BASIC/RT11
LANGUAGE REFERENCE MANUAL

DEC-11-LBACA-D-D

Order additional copies as directed on the Software Information page at the back of this document.

digital equipment corporation · maynard. massachusetts

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 manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

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

The HOW TO OBTAIN SOFTWARE INFORMATION page, located at the back of this document, explains the various services available to DIGITAL software users.

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:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

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

7/75-14

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).

Technical changes from previous versions are indicated by bars in the outer margin. Changes made after the DEC-11-LBACA-C-D version also have the date of change printed in the lower left hand corner of the page. Any previous change bars on a revised page have been deleted.

CONTENTS

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.

If the computer is turned off while BASIC is operating, the ?PWF error message is output when power is turned on again. The user program is not destroyed and BASIC returns to the READY state with all files closed.

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 will cause BASIC/RT11 to print the symbol "↑". After any character on the keyboard is pressed (it will not be printed) the program will input data from the high speed paper tape reader and store 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:

IF END #n { THEN / GO TO } line number

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	UNACCEPTABLE 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	FOR I1 = 1 TO 10 FOR I2 = 1 TO 10 NEXT I1 NEXT I2
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	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:

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

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.

NOTE

Execution of the OVERLAY statement may cause the data pointer to change. Any program employing both the OVERLAY and DATA statements should have a RESTORE statement executed after the OVERLAY statement. This will cause the data pointer to be at the start of the first DATA statement in the merged 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 180

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 except when the file description is "PR:" in which case BASIC assumes the name PR:.

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

```
OLD
OLD FILE NAME--TEST 1
```

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

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

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

```
OLD "PR:RESEQ"
↑
```

clears user area and inputs RESEQ from the high speed Paper tape Reader after any character is typed in response to the prompt "↑".

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.
```

There are several variations of the LIST command which 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
```

8.2.1 System Function Table When Using LPS or GT Support

When either the Laboratory Peripheral System (LPS) or GT graphic support is also being linked with BASIC, the user written assembly language routines must be defined in FTBL.MAC, a function table supplied in source form.

The following instructions to the RT-11 EDIT program will produce a function table that includes the AND, OR, and REV routines in the previous section and the LPS and/or GT support routines when assembled with PERPAR.MAC as described in the appropriate appendix:

```
.R EDIT
*EBFTBL.MAC$R$$
*FFTBL:$$
*I
        .GLOBL  ANDFN,ORFN,REVFN
        .ASCII  'AND'
        .BYTE   0
        .WORD   ANDFN
        .ASCII  'OR'
        .BYTE   0,0
        .WORD   ORFN
        .ASCII  'REV'
        .BYTE   0
        .WORD   REVFN
$EX$$
```

$ represents the ALTMODE key.

The FTBL object module should be linked with the FUN2 module described in section 8.3.1, GETARG (when needed), PERVEC RTINT, the appropriate LPS and GT object modules and the BASIC object modules.

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)).

THIS PAGE PURPOSELY LEFT BLANK

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 .ASCII 'MESSAGE' .EVEN	This routine stops execution of the BASIC program and types the message: ?MESSAGE AT LINE**** 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 contains the address of the symbol table entry for the variable. 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. 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`

MSG	JSR R1,MSG .ASCII 'MESSAGE' .BYTE 0 .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.
NUMSGN	JSR PC,NUMSGN .WORD ROUTINE	This subroutine converts the number contained in the FAC to ASCII, and saves it via the specified ROUTINE. The ROUTINE is called by a "JSR PC" instruction and preserves all register contents.
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", which starts at the address contained in R5, will not change from release to release; if new ones are added, they will be inserted at the end of the users area. Therefore, these values may be set by MACRO equate statements in the user's source program (e.g. FAC1 = 40). The most commonly-used user area 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

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
```

8.8.1 Background Routine with LPS or GT Support

When either the Laboratory Peripheral System (LPS) or GT graphic support is also being linked with BASIC a user written background routine must be defined in RTINT.MAC. The LPS display module LPS4 is a background routine itself and only one background routine may be linked with BASIC.

The following instructions to the RT-11 EDIT program will define the background routine in RTINT:

(Ⓢ) represents the ALTMODE key.

```
.R EDIT
*EBRTINT.MACⓈRⓈⓈ
*F.WORD FTBLⓈⓈ
*I
        .GLOBL  BKG
        .WORD   BKG
ⓈEXⓈⓈ

.
```

The module defining the background routine should now be of the form

```
        .C SECT BKGMOD
        .GLOBL BKG
BKG:     ;START OF BACKGROUND ROUTINE
           .
           .
           .
         RTS PC
        .END
```

The BKGMOD object module should be linked with FTBL, PERVEC, RTINT, the appropriate LPS and GT object modules, and the BASIC object modules.

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
?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	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.
?PWF	POWER FAIL AT LINE xxxxx	A power fail interrupt occurred while the specified program line was executing. All files are closed.
?RBG	RETURN BEFORE GOSUB AT LINE xxxxx	A RETURN was encountered before execution of a GOSUB statement.
?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
```

```
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
```

```
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$)
```

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-A-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

Appendix D is reserved for future use.

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.
?PWF	POWER FAIL AT LINE xxxxx	A power fail occurred while the specified line was being executed.
?RBG	RETURN BEFORE GOSUB AT LINE xxxxx	A RETURN was encountered before execution of a GOSUB statement.
?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 isn't 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 xxxx	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 $0 \leq x < 256$
LEN(A$)		No errors
ASC(A$)	?ARG	A$ is not a string of length 1
CHR$(X)	?ARG	X is not in the range $0 \leq x < 256$

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
$NO STR=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:400/C

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

GETARG is the general argument interface module listed in Appendix H. In an overlay system, here 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:400/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:400/C
TRANSFER ADDRESS =
GO
*BASICE/O:1/C
*BASICX[,GETARG]/O:1/C
*BASICH/O:2
```

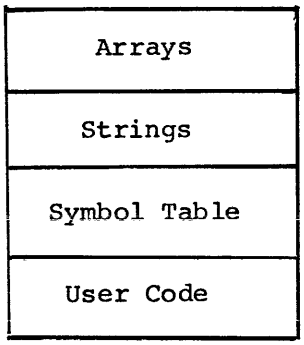
There are three additional object modules (FPMP.FPU, FPMP.EAE, FPMP.EIS) which allow BASIC/RT11 to be linked for special arithmetic hardware.

Processor	Replace FPMP.OBJ With
EAE hardware	FPMP.EAE
PEP-11/ro 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.

APPENDIX I

LABORATORY PERIPHERAL SYSTEM SUPPORT

I.1 INTRODUCTION

LPS support for BASIC/RT11 allows a user to completely utilize the LPS (Laboratory Peripheral System) hardware. LPS support enables the user to sample and display in real-time a variety of data from analog to digital converters, digital input/output, and external events. Sampling is controlled by crystal clocks and/or Schmitt triggers in which the user may specify such parameters as sampling rate and response time thus allowing maximum flexibility.

Since BASIC is a higher level language, even the novice programmer can solve complex data acquistion problems with a minimum amount of effort. All LPS commands are issued by the BASIC CALL statement allowing experienced PDP-11 assembly language programmers to easily include or modify the commands to meet particular (or special) requirements.

I.2 DESCRIPTION OF COMMANDS

BASIC contains 19 commands to control the following options on the LPS hardware:

LPSAD-12	12-bit ADC, sample and hold, 8-channel multiplexer, and LED (Light Emitting Diodes) 6-digit programmable decimal readout display.
LPSAD-NP	Direct memory access (DMA) option for the LPSAD-12 ADC.
LPSAM	8-channel expansion multiplexer.
LPSSH	Second sample and hold for a dual sample and hold configuration.
LPSKW	Programmable real-time clock and two Schmitt triggers.
LPSVC	Display control including two 12-bit DACS. This controller is capable of handling Digital's VR14 and VR20 scopes. Also, Tektronix's RM503, 602, 604, 611, and 613 scopes.
LPSDR	16-bit buffered digital I/O with drive capabilities and two programmable n.o. (normally open) relays.

The 19 commands are divided into 5 categories depending on their function. Each category is supplied as a separate module allowing the user to include only the modules necessary for his experiment.

The following list is a summary of the commands available for controlling LPS hardware and a brief description of each:

MODULE 0 (This module is always required.)

USE	Define array(s) to be used for storage of data.
ACC	Allow access to an entire array.
RDB	Return the next data point from a specified buffer.

MODULE 1 (ANALOG TO DIGITAL CONVERSION)

ADC	Initiate an A/D conversion on a specified channel and return the result to the user.
RTS	Perform real-time buffered/clocked sampling of the A/D.
LED	Display a numeric value on the Light Emitting Diodes.

MODULE 2 (REAL-TIME CLOCK)

SETR	Set clock running at a designated rate and mode.
SETC	Set clock running at a designated rate and initiate some action after a specified number of seconds have elapsed.
HIST	Perform histogram sampling using a timed Schmitt trigger.
WAIT	Wait for a specified event to occur.

MODULE 3 (DIGITAL I/O)

DIR	Read Digital Input register.
DOR	Write Digital Output register.
DRS	Perform sampling of the Digital Input register.
REL	Close or open one of two relays.

MODULE 4 (DISPLAY)

CLRD	Define display buffer and optionally clear or scale the data in it.
PUTD	Put data into data buffer.
DIS	Display data with constant x and variable y whenever BASIC is waiting for I/O.
FSH	Display a complete sweep of data with constant x and variable y.
DXY	Display data with variable x and y values whenever BASIC is waiting for I/O.

Module 0 is the main module and contains not only the USE, ACC, and RDB commands but also all necessary support routines for the other modules. Therefore, it must be included while the other modules are optional.

Data buffers used by the LPS commands differ slightly from the normal arrays in BASIC in that they use only one word of storage per data element rather than two. This is because all LPS data is no larger than 2**16-1 and can be stored as unsigned 16-bit data. All data buffers must first be defined by a USE command before use as a data buffer; however, the USE command allows the user to partition and equivalence arrays for ease in displaying and manipulating common data. All data buffers defined in the USE command are circular with

internal pointers keeping track of where data is to be placed next and/or retrieved.

Section I.8 describes how the user builds a load module containing only the modules necessary for his application.

Section I.9 contains a complete list of all LPS commands and their structures.

Section I.10 describes the options necessary to utilize each command.

Section I.12 contains several example BASIC programs utilizing the LPS commands.

I.3 MODULE 0 (REQUIRED MODULE)

I.3.1 "USE" (A,B,C...)

Define buffer areas for use with the ACC, RDB, RTS, HIST, DRS, CLRD, PUTD, DIS, FSH, and DXY commands. This command sets up internal pointers allowing circular buffering and data overrun and/or nonexistent data checking. A maximum of five buffers may be specified, all of which must be given in a single USE statement. Currently the maximum is set at five; however, through reassembly of module 0, the user can easily change this maximum. All areas defined in the USE must have been previously dimensioned in a DIM statement.

The USE statement allows the user to define buffer areas required for storage of data. These areas may be a partitioned array which can be equivalenced to one large array. The following examples illustrate all aspects of USE. Note that the size of an area defined in a DIM statement is one half that desired. This results because BASIC uses two words to store data whereas the LPS data is stored in one word.

Example:

Define areas A, B, and C to have 100, 200, and 300 data points respectively.

```
10 DIM A(50),B(100),C(150)
20 CALL "USE"(A,B,C)
```

Define area A to consist of three parts, the first having 10 data points and the second and third having 20 each. Then define a final area having access to all of the array A (including the zero subscript element).

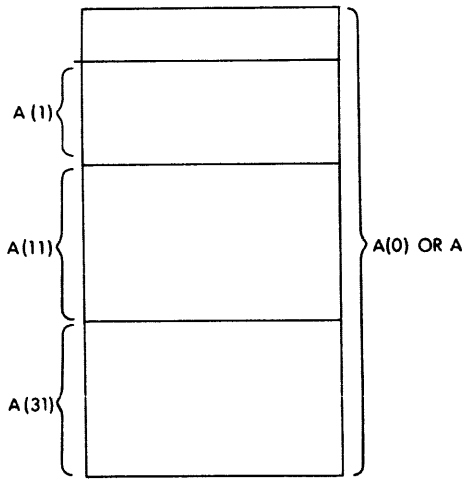
NOTE

Read the USE statement from left to right establishing the separate areas based on previously defined members of the same array. Only when the starting address of the next array is less than the previous one will entire access to the array be allowed by the following array.

The subscripts in the declaration are used to define pseudo partition names which can be used in other LPS statements which require arrays.

```
10 DIM A(25)
20 CALL "USE"(A(1),A(11),A(31),A)
```

In the preceding example, each declaration of the array A has a unique description. A, which is equivalent to A(0), is different than A(1), A(11), and A(31). The following figure illustrates the partitioning of the array A due to the second example.



In the example, the partitioning occurs as follows: A(1) defines a buffer array starting at position A(1) in the array A and ending at the last position in the array. Since A(11) is declared immediately following it, the end of the array for A(1) is redefined to be one less than the A(11) position. At this point, A(1) goes from the 1st position in the array to the 10th, and A(11) goes from the 11th position in the array to the 50th. When A(31) is declared immediately following the A(11), A(11) is redefined to go from the 11th position in the array to the 30th and A(31) goes from position A(31) to the 50th position. Now the partitioned array consists of three parts. The first part is called A(1) and is 10 locations in size. The second part is called A(11) and is 20 locations in size. The third part is called A(31) and is also 20 locations in size. The final declaration in the example is A or A(0) (both are equivalent), which allows access to the entire array A. This happens because the position in the array A of A(0) is less than the last declaration in the USE, i.e., A(31), and a new partitioning is started. This new partitioning begins at A(0) and proceeds until the end of the array A. The rules to continue from this point are the same as previously discussed and further partitioning could be defined if necessary. Note that every declaration in the USE statement must be unique, i.e., a statement of the form "USE"(A,A(0)) results in the first array A having an area of zero length. Since the second array is not unique in name, any reference to it later by other LPS statements actually refers to the array A and not A(0). Since A(0) has zero length, the buffer declaration is useless.

I.3.2 "ACC"(BUF)

Access entire buffer BUF. This command resets all buffer pointers of the array BUF to allow full access to it by the RDB and PUTD commands. The PUTD pointer is placed at the end of the array and the RDB pointer is placed at the beginning.

Example:

Allow full access to the array H and the array A(11).

```
10 DIM A(25),H(20)
20 CALL "USE"(A(1),A(11),A(31),A,H)
30 ....
40 ....
 ..
 ..
 ..
100 CALL "ACC"(H)
110 CALL "ACC"(A(11))
120 ....
130 ....
 ..
 ..
 ..
```

I.3.3 "RDB"(BUF,var)

Return the next data point from the specified buffer. Returns values of 65535>=var>=0 for good data. Bad data (defined as overrun) is returned as a minus one. If no data exists yet, a minus 2 will be returned.

When the referenced buffer refers to analog sampling (RTS function), the values returned are in the range 4095>=var>=0.

When the referenced buffer refers to a clocked histogram sampling (HIST function), the values returned are in the range 65535>=var>=0. These values are either the number of ticks accumulated or the number remaining depending on the clock mode.

When the referenced buffer refers to a Digital I/O operation (DRS function), a value between 65535>=var>=0 is returned from the next position in the specified buffer.

Example:

Assume that the array X has 100 data values previously entered by an RTS command. Print out the data making sure that data overrun did not occur and that 100 data points were indeed taken.

```
 ..
 ..
100 FOR I=1 TO 100
110 CALL "RDB"(X,Z)
120 IF Z >= 0 GO TO 160
130 IF Z =-2 GO TO 180
140 PRINT "BAD DATA AT EVENT";I
150 GO TO 190
160 PRINT Z
```

```
170 GO TO 190
180 PRINT "NO DATA AT EVENT";I
190 NEXT I
 ..
 ..
```

I.4 MODULE 1 (A/D CONVERSION AND NUMERIC READOUTS)

I.4.1 "ADC"(chan,var)

Initiate an A/D conversion from the specified channel (0 =chan =15), wait for it to complete, and return the conversion as a floating point result in "var" (0 =var =4095). The A/D cannot be currently involved in a Real-Time Sampling (RTS) operation.

Example:

Sample the A/D from channels 4 and 5 and save the results in the arrays A4 and A5 respectively. Assume 100 samples are to be taken.

```
10 DIM A4(100),A5(100)
20 FOR I=1 TO 100
30 CALL "ADC"(4,A4(I))
40 CALL "ADC"(5,A5(I))
50 NEXT I
```

I.4.2 "RTS"(BUF,sc,nsc,npts,mode)

Perform real time buffered/clocked sampling of the A/D. The A/D can be enabled in a variety of options depending on the mode specified. The normal mode of operation (mode=0) causes the A/D to sample whenever Schmitt trigger 1 fires. A mode of 2 causes the A/D to sample whenever the clock overflows. To enable other options, merely add their code number to the mode. The following list describes options available (all options are normally disabled):

Code	Option
+1	Enable burst mode (used only with DMA)
+2	Enable clock, disable Schmitt trigger 1
+4	Enable dual sample and hold
+8	Enable DMA

The A/D will be started by a clock overflow or the firing of Schmitt trigger 1. Pointers are used to determine if good data exists in the buffer arrays or if data wraparound occurs. Since data is stored in circular buffers (excluding DMA operations), pointers are used to ensure that the incoming data rate does not exceed the removal rate. Data returned as minus 2 (-2) indicates that data overrun occurred. The buffer pointers are reset initially before the sampling operation begins.

A/D channels are sampled on every clock overflow or firing of Schmitt trigger 1 with the result stored in consecutive data cells. Data is stored in a format identical to that read from the A/D. When a clock

overflow or Schmitt trigger firing occurs, the A/D samples the first channel specified by "sc" and then samples the next "nsc"-1 consecutive channels. Sampling then continues until "npts" clock overflows or Schmitt triggers have been received. If "npts" is specified as zero, any previous RTS sampling will be disabled.

In dual sample and hold mode, the "nsc" parameter is the number of pairs of channels to read per event.

DMA operations may or may not use dual sample and hold. DMA allows direct hardware storage of A/D results into a specified buffer array. A maximum of 4096 samples may be taken at any one time with removal of data allowed only when the buffer is completely filled. The "nsc" parameter is ignored and is considered to be 1.

RTS operations do not interfere with other sampling operations (i.e., DRS and HIST) and all may be in progress simultaneously.

Example:

Set up the A/D to read data from channels 0 through 3 and store the results in the array A. Schmitt triggers are to be used to fire the A/D. Note that a dimension of 100 allows 200 data points. Since 4 channels are to be sampled, 50 Schmitt triggers will be required to complete the request.

```
10 DIM A(100)
20 CALL "USE"(A)
30 CALL "RTS"(A,0,4,100*2/4,0)
```

I.4.3 "LED"(var)

LED will display the floating point value of the variable "var". Up to six positive or five negative digits can be displayed in the LEDs. An optional decimal point may also be included. Numbers which cannot be accurately displayed (i.e., E numbers or 6-digit negative numbers) are shown as all minus signs.

Example:

Display the value 5.632 on the LEDs.

```
10 A = 5.632
20 CALL "LED"(A)
```

or

```
10 CALL "LED"(5.632)
```

I.5 MODULE 2 (REAL-TIME CLOCK)

I.5.1 "SETR"(rate,mode,preset)

Set clock Rate will set the clock running in the specified mode and at the designated rate. The preset value is the clock counter value. The interrupt enable is always set.

Values of Rate

0 No rate selected
1 1 MHz
2 100 kHz
3 10 kHz
4 1 kHz
5 100 Hz
6 Schmitt trigger 1 (meaningful only for HIST operations)
7 Line frequency (50 Hz or 60 Hz)

Values of Mode

0 Single interval mode. Counter counts from preset value to overflow and stops.

1 Repeated interval mode. Counter counts from preset value to overflow, transfers buffer/preset register to the counter, and begins again.

2 External event timing mode. The counter is free running, and a pulse from Schmitt trigger 2 will transfer contents from the counter to the buffer/preset register and then continue counting.

3 Event timing from zero base mode is the same as mode 2 except when the transfer of the counter to the buffer/preset register is done, the counter is cleared and the count begins from zero.

mode+4 To start clock only after Schmitt trigger 1 fires.

Example:

Set the clock running to interrupt once every second. A 100Hz frequency will be used and the clock mode is 1.

```
CALL "SETR"(5,1,100)
```

I.5.2 "SETC"(rate,time)

Set Clock to that specified by "rate" and time. The clock status register is set to rate and will run for time seconds. A clock interrupt will then occur which may be used to initiate any of the clock controlled functions. The time argument is evaluated as ticks = time in seconds times the clock rate specified in rate, e.g., if the clock rate was 10kHz, then ticks = time in seconds times 10kHz. The ticks are entered into the clock preset/buffer register. The clock always runs in mode 0.

Legal values of rate are: 4, 5, and 7 (see SETR for explanation of rates).

Example:

Set the clock to interrupt in 10 seconds using a 100Hz frequency.

```
CALL "SETC"(5,10)
```

I.5.3 "HIST"(BUF,npts)

Histogram - Timed Schmitt trigger. The HIST command inputs "npts" values from the clock preset/buffer register and stores them into the specified buffer, BUF, whenever Schmitt trigger 2 fires. The clock must be in mode 2 or 3 to be meaningful. The RDB function is used to retrieve the data. The buffer is always circular with the data pointers initially reset before the sampling operation begins.

If "npts" is given as zero, the HIST sampling will be disabled.

HIST operations do not interfere with other sampling operations (i.e., RTS and DRS) and all may be in progress simultaneously.

Example:

Collect a timed histogram between external events (Schmitt trigger 2) and store the results in array T. The clock will run at 1kHz and 100 intervals are required.

```
10 DIM T(50)
20 CALL "USE"(T)
30 CALL "HIST"(T,100)
40 CALL "SETR"(4,3,1)
```

I.5.4 "WAIT"(n)

Disable further program execution and wait until the specified event "n" occurs. "n" is defined as follows:

```
n=0       Wait for clock overflow.
n=1       Wait for Schmitt trigger 1 to fire.
n=2       Wait for clock overflow or Schmitt trigger 1 to fire.
n<>0,1,2  Returns immediately.
```

Example:

Wait for clock overflow.

```
10 CALL "SETR" (5,1,100)
20 CALL "WAIT" (0)
```

I.6 MODULE 3 (DIGITAL I/O)

I.6.1 "DIR"(n,var,NEWCSR)

Read Digital Input Register. If n=0, input is four BCD digits converted to a floating point number. If n<>0, then the binary result read from the register is directly converted to a floating point number. The Digital Input Register is read via an internal load request and does not respond to interrupts. The result is placed in "var", where 0<=var<=65535.

The new CSR register setting is returned in NEWCSR.

Example:

Read Digital Input Register as a binary number, convert to a floating point number, and put result into Y.

```
40 CALL "DIR"(1,Y,N)
```

I.6.2 "DOR"(m,n,NEWDOR)

Write Digital Output Register. Selected bits in the register may be set or cleared. If m=0, set bits in register, otherwise clear bits in register. "n" is the bit pattern used to set or clear the Output register. The new result in the Output register is returned as a floating point number in NEWDOR.

Example:

Turn on (set) bits 1 and 2 of the Digital Output Register.

```
40 CALL "DOR"(0,BIN"110",N)
```

Clear Digital Output Register.

```
40 CALL "DOR"(1,OCT"177 777",N)
```

or

```
40 CALL "DOR"(1,-1,N)
```

I.6.3 "DRS"(BUF,mode,npts,M,NEWCSR)

Digital Readout Sampling. Samples the Digital Input Register in a similar fashion as the RTS function. When M=0, each time the clock fires (or Schmitt trigger, which can trigger the clock), the Digital Input Register is read, possibly converted from BCD to binary (depending on the mode), and stored in the circular buffer BUF. The circular buffer pointers are initially reset before the sampling operation begins.

If the DRS is not clock driven, M<>0, it may be driven by digital inputs, i.e. whenever a new value is received by the input register, the value is immediately read in and stored in the buffer BUF. If the

mode=0, then read BCD, otherwise read binary directly. If "npts" is given as zero, the DRS sampling will be disabled.

The new setting of the digital control status register is returned in NEWCSR.

Example:

Read the Digital Input Register once every one tenth of a second for 100 readings and store the results in array A.

```
10 DIM A(50)
20 CALL "USE"(A)
30 CALL "DRS"(A,0,100,0,N)
40 CALL "SETR"(5,1,10)
```

I.6.4 "REL"(s,dir)

Close or open relay "s". This command opens relay "s" (s = 1 or 2) if "dir" is equal to zero, otherwise it closes it.

Example:

Open relay 1 and close relay 2.

```
100 CALL "REL"(1,0)
110 CALL "REL"(2,1)
```

I.7 MODULE 4 (DISPLAY)

I.7.1 "CLRD"(BUF,size,scale)

Define display buffer having fixed delta x values. BUF is the name of the buffer to be displayed, and contains single word values. Values in the range 4095>=value>=0 are displayed while values outside this are not. The size of the buffer is the number of Y points to display and must be <= the number of points defined in the DIM and USE commands. The delta x is calculated as 4096/size and may be fractional.

If scale equals 0, CLRD will set all buffer values to -1 (non-displayable values). If scale does not equal 0, CