIZED NUMBERS:
 .7785034E-1  .1632385      .2787781      .2035217
READY
RUNNH
RANDOMIZED NUMBERS:
 .8417053     .1678467E-2   .4347229      .5932312
READY
RUNNH
RANDOMIZED NUMBERS:
 .6651917     .2846375      .7210999      .7648621
READY
```

Removing the RANDOMIZE statement and changing line 25:

```
15
25 PRINT "REPRODUCIBLE RANDOM NUMBER SET."
```

program output is as follows.

```
RUNNH
REPRODUCIBLE RANDOM NUMBER SET.
 .0407319     .528293       .803172       .0643915
READY

RUNNH
REPRODUCIBLE RANDOM NUMBER SET.
 .0407319     .528293       .803172       .0643915
READY

RUNNH
REPRODUCIBLE RANDOM NUMBER SET.
 .0407319     .528293       .803172       .0643915
READY
```

5.7 PROGRAM CONTROL

The statements described in the following paragraphs cause the execution of a program to jump to a different line either unconditionally or depending upon some condition within the program.

5.7.1 GO TO Statement

The GO TO statement is used when it is desired to unconditionally transfer to some line other than the next sequential line in the program. In other words, a GO TO statement causes an immediate jump to a specified line, out of the normal consecutive line number order of execution. The general format of the statement is as follows:

GO TO line number

The line number to which the program jumps can be either greater or less than the current line number. It is thus possible to jump forward or backward within a program.

For example,

```
10 LET A=2
20 GO TO 50
30 LET A=SQR(A+14)
50 PRINT A,A*A
```

causes the following to be printed:

```
2              4
```

When the program encounters line 20, control transfers to line 50; line 50 is executed, control then continues to the line following line 50. Line 30 is never executed. Any number of lines can be skipped in either direction.

When written as part of a multiple statement line, GO TO should always be the last statement on the line (except for REM statements), since any statement following the GO TO on the same line is never executed. For example:

```
110 LET A=ATN(B2)\ PRINT A\ GO TO 50
```

5.7.2 IF THEN, IF GO TO and IF END Statements

The IF THEN statement is used to transfer conditionally from the normal consecutive order of statement numbers, depending upon the truth of some mathematical relation or relations. The basic format of the IF statement is as follows:

IF expression rel.op. expression {THEN | GO TO} line number

where expression — is an arithmetic or string expression. Expressions cannot be mixed; both must be string or both must be numeric. Numeric comparisons are handled as described in Section 2.5.2. String comparisons are performed on the ASCII values of the strings as described in Section 3.3.2.

rel.op. — is one of the relational operators described in section 2.5.2.

line number — is the line of the program to which control is conditionally passed.

If the relation is true, control passes to the line number specified. If the relation is false, control passes to the next statement in sequence.

Examples:

```
10 IF A=B THEN 20\PRINT "A<>B"
15 STOP
20 PRINT A+B

10 IF A<>10 GO TO 20\PRINT A
15 STOP
20 D=A+B*C

10 IF A$<B$ THEN 20\STOP
20 PRINT A$
```

BASIC/RT11 provides a special form of the IF statement used to detect an end of file condition on a sequential file. The form of the statement is:

```
          ( THEN  )
IF END #n {       } line number
          ( GO TO )
```

where #n represents the logical file number.

If the next input statement executed for the sequential file (#n) would detect an end of file (and an OUT OF DATA error message) then the branch to the line number is taken. The following example illustrates the use of the IF END statement:

```
10 OPEN "TEST" AS FILE #1
20 IF END #1 THEN 100
30 INPUT #1: A$
40 PRINT A$
50 GO TO 20
100 PRINT "END OF FILE"
110 STOP
```

The program prints out the contents of the ASCII file "TEST.DAT", followed by the message

```
END OF FILE
```

5.7.3 FOR-NEXT Statements

FOR and NEXT statements define the beginning and end of a program loop. (A loop is a set of instructions which are repeated over and over again, each time being modified in some way until a terminal condition is reached.) The FOR statement is of the form:

```
FOR variable = expression1 TO expression2 STEP expression3
```

where

variable	must be a nonsubscripted numeric variable.
expression	is an arithmetic expression which may be noninteger.

The variable is the index; expression1 is the initial value of the index; expression2, the index terminal value (the value which the index reaches before execution of the loop halts) and expression3, the increment value.

For positive STEP values, the loop is executed until the control variable is greater than its final value. For negative STEP values, the loop continues until the control variable is less than its final value.

For example:

```
15 FOR K=2 TO 20 STEP 2
```

causes program execution of the designated loop as long as K is less than or equal to 20. Each time through the loop, K is incremented by 2, so the loop is executed a total of 10 times. When K=20, program control passes to the line following the associated NEXT statement.

The NEXT statement signals the end of the loop which began with the FOR statement. The NEXT statement is of the form:

```
NEXT variable
```

where the variable is the same variable specified in the FOR statement. There must be only one NEXT statement for each FOR statement. Together the FOR and NEXT statements define the boundaries of the program loop. When execution encounters the NEXT statement, the computer adds the STEP expression value to the variable and checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement.

If the STEP expression and the word STEP are omitted from the FOR statement, +1 is the assumed value. Since +1 is a common STEP value, that portion of the statement is frequently omitted.

The expressions within the FOR statement are evaluated once upon initial entry to the loop. The test for completion of the loop is made prior to each execution of the loop. (If the test fails initially, the loop is never executed.)

The index variable can be modified within the loop. When control falls through the loop, the index variable retains the last value used within the loop.

The following is a demonstration of a simple FOR-NEXT loop. The loop is executed 10 times; the value of I is 10 when control leaves the loop; and +1 is the assumed STEP value:

```
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
40 PRINT I
```

The loop itself is lines 10 through 30. The numbers 1 through 10 are printed when the loop is executed. After I=10, control passes to line 40 which causes 10 to be printed again. If line 10 had been:

```
10 FOR I = 10 TO 1 STEP -1
```

the value printed by line 40 would be 1.

```
10 FOR I = 2 TO 44 STEP 2
20 LET I = 44
30 NEXT I
```

The above loop is only executed once since the value of I=44 has been reached and the termination condition is satisfied.

If, however, the initial value of the variable is greater than the terminal value, the loop is not executed at all. The loop set up by the statement:

```
10 FOR I = 20 TO 2 STEP 2
```

will not be executed, although a statement like the following will initialize execution of a loop properly:

```
10 FOR I=20 TO 2 STEP -2
```

FOR loops can be nested but not overlapped. The depth of nesting depends upon the amount of user storage space available (in other words, upon the size of the user program and the amount of core available). Nesting is a programming technique in which one or more loops are completely within another loop. The field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) must not cross the field of another loop.

ACCEPTABLE NESTING TECHNIQUES

Two Level Nesting

```
FOR I1 = 1 TO 10
FOR I2 = 1 TO 10
NEXT I2
FOR I3 = 1 TO 10
NEXT I3
NEXT I1
```

Three Level Nesting

```
FOR I1 = 1 TO 10
FOR I2 = 1 TO 10
FOR I3 = 1 TO 10
NEXT I3
FOR I4 = 1 TO 10
NEXT I4
NEXT I2
NEXT I1
```

UNACCEPTABLE NESTING TECHNIQUES

```
FOR I1 = 1 TO 10
FOR I2 = 1 TO 10
NEXT I1
NEXT I2
```

```
FOR I1 = 1 TO 10
FOR I2 = 1 TO 10
FOR I3 = 1 TO 10
NEXT I3
FOR I4 = 1 TO 10
NEXT I4
NEXT I1
NEXT I2
```

An example of nested FOR-NEXT loops is shown below:

```
5  DIM X(5,10)
10 FOR A=1 TO 5
20 FOR B=2 TO 10 STEP 2
30 LET X(A,B)= A+B
```

```
40 NEXT B
50 NEXT A
55 PRINT X(5,10)
```

When the above statements are executed, BASIC prints 15 when line 55 is processed.

It is possible to exit from a FOR-NEXT loop without the control variable reaching the termination value. A conditional or unconditional transfer can be used to leave a loop. Control can only transfer into a loop which had been left earlier without being completed, ensuring that termination and STEP values are assigned.

Both FOR and NEXT statements can appear anywhere in a multiple statement line. For example:

```
10 FOR I=1 TO 10 STEP 5\ NEXT I\ PRINT "I=";I
```

causes:

```
I= 6
```

to be printed when executed.

5.7.4 GOSUB and RETURN Statements

The GOSUB statement causes execution of a block of statements called a subroutine. The RETURN statement causes program control to return to the statement following the GOSUB.

A subroutine is a section of code performing some operation required at more than one point in the program. Sometimes a complicated I/O operation for a volume of data, a mathematical evaluation which is too complex for a user-defined function, or any number of other processes may be best performed in a subroutine.

More than one subroutine can be used in a single program, in which case they can be placed one after another at the end of the program (in line number sequence). A useful practice is to assign distinctive line numbers to subroutines; for example, if the main program uses line number 0 up to 199, use 200 and 300 as the first numbers of two subroutines.

Subroutines are usually placed physically at the end of a program before DATA statements, if any, and always before the END statement. The program begins execution and continues until it encounters a GOSUB statement of the form:

```
GOSUB line number
```

where the line number following the word GOSUB is that of the first line of the subroutine. Control then transfers to that line of the subroutine. For example:

```
50 GOSUB 200
```

Control is transferred to line 200 in the user program. The first line in the subroutine can be a remark or any executable statement.

Having reached the line containing a GOSUB statement, control transfers to the line indicated after GOSUB; the subroutine is processed until BASIC encounters a RETURN statement of the form:

```
RETURN
```

which causes control to return to the statement following the calling GOSUB statement. A subroutine is always exited via a RETURN statement.

Before transferring to the subroutine, BASIC internally records the next sequential statement to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this statement. In this way, no matter how many subroutines there are or how many times they are called, BASIC always knows where to transfer control next. The following program demonstrates the use of GOSUB and RETURN.

```
1   REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
10  DEF FNA(X)= ABS(INT(X))
20  INPUT A,B,C
30  GOSUB 100
40  LET A=FNA(A)
50  LET B=FNA(B)
60  LET C=FNA(C)
70  PRINT
80  GOSUB 100
90  STOP
100 REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
110 REM - OF THE EQUATION:  AX↑2 + BX + C = 0
120 PRINT "THE EQUATION IS    " A "*X↑2 + " B"*X  + " C
130 LET D=B*B - 4*A*C
140 IF D<>0 THEN 170
150 PRINT "ONLY ONE SOLUTION... X="; -B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS...X =(";
185 PRINT (-B+SQR(D))/(2*A);") AND (";(-B-SQR(D))/(2*A;")"
190 RETURN
200 PRINT "IMAGINARY SOLUTION...X=(";
205 PRINT -B/(2*A);"+"; SQR(-D)/(2*A);"I) AND (";
207 PRINT -B/(2*A);"-"; SQR(-D)/(2*A);"*I)"
210 RETURN
900 END
```

Subroutines can be nested; that is, one subroutine can call another subroutine. If the execution of a subroutine encounters a RETURN statement, it returns control to the line following the GOSUB which called that subroutine. Therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and can have more than one RETURN statement. It is possible to transfer to the beginning or any part of a subroutine; multiple entry points and RETURNs make a subroutine more versatile. Up to 20 levels of GOSUB nesting are allowed.

5.8 PROGRAM TERMINATION

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

The CHAIN statement also causes execution to cease but in addition, loads and executes a previously stored program.

5.8.1 END Statement

The END statement is the last statement in a BASIC program and is of the form:

```
END
```

The line number of the END statement must be the largest line number in the program, since any lines having line numbers greater than that of the END statement are not executed (although they are saved with the SAVE command. The END statement is optional. When an END statement is executed, program execution stops; all open files are automatically closed. If the program does not have an END or STOP statement, the open files are not closed.

5.8.2 STOP Statement

The STOP statement causes termination of program execution and can occur several times throughout a single program with conditional jumps determining the actual end of the program. The STOP statement is of the form:

```
STOP
```

and causes the message:

```
STOP AT LINE nnn
```

where nnn is the statement number of the STOP statement.

Execution of a STOP statement causes the message:

```
READY
```

to be printed on the terminal and all open files are automatically closed. This signals that the execution of a program has been terminated or completed, and BASIC is able to accept further input.

5.8.3 CHAIN Statement

The CHAIN statement terminates execution of the program currently in core then loads and executes the specified program. The execution of this previously stored program begins at the lowest line number unless another line number is specified. This allows a large program to be broken into segments and then linked together for execution with CHAIN.

The form of the command is

```
CHAIN "dev:filnam.ext" LINE number
```

The file descriptor (dev: filnam.ext) may be a literal string or a non-subscripted string variable name.

CHAIN closes all files which are open then opens and closes the program file containing the program to be executed. The default extension is .BAS.

All variables used in the current program are erased when the CHAIN statement is executed. If variables are to be passed to the next program, they must be stored in a file which is then read by the new program.

Examples:

```
CHAIN "DT1:PART2" LINE 10
```

Halts execution of current program then loads program, PART2.BAS, from DECtape unit 1 and begins execution at line 10.

5.9 FILE CONTROL

Any RT-11 file may be used or created by a BASIC program, including EDIT and MACRO files. A file may be used in one of two ways: first as an ASCII "sequential" file, as if it were typed at the terminal. Here are examples of statements which access sequential files:

```
INPUT #1:  A$, B, C
PRINT #2:  "ANSWERS:" X;Y
```

Alternatively, a file may be used as a random-access binary "virtual memory" file, as if each item were an element of a large array. The following are examples of statements which access virtual memory files.

```
LET A=(VF1(I)+VF1(J))/2
LET VF2(K)=A*3*SIN(X)
```

A virtual memory file may consist of string or numeric data, as explained below.

A sequential data file is limited in its applications and depends upon a strictly sequential treatment of I/O. With virtual data storage, reference can be made to any element within the file regardless of where that element resides.

The file control statements, OPEN and CLOSE, provide access to sequential and virtual memory files.

The OVERLAY statement overlays the program currently in memory with the specified file and continues execution.

NOTE

If a disk or DECtape is the device in an OPEN, CLOSE, OVERLAY, or CHAIN statement or OLD, SAVE, or REPLACE command (see Chapter 7) and the device is not on line a ?M-DIR I/O ERR? will be printed and control will return to the RT-11 monitor which will give an ?ILL CMD? message to the first command input. BASIC may then be reloaded by the RUN command but the stored program will be lost. This also occurs when a device is WRITE locked and the BASIC program attempts to output to it.

5.9.1 OPEN Statement

The OPEN statement opens files for input or output by the BASIC program and has two forms, one for sequential files and one for virtual.

For sequential files, the format is:

```
                              ( INPUT  )
OPEN "dev:filnam.ext" FOR     {        } AS FILE #digit DOUBLE BUF
                              ( OUTPUT )
```

where "dev: filnam.ext" may be a literal string or a scalar string variable name.

digit is a logical unit number in the range 1-7. The maximum number of files which may be opened at one time is 14 (7 sequential and 7 virtual).

If FOR OUTPUT or FOR INPUT is not included in the specification, the file is open for input. If the file name is omitted, the current program name is used. Thus, if the program name is TEST, then the statement to open file #1 will open the file DK:TEST.DAT. If the extension is omitted, .DAT (data) is assumed.

This form of the statement opens the specified file (or a non-file structured device) as a sequential ASCII file with logical unit number <expression>. The file is either for input or output as specified. Once opened, an input file may be read by the INPUT # statement, and an output file may be written by the PRINT # statement.

The OPEN statement can be used to specify the number of blocks to be assigned to an output file on disk or DECtape in the form:

```
...OUTPUT(blocks)...
```

Each block holds 512 ASCII characters including carriage return and line feed. If the program then attempts to write past the end of the file created, the message ?FTS (File Too Short) results. If the number of blocks is not specified, one half of the largest available group of blocks is used.

There is a 256-word input/output buffer associated with every file. Output to a file actually occurs only after the buffer is filled or the file is closed. For example, execution of a PRINT # statement where the device is the line printer will produce no visible output until the buffer is filled or the file is closed. DOUBLE BUF is optional and if specified a second 256-word I/O buffer is allotted to the file. Using DOUBLE BUF improves the execution speed of programs with extensive I/O but requires more memory.

Examples:

```
OPEN "ABC" FOR OUTPUT(5) AS FILE #1
```

Creates ABC.DAT on disk as logical file 1 and allocates 5 blocks.

```
LET A$=XYZ
OPEN A$ AS FILE #2 DOUBLE BUF
```

Opens disk file XYZ.DAT as logical file 2 and allocates two 256-word I/O buffers for input.

OPEN "ALTO.MAC" FOR INPUT AS FILE #3
Opens disk file ALTO.MAC as logical file 3 for input.

OPEN "LP:" FOR OUTPUT AS FILE #1
Opens the specified device "LP:" for output as file #1. If FOR OUTPUT were not specified, input would be assumed and an error message would result since the line printer is a write-only device.

The virtual memory file OPEN statement has the form:

OPEN "dev:filnam.ext" FOR {INPUT | OUTPUT} AS FILE VFnx(dimension)=string length

where

dev: filnam.ext	may be a literal string or a scalar string variable.
n	is a number in the range 1-7 representing the virtual file logical unit number. The maximum number of files which may be opened at one time is 14 (7 sequential and 7 virtual).

x is the type of virtual file as follows:

type	File data type
blank or null	The file consists of 2-word floating-point numbers.
%	The file consists of 1-word signed integers.
$	The file consists of strings of a given length. This length is 32 characters, unless otherwise specified.

(dimension)	is maximum subscript to be used in referencing the virtual file.
=string length	may be included for string virtual files to indicate the length of the strings in the file. The values which can be specified are 1,2,4,8,16,32,64 and 128. The default value is 32.

This form of the statement opens the specified file as the virtual file VFn. This special file is distinct from a sequential file <digit>.

If FOR OUTPUT is specified, the system allocates blocks to accommodate the maximum dimension specified. Any previous file with the same name will be deleted. FOR OUTPUT should only be specified to create a new file. To allow output to an existing virtual file neither FOR INPUT nor FOR OUTPUT should be specified. If the device cannot accommodate

the blocks specified, the message ?NER (Not Enough Room) results. As with sequential files, the number of blocks to be assigned to an output file can be specified after the phrase FOR OUTPUT. The number of blocks so specified overrides the maximum subscript specified if any. If neither is specified, the largest block number written becomes the length of the file.

The following table can be used to calculate the number of blocks needed for a file.

file type	# bytes per element	# elements per block
blank (floating point)	4	128
% (integer)	2	256
$	32	16
$=1	1	512
$=2	2	256
$=4	4	128
$=8	8	64
$=16	16	32
$=32	32	16
$=64	64	8
$=128	128	4

If the phrase FOR INPUT is included, then the file is write-protected; it may only be read by the program. If the phrase FOR OUTPUT is specified, a new file is created and can be used for input or output. If FOR INPUT or FOR OUTPUT is not specified an existing file is opened for input and/or output.

Once a virtual file has been opened, its elements may be used as any other variables in the BASIC program. A virtual file element may only be set by an assignment statement.

Examples:

```
OPEN "TEST" AS FILE VF1$(2000)=8
```

Opens the file TEST.DAT on disk as virtual memory file 1 containing 2000 string elements; each one 8 bytes long. This file is now available for input and output operations. A program reference to file element 2001 causes an error.

```
OPEN "TEST" FOR OUTPUT AS FILE VF2$(500)
```

Creates a file TEST.DAT on disk for output as virtual memory file 2 with 500 string elements, each 32 bytes long.

```
OPEN "TEST" FOR INPUT AS FILE VF3
```

Opens the file TEST.DAT for input only operations as virtual memory file 3, it consists of floating point numbers.

```
LET A$="TEST"
OPEN A$ FOR OUTPUT (10) AS FILE VF4%(50)
```

Creates the file TEST.DAT and opens it for input or output as virtual memory file 4 with 10 blocks. The number of blocks overrides the number of elements (50).

These files can then be used in BASIC operations as follows:

LET A = B + VF3(I)/2
: Uses the value of virtual file element VF3(I) in computing an expression.

PRINT "VARIABLE", N, VF4(N)
: Uses the value of integer virtual memory file element VF4(N) in a print list.

LET VF3(2*N+1) = (A + B)/2
: Sets the value of virtual memory file element VF3(2*N+1) to the value of the expression (A+B)/2.

LET VF1(10) = "ABCD"
: Sets the value of string virtual memory file element VF1(10) to "ABCD". The string will be truncated or lengthened and filled with blanks to the appropriate length, as specified in the OPEN statement.

5.9.2 CLOSE Statement

The CLOSE statement closes the logical file specified and has the form

CLOSE file identification

where file identification contains the file numbers of the form:

#n	for sequential files
VFn	for virtual memory files

where n is a digit in the range 1 to 7.

If no file identification is specified, all open files are closed.

If a file is referenced after a CLOSE, the message ?FNO (File Not Open) is printed.

NOTE

In addition to CLOSE, the SCRATCH, NEW, OLD and CLEAR commands, the END, STOP and CHAIN statements and the ?FIO error routine close all open file when executed.

Examples:

CLOSE #1	Closes the sequential file associated with logical unit 1.
CLOSE VF3	Closes the virtual memory file associated with logical unit 3.

5.9.3 OVERLAY Statement

The OVERLAY statement causes the program currently in core to be "overlaid" or merged with the specified file, which also contains a BASIC program.

The form of the OVERLAY statement is:

```
OVERLAY "file descriptor"
```

All variables and arrays defined keep their current values. All data files remain open. If a program line in the new program has a line number identical to one in the current program, the current program line is replaced by the new program line. After the overlaid program has been merged with the current program, execution continues at the first program line which now follows the statement number of the OVERLAY statement. Thus, programs can be segmented into separate files as with the CHAIN statement, and data can be communicated among segments in the arrays, and a very long program can be divided up into several smaller overlay segments.

The new program must not contain DIM, RANDOMIZE, or DEF statements. If a DEF statement in the current program is overlaid, the function will no longer be defined.

As an example:

Main Program

```
 10 DIM A (100)
 20 FOR I = 0 TO 100
 30 LET A (I) = SQR (I)
 40 NEXT I
 50 DEF FNS(I) = SQR (A (I))
 60 OPEN "LP:" FOR OUTPUT AS FILE #1
100 FOR I = 0 TO 100
110 PRINT  #1: A  (I),
120 NEXT I
900 OVERLAY "OV1"
910 GO TO 100
```

Overlay Section, file OV1.BAS

```
100 PRINT #1:    "FIRST OVERLAY"
110 FOR J = 0 TO 100
120 PRINT #1:    FNS(J),
130 NEXT J
140 STOP
```

Execution of the main program sets the elements of A to the square root of I; the function FNS(I) is set to the square root of A(I), or the fourth root of I. The main program then prints out the elements of A on the line printer.

The execution of the OVERLAY statement causes the file

```
"DK:OV1.BAS"
```

to be edited into the program.

The program in memory is now:

```
 10  DIM A(100)
 20  FOR I = 0 TO 100
 30  LET A(I) = SQR(I)
 40  NEXT I
 50  DEF FNS(I) = SQR (A(I))
 60  OPEN "LP:" AS FILE #1
100  PRINT #1: "FIRST OVERLAY"
110  FOR J = 1 TO 100
120  PRINT #1: FNS (J),
130  NEXT J
140  STOP
900  OVERLAY "OV1"
910  GO TO 100
```

Control now passes to statement 910, which is the first statement following statement 900 in the merged program.

Execution at statement 100 causes

"FIRST OVERLAY"

to be printed, followed by the fourth roots of the numbers from 0 to 100.

Finally, "STOP AT LINE 140" is output at the terminal.

An overlay statement executed in the immediate mode (without a line number) will act like an OLD command, except that the program currently in core is not scratched. Instead, the program lines in the specified file will be edited into the program, just as if they were typed in via the console.

A very useful application of this feature is when the BASIC programmer has a "library" of GOSUB subroutines to edit into his program. The procedure is as follows.

Type in the BASIC program as if there were subroutines at specific (high) statement numbers such as 1000, 2000, etc. Then SAVE the program. The next step is to resequence the required library routines using the BASIC program RESEQ (see Chapter 10) so that they begin at the correct statement numbers. Then read in the saved program again with the OLD command. Finally, edit in the subroutines with immediate mode OVERLAY statements such as

```
OVERLAY "SUB1"
OVERLAY "SUB2"
```

Finally, a REPLACE command will update the saved program.

CHAPTER 6

BASIC/RT-11 FUNCTIONS

6.1 ARITHMETIC FUNCTIONS

BASIC provides eleven functions to perform certain standard mathematical operations such as square roots, logarithms, etc.

These functions have three-letter call names followed by a parenthesized argument. They are pre-defined and may be used anywhere in a program.

Call Name	Function
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent of x as an angle in radians in range + or - pi/2.
BIN(x$)	Computes the integer value from a string of blanks (ignored), zeroes, and ones (binary integer).
COS(x)	Returns the cosine of x radians.
EXP(x)	Returns the value of e↑x where e=2.71828...
INT(x)	Returns the greatest integer less than or equal to x, (INT(-.5)=-1).
LOG(x)	Returns the natural logarithm of x.
OCT(x$)	Computes an integer value from a string of blanks (ignored) and digits from 0 to 7 (octal integer).
RND(x)	Returns a random number greater than or equal to 0 and less than 1.
SGN(x)	Returns a value indicating the sign of x.
SIN(x)	Returns the sine of x radians.
SQR(x)	Returns the square root of x.
TAB(x)	Causes the terminal type head to tab to column number x. Valid in PRINT statement only (refer to paragraph 5.5.1.5).

The argument x to the functions can be a constant, a variable, an expression, or another function. A square bracket cannot be used as the first enclosing character for the argument x, e.g., SIN[x] is illegal.

Function calls, consisting of the function name followed by a parenthesized argument, can be used as expressions or as elements of expressions anywhere that expressions are legal.

Values produced by the functions SIN(x), COS(x), ATN(x), SQR(x), EXP(x), and LOG(x) have six significant digits.

6.1.1 Sine and Cosine Functions, SIN(x) and COS(x)

The sine and cosine functions require an argument angle expressed in radian measure. If the angle is stated in degrees, conversion to radians may be done using the identity:

<radians> = <degrees> (pi/180)

In the following example program, 3.14159265 is used as a nominal value for pi. P is set equal to this value at line 20. At line 40 the above relationship is used (in the expression within the LET statement) to convert the input value into radians.

```
10 REM - CONVERT ANGLE (X) TO RADIANS, AND
11 REM - FIND SIN AND COS
20 LET P = 3.14159265
25 PRINT "DEGREES", "RADIANS", "SINE", "COSINE"
30 INPUT X
40 LET Y = X*P/180
60 PRINT X, Y, SIN(Y), COS(Y)
70 GO TO 30
RUNNH
DEGREES        RADIANS        SINE           COSINE
?0
 0              0              0              1
?10
 10             .174533        .173648        .984808
?20
 20             .349066        .34202         .939693
?30
 30             .523598        .5             .866025
?360
 360            6.28319       -3.7457E-07     1
?45
 45             .785398        .707107        .707107
?↑C
.REENTER
READY
```

6.1.2 Arctangent Function, ATN(x)

The arctangent function returns a value in radian measure, in the range +pi/2 to -pi/2 corresponding to the value of a tangent supplied as the argument (X).

In the following program, input is an angle in degrees. Degrees are then converted to radians at line 40. At line 50 the radian value (Y) is used with the SIN and COS functions to derive the tangent of the input angle according to the identity:

$$TAN(X) = \frac{SIN(X)}{COS(X)}$$

At line 70 the tangent value, Z, is supplied as argument to the ATN function to derive the value found in column 4 of the printout under the label ATN(X). Also in line 70 the radian value of the arctangent function is converted back to degrees and printed in the fifth column of the printout as a check against the input value shown in the first column.

```
10 LET P = 3.14159265
20 PRINT "SUPPLY AN ANGLE IN DEGREES"
25 PRINT "ANGLE","ANGLE","TAN(X)","ATAN(X)","ATAN(X)"
26 PRINT "(DEGS)","(RADS)",,,"(DEGS)"
30 INPUT X
40 LET Y = X*P/180
45 IF ABS(COS(Y))<.01 THEN 100
50 LET Z = SIN(Y)/COS(Y)
70 PRINT X,Y,Z,ATN(Z),ATN(Z)*180/P
85 PRINT
90 GO TO 30
100 PRINT "ANGLE ERROR"
110 GO TO 30
RUNNH
SUPPLY AN ANGLE IN DEGREES
ANGLE   ANGLE       TAN(X)              ATAN(X)             ATAN(X)
(DEGS)  (RADS)                                              (DEGS)
?0
 0       0           0                   0                   0

?45
 45      .785398     .999999             .785398             45

?10
 10      .174533     .176327             .174533             10

?↑C
.REENTER
READY
```

Note that the tangent of an odd multiple of pi/2 radians is not defined. Since the cosine of such an angle is 0, the statement on line 50 would be dividing by 0 and the statement on line 45 checks for an angle close to the odd multiple of pi/2 radians to circumvent this problem.

6.1.3 Square Root Function, SQR(x)

This function derives the square root of any positive value as shown below.

```
10 INPUT X
20 LET X = SQR(X)
30 PRINT X
40 GO TO 10
RUNNH
?16
```

```
 4
?100
 10
?1000
 31.6228
?123456789
 11111.1
?17
 4.12311
?25E2
 50
?1970
 44.3847
?↑C
.REENTER
READY
```

6.1.4 Exponential Function, EXP(x)

The exponential function raises the number e to the power x. EXP is the inverse of the LOG function. The relationship is

```
LOG(EXP(X)) = X
```

The following program prints the exponential equivalent of an input value. Note that the output values derived below are used as input to the LOG function in Section 6.1.5.

```
10 INPUT X
20 PRINT EXP(X)
40 GO TO 10
99 END
RUNNH
?4
 54.5981
?10
 22026.5
?9.421006
 12345
?4.60517
 100
?25
 7.20049E+10
?↑C

.REENTER

READY
```

6.1.5 Logarithm Function, LOG(x)

The LOG function derives the logarithm to the base e of a given value. In the following program at line 20, the LOG function is used to convert an input value to its logarithmic equivalent.

```
10 INPUT X
20 PRINT LOG(X)
```

```
30 GO TO 10

RUNNH
?54.59815
 4
?22026.47
 10
?12345
 9.42101
?100
 4.60517
?.720049E11
 25
?↑C
.REENTER
READY
```

Logarithms to the base e may easily be converted to any other base using the following formula:

$$\log_a N = \frac{\log_e N}{\log_e a}$$

where a represents the desired base. The following program illustrates conversion to the base 10.

```
 1 REM - CONVERT BASE E LOG TO BASE 10 LOG.
 5 PRINT "VALUE","BASE E LOG","BASE 10 LOG"
15 INPUT X
17 PRINT X,
20 PRINT LOG(X),
40 PRINT LOG(X)/LOG(10)
50 GO TO 15
60 END
RUNNH
VALUE         BASE E LOG     BASE 10 LOG
?4
 4             1.38629        .60206
?250
 250           5.52146        2.39794
?5
 5             1.60944        .69897
?60
 60            4.09434        1.77815
?100
 100           4.60517        2
?↑C
.REENTER
READY
```

An attempt to do a LOG(0) or logarithm of a negative number causes the ?ARG error message.

6.1.6 Absolute Function, ABS(x)

The ABS function returns an absolute value for any input value. Absolute value is always positive. In the following program, various input values are converted to their absolute values and printed.

```
10 INPUT X
20 LET X = ABS(X)
30 PRINT X
40 GO TO 10
RUNNH
?-35.7
 35.7
?2
 2
?25E10
 2.50000E+11
?105555567
 1.05556E+08
?10.12345
 10.1234
?-44.555566668899
 44.5556
?↑C
.REENTER
READY
```

6.1.7 Integer Function, INT(x)

The integer function returns the value of the greatest integer not greater than x. For example:

```
PRINT INT(34.67)
 34

PRINT INT(-5.1)
-6
```

The INT of a negative number is a negative number with the same or larger absolute value, i.e., the same or smaller algebraic value. For example:

```
PRINT INT(-23.45)
-24

PRINT INT(-14.39)
-15

PRINT INT(-11)
-11
```

The INT function can be used to round numbers to the nearest integer, using INT(X+.5). For example:

```
PRINT INT(34.67+.5)
 35
PRINT INT(-5.1+.5)
-5
```

INT(X) can also be used to round to any given decimal place or integral power of 10, by using the following expression as an argument:

```
(X*10↑D+.5)/10↑D
```

where D is an integer supplied by the user.

```
10 REM - INT FUNCTION EXAMPLE.
15 PRINT
20 PRINT "NUMBER TO BE ROUNDED:"
25 INPUT A
40 PRINT "NO. OF DECIMAL PLACES:"
45 INPUT D
60 LET B = INT(A*10↑D + .5)/10↑D
70 PRINT "A ROUNDED = " B
80 GO TO 15
90 END

RUNNH

NUMBER TO BE ROUNDED:
?55.65842
NO. OF DECIMAL PLACES:
?2
A ROUNDED =  55.66

NUMBER TO BE ROUNDED:
?78.375
NO. OF DECIMAL PLACES:
?-2
A ROUNDED =  100

NUMBER TO BE ROUNDED:
?67.38
NO. OF DECIMAL PLACES:
?-1
A ROUNDED =  70

NUMBER TO BE ROUNDED:
?↑C
.REENTER
READY
```

6.1.8 Random Number Function, RND(x)

The random number function produces a random number, or random number set, between 0 and 1. If the RANDOMIZE statement is not present in the program; the numbers are reproducible in the same order for later checking of a program. The argument (x) is not used and can be any number (but cannot be a string expression); it serves only to standardize all BASIC function representations. The form RND is also legal. For example:

```
10 REM - RANDOM NUMBER EXAMPLE.
25 PRINT "RANDOM NUMBERS:"
30 FOR I = 1 TO 15
40 PRINT RND(0),
50 NEXT I
60 END
RUNNH
RANDOM NUMBERS:
 .1002502      .9648132     .8866272      .6364441   .8390198
 .3061218      .285553      .9582214      .1793518   .4521179
 .9854126E-1   .5221863     .2462463      .7778015   .450592

READY
```

To obtain random digits from 0 to 9, change line 40 to read:

```
40 PRINT INT(10*RND(0)),
```

and run the program again. This time the results will be printed as follows.

```
RUNNH
RANDOM NUMBERS:
 1             9            8             6          8
 3             2            9             1          4
 0             5            2             7          4

READY
```

It is possible to generate random numbers over a given range. If the open range (A,B) is desired, use the expression:

```
(B-A)*RND(0)+A
```

to produce a random number in the range A<n<B.

The following program produces a random number set in the open range 4,6 (the extremes, 4 and 6, are never reached).

```
10 REM - RANDOM NUMBER SET IN OPEN RANGE 4,6.
20 FOR B = 1 TO 15
30 LET A = (6-4) * RND(0) +4
40 PRINT A,
50 NEXT B
60 END

RUNNH
 4.2005           5.92962     5.77325       5.27288    5.67804
 4.61224          4.57110     5.91644       4.35870    4.90423
 4.19708          5.04437     4.49249       5.55560    4.90118

READY
```

6.1.9 Sign Function, SGN(x)

The sign function returns the value 1 if x is a positive value, 0 if x is 0, and -1 if x is negative. For example:

```
PRINT SGN(3.42)
 1

PRINT SGN(-42)
-1

PRINT SGN(23-23)
 0
```

The following example program illustrates the use of the SGN function.

```
10 REM- SGN FUNCTION EXAMPLE.
20 READ A,B,C
25 PRINT "A  = "A, "B  = "B, "C  = "C
30 PRINT "SGN(A) ="SGN(A), "SGN(B) ="SGN(B),
40 PRINT "SGN(C) ="SGN(C)
50 DATA -7.32, .44, 0
60 END
RUNNH
A  = -7.32    B  =  .44     C  =  0
SGN(A) =-1    SGN(B) = 1    SGN(C) = 0

READY
```

6.1.10 Binary Function, BIN(x$)

The BIN function computes the integer value of a string of 1's and 0's. Spaces are ignored (allowing input in convenient bit groupings), and the parentheses around the argument are not required.

For example,

```
PRINT BIN ('100101001')
297
```

The binary number is treated as a signed 2's complement integer and its absolute value may not be larger than $2^{15}-1$.

For example,

```
PRINT BIN ('1  111  111  111  111  111')
-1
```

6.1.11 Octal Function, OCT(x$)

The OCT function computes an integer value from a string of blanks (ignored) and digits from 0 to 7. Spaces are ignored (allowing input in convenient spacing), and the parentheses around the argument are not required.

For example,

```
PRINT OCT ('177777')
-1
```

The number is treated as a signed 2's complement and its absolute value may not be larger than $2^{15}-1$.

6.2 USER DEFINED FUNCTIONS

In some programs it may be necessary to execute the same sequence of statements or mathematical formulas in several different places. BASIC allows definition of unique operations or expressions and the calling of these functions in the same way as the square root or trig functions.

These user-defined functions consist of a function name: the first two letters of which are FN followed by a third letter. For example:

legal	illegal
FNA	FNA1
	FN2

Each function is defined once and the definition may appear anywhere in the program. The defining or DEF statement is formed as follows:

```
DEF  FNa (variable list) = expression
```

where a — is an alphabetic character which becomes part of the function name. The expression, however, need not contain all the arguments.

variable list — may consist of one to five dummy variables.

expression — (to the right of the equal sign) may contain the variables named in the variable list.

For example:

```
10 DEF FNA(S) = S↑2
```

causes a later statement:

```
20 LET R=FNA (4) + 1
```

to be evaluated as R=17. As another example:

```
50 DEF FNB(A,B) = A+X↑2
60 LET Y=FNB(14.4,R3)
```

causes the function to be evaluated using the current value of the variable X squared +14.4. In this case the dummy argument B (which becomes the actual argument R3 in the function call) is unused.

The two following programs

Program #1:

```
10 DEF FNS(A) = A↑A
20 FOR I=1 TO 5
30 PRINT I, FNS(I)
40 NEXT I
50 END
```

Program #2:

```
10 DEF FNS(X) = X↑X
20 FOR I=1 TO 5
30 PRINT I, FNS(I)
40 NEXT I
50 END
```

cause the same output:

```
RUNNH
 1            1
 2            4
 3            27
 4            256
 5            3125

READY
```

The arguments in the DEF statement can be seen to have no significance; they are strictly dummy variables. (A DEF statement with no arguments is illegal.) The function itself can be defined in the DEF statement in terms of numbers, variables, other functions, or mathematical expressions. For example:

```
10 DEF FNA(X) = X↑2+3*X+4
20 DEF FNB(X) = FNA(X)/2 + FNA(X)
30 DEF FNC(X) = SQR(X+4)+1
```

The statement in which the user-defined function appears can have that function combined with numbers, variables, other functions, or mathematical expressions. For example:

```
40 LET R = FNA(X+Y+Z)*N/(Y↑2+D)
```

A user-defined function can be a function of one to five variables, as shown below:

```
25 DEF FNL(X,Y,Z) = SQR(X↑2 + Y↑2 + Z↑2)
```

A later statement in a program containing the above user-defined function might look like the following:

```
55 LET B = FNL(D,L,R)
```

where D, L, and R have some values in the program.

The number of arguments with which a user-defined function is called must agree with the number of arguments with which it was defined. For example:

```
10 DEF FNA (X) = X*2 + X/2
20 PRINT FNA(3,2)
```

causes the error message:

```
?ARG AT LINE 20
```

When calling a user-defined function, the parenthesized arguments can be any legal expressions. The value of each expression is substituted for the corresponding function variable. For example:

```
10 DEF FNZ(X)=X↑2
20 LET A=2
30 PRINT FNZ(2+A)
```

line 30 causes 16 to be printed.

If the same function name is defined more than once, an error message is printed.

```
10 DEF FNX(X)=X↑2
20 DEF FNX(X)=X+X
%IDF AT LINE 20
```

and the program cannot be executed until corrected.

The function variable need not appear in the function expression as shown below:

```
10 DEF FNA (X) = 4 +2
20 LET R = FNA(10)+1
30 PRINT R
40 END
RUNNH
 7
```

The program in Figure 6-1 contains examples of a multi-variable DEF statement in lines 10, 25, and 40.

```
1 REM MODULUS ARITHMETIC PROGRAM
5 REM FIND X MOD M
10 DEF FNM(X,M)=X-M*INT(X/M)
15 REM
20 REM FIND A+B MOD M
25 DEF FNA(A,B,M)=FNM(A+B,M)
30 REM
35 REM FIND A*B MOD M
40 DEF FNB(A,B,M)=FNM(A*B,M)
41 REM
45 PRINT
50 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M"
55 PRINT "GIVE ME AN M";\INPUT M
60 PRINT \PRINT "ADDITION TABLES MOD "M
65 GOSUB 800
70 FOR I=0 TO M-1
75 PRINT I;"  ";
80 FOR J=0 TO M-1
85 PRINT FNA(I,J,M);
90 NEXT J\PRINT \NEXT I
100 PRINT \PRINT \
110 PRINT "MULTIPLICATION TABLES MOD "M
120 GOSUB 800
130 FOR I=0 TO M-1
140 PRINT I;"  ";
150 FOR J=0 TO M-1
160 PRINT FNB(I,J,M);
170 NEXT J\PRINT \NEXT I
180 STOP
800 REM SUBROUTINE FOLLOWS:
810 PRINT \PRINT TAB(5);0;
820 FOR I=1 TO M-1
830 PRINT I;\NEXT I\PRINT
840 FOR I=1 TO 3*M+4
850 PRINT "-";\NEXT I\PRINT
860 RETURN
870 END
```

Figure 6-1 Modulus Arithmetic

```
RUNNH

ADDITION AND MULTIPLICATION TABLES MOD M
GIVE ME AN M?7

ADDITION TABLES MOD  7

      Ø  1  2  3  4  5  6
-------------------------
Ø     Ø  1  2  3  4  5  6
1     1  2  3  4  5  6  Ø
2     2  3  4  5  6  Ø  1
3     3  4  5  6  Ø  1  2
4     4  5  6  Ø  1  2  3
5     5  6  Ø  1  2  3  4
6     6  Ø  1  2  3  4  5


MULTIPLICATION TABLES MOD  7

      Ø  1  2  3  4  5  6
-------------------------
Ø     Ø  Ø  Ø  Ø  Ø  Ø  Ø
1     Ø  1  2  3  4  5  6
2     Ø  2  4  6  1  3  5
3     Ø  3  6  2  5  1  4
4     Ø  4  1  5  2  6  3
5     Ø  5  3  1  6  4  2
6     Ø  6  5  4  3  2  1

STOP AT LINE 18Ø

READY
```

Figure 6-1 (Cont.) Modulus Arithmetic

6.3 STRING FUNCTIONS

Like the intrinsic mathematical functions (e.g., SIN, LOG), BASIC contains various functions for use with character strings. These functions allow the program to concatenate two strings, access part of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, search for a substring within a larger string, and perform other useful operations. The various functions available are summarized in Table 6-1.

Table 6-1

String Functions

Function Code	Meaning
ASC(x$)	Returns the seven-bit internal code for the one-character string (x$) as a decimal number. If the argument is a null string or contains more than one character, the ?ARG error message is output.
CHR$(x)	Generates a one-character string having the ASCII value of x where x is a number greater than or equal to 0 and less than or equal to 255. Refer to Appendix B. For example: CHR$(65) is equivalent to "A". Arguments greater than 127 are treated modulo 128. Only one character can be generated.
DAT$	Returns the current date, as set by the RT-11 Monitor, in the form 07-MAY-73.
LEN(x$)	Returns the number of characters in the string x$ (including trailing blanks). For example: PRINT LEN(A$) 26
POS(x$,y$,z)	Searches for and returns the position of the first occurrence of y$ in x$ starting with the zth position. If the string y$ is not found in the string x$, then 0 is returned. If x$ is a null string, 0 is returned. If y$ is a null string, the character position of z is returned.
SEG$(x$,y,z)	Returns the string of characters in positions y through z in x$. If y =<0, 1 is assumed. If z =< 0, a null string is returned. If z > the length of (x$), the string to end of x$ is returned. If z < y, a null string is returned. If y > LEN(x$), a null string is returned.

Table 6-1 (Cont.)

String Functions

Function Code	Meaning
STR$(x)	Returns the string which represents the numeric value of x as it would be printed by a PRINT statement but without a leading or trailing blank.
TRM$(X$)	Returns X$ with trailing blanks removed (trimmed).
VAL(x$)	Returns the number represented by the string x$. If x$ does not represent a number, the ?ARG error message is output.

In the above examples, x$ and y$ represent any legal string expressions and x, y, and z represent any legal arithmetic expressions.

6.3.1 User-Defined String Functions

Character string functions can be written in the same way as numeric functions. (See Section 6.2.)

User-defined string functions return character string values, although both numeric and string values can be used as arguments to the function.

```
10 DEF FNL(A$,X)=A$&STR$(X)
```

The following function combines two strings into one string:

```
10 DEF FNC(X$,Y$)=X$&Y$
```

Numbers cannot be used as arguments in a function where strings are expected or vice versa. Line 80 is unacceptable:

```
10 DEF FNA(A$) = CHR$(LEN(A$)+1)
80 LET Z=FNA(4)
```

The message:

```
?NSM AT LINE 80
```

is printed.

CHAPTER 7

EDITING COMMANDS

BASIC provides key commands which can be used to halt program execution, erase characters or delete lines. Table 7-1 provides an explanation of each of the key commands.

Table 7-1

Key Commands

Key	Explanation
CTRL/C	Interrupts execution of a command or program and returns control to the RT-11 monitor BASIC can be restarted without loss of the current program by using the monitor RE command. A control command is typed by holding down the CTRL key while typing the letter key.
CTRL/O	Stops output on the terminal but does not halt execution until an input statement is encountered or the program terminates. If CTRL/O is typed again, type out resumes. If desired, immediate mode statements can be used to print the results of the program after a CTRL/O suppresses output.
(Shift O) or RUBOUT	Deletes the last character typed and echoes as a backarrow on the terminal. For example, RT-11 BASC←IC RUBOUT typed here. Spaces as well as characters may be erased. On a VT05 or an LA30, the underscore key (-) is used instead of RUBOUT to delete characters.
ALTMODE or CTRL/U	Deletes the entire current line (provided the RETURN key has not been typed). BASIC displays DELETED at the end of the line. For example: OS BASIC DELETED ↑ ALTMODE typed here. On some terminals, the ESCAPE key must be used.

If the RETURN key has already been typed, a program line can be corrected by typing the appropriate line number and retyping the line correctly.

The line can be deleted by typing the RETURN key immediately after the line number; removing both the line number and line from the program.

If the line number of a line not needing correction is accidentally typed, the RUBOUT key (also SHIFT/O, ALTMODE, ESC or CTRL/U) may be used to delete the number(s); then the correct number can be typed. Assume the line:

```
10 IF A>5 GO TO 230
```

is correct. A line 15 is to be inserted, but

```
10 LET
```

is typed by mistake. The correction is made as follows:

```
10 LET←←←←←5 LET X=X-3
```

Line 10 remains unchanged, and line 15 is entered.

Following an attempt to run a program, error messages may be output on the terminal indicating illegal characters or formats, or other user errors in the program. Most errors can be corrected by typing the line number(s) and the correction(s) and then rerunning the program. As many changes or corrections as desired may be made before each program run.

The following editing commands are entered in immediate mode and terminated by the RETURN key. These commands are used to erase a program in core, assign a program name and list, punch or run a program.

7.1 SCRATCH COMMAND

The SCRATCH (or SCR) command clears the storage area set up by BASIC (refer to Appendix G). This deletes any commands, programs arrays, strings or symbols currently stored by BASIC.

SCRATCH should be used before entering a new program from the terminal keyboard to be sure no old program lines will be mixed into the new program and to clear out the symbol table area.

Example:

```
SCR

READY
10 READ A
.
.
.
```

clears the storage area and inserts the program being input at the keyboard.

7.2 OLD COMMAND

The OLD command (OLD) erases the contents of the storage area (SCRATCH and CLEAR) and inputs the program via the specified device.

The form of the command is:

```
OLD "dev:filnam.ext"
```

If the file descriptor (dev:filnam.ext) is not specified as part of the OLD command, BASIC prints:

```
OLD FILE NAME--
```

and waits for the file description and the return key. Type the name of the file containing the BASIC program (do not enclose the filename in quotation marks). If a filename is not entered, BASIC assumes the name NONAME.

In the examples of OLD commands that follow, the computer printout is underlined

```
OLD
OLD FILE NAME--TEST1
```

clears user area and inputs program TEST1.BAS from Disk (DK).

```
OLD "DT1:PROG1"
```

clears user area and inputs program PROG1.BAS from DECtape unit 1.

```
OLD "PR:RESEQ"
```

clears user area and inputs the program RESEQ from the high speed paper tape reader.

7.3 LIST/LISTNH COMMANDS

The LIST command prints the specified lines of the user program currently in memory on the terminal. The program name, date and the BASIC version number are output as a header line for the lines being listed. The form of the LIST command is:

```
LIST statement no.-statement no.
```

Several variations of the LIST command can be used:

LIST statement no.	Lists only the specified line.
LIST-statement no.	Lists from the beginning of the program to and including the specified line.
LIST statement no.- LIST statement no.-END	Lists from the specified line to the end of the program.

LIST statement no.-statement no.
Lists the specified section of the program.

If no statement number is specified, the entire program is listed. If the statement number specified does not exist, the first line of the program is listed.

Typing LIST followed by the statement number causes the header line and the line specified to be listed. The LISTNH command also prints the lines currently in core but suppresses the header line.

Type CTRL/O (depress the CTRL key and type the O key) to suppress an undesired listing. BASIC returns to the READY message when command execution is complete.

The lines listed may differ slightly from those entered because:

1. Certain characters while acceptable to BASIC are stored in a standard manner when they appear outside of quotation marks.

Character typed	Character stored
]	)
[	(
=<	<=
=>	>=
><	<>

2. Literals are stored to 24 bits of accuracy. Those with more than 24 bits are truncated to 24 bits.

3. Although literal storage is 24 bits, output is truncated to 6 decimal digits.

4. Literals are output in standard BASIC format, regardless of how they were input, for example,

```
10 LET X=3.0+1.0000001
20 PRINT X-1E7
LIST
10 LET X=3+1
20 PRINT X-1.00000E+07
```

5. Spaces in the input program are ignored, except within strings and REM statements. The LIST command prints the program with spaces inserted to separate keywords and line numbers from numeric information. The listed program is therefore easier to read. In the case of an IF...GO TO statement, no space is typed before the GO TO keyword.

Examples:

LISTNH 100 lists line 100.

LIST-10	lists the header line and the program lines up to line 10.
LIST 10-20	lists the header line and lines 10 to 20 of the program in memory.

7.4 SAVE COMMAND

The SAVE command creates an ASCII file and saves the BASIC program currently in memory as specified in the file descriptor. A SAVEd program can be retrieved with the OLD command or CHAIN or OVERLAY statement. The form of the command is:

```
SAVE "dev:filnam.ext"
```

If no file descriptor is specified, it is assumed to be DK: name.BAS where name is the current program name.

The SAVE command can be used to list the program currently in memory on the line printer.

If the file specified already exists, then the error message

```
?RPL or USE REPLACE
```

is typed on the console.

Examples:

SAVE "DT1:PROG1"	outputs program in core to DECtape unit 1 as PROG1.BAS.
SAVE	outputs program to the system device with current program name, and extension BAS.
SAVE "LP:"	lists the program on the line printer.

7.5 REPLACE COMMAND

The REPLACE command is just like the SAVE command, except that it replaces, or updates a file previously created by SAVE. The destinction between creation and replacement of files prevents the user from inadvertently destroying programs which he has previously saved.

The form of the command is

```
REPLACE "dev:filnam.ext"
```

If no file descriptor is specified, it is assumed to be DK:filnam.bas where filnam is the current program name.

7.6 RUN/RUNNH COMMANDS

After the user program is entered into memory, it can be executed by typing the command

```
RUN
```

and the RETURN key. The RUN command causes a header line (program name, date and BASIC version number) to be printed before the program is executed.

When BASIC is first loaded or when a SCR command is executed, the user program name is set to NONAME until a RENAME command is executed.

The program is scanned; arrays are created in core and then the program is executed. Any appropriate error messages are printed and when the END or STOP statement is encountered, execution halts and a message is printed. Execution of a program can be halted before executing an END or STOP statement by using the CTRL/C, RE combination to return BASIC to a READY message.

After execution, the variables used in a program remain accessible for use in immediate mode until a SCRATCH, CLEAR or another RUN command is executed.

The RUNNH command also executes the program in core but suppresses the header line.

Example:

```
RUN
PROG1  03-JUN-73   BASIC V01-05
10

RUNNH
10
```

7.7 CLEAR COMMAND

The CLEAR command clears the contents of the user array and string buffers. This command is generally used when a program has been executed and then edited. Before it is rerun, the array and string buffers are set to zeros and nulls by the CLEAR command to provide more memory.

These buffers will be filled again when the RUN command is executed.

Example:

```
10 A=10
20 PRINT A
CLEAR

READY

RUNNH
10

READY
```

7.8 RENAME COMMAND

The RENAME command assigns the specified name to the program currently in memory. The form of the command is:

RENAME "filnam"

followed by the RETURN key. The filnam is optional and if not specified BASIC responds with

FILE NAME--

Type the 1 to 6 character program name (don't enclose the name in quotation marks) followed by a carriage return. If a device or extension are specified with the file name they are ignored. The characters in the program name may consist of A-Z or 1-9.

If more than 6 characters are entered, the excess characters are ignored. Blanks are also ignored. If no name is specified in answer to the FILE NAME message, the default name, NONAME, is used. The program itself does not change.

7.9 NEW COMMAND

The NEW command clears the storage area set up by BASIC (same as SCRATCH) and assigns the specified name to the program currently in memory (same as RENAME).

The form of the command is:

NEW "filnam"

If the file name is not specified as part of the NEW command, BASIC prints:

NEW FILE NAME--

and waits for the file name and RETURN key to be typed. Type the file name, (do not enclose in quotation marks) and the RETURN key. If specified, device or extension are ignored.

CHAPTER 8

USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

RT-11 BASIC has a facility which allows experienced PDP-11 assembly language programmers to interface their own assembly language routines to BASIC. This facility permits the user to add functions to BASIC which can operate directly on special purpose peripheral devices. This chapter describes in some detail the internal characteristics of BASIC during the execution of a BASIC program, and is intended to serve as a programming guide for the creation of such user-coded assembly language functions. This material assumes the user is familiar with PDP-11 assembly language. For additional information on this subject, refer to the RT-11 System Reference Manual DEC-11-ORUGA-A-D.

The CALL statement is used to reference these assembly language routines from the BASIC program.

8.1 CALL STATEMENT

The CALL statement can be inserted anywhere in the BASIC program and has the form:

CALL string expression (argument list)

Where		
	string expression	specifies the name (up to 4 characters) assigned to the assembly language routine to be called. This name is assigned via the System Function Table, as described in Section 8.2. The routine named must be linked with the BASIC system with the Linker.
	argument list	is the optional list of arguments to the assembly language routine, separated by commas. There may be any number of arguments to a routine, as long as the CALL statement fits on one line. The elements of the argument list are expressions, variable names, and array elements. These may include values passed to the user routine, and variables set by it.

In BASIC without strings, string variables are not allowed but a literal string, enclosed in quotes, may be used in the CALL statement.

Examples:

CALL "AND" (A,B,C)	Calls the routine assigned the name AND in the System Function Table which sets the variable C to the value of A ANDed with the value of B.

CALL "OR" (A,B,C)	Calls the routine named OR, which OR's the values of A and B, storing the result in C.
LET F$="REV" CALL (F$) (A$,B$)	Calls the routine named REV which sets the string B$ equal to the string A$ with the characters in reverse order.

8.2 SYSTEM FUNCTION TABLE

For a routine to be accessible from the CALL statement, it must be defined in the special System Function Table. The first word of the BASICR CSECT contains the address of this table. The table consists of a series of 3-word entries, followed by a 0 byte indicating the end of the table. Each entry defines one user routine. The first two words of the entry contain the ASCII characters of the routine name to be used in the CALL statement. Those names with less than four characters are followed with 0 bytes to fill the remainder of the two words. The third word of the entry contains the address of the function.

The following source program generates a system functon table based on the sample CALL statements in section 8.1:

```
; FUNCTION TABLE DEFINITION
        .GLOBL  ANDFN, ORFN, REVFN
        .CSECT  BASICR
        .WORD   FUNTAB
        .CSECT  FUN1
FUNTAB:
        .ASCII  'AND'           ;ASCII NAME OF FUNCTION
        .BYTE   0               ;(4 BYTES)
        .WORD   ANDFN           ;ADDRESS OF FUNCTION ROUTINE
        .ASCII  'OR'
        .BYTE   0,0
        .WORD   ORFN
        .ASCII  'REV'
        .BYTE   0
        .WORD   REVFN
                                ;INSERT NEW ENTRIES HERE
        .BYTE   0               ;END-TABLE FLAG
        .END
```

To produce a BASIC system with the functions defined in the example, link the following modules with LINK.

BASICR BASICE BASICX	Basic object modules, starting at location 400
FPMP	Object module (floating point math package)
FUN1	Object module, produced from the above source.
FUN2	Object module produced from the source in section 8.3.1.

GETARG — Object module, produced from the source shown in Appendix H.

BASICH — BASIC High object module.

Use the LINK command string:

```
*BASIC=BASICR,FPMP,BASICE,BASICX/B:400/C

*FUN1,FUN2,GETARG,BASICH
```

8.3 WRITING ASSEMBLY LANGUAGE ROUTINES

The user's assembly language routine must interface with the BASIC system to pass its arguments to and from the calling BASIC program.

If the user's routine does not accept a variable number of arguments, then the general subroutines GETARG,STORE, and SSTORE, which are listed in Appendix H, may be used to interface the user routines with BASIC. The routine GETARG checks the syntax of the CALL statement, and the argument types. It accesses the routine arguments as specified in the CALL statement, and stores references to them in a table, addressed by R0.

Argument Type	Stored in table at (R0)
1 - Input numeric expression	two words, the expression value
2 - Output numeric target variable	three words, used by STORE subroutine
3 - Input string expression	zero words are stored in table, string pointer is returned on the stack
4 - Output string target variable	three words, used by SSTORE subroutine

To store target variables (argument types 2 and 4), the user routine addresses the corresponding three-word entry in the table set up by GETARG and calls the subroutine STORE for numeric target variables, and SSTORE for string target variables. The examples in section 8.3.1 show how these routines are used.

Once the user routine has called GETARG to reference its arguments, it may use any registers except R5 for calculations. The routine must return via an "RTS PC" instruction, with the stack unchanged.

The GETARG, STORE, and SSTORE subroutines assume that all arguments to the user routines will be in the CALL statement. In the case of a user routine which handles optional arguments, it may use the system subroutines described below in section 8.4 to pass the arguments to and from BASIC. Each of the routines named is a .GLOBL symbol.

When the CALL statement is executed, the user's assembly language routine is called by the instruction:

```
JSR PC, routine address
```

When the user routine is entered, these registers contain information about the calling sequence:

R1 — is a pointer to the translated code of the CALL statement. (See section 8.7 for the format of the translated code.)

If the routine has an argument list, R1 points to the 1-byte token (refer to section 8.7.2 for an explanation of tokens) which represents the left parenthesis in the calling sequence. This token has the value .LPAR.

```
                R1
                ↓
CALL "AND"      (A,B,C)
```

The 1-byte values of code bytes (tokens) .LPAR, .COMMA and .RPAR (right parenthesis) are global symbols. These are not the same as the ASCII representation of these characters.

R4 — Contains the low limit of the stack. If the stack is used heavily, the function must check that it never goes below this limit. (If it does, transfer control to ERRPDL, a global location in BASIC.)

R5 — Contains the address of the "user area", which must be preserved for all calls to BASIC subroutines.

Once the argument references are no longer required by the function R0 through R5 may be used in any way. R0, R2, and R3 need not be preserved in any case.

The function may use the stack, but must return via an

```
RTS PC
```

instruction with the stack unchanged.

The user routine can not use the TRAP instruction, as it is reserved for use by the BASIC system program.

A user routine which does not use the GETARG subroutine should verify the syntax of the invoking CALL statement by checking that the left parenthesis, comma and right parenthesis tokens are contained in the code where expected. (.LPAR, .COMMA and .RPAR are the global values of these 1-byte tokens, respectively.)

In general, arguments which are expression values are passed to the user by the subroutine EVAL, as described in section 8.4. The program can then obtain the value of the expression from the floating accumulator or FAC (FAC1(R5) and FAC2(R5)).

Arguments are passed from the user routine back to BASIC by first calling GETVAR to address the target variable and then calling STOVAR for numeric results and STOSVAR for string results to store the new value in the BASIC variable. These routines are also described in section 8.4.

The example in section 8.3.1 contains the code for both of these types of argument transfer.

8.3.1 Sample User Functions

The following source program shows how the routines AND, OR and REV in the example above would interface with the BASIC system to pass their arguments to the calling program. Each of the system subroutines used in the example is described in section 8.4.

```
; FUN2 - SAMPLE USER FUNCTIONS
         .TITLE  FUN2
         .GLOBL  ANDFN, ORFN, REVFN
         .GLOBL  GETARG, STORE, SSTORE
R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
FAC1=40
FAC2=42
;
;
; "AND"  (A,B,C)
ANDFN:   MOV     #TABLE,R0        ;ADDRESS VARIABLE STORAGE AREA
         JSR     PC,GETARG        ;CHECK SYNTAX AND SET ARGS
         .BYTE   1,1,2,0          ;(ARG TYPES)
         .EVEN
         MOV     #FAC1,R3
         ADD     R5,R3            ;ADDRESS FAC1(R5) IN R3
         MOV     A1,R2
         COM     R2
         MOV     B1,(R3)
         BIC     R2,(R3)+         ;FAC1(R5) IS A1 (AND) B1
         MOV     A2,R2
         COM     R2
         MOV     B2,(R3)
         BIC     R2,(R3)          ;FAC2(R5) IS A2 (AND) B2
         MOV     #C,R0            ;ADDRESS C
         JSR     PC,STORE         ;STORE FAC1,FAC2 IN C

         RTS     PC
; "OR"   (A,B,C)
ORFN:    MOV     #TABLE,R0        ;ADDRESS ARGUMENT TABLE
         JSR     PC,GETARG        ;CHECK SYNTAX AND GET ARGS
         .BYTE   1,1,2,0          ;(ARG TYPES)
         .EVEN
         MOV     #FAC1,R3         ;ADDRESS FAC1(R5) IN R3
```

```
        ADD     R5,R3
        MOV     A1,(R3)
        BIS     B1,(R3)+          ;FAC1(R5) IS A1 (OR) B1
        MOV     A2,(R3)
        BIS     B2,(R3)           ;FAC2(R5) IS A2 (OR) B2
        MOV     #C,R0             ;ADDRESS C
        JSR     PC,STORE          ;STORE FAC1,FAC2 IN C
        RTS     PC
; "REV" (A$,B$)
REVFN:  MOV     #TABLE,R0         ;ADDRESS ARG AREA
        JSR     PC,GETARG         ;CHECK SYNTAX AND GET ARGS
        .BYTE   3,4,0             ;(ARG TYPES)


        .EVEN
        CMP     (SP),#-1          ;CHECK NULL STRING

        BEQ     REVX
        CLR     R2
        MOV     (SP),R3
        BISB    (R3)+,R2          ;R2 IS STRING LENGTH
        CMPB    (R3)+,(R3)+       ;R3 ADDRESSES CHARS
REV1:   DEC     R2                ;ADDRESS NEXT PAIR OF BYTES
        MOV     R3,R0             ;TO SWITCH
        ADD     R2,R0
        CMP     R0,R3             ;CHECK DONE--REACHED MIDDLE
        BLOS    REVX
        MOVB    (R0),R1           ;EXCHANGE ANOTHER PAIR
        MOVB    (R3),(R0)         ;OF BYTES
        MOVB    R1,(R3)+
        DEC     R2
        BR      REV1
REVX:   MOV     #B$,R0            ;ADDRESS B$

        JSR     PC,SSTORE         ;STORE STRING ON STACK

        RTS     PC
;
; ARGUMENT AREA
TABLE:
A1:     .WORD   0                 ;VALUE OF A (2 WORDS)
A2:     .WORD   0
B1:     .WORD   0                 ;VALUE OF B (2 WORDS)
B2:     .WORD   0
C:      .WORD   0,0,0             ;ADDRESS OF C (3 WORDS)


;
.=TABLE
                                  ;POINTER TO A$ IS ON STACK
B$:     .WORD   0,0,0             ;ADDRESS OF B$ (3 WORDS)


;
        .END
```

8.4 SYSTEM ROUTINES IN BASIC

The routines described below are all global symbols and are available to the user functions:

Routine Name (Global)	Call	Description
BOMB	TRAP 0 .ASCIZ 'MESSAGE' .EVEN	This routine stops execution of the BASIC program and types the message: ?MESSAGE AT LINE xxxx If the $LONGER option is specified, the '?' character is omitted. BASIC then types the READY message.
ERRPDL	JMP ERRPDL	Called when the stack pointer (SP) goes below the value in R4. Causes execution to halt and types out ?ETC AT LINE xxxxx. There are 20 extra "buffer" words on the stack. If the user routine will definitely not use more than this many words on the stack, the routine need not check for a stack overflow.
ERRSYN	JMP ERRSYN	Syntax error. Stops execution and prints out ?SYN AT LINE xxxxx.
ERRARG	JMP ERRARG	Argument error. Stops execution and prints out ?ARG AT LINE xxxxx.
EVAL	JSR PC,EVAL	Evaluate expression. R1 points to the start of the expression in the code. EVAL sets the carry bit as follows: carry = 0: The expression is numeric. The value of the expression is contained in the floating accumulator (FAC1 and FAC2). carry = 1: A string expression. If the string is non-null, the top of the stack is an indirect pointer to the string. (See section 8.6 for the format of string variables.) If the string is null, the top of the stack is the value 177777. In both cases, R1 is moved to point to the byte following the expression in the code. If it detects an error in the expression, EVAL branches to the appropriate error routine.

Routine Name (Global)	Call	Description
GETVAR	JSR PC,GETVAR	Address variable or array element. R2 must contain the address of the symbol table entry for the variable and R1 must point to the next byte beyond the second byte of the symbol table offset on call. GETVAR looks up and saves the address of the variable reference, so that a subsequent STOVAR or STOSVAR will store a value in the addressed variable. GETVAR destroys the FAC when addressing an array element; R1 is left unchanged unless the variable is subscripted, in which case R1 is advanced past the right parenthesis. To address the symbol table entry, precede the GETVAR call with the code:

```
MOVB    (R1)+,R2    ;FIRST BYTE OF
                    ;OFFSET
BMI     ESYN        ;IF NEGATIVE, ERROR
SWAB    R2
BISB    (R1)+,R2    ;GET 2ND HALF OF
                    ;OFFSET
ADD     (R5),R2     ;ADD BASE OF SYMBOL
                    ;TABLE
```

Routine Name (Global)	Call	Description
MSG	JSR R1,MSG .ASCIZ 'MESSAGE' .EVEN	Print message on console. Prints the ASCII characters specified after the JSR instruction up to the 0-byte. MSG prints only those characters specified in the calling sequence plus padding characters specific to the terminal in use. The calling program must insert a carriage return where required. MSG clears the CTRL/O condition.
STOVAR	JSR PC,STOVAR	Store numeric variable. Stores the FAC in the variable or array element last referenced by GETVAR. If it was a string variable, STOVAR stops execution of the program, and produces the ?NSM error message.
STOSVAR	JSR PC,STOSVAR	Store string variable. Stores the top of the stack in the variable or array element last referenced by GETVAR, and pops one word from the stack. If it was a numeric variable, STOSVAR stops execution of the program and produces the ?NSM error message.

Routine Name (Global)	Call	Description
INT	JSR PC,INT	Integerize the FAC. Sets the value of the FAC to the greatest integer contained in the previous contents of the FAC. The number is expressed in the BASIC integer format if possible.
MAKEST	JSR PC,MAKEST	Make non-null string variable. The top of the stack contains the length of the string to be created. R2 contains an indirect pointer to (the start of the ASCII characters to fill the string) -3. MAKEST returns an indirect pointer to the string on the top of the stack. (Called MAKESTR in sources.)

In addition, the user program may call the following FPMP-11 routines, which are documented in the FPMP-11 User's Manual (DEC-11-NFPMA-A-D).

$POLSH	Enter "Polish Mode"
$IR	Integer-to-Real Conversion
$MLR	Multiply Real
$DVR	Divide Real
$ADR	Add Real
$SBR	Subtract Real
SIN	Sine Function
COS	Cosine Function
SQRT	Square Root Function
ALOG	Logarithm Function (Base e)
ATAN	Arctangent Function
EXP	Exponentiation Function

The following list contains all the .GLOBL symbols available to the user's assembly language routines. Other .GLOBL's may not exist in future releases of BASIC.

GLOBL Symbol	Description
BOMB	Error routine, called by TRAP 0
ERRARG	Argument error
ERRPDL	Stack overflow error
ERRSYN	Syntax error
EVAL	Evaluate expression

GLOBL Symbol	Description	
GETVAR	Address variable	
INT	Integerize floating accumulator	
MAKEST	Create a string	
MSG	Print a message on the terminal	
NUMSGN	Convert from numeric to ASCII	
STOSVAR	Store string variable	
STOVAR	Store numeric variable	
.COMMA	comma token	,
.DQUOT	double quote token	"
.EOL	end-line token	\
.LPAR	left-parenthesis token	(
.RPAR	right-parenthesis token	)
.SQUOT	single quote token	'

The offset of system variables in the "user area" starts at the address contained in R5. The most commonly-used user offsets are described below:

User area offset			Description
SYMBOLS	=	0	Address of symbol table
CODE	=	16	Address of stored program
LINE	=	20	Address of input line buffer
VARSAVE	=	22	Saved symbol table entry address
SS1SAVE	=	24	Saved first array subscript
SS2SAVE	=	26	Saved second array subscript
LINENO	=	30	Line number being executed
FAC1	=	40	Floating accumulator, upper word
FAC2	=	42	Floating accumulator, lower word
PROGNM	=	142	Program name, 6 ASCII bytes

8.5 REPRESENTATION OF NUMBERS IN BASIC

The value stored in the floating accumulator (FAC1(R5) and FAC2(R5)) by EVAL is always two words long: FAC1(R5) contains the high-order, and FAC2(R5), the low-order portion. If FAC1(R5) is non-zero, then the number is stored as a two-word floating-point number, in this format:

Word	Bit(s)	Description
FAC1(R5)	15	Sign bit, set if the number is negative.
	14-7	Exponent, with a bias of 200 octal.
	6-0	The second through eighth significant bits of mantissa. The first significant bit is always an assumed 1.
FAC2(R5)	15-0	The 9th through 24th significant bits of mantissa.

If FAC1(R5) is zero then FAC2(R5) contains the integer value of the number in 2's complement form. Note that the integers from -32,768 to +32,768 do not have a unique representation: they may be stored in the floating-point or integer form. For example, the number represented by

```
FAC1:    40640     ;Floating-point "5"
FAC2:    0
```

has the same value as

```
FAC1:    0
FAC2:    5         ;Integer "5"
```

The subroutine INT, described in sections 6.1.7 and 8.4, converts a number from the floating point representation to an integer.

8.6 REPRESENTATION OF STRINGS IN BASIC

Non-null strings are represented as follows:

Byte(s)	Contents
0	The length of the string, N
1 and 2	An internal "back-pointer" used by BASIC. Do not change this value.
3 to 0(2+N)	The ASCII characters of the string
3+N	The length of the string, N

A null string is not stored in BASIC; rather, the indirect pointer to the string has the value 177777.

8.7 FORMAT OF TRANSLATED BASIC PROGRAM

When the user inputs a BASIC program, the BASIC system does not store the program exactly as it is typed or read from the input file. Instead, it translates the program to an intermediate form which can be used in two different ways. The intermediate code can be "un-translated" by the LIST or SAVE commands to produce an ASCII program which looks very similar to the input program, or the translated code can be very quickly interpreted by the RUN command to provide swift execution of a program under BASIC/RT-11.

8.7.1 Symbol Table Format

As the BASIC program is input, the system builds a symbol table in core at the indirect address 0(R5). There are four different types of symbol table entries, as shown in Table 8-1.

Table 8-1

Symbol Table Entries

Symbol Table Definition	Description
Line Number	This entry is two words long, with this format: Word 1: Line number as an unsigned 16-bit integer. The highest number allowed is 177774 octal or 65,532 decimal. Word 2: The address of the specified line in the stored translated program.
Numeric Scalar	This is five words long, with this format: Word 1: Constant 177775 Word 2: High-order Scalar Value Word 3: Low-order Scalar Value Word 4: Constant 0 Word 5: ASCII scalar name, the second byte is 0 if the name is only one character.
Numeric Array	This entry is five words long, with this format: Word 1: Constant 177776 Word 2: Address of array Word 3: Maximum value of first subscript (SS1MAX below) Word 4: Maximum value of second subscript or -1 if the array is singly-dimensioned

(Continued on next page)

Table 8-1 (Cont.)

Symbol Table Entries

Symbol Table Definition	Description
	Word 5: ASCII name The scalar with the same name as an array is stored internally as the first element of the array. The address of the array is actually the address of this element. The arrays are stored with the first subscript varying the fastest; each element of the array takes up two words. The address of the (M,N) element in the array is the array address plus the quantity: 4*(N*SS1MAX+M+1)
String	This entry is five words long, with this format: Word 1: Constant 177777 Word 2: Array Address, or string pointer, if Word 3=-1 Word 3: Maximum value of first subscript (SS1MAX below), or -1 if not a string array Word 4: Maximum value of second subscript, or -1 if the array is singly-dimensioned or scalar Word 5: ASCII string name, with the '$' character omitted Strings and string arrays are stored as 1-word pointers to the strings, or the flag 177777 for a null string. If a string is dimensioned or used as a string array, the scalar string with the same name is stored as the first entry in an array. Otherwise, the pointer to the scalar string is stored directly in the symbol table entry, as indicated above. The address of the pointer to the (M,N) element in the array is then the array address plus the quantity: 2*(N*SS1MAX+M+1)

8.7.2 Translated Code

After the line is input, the TRAN subroutine is called to translate it to the internal format. TRAN scans the input line from left to right, and translates it as described below.

All references to line numbers or variable names are stored as the two-byte offset into the symbol table of the entry for that variable name. The symbol table entries for all numeric variables are initially scalars, and are changed to dimensioned arrays when the RUN statement is executed. This two-byte offset is, of course, not negative; therefore, it may be distinguished from the "keyword tokens" described below. It is not necessarily aligned to a word boundary.

All sequences of characters used as a single unit by the BASIC language are defined as "Keywords". The following are examples of keywords:

```
LET
INPUT
STEP
+
(
)
SIN(
GO TO
RANDOMIZE
```

TRAN scans the characters in the program line for the occurrence of any of the keywords, disregarding blanks. When one is found, the corresponding 1-byte system "token" is stored in the saved program. Thus, only one byte in the stored program is required to store such keywords as GOSUB and RANDOMIZE. All of the tokens have the high-order bit set.

At the end of every line in the code, there is a special ".EOL" token. At the end of the program there is an ".EOF" token.

The values of the tokens may be found in a listing of BASIC. Since they are only used internally, some of the values may be different for different versions of BASIC.

When an integer literal is encountered in the program following a GOSUB, GO TO, THEN, LIST, or LISTNH keyword, or as the first element on a line, it is stored as a symbol table reference to a line number entry.

When TRAN finds any other literal numeric value in the input program line, it stores it in the translated program in one of the following forms:

1-Byte Literal — An integer constant in the range 0-255 is stored as two bytes in the translated program:

```
Byte 1: constant 375
Byte 2: 1-byte value
```

1-word Literal — An integer constant with an absolute value less than 32,768 which is not in the range 0-255 is stored as three bytes in the translated program:

```
Byte 1: Constant 376
Bytes 2-3: 2-byte value
```

2-word literal — Any other numeric constant is stored as five bytes in the translated program:

```
Byte 1: constant 374
Bytes 2-5: 4-byte floating point value of
           the literal, as described in
           section 8.5.
```

Certain Keywords translate into tokens which are followed by special "extra bytes" when they are translated, as described below.

Keyword	Translated code
' or "	When the first quote character is encountered, TRAN outputs the corresponding token, followed by a .TEXT token, with the value 377. Next follow all of the ASCII characters in the program line up to the closing quote character. Finally, TRAN outputs a 0 byte and a matching close-quote token to the translated program.
FN	A special byte is placed in the translated code after the FN token. It contains a function number to represent the function name, as follows:

Function Number (octal)	Function Name
0	FNA
2	FNB
4	FNC
6	FND
.	.
.	.
.	.
62	FNZ

Keyword	Translated code
NEXT	Ten extra bytes are output to the translated code following the NEXT statement; these are required at execution time for the proper nesting of FOR-NEXT loops.
REM	The REM token in the code is followed by a .TEXT token, and then the remaining characters on the line.

Any sequence of characters which cannot be translated into a token, and is not a symbol table reference or literal, is translated as the .TEXT token, followed by the remaining characters on the line. The BASIC language does not allow a program to have two variable names together without a character in between. If this occurs, the remainder of the line will be translated as described above. When any such translated program line is executed, it will produce a syntax error.

8.8 BACKGROUND ASSEMBLY LANGUAGE ROUTINE

BASIC/RT11 provides for the execution of a "background" assembly language subroutine during its idle-time, that is, when it is waiting for terminal input. An example of such a background routine is one that displays data from an array on a CRT. This array could be filled with data by CALL statements, and displayed by the background subroutine. The background subroutine is called by a JSR PC instruction. It must preserve all register contents, and exit with the stack intact. This subroutine should be of limited duration, such as one loop through the display buffer. In the case of a long idle-time, the subroutine will be evoked many times. The routine may use the same GLOBL symbols as one called by the CALL statement, but there are no arguments passed to or from BASIC.

To use a background subroutine, it must be linked with BASIC, and the address of the subroutine must be specified in the word following the function table address (FNTBL) in the CSECT BASICR. If no background routine is specified, the contents of this word should not be changed. The following source code generates the information necessary to include a background subroutine, BKG:

```
        .CSECT   BASICR

        .=.+2             ;SKIP OVER FNTBL

        .WORD    BKG      ;ADDRESS OF BACKGROUND ROUTINE

        .CSECT   BKGMOD

BKG:                      ;START OF BACKGROUND RTN
          .
          .
          .

        RTS      PC

        .END
```

To create a version of BASIC with this module included, assemble it as the object module BKGMOD. It may then be linked by the LINK command string:

```
*BASIC.BKG=BASICR,FPMP,BASICE,BASICX/B:400/C
*BKGMOD,BASICH
```

CHAPTER 9

ERROR MESSAGES

When BASIC encounters an error, execution of the command or statement in error halts. An error message and then the READY message are printed.

The BASIC error messages are printed in one of the following formats:

```
        message
    or
        message AT LINE xxxxx
```

where xxxxx is the line number of the statement containing the error. Error messages in immediate mode do not include AT LINE xxxxx. Table 9-1 lists the error messages produced by BASIC. Normally the abbreviated message is printed unless long messages are specified at assembly. (Refer to Appendix F.)

Table 9-1

BASIC Error Messages

Abbreviation	Message	Explanation
?ARG	ARGUMENT ERROR AT LINE xxxxx	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
?ATL	ARRAYS TOO LARGE AT LINE xxxxx	There is not enough room in the core available for the arrays specified in the DIM statements.
?BDR	BAD DATA READ AT LINE xxxxx	Item input from DATA statement list by READ statement is bad.
?BRT	BAD DATA-RETYPE FROM ERROR	Item entered to input statement is bad.
?BSO	BUFFER STORAGE OVERFLOW	Not enough room available in file buffers.
?DCE	DEVICE CHANNEL ERROR AT LINE xxxxx	The device channel number specified for a sequential or virtual memory file is out of range (1-7) or has been opened, or OPEN statement tried to open a virtual memory file on a non-file structured device.
?DNR	DEVICE NOT READY	An I/O device referenced by an OLD, SAVE, or PRINT command is not on-line or the file does not contain any legal BASIC program lines.

(Continued on next page)

Table 9-1 (Cont.)

BASIC Error Messages

Abbreviation	Message	Explanation
?DVO	DIVISION BY 0 AT LINE xxxxx	Program attempted to divide some quantity by 0.
?ETC	EXPRESSION TOO COMPLEX AT LINE xxxxx	The expression being evaluated caused the stack to overflow usually because the parentheses are nested too deeply. The degree of complexity that produces this error varies, according to the amount of space available in the stack at the time. Breaking the statement up into several simpler ones eliminates the error.
?FDE	FILE DATA ERROR	Tried to write an element on an integer virtual memory file outside the range (x)<32,768.
?FIO	FILE I/O ERROR	An I/O error occurred. All files are automatically closed.
?FNF	FILE NOT FOUND	The file requested was not found on the specified device.
?FNO	FILE NOT OPEN	The sequential or virtual memory file referenced is not open.
?FTS	FILE TOO SHORT	The sequential file space allocated to an output file is inadequate.
?FWN	FOR WITHOUT NEXT AT LINE xxxxx	The program contains a FOR statement without a corresponding NEXT statement to terminate the loop.
?GND	GOSUBS NESTED TOO DEEPLY AT LINE xxxxx	Program GOSUB nested to more than 20 levels.
?IDF	ILLEGAL DEF AT LINE xxxxx	The define function statement contains an error.
?IDM	ILLEGAL DIM AT LINE xxxxx	Syntax error in a dimension statement.
?ILN	ILLEGAL NOW	Execution of INPUT statement was attempted in immediate mode.
?ILR	ILLEGAL READ	Tried to read on a sequential file open for output.
?LTL	LINE TOO LONG	The line being typed is longer than 120 characters; the line buffer overflows.

(Continued on next page)

Table 9-1 (Cont.)
BASIC Error Messages

Abbreviation	Message	Explanation
?NBF	NEXT BEFORE FOR AT LINE xxxxx	The NEXT statement corresponding to a FOR statement precedes the FOR statement.
?NER	NOT ENOUGH ROOM	There is not enough room on the selected device for the specified number of output blocks.
?NPR	NO PROGRAM	The RUN command has been specified, but no program has been typed in.
?NSM	NUMBERS AND STRINGS MIXED AT LINE xxxxx	String and numeric variables may not appear in the same expression, nor may they be set = to each other; for example, A$=2.
?OOD	OUT OF DATA AT LINE xxxxx	The data list was exhausted and a READ requested additional data.
?OVF	OVERFLOW AT LINE xxxxx	The result of a computation is too large for the computer to handle.
?PTB	PROGRAM TOO BIG	The line just entered caused the program to exceed the user code area.
?RBG	RETURN BEFORE GOSUB AT LINE xxxxx	A RETURN was encountered before execution of a GOSUB statement.
?RPL	USE REPLACE	File already exists. Use REPLACE command.
?SOB	SUBSCRIPT OUT OF BOUNDS AT LINE xxxxx	The subscript computed is greater than 32,767 or is outside the bounds defined in the DIM statement.
?SSO	STRING STORAGE OVERFLOW AT LINE xxxxx	There is not enough core available to store all the strings used in the program.
?STL	STRING TOO LONG AT LINE xxxxx	The maximum length of a string in a BASIC statement is 255 characters.
?SYN	SYNTAX ERROR AT LINE xxxxx	The program has encountered an unrecognizable statement. Common examples of syntax errors are misspelled commands and unmatched parentheses, and other typographical errors.

(Continued on next page)

Table 9-1 (Cont.)

BASIC Error Messages

Abbreviation	Message	Explanation
?TLT	LINE TOO LONG TO TRANSLATE	Lines are translated as entered and the line just entered exceeds the area available for translation.
?UFN	UNDEFINED FUNCTION AT LINE xxxxx	The function called was not defined by the program or was not loaded with BASIC.
?ULN	UNDEFINED LINE NUMBER AT LINE xxxxx	The line number specified in an IF, GO TO or GOSUB statement does not exist anywhere in the program.
?WLO	WRITE LOCKOUT	Tried to write on a sequential or virtual file opened for input only.
?↑ER	↑ ERROR AT LINE xxxxx	The program tried to compute the value A↑B, where A is less than 0 and B is not an integer. This produces a complex number which is not represented in BASIC.

When the message ?DNR AT LINE xxxxx is printed because the device referenced is not on-line, turn the device on and issue a GO TO xxxxx statement. Execution of the program resumes at the line (xxxxx) specified. This message may also indicate that a program file does not contain any legal BASIC program lines.

When the message ?OOD AT LINE xxxxx is printed because the file referenced by an INPUT#1 statement is not ready, prepare the file and issue a GO TO statement to resume execution.

Function Errors

The following errors can occur when a function is called improperly.

?ARG	The argument used is the wrong type. For example, the argument was numeric and the function expected a string expression.
?SYN	The wrong number of arguments was used in a function, or the wrong character was used to separate them. For example, PRINT SIN(X,Y) will produce a syntax error.

In addition, the functions give the errors listed below.

FNa(...)	?UFN	The function a has not been defined (function cannot be defined by an immediate mode statement).
RND or RND(X)		No errors
SIN(X)		No errors

COS(X)		No errors
SQR(X)	?ARG	X is negative
ATN(X)		No errors
EXP(X)	?↑ER	X is greater than 87
LOG(X)	?ARG	X is negative or 0
ABS(X)		No errors
INT(X)		No errors
SGN(X)		No errors
TAB(X)	?ARG	X is not in the range 0<x<256
LEN(A$)		No errors
ASC(A$)	?ARG	A$ is not a string of length 1
CHR$(X)	?ARG	X is not in the range 0<x<256
POS(A$,B$,N)		No errors
SEG$(A$,N1,N2)		No errors
VAL(A$)	?ARG	A$ is not a valid numeric expression
STR$(X)		No errors
TRM$(A$)		No errors
BIN(X$)	?ARG	Character other than blank, 0 or 1 in string
OCT(X$)	?ARG	Character other than blank or 0 through 7

CHAPTER 10

DEMONSTRATION PROGRAMS

PROGRAM #1:

```
50 REM PROGRAM TO CALCULATE E BY AN INFINITE SERIES
100 LET E=1
110 LET I=I+1
120 LET D=1
130 FOR J=1 TO I
140 LET D=D*J
150 NEXT J
160 LET E=E+1/D
170 PRINT E
180 GO TO 110
999 END

RUNNH

 2
 2.5
 2.66666
 2.70833
 2.71666
 2.71805
 2.71825
 2.71827
 2.71828
 2.71828
 2.71828
 2.71828
```

PROGRAM #2:

```
50 REM PROGRAM TO ROUND OFF DECIMAL NUMBERS
100 PRINT "WHAT NUMBER DO YOU WISH TO ROUND OFF";
110 INPUT N
120 PRINT "TO HOW MANY PLACES";
130 INPUT Y
140 PRINT
150 LET A=INT(N*10↑Y+0.5)/(10↑Y)
160 PRINT N "=" A "TO" Y "DECIMAL PLACES."
170 PRINT
180 GO TO 100
190 END

RUNNH
WHAT NUMBER DO YOU WISH TO ROUND OFF?56.0237
TO HOW MANY PLACES?2

56.0237 = 56.02 TO 2 DECIMAL PLACES.

WHAT NUMBER DO YOU WISH TO ROUND OFF?8.449
TO HOW MANY PLACES?1
```

```
8.449 = 8.4 TO 1 DECIMAL PLACES.

WHAT NUMBER DO YOU WISH TO ROUND OFF?3.685
TO HOW MANY PLACES?2

3.685 = 3.69 TO 2 DECIMAL PLACES.

WHAT NUMBER DO YOU WISH TO ROUND OFF?3.67449
TO HOW MANY PLACES?2

3.67499 = 3.67 TO 2 DECIMAL PLACES.
```

PROGRAM #3:

```
5 REM PROGRAM TO PLOT SINE WAVE
10 FOR X=0 TO 19 STEP .25
20 LET Q=30+30*SIN(X)
30 FOR B=1 TO Q
40 PRINT " ";
50 NEXT B
60 PRINT "X"
70 NEXT X
80 END
```

PROGRAM #4:

The following BASIC program uses another BASIC program file as data, and resequences its line numbers.

```
90 REM - PROGRAM TO RESEQUENCE BASIC PROGRAMS
100 DIM L(500),M(500),K$(2)
110 READ D
120 DATA 500
130 READ K$(0),K$(1),K$(2)
140 DATA "GO TO ","THEN ","GOSUB "
150 PRINT "RESEQUENCE"
160 PRINT "OLD FILE";
170 INPUT P$
180 PRINT "NEW FILE";\REM - MAY HAVE SAME NAME
190 INPUT Q$
200 PRINT "START INPUT LINE, START OUTPUT LINE, INTERVAL SIZE";
210 INPUT L0,L1,I1
220 IF Q$<>"" THEN 230 \LET Q$=P$
230 LET P$=P$&".BAS"
240 LET Q$=Q$&".BAS"
260 IF L1<>0 THEN 270 \LET L1=10
270 IF I1<>0 THEN 280 \LET I1=10
280 OPEN P$ AS FILE #1
290 LET C=-1
300 IF END #1 THEN 410
310 INPUT #1:L$
320 LET L2=L2+1
330 LET T=POS(L$," ",1)
340 LET S$=SEG$(L$,1,T-1)
350 LET S=VAL(S$)
```

```
360 IF S<L0 THEN 300
370 LET C=C+1
380 IF C>D THEN 2000
390 LET L(C)=S
400 GO TO 300
410 LET S=INT(L1)
420 FOR I=0 TO C
430 LET M(I)=S
440 IF S>65530 THEN 2010
450 LET S=S+I1
460 NEXT I
470 RESTORE #1
480 OPEN Q$ FOR OUTPUT AS FILE #2
490 OPEN "LP:" FOR OUTPUT AS FILE #3
500 FOR I=1 TO L2
510 INPUT #1:L$
520 LET C2=POS(L$," ",1)-1
530 LET C1=1
540 GOSUB 1000
550 FOR J=0 TO 2
560 LET C1=1
570 LET C1=POS(L$,K$(J),C1)
580 IF C1=0 THEN 700
590 LET C1=C1+LEN(K$(J))
600 LET C2=POS(L$," ",C1)-1
610 LET E=POS(L$,"\",C1)
620 IF E<>0 THEN 630 \LET E=256
630 LET Q1=POS(L$,"'",C1)
640 LET Q2=POS(L$,'"',C1)
650 IF C2<>0 THEN 660 \LET C2=E-1
660 IF (E-Q1)*Q1>0 THEN 570
670 IF (E-Q2)*Q2>0 THEN 570
680 GOSUB 1000
690 GO TO 570
700 NEXT J
710 PRINT #2:L$
720 PRINT #3:L$
730 NEXT I
740 PRINT "DONE"\STOP
1000 LET S$=SEG$(L$,C1,C2)
1010 LET S=VAL(S$)
1020 IF S>=L0 THEN 1030 \RETURN
1030 FOR K=0 TO C
1040 IF L(K)=S THEN 1070
1050 NEXT K
1060 RETURN
1070 LET L1$=SEG$(L$,1,C1-1)
1080 LET L3$=SEG$(L$,C2+1,256)
1090 LET L2$=STR$(M(K))
1100 LET L$=L1$&L2$&L3$
1110 RETURN
2000 PRINT "TOO MANY LINES"\STOP
2010 PRINT "LINE NO, TOO BIG"\STOP
2020 END
```

PROGRAM #5:

```
100 PRINT "OCTAL DUMP"\REM THIS PROGRAM PRINTS AN OCTAL
110 PRINT "FILE NAME";\REM DUMP OF THE SPECIFIED FILE
120 INPUT F$
130 PRINT "START BLOCK,#BLOCKS"
140 INPUT B1,B2
190 OPEN F$ FOR INPUT AS FILE VF1%
200 OPEN "LP:" FOR OUTPUT AS FILE #1
210 PRINT #1:"OCTAL DUMP OF FILE ";F$
220 FOR B=B1 TO B1+B2-1
230 PRINT #1
240 PRINT #1:"BLOCK";B
250 FOR L=0 TO OCT'37'
260 LET V=L*16
270 GOSUB 1000
280 PRINT #1:SEG$(V$,4,6);"/";
290 FOR S=0 TO 7
300 LET V=VF1(B*256+L*8+S)
310 GOSUB 1000
320 NEXT S
340 PRINT #1
350 NEXT L
360 PRINT #1
370 NEXT B
380 STOP
1000 LET V1=V\REM                THIS SUBROUTINE CONVERTS
1005 REM                         INTEGER V
1010 LET V$=""\REM               TO ASCII STRING V$,WHICH
1020 LET V1$='0'\REM             IS THE OCTAL VALUE OF V
1030 IF V>=0 THEN 1060 \REM      USES V1,V2,V3,V1$,V2$
1040 LET V1=V1+2↑15\REM          V IS PRESERVED
1050 LET V1$='1'
1060 FOR I=1 TO 5
1070 LET V3=INT(V1/8)
1080 LET V2$=STR$(V1-V3*8)
1090 LET V$=V2$&V$
1100 LET V1=V3
1110 NEXT I
1120 LET V$=V1$&V$
1130 RETURN
9000 END
```

APPENDIX A

BOOTSTRAPPING THE RT-11 SYSTEM

Complete bootstrapping instructions may be found in section 2.1 of the RT-11 SYSTEM REFERENCE MANUAL (DEC-11-ORUGA-C-D). For the user's convenience the instructions for systems with the BM792-YB hardware bootstrap follow:

1. Write-enable unit Ø of the system device.
2. Press the HALT switch.
3. Load 1731ØØ in the Switch Register.
4. Press the ADDR switch.
5. Load 1774Ø6 in Switch Register (177344 for DECtape) and press START.

The system responds with

Enter the DATE command, for example

```
.DA 11-SEP-73
```

and then

```
.R BASIC
```

If the RT-11 system is already in core, just type:

```
.R BASIC
```

APPENDIX B

ASCII CHARACTER SET

ASCII 7-BIT OCTAL CODE	CHAR.
000	NUL
001	SOH
002	STX
003	ETX
004	EOT
005	ENQ
006	ACK
007	BEL
010	BS
011	HT
012	LF
013	VT
014	FF
015	CR
016	SO
017	SI
020	DLE
021	DC1
022	DC2
023	DC3
024	DC4
025	NAK
026	SYN
027	ETB
030	CAN
031	EM
032	SUB
033	ESC
034	FS
035	GS
036	RS
037	US
040	SP
041	!
042	"
043	#
044	$
045	%
046	&
047	'
050	(
051	)
052	*
053	+
054	,
055	-
056	.
057	/

ASCII 7-BIT OCTAL CODE	CHAR.
060	0
061	1
062	2
063	3
064	4
065	5
066	6
067	7
070	8
071	9
072	:
073	;
074	<
075	=
076	>
077	?
100	@
101	A
102	B
103	C
104	D
105	E
106	F
107	G
110	H
111	I
112	J
113	K
114	L
115	M
116	N
117	O
120	P
121	Q
122	R
123	S
124	T
125	U
126	V
127	W
130	X
131	Y
132	Z
133	[
134	
135	]
136	↑
137	←
140	`
141	a
142	b
143	c
144	d
145	e
146	f
147	g
150	h
151	i
152	j

ASCII 7-BIT OCTAL CODE	CHAR.
153	k
154	l
155	m
156	n
157	o
160	p
161	q
162	r
163	s
164	t
165	u
166	v
167	w
170	x
171	y
172	z
173	{
174		
175	}
176	~
177	DEL

APPENDIX C

STATEMENTS, COMMANDS, FUNCTIONS

C.1 RT-11 BASIC STATEMENTS

The following summary of BASIC statements defines the general format for the statement and gives a brief explanation of its use.

CALL "function name" (argument list)
: Used to call assembly language user functions from a BASIC program.

CHAIN "dev:filnam.ext" LINE number
: Terminates execution of user program, loads and executes the specified program starting at the line number if included.

CLOSE [VFn / #n]
: Closes the logical file specified. If no file is specified, closes all files which are open.

DATA data list
: Used in conjunction with READ to input data into an executing program.

DEF FNfunction (argument)=expression
: Defines a user function to be used in the program.

DIM variable(n), variable(n,m),variable$(n),variable$(n,m)
: Reserves space for lists and tables according to subscripts specified after variable name.

END
: Placed at the physical end of the program to terminate program execution.

FOR variable = expression1 TO expression2 STEP expression3
: Sets up a loop to be executed the specified number of times.

GOSUB line number
: Used to transfer control to the first line of a subroutine.

GO TO line number
: Used to unconditionally transfer control to other than the next sequential line in the program.

IF expression rel.op. expression {THEN / GO TO} line number
: Used to conditionally transfer control to the specified line of the program.

IF END #n {THEN / GO TO} line number
: Used to test for end file on sequential input file #n.

INPUT list	Used to input data from the terminal keyboard or papertape reader.
INPUT #expression: list	Inputs from a particular device.
[LET] variable = expression	Used to assign a value to the specified variable(s).
[LET] VFn(i)=expression	Used to set the value of a virtual memory file element.
NEXT variable	Placed at the end of a FOR loop to return control to the FOR statement.

OPEN file [FOR {INPUT / OUTPUT}] [(b)] AS FILE #digit [DOUBLE BUF]

Opens a sequential file for input or output as specified. File may be of the form "dev:filnam.ext" or may be a scalar string variable. The number of blocks can be specified by b.

OPEN file [FOR {INPUT / OUTPUT}] [(b)] AS FILE VFdigitx (dimension) = string length

Opens a virtual memory file for input or output. x represents the type of file: floating point (blank), integer (%), or character strings ($). File may be of the form "dev:fil.ext" or may be a scalar string variable. The number of blocks can be specified by b.

OVERLAY "file descriptor"	Used to overlay or merge program currently in core with specified file and continue execution.
PRINT list	Used to output data to the terminal. The list can contain expressions or text strings.
PRINT "text"	Used to print a message or a string of characters.
PRINT #expression: expression list	Outputs to a particular output device, as specified in an OPEN statement.
PRINT TAB(x);	Used to space to the specified column.
RANDOMIZE	Causes the random number generator to calculate different random numbers every time the program is run.
READ variable list	Used to assign the values listed in a DATA statement to the specified variables.

REM comment	Used to insert explanatory comments into a BASIC program.
RESTORE	Used to reset data block pointer so the same data can be used again.
RESTORE #n	Rewinds the input sequential file #n to the beginning.
RETURN	Used to return program control to the statement following the last GOSUB statement.
STOP	Used at the logical end of the program to terminate execution.

C.2 Commands

The following key commands halt program execution, erase characters or delete lines.

Key	Explanation
ALTMODE	Deletes the entire current line. Echoes DELETED message (same as CTRL/U). On some terminals the ESC key must be used.
CTRL/C	Interrupts execution of a command or program and returns control to the RT-11 monitor. BASIC can be restarted without loss of the current program by using the monitor RE command.
CTRL/O	Stops output to terminal and returns BASIC to READY message when program or command execution is completed.
CTRL/U	Deletes the entire current line. Echoes DELETED message (same as ALTMODE).
←	(SHIFT/O) Deletes the last character typed and echoes a backarrow (same as RUBOUT). On VT05 or LA30 use the underscore (-) key.
RUBOUT	Deletes the last character typed and echoes a backarrow (same as ←).

The following commands list, punch, erase, execute and save the program currently in core.

Command	Explanation
CLEAR	Sets the array and string buffers to nulls and zeroes.
LIST	Prints the user program currently in core on the terminal.
LIST	line number
LIST	-line number
LIST	line number-[END]

Command	Explanation
LIST	line number-line number Types out the specified program line(s) on the terminal.
LISTNH line number LISTNH -line number LISTNH line number-[END] LISTNH line number-line number	Lists the lines associated with the specified numbers but does not print a header line.
NEW "filnam"	Does a SCRatch and sets the current program name to the one specified.
OLD"file"	Does a SCRatch and inputs the program from the specified file.
RENAME "filnam"	Changes the current program name to the one specified.
REPLACE "dev:filnam.ext"	Replaces the specified file with the current program.
RUN	Executes the program in core.
RUNNH	Executes the program in core area but does not print a header line.
SAVE "dev:filnam.ext"	Outputs the program in core as the specified file.
SCRatch	Erases the entire storage area.

C.3 Functions

The following functions perform standard mathematical operations in BASIC.

Name	Explanation
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent of x as an angle in radians in the range + or - pi/2.
BIN(x$)	Computes the integer value from a string of blanks, 1's and 0's.
COS(x)	Returns the cosine of x radians.
EXP(x)	Returns the value of e↑x where e=2.71828.
INT(x)	Returns the greatest integer less than or equal to x.

Name	Explanation
LOG(x)	Returns the natural logarithm of x.
OCT(x$)	Computes an integer value from a string of blanks and digits from 0 to 7.
RND(x)	Returns a random number between 0 and 1.
SGN(x)	Returns a value indicating the sign of x.
SIN(x)	Returns the sine of x radians.
SQR(x)	Returns the square root of x.
TAB(x)	Causes the terminal type head to tab to column number x.

The string functions are:

ASC(x$)	Returns as a decimal number the seven-bit internal code for the one-character string (x$).
CHR$(x)	Generates a one-character string having the ASCII value of x.
DAT$	Returns the current date in the format 07-MAY-73.
LEN(x$)	Returns the number of characters in the string (x$).
POS(x$,y$,z)	Searches for and returns the position of the first occurrence of y$ in x$ starting with the zth position.
SEG$(x$,y,z)	Returns the string of characters in positions y through z in x$.
STR$(x)	Returns the string which represents the numeric value of x.
TRM$(x$)	Returns x$ without trailing blanks.
VAL(x$)	Returns the number represented by the string (x$).

APPENDIX D

GETARG, STORE, SSTORE LISTING

```
; GETARG, STORE, SSTORE :  SUBROUTINES FOR
; LINKAGE OF ASSEMBLER SUBROUTINES TO BASIC
;
        .TITLE  GETARG  29-AUG-73
        .GLOBL  GETARG, STORE
        .GLOBL  EVAL, GETVAR, ERRARG, ERRSYN
        .GLOBL  .LPAR, .COMMA, .RPAR, .EOL
        .GLOBL  STOVAR, .SQUOT, .DQUOT
        .IFNDF  $NOSTR
        .GLOBL  SSTORE, STOSVAR
        .ENDC   ;$NOSTR
        .CSECT  GET
;
;$NOSTR =       1       ;DELETE ';' TO ASSEMBLE FOR
;                       ;BASIC WITH NO STRINGS
R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
NVAL=4
        .IFDF   $NOSTR
NVAL=3
        .ENDC   ;$NOSTR
.TEXT=377
FAC1=40
FAC2=42
VARSAV=22
```

```
;-------------------------------------------------------------
; SUBROUTINE 'GETARG'   CALLED BY MOV  #TABLE,R0
;                                 JSR  PC,GETARG
;                                 .BYTE N1,N2,...,0
;                                 .EVEN
;                       WHERE TABLE IS THE ADDRESS OF A
;                       TABLE TO HOLD THE ARG REFERENCES.
;                       N1,N2,ETC. INDICATE THE ARG TYPES:
;                       1       INPUT NUMERIC EXPRESSION. 2
;                               (THE EXPRESSION VALUE) ARE
;                               STORED IN TABLE.
;                       2       OUTPUT NUMERIC VARIABLE. 3 WORDS
;                               ARE STORED IN TABLE.
;                       STRING VERSION ONLY:
;                       3       INPUT STRING EXPRESSION. NO WORDS
;                               ARE STORED IN TABLE. THE STRING
;                               POINTER IS ON THE STACK.
;                       4       OUTPUT STRING VARIABLE. 3 WORDS
;                               ARE STORED IN TABLE.
;                       NO STRING VERSION:
;                       3       INPUT STRING LITERAL. 2 WORDS
;                               ARE STORED IN TABLE. WORD 1 CON-
;                               TAINS THE START OF THE ASCII STRING.
;                               WORD 2 CONTAINS THE LENGTH OF THE
;                               STRING IN BYTES.
;                       CHECKS THE SYNTAX OF THE CALLING
;                       STATEMENT AND FINDS THE REQUESTED
;                       ARGUMENT REFERENCES, STORING THEM
;                       CONSECUTIVELY IN TABLE.
GETARG: MOV     (SP)+,R3        ;ADDR OF CALL IN R3
        MOVB    (R3)+,R2        ;GET 1ST BYTE IN R2
        BLE     GETX            ;NO ARGS, EXIT
        CMPB    (R1)+,#.LPAR    ;CHECK STARTING '('
        BNE     GETERS          ;NO, SYNTAX ERROR
        BR      GET2            ;ENTER LOOP
GET1:   CMPB    (R1)+,#.COMMA   ;CHECK ',' BETWEEN ARGS
        BNE     GETERS          ;NO, SYNTAX ERROR
GET2:   CMP     R2,#NVAL        ;CHECK VALID BYTE
        BHI     GETERA
        ASL     R2
        MOV     R0,R0S          ;SAVE REGS
        MOV     R3,R3S
        MOV     BRTAB-2(R2),PC  ;BRANCH TO ROUTINE
```

```
; NUMERIC EXPRESSION
NUMEXP: JSR     PC,EVAL         ;EVALUATE!
        BCS     GETERA          ;STRING IS BAD
        MOV     R0S,R0          ;RESTORE TABLE POINTER
        MOV     FAC1(R5),(R0)+
        MOV     FAC2(R5),(R0)+  ;SAVE VALUE
        BR      NXTARG
; STRING EXPRESSION
STREXP:
        .IFNDF  $NOSTR
        JSR     PC,EVAL         ;EVALUATE!
        BCC     GETERA          ;NUMERIC IS BAD
        MOV     R0S,R0          ;RESTORE TABLE POINTER
        BR      NXTARG
        .ENDC   ;$NOSTR
        .IFDF   $NOSTR
        MOVB    (R1)+,-(SP)     ;LOOK FOR STRING LITERAL
        CMPB    (SP),#.SQUOT    ;CHECK QUOTE CHAR.
        BEQ     STR1
        CMPB    (SP),#.DQUOT
        BNE     GETERS
STR1:   CMPB    (R1)+,#.TEXT    ;CHECK .TEXT TOKEN NEXT
        BNE     GETERS
        MOV     R0S,R0          ;RESTORE TABLE POINTER
        MOV     R1,(R0)+        ;SAVE STRING ADDRESS IN TABLE
        CLR     R2              ;NOW FIND LENGTH
STR2:   TSTB    (R1)+           ;END OF STRING IS BYTE 00
        BEQ     STR3
        INC     R2              ;COUNT
        BR      STR2
STR3:   MOV     R2,(R0)+        ;SAVE LENGTH IN TABLE
        CMPB    (SP)+,(R1)+     ;CHECK MATCHING CLOSE QUOTE
        BNE     GETERS
        BR      NXTARG
        .ENDC   ;$NOSTR
; NUMERIC TARGET VARIABLE
NUMVAR: CLR     -(SP)           ;REMEMBER IT'S NUMERIC
        .IFNDF  $NOSTR
        BR      VAR1
; STRING TARGET VARIABLE
STRVAR: MOV     R2,-(SP)        ;REMEMBER IT'S STRING
        .ENDC   ;$NOSTR
VAR1:   MOVB    (R1)+,R2        ;GET SYMTAB REF IN R2
        BMI     GETERS
        SWAB    R2
        BISB    (R1)+,R2
        ADD     (R5),R2
        JSR     PC,GETVAR       ;ADDRESS VARIABLE
        MOV     R0S,R0          ;RESTORE TABLE POINTER
        MOV     R5,R2           ;ADDRESS VARSAV
        ADD     #VARSAV,R2
        MOV     (R2),R3         ;SAVE A COPY
        MOV     (R2)+,(R0)+     ;MOVE 3 WORDS INTO TABLE
        MOV     (R2)+,(R0)+
        MOV     (R2),(R0)+
        TST     (SP)+           ;STRING OR NUM
        BNE     VAR2
        CMP     (R3),#-1        ;NUMERIC, CHECK TYPE AGREES
        BEQ     GETERA
        BR      NXTARG
VAR2:   CMP     (R3),#-1
        BNE     GETERA
```

```
; GO TO NEXT ARGUMENT
NXTARG: MOV     R3S,R3
        MOVB    (R3)+,R2        ;GET NEXT BYTE IN R2
        BGT     GET1            ;LOOP TILL BYTE IS 0
        CMPB    (R1)+,#.RPAR    ;CHECK CLOSING ')'
        BNE     GETERS
GETX:   CMPB    (R1)+,#.EOL     ;AND END-LINE TOKEN
        BNE     GETERS
        INC     R3              ;MAKE SURE R3 IS EVEN
        ASR     R3
        ASL     R3
        JMP     (R3)
R0S:    .WORD   0
R3S:    .WORD   0
GETERA: JMP     ERRARG
GETERS: JMP     ERRSYN
BRTAB:  .WORD   NUMEXP
        .WORD   NUMVAR
        .WORD   STREXP
        .IFNDF  $NOSTR
        .WORD   STRVAR
        .ENDC   ;$NOSTR
```

```
;-------------------------------------------------------------
; SUBROUTINE 'STORE'      CALLED BY JSR PC,STORE
;                         R0 POINTS TO 3-WORD ARG REFERENCE
;                         SET UP BY GETVAR
;                         SAVES THE VALUE OF THE FAC
;                         IN THE SPECIFIED NUMERIC VARIABLE
STORE:  MOV     R5,R2                   ;ADDRESS VARSAV
        ADD     #VARSAV,R2
        MOV     (R0)+,(R2)+             ;MOVE FROM TABLE TO USER AREA
        MOV     (R0)+,(R2)+
        MOV     (R0),(R2)
        JSR     PC,STOVAR               ;STORE IT
        RTS     PC
;
        .IFNDF  $NOSTR
;-------------------------------------------------------------
; SUBROUTINE 'SSTORE'     CALLED BY JSR PC,SSTORE
;                         R0 POINTS TO 3-WORD ARG REFERENCE
;                         SET UP BY GETVAR
;                         STRING POINTER IS AT THE TOP OF STK
;                         SAVES THE STRING AT TOP OF STK
;                         IN THE SPECIFIED STRING VARIABLE
SSTORE: MOV     R5,R2
        ADD     #VARSAV,R2              ;ADDRESS VARSAV
        MOV     (R0)+,(R2)+             ;MOVE FROM TBL TO USER AREA
        MOV     (R0)+,(R2)+
        MOV     (R0),(R2)
        MOV     (SP),R3                 ;SWITCH RETURN & STRING PTR
        MOV     2(SP),(SP)
        MOV     R3,2(SP)
        JSR     PC,STOSVAR              ;STORE STRING
        RTS     PC                      ;RETURN
        .ENDC   ;$NOSTR
        .END
```

APPENDIX E

BASIC ERROR MESSAGES

Abbreviation	Message	Explanation
?ARG	ARGUMENT ERROR AT LINE xxxxx	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
?ATL	ARRAYS TOO LARGE AT LINE xxxxx	There is not enough room in the core available for the arrays specified in the DIM statements.
?BDR	BAD DATA READ AT LINE xxxxx	Item input from DATA statement list by READ statement is bad.
?BRT	BAD DATA-RETYPE FROM ERROR	Item entered to input statement is bad.
?BSO	BUFFER STORAGE OVERFLOW at line xxxxx	Not enough room available in file buffers.
?DCE	DEVICE CHANNEL ERROR AT LINE xxxxx	The device channel number specified for a sequential or virtual memory file is out of range (1-7), or tried to open a virtual memory file on a non-file structured device.
?DNR	DEVICE NOT READY	An OLD command read a file which did not have any BASIC statements.
?DV0	DIVISION BY 0 AT LINE xxxxx	Program attempted to divide some quantity by 0.
?ETC	EXPRESSION TOO COMPLEX AT LINE xxxxx	The expression being evaluated caused the stack to overflow usually because the parentheses are nested too deeply. The degree of complexity that produces this error varies according to the amount of space available in the stack at the time. Breaking the statement up into several simpler ones eliminates the error.
?FDE	FILE DATA ERROR AT LINE xxxxx	Tried to write an element on an integer virtual memory file outside the range (x)<32,768.
?FIO	FILE I/O ERROR AT LINE xxxxx	An I/O error occurred. All files are automatically closed.

Abbreviation	Message	Explanation
?FNF	FILE NOT FOUND AT LINE xxxxx	The file requested was not found on the specified device.
?FNO	FILE NOT OPEN AT LINE xxxxx	The sequential or virtual memory file referenced is not open.
?FTS	FILE TOO SHORT AT LINE xxxxx	The sequential file space allocated to an output file is inadequate.
?FWN	FOR WITHOUT NEXT AT LINE xxxxx	The program contains a FOR statement without a corresponding NEXT statement to terminate the loop.
?GND	GOSUBS NESTED TOO DEEPLY AT LINE xxxxx	Program GOSUBS nested to more than 20 levels.
?IDF	ILLEGAL DEF AT LINE xxxxx	The DEF statement contains an error.
?IDM	ILLEGAL DIM AT LINE xxxxx	Syntax error in a dimension statement.
?ILN	ILLEGAL NOW	AN ATTEMPT WAS MADE TO EXECUTE AN INPUT statement in immediate mode.
?ILR	ILLEGAL READ AT LINE xxxxx	Tried to open a write-only device for input or tried to read on a sequential file open for output.
?LTL	LINE TOO LONG	The line being typed is longer than 120 characters; the line buffer overflows.
?NBF	NEXT BEFORE FOR AT LINE xxxxx	The NEXT statement corresponding to a FOR statement precedes the FOR statement.
?NER	NOT ENOUGH ROOM AT LINE xxxxx	There is not enough room on the selected device for the specified number of output blocks.
?NPR	NO PROGRAM	The RUN command has been specified, but no program has been typed in.
?NSM	NUMBERS AND STRINGS MIXED AT LINE xxxxx	String and numeric variables may not appear in the same expression, nor may they be set equal to each other as in A$=2.

Abbreviation	Message	Explanation
?OOD	OUT OF DATA AT LINE xxxxx	The data list was exhausted and a READ requested additional data.
?OVF	OVERFLOW AT LINE xxxxx	The result of a computation is too large for the computer to handle.
?PTB	PROGRAM TOO BIG	The line just entered caused the program to exceed the user code area.
?RBG	RETURN BEFORE GOSUB AT LINE xxxxx	A RETURN was encountered before execution of a GOSUB statement.
?RPL	USE REPLACE	File already exists. Use REPLACE command.
?SOB	SUBSCRIPT OUT OF BOUNDS AT LINE xxxxx	The subscript computed is greater than 32,767 or is outside the bounds defined in the DIM statement.
?SSO	STRING STORAGE OVERFLOW AT LINE xxxxx	There is not enough core available to store all the strings used in the program.
?STL	STRING TOO LONG AT LINE xxxxx	The maximum length of a string in a BASIC statement is 255 characters.
?SYN	SYNTAX ERROR AT LINE xxxxx	The program has encountered an unrecognizable statement. Common examples of syntax errors are misspelled commands and unmatched parentheses, and other typographical errors.
?TLT	LINE TOO LONG TO TRANSLATE	Lines are translated as entered and the line just entered exceeds the area available for translation.
?UFN	UNDEFINED FUNCTION AT LINE xxxxx	The function called was not defined by the program or was not loaded with BASIC.
?ULN	UNDEFINED LINE NUMBER AT LINE xxxxx	The line number specified in an IF, GO TO or GOSUB statement does not exist anywhere in the program.
?WLO	WRITE LOCKOUT AT LINE xxxxx	Tried to open a read-only device for output, or tried to write on a sequential or virtual file opened for input only.

Abbreviation	Message	Explanation
?↑ER	↑ERROR AT LINE xxxxx	The program tried to compute the value A↑B, where A is less than 0 and B is not an integer. This produces a complex number which is not represented in BASIC.

Function Errors

The following errors can occur when a function is called improperly.

?ARG	The argument used is the wrong type. For example, the argument was numeric and the function expected a string expression.
?SYN	The wrong number of arguments was used in a function, or the wrong character was used to separate them. For example, PRINT SIN(X,Y) produces a syntax error.

In addition, the functions give the errors listed below.

FNa(...)	?UFN	The function a has not been defined (function cannot be defined by an immediate mode statement).
RND or RND(X)		No errors
SIN(X)		No errors
COS(X)		No errors
SQR(X)	?ARG	X is negative
ATN(X)		No errors
EXP(X)	?↑ER	X is greater than 87
LOG(X)	?ARG	X is negative or 0
ABS(X)		No errors
INT(X)		No errors
SGN(X)		No errors
TAB(X)	?ARG	X is not in the range $0 \le x < 256$
LEN(A$)		No errors
ASC(A$)	?ARG	A$ is not a string of length 1
CHR$(X)	?ARG	X is not in the range $0 \le x < 256$

DAT$		No errors
POS(A$,B$,N)		No errors
SEG$(A$,N1,N2)		No errors
TRM$(A$)		No errors
VAL(A$)	?ARG	A$ is not a valid numeric expression
STR$(X)		No errors
BIN(x$)	?ARG	Character other than blank, 0 or 1 in string
OCT(x$)	?ARG	Character other than blank or 0 through 7 in string

APPENDIX F

ASSEMBLING AND LINKING BASIC

F.1 ASSEMBLING BASIC/RT11

The source program of BASIC/RT11 consists of three source files:
A 16K system is required to assemble BASIC.

```
BASICL.MAC
BASICH.MAC
FPMP.MAC
```

It is necessary to create the files BASICR, BASICE, and BASICX which consist of only one line of code each. They specify the conditionals necessary to assemble BASICL into the three object modules BASICR.OBJ, BASICE.OBJ and BASICX.OBJ.

They are created using the EDIT program, as follows:

```
($) Represents the Altmode key
.R EDIT
*EWBASICR.MAC ($)($)
*IBASICR=1
($) EX ($)($)

.R EDIT
*EWBASICE.MAC ($)($)
*IBASICE=1
($) EX ($)($)

.R EDIT
*EWBASICX.MAC ($)($)
*IBASICX=1
($) EX ($)($)
```

If any other options are desired, include the conditionals for them in these files. For example:

```
$NOSTR=1         ;NO STRINGS
$LONGER=1        ;LONG ERROR MESSAGES
$NOVF=1          ;NO VIRTUAL MEMORY FILES
$NOPOW=1         ;NO POWER-FAIL OPTION
$STKSZ=n         ;PROGRAM STACK SIZE
                 ;IN BYTES (DEFAULT IS
                 ;200 (OCTAL) BYTES
```

If BASIC is to run on an 8K system, the $NOSTR conditional must be specified.

For example, to create a BASIC with no strings, no virtual memory files, and a stack size of 300 (octal) the BASICR, BASICE, and BASICX files should be created using the EDIT program, as follows

```
.R EDIT
*EWBASICR.MAC ($)($)
*IBASICR=1
$NOSTR=1
$NOVF=1
$STKSZ=300
($) EX ($)($)
```

```
.R EDIT
*EWBASICE.MAC ($)($)
*IBASICE=1
$NOSTR=1
$NOVF=1
$STKSZ=300
($) EX ($)($)

.R EDIT
*EWBASICX.MAC ($)($)
*IBASICX=1
$NOSTR=1
$NOVF=1
$STKSZ=300
($) EX ($)($)
```

($) represents the Altmode key.

To assemble Basic, type the following as input to the MACRO Assembler:

```
*BASICR=BASICR,BASICL
*BASICE=BASICE,BASICL
*BASICX=BASICX,BASICL
*BASICH=BASICH
*FPMP=FPMP
```

This produces the five object modules

BASICR	BASIC Root section
BASICE	BASIC Edit overlay
BASICX	BASIC EXecution overlay
FPMP	Floating Point Math Package
BASICH	BASIC High section, with once-only code and optional functions

F.1.1 Floating Point Math Package

Assembly of the FPMP source file produces a "standard" FPMP for BASIC, which runs on any PDP-11, but will not make use of special arithmetic hardware. All of the routines needed for the full complement of BASIC arithmetic functions are included. A non-standard FPMP may be specified, as outlined in the table below:

FPMP Assembly Parameters

Parameter	Default Value	Description
MIN	undefined	Define to eliminate code for BASIC functions SIN, COS, SQR, and ATN. When linked, the functions are listed as "undefined references". However, when executed by a BASIC program, they produce a ?UFN (UNDEFINED FUNCTION) error.

Parameter	Default Value	Description
FPU	undefined	Define to assemble a version for the PDP-11/45 FPU hardware.
EAE	undefined	Defined to assemble for the EAE hardware.
MULDIV	undefined	Define to assemble for the PDP-11/40 extended instruction set (EIS) or the 11/45 processor.

If MIN is defined, then the following parameters may be specified to include the SIN, COS, ATN, and SQR functions, selectively

CND$37	1	Define (only if MIN is specified) to include the code for the SIN and COS functions.
CND$39	1	Define (only if MIN is specified) to include the code for the ATN function.
CND$41	1	Define (only if MIN is specified) to include the code for the SQR function.

To assemble the Floating Point Match Package with conditionals it is necessary to use the EDIT program to either insert the conditionals in the beginning of the FPMP.MAC file or create a new file, FPMPC.MAC which will be assembled with FPMP.MAC. For example, to create the FPMP with the ATN function excluded, with the SIN, COS, and SQR function included, and to run with the EAE hardware, the file FPMPC.MAC is created by the EDIT program, as follows:

```
.R EDIT
*EWFPMPC.MAC $$
*IMIN=1
EAE=1
CND$37=1
CND$41=1
$ EX $$
```

The MACRO assembly instructions would then be:

```
*BASICR=BASICR,BASICL
*BASICE=BASICE,BASICL
*BASICX=BASICX,BASICL
*FPMP=FPMPC,FPMP
```

F.2 LINKING BASIC/RT11

The five object modules (BASICR, BASICE, BASICX, FPMP, BASICH) may be linked with or without an overlay structure. The overlay option has the advantage that sections of BASIC which are not required at the same time occupy the same core space alternately when they are used; the disadvantage is that BASIC will run somewhat slower, and there will be I/O time spent when switching overlay segments in and out of core. When BASIC is linked to run in an 8K system, it must use the overlay option.

To link BASIC without overlays, type the following command string to the Linker (LINK):

```
*BASIC,BASIC=BASICR,FPMP,BASICE,BASICX,BASICH/B:4ØØ
```

To link BASIC with overlays, use this LINK command sequence:

```
*BASIC,BASIC=BASICR,FPMP/T/B:4ØØ/C
TRANSFER ADDRESS =
GO
*BASICE/O:1/C
*BASICX/O:1/C
*BASICH/O:2
```

F.2.1 Linking BASIC/RT11 with User Functions

The System Function Table address used by the CALL statement to link the user's assembly language routines must be set in the first word of the BASICR control section.

The source code for the System Function Table and the actual function routines must be broken into two separate source files. The source file FUN1 consists of the System Function Table definition, with this general outline:

```
                Function entry points

        .GLOBL  FN1, FN2
        .CSECT  BASICR
        .WORD   FUNTAB

        .CSECT  FUN1
FUNTAB: (function table entries for FN1,FN2,...)
```

The source and file FUN2 consists of the code for the function routines, with this general outline:

```
        .GLOBL  FN1,FN2,...
        .CSECT  FUN2

FN1:
                (The user function routines)

FN2:
```

To link BASIC with the user functions in a non-overlay system, type this command string to the Linker:

```
*BASIC=BASICR,FPMP,BASICE,BASICX/B:4ØØ/C

*FUN1,FUN2[,GETARG],BASICH
```

GETARG is the general argument interface module listed in Appendix H. In an overlay system, there are two possible ways in which to link BASIC with the user functions.

If the user function routines contain no data which must be preserved from one function call to the next, that is, if the code for the routines may be refreshed at the beginning of each function call, then the routines may be incorporated into the execution overlay by using this LINK command string:

```
*BASIC,BASIC=BASICR,FPMP,FUN1/T/B:4ØØ/C
TRANSFER ADDRESS =
GO
*BASICE/O:1/C
*BASICX,FUN2[,GETARG]/O:1/C
*BASICH/O:2
```

In this case, the function routines (in the module FUN2) occupy space in the first overlay segment which is normally unused, since the Edit overlay segment (BASICE) is about 250 words longer in the 8K no-string system than the Execution overlay segment (BASICX). These first 250 words of storage are "free" in this case.

In the case where FUN2 may not be read in anew whenever it is used, type this command string to the Linker:

```
*BASIC=BASICR,FPMP,FUN1,FUN2/T/B:4ØØ/C
TRANSFER ADDRESS =
GO
*BASICE/O:1/C
*BASICX[,GETARG]/O:1/C
*BASICH/O:2
```

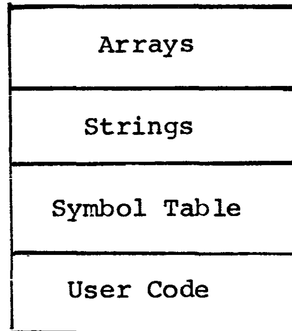
There are three additional object modules (FPMP.FPU, FPMP.EAE, FPMP.EIS) which allow BASIC/RT11 to be linked for special arithmetic hardware.

Processor	Replace FPMP.OBJ With
EAE hardware	FPMP.EAE
PDP-11/40 extended processor or PDP-11/45 processor	FPMP.EIS
PDP-11/45 FPU hardware	FPMP.FPU

APPENDIX G

BASIC CORE MAP

BASIC stores a user program in core in the following format:



The symbol table and user code area are created when the program is entered. When the RUN command is given the user program is scanned and arrays are set up. The string buffer is created during program execution.

The SCRatch command (refer to paragraph 7.1) clears all the user code, symbol table, strings and arrays from core. The CLEAR command clears the arrays and strings but does not affect the user code or symbol table.

The total amount of core storage required to store a BASIC program depends upon the following parameters:

Parameter	Definition	Examples
L	Number of lines in the BASIC program	
K	Number of keywords per line	
R	Number of symbol references per line. There are 3 symbol references in the line:	LET A=B*C+1
S	Total number of symbols used in the program.	
I1	Total number of integer literals in the range 0 x 255	FOR I=1 TO N
I2	Total number of integer literals in the range $-32,768 \leq x \leq 0$ or $256 \leq x \leq 32,767$	LET X=50000
F	Number of non-integer literals and integer literals not in the above ranges	LET Y=X*2.5

Parameter	Definition	Examples
T	Total number of literal strings in the program	LET A$="ABC"
C	Total number of characters inside quotation marks (literal strings)	(C=3 IN THE ABOVE LINE)

The number of bytes required to store the program is then:

    L*(K + 2*R + 7) + 10*S + 2*I1 + 3*I2 + 5*F + 2*T + C+ 1

When the BASIC program is running, the following additional array and string storage is required. For each numeric array, the number of bytes allocated is

    4*(SS1MAX+2)

for a singly-dimensioned array.

or

    4*[(SS1MAX+1)*(SS2MAX+1)+1]

for a doubly-dimensioned array.

Where SS1MAX and SS2MAX are the maximum values of the first and second array subscripts, respectively. For each string array, the number of bytes allocated is

    2*(SS1MAX+2)

for a singly-dimensioned array or

    2*[(SS1MAX+1)*(SS2MAX+1)+1]

for a doubly-dimensioned array.

Where SS1MAX and SS2MAX are the maximum values of the first and second array subscripts, respectively.

For each non-null string scalar or array element of length N currently defined in the BASIC program, N+4 bytes of string storage are required.

INDEX

Arithmetic,
  Functions, 6-1
  Operations, 2-4
ASC Function, 6-15
Assembly,
  Instructions, F-1
  Language routines, 8-1, 8-3
Assignment Statement, 5-2
ATN Function, 6-2

Background Subroutine, 8-15
BIN Function, 6-9
BOMB system routine, 8-7
Buffers,
  I/O, 5-22

CALL Statement, 8-1
CHAIN Statement, 5-20
CHR Function, 6-15
CLEAR command, 7-6
CLOSE Statement, 5-25
Comma usage, 5-6
Command summary, C-3
Commands, Key, 7-1
Concatenation, 3-2
Conditional Transfer, 5-14
Core map, G-1
COS Function, 6-2
CTRL/C Command, 1-2

DAT Function, 6-15
DATA Statement, 5-10
Debugging, Program, 4-1
Demonstration programs, 10-1
~~Dialogue,~~ 1-1
Dimension Statement, 5-3

END Statement, 5-20
ERRARG system routine, 8-7
Error message summary, E-1
Error messages, 9-1
ERRPDL system routine, 8-7
ERRSYN system routine, 8-7
EVAL system routine, 8-7
EXP Function, 6-4
Exponential format, 2-1
Expressions, 2-4

File control, 5-21
Files,
  Sequential, 5-21
  Virtual memory, 5-21
Floating point format, 2-1
FOR loops, nested, 5-17
FOR Statement, 5-15
FPMP routines, 8-9
Function arguments, 6-1
Function selection, 1-1
Function summary, C-4

Functions
  ABS, 6-6
  ASC, 6-15
  ATN, 6-2
  BIN, 6-9
  CHR, 6-15
  COS, 6-2
  DAT, 6-15
  EXP, 6-4
  INT, 6-6
  LEN, 6-15
  LOG, 6-4
  OCT, 6-9
  POS, 6-15
  RND, 6-7
  SEG, 6-15
  SGN, 6-1
  SIN, 6-2
  SQR, 6-3
  STR, 6-16
  TAB, 6-1
  TRM, 6-16
  VAL, 6-16
Functions,
  Arithmetic, 6-1
  Optional, 1-2
  String, 6-15
  User defined, 6-10
  User defined string, 6-16
Functions system routines,
  Sample user, 8-3

GETVAR system routine, 8-8
GO TO Statement, 5-13
GOSUB nesting, 5-19
GOSUB Statement, 5-18

I/O Buffers, 5-22
IF END Statement, 5-14
IF GO TO Statement, 5-14
IF THEN Statement, 5-14
Immediate mode restrictions, 4-2
Immediate statement execution, 4-1
Input device selection, 5-9
INPUT Statement, 5-8
Input/Output Statements, 5-4
INT Function, 6-6
INT system routine, 8-9
Integer Numbers, 2-1

Key Commands, 7-1

Leading and Trailing Zeroes, 2-1
LEN Function, 6-15
LET Statement, 5-2
Linking instructions, F-3

LIST command, 7-3
LISTNH command, 7-3
Load procedure, A-1
Loading BASIC, 1-1
LOG Function, 6-4
Loop, Program, 5-15

MAKEST system routine, 8-9
Monitor,
  Return to the, 1-2
MSG system routine, 8-8
Multiple statements,
  immediate mode, 4-2

Nested FOR loops, 5-17
NEW command, 7-7
NEXT Statement, 5-15
Numbers, 8-11
  Integer, 2-1
  Real, 2-1
NUMSGN system routine, 8-8

OCT Function, 6-9
OLD command, 7-3
OPEN Statement, 5-22
Optional Functions, 1-2
Output device selection,
  5-7
OVERLAY Statement, 5-26

POS Function, 6-15
Power off, 1-3
PRINT Statement, 5-4
Printing Strings, 5-5
Printing Variables, 5-4
Program,
  Control, 5-13
  Debugging, 4-1
  Loop, 5-15
  Termination, 5-20

RANDOMIZE Statement, 5-12
READ Statement, 5-10
Real Numbers, 2-1
Relational operations,
  strings, 3-2
Relational operators, 2-6
REMARK Statement, 5-1
RENAME command, 7-7
REPLACE command, 7-5
RESTORE, 5-11
Restrictions,
  Immediate mode, 4-2
RETURN Statement, 5-18
Return to the Monitor, 1-2
RND Function, 6-7
RUN command, 7-6
RUNNH command, 7-6

Sample user functions
  system routines, 8-3
SAVE command, 7-4
SCRATCH commands, 7-2
SEG Function, 6-15
Semicolon (;) usage, 5-6
Sequential Files, 5-21
SGN Function, 6-1
SIN Function, 6-2
SQR Function, 6-3
Statement, 5-11
  Assignment, 5-2
  CALL, 8-1
  CHAIN, 5-20
  CLOSE, 5-25
  DATA, 5-10
  Dimension, 5-3
  END, 5-20
  FOR, 5-15
  GO TO, 5-13
  GOSUB, 5-18
  IF END, 5-14
  IF GO TO, 5-14
  IF THEN, 5-14
  Immediate execution, 4-1
  INPUT, 5-8
  LET, 5-2
  NEXT, 5-15
  OPEN, 5-22
  OVERLAY, 5-26
  PRINT, 5-4
  RANDOMIZE, 5-12
  READ, 5-10
  REMARK, 5-1
  RETURN, 5-18
  STOP, 5-20
  Summary, C-1
Statements,
  Input/Output, 5-4
STEP values, 5-16
STOP Statement, 5-20
STOSVAR system routine, 8-8
STOVAR system routine, 8-8
STR Function, 6-16
String functions, 6-15
  User defined, 6-16
String operations, 3-2
String Variables,
  Subscripted, 3-1
Strings, 3-1, 8-11
  Printing, 5-5
Subroutine,
  Background, 8-15
Subroutine execution, 5-18
Subscripted String
  Variables, 3-1
Subscripted Variables, 2-2
Symbol table format, 8-12
System function table, 8-2
System routines,
  Sample user functions, 8-3

BASIC/RT11 Language
Reference Manual
DEC-11-LBACA-E-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Performance Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

____
____
____
____
____
____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____
____
____
____
____
____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

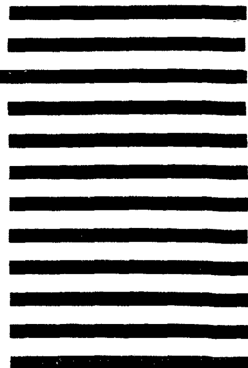
City ______________ State ____________ Zip Code or Country ____________

If you require a written reply, please check here. ☐

Please cut along this line.

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --



BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.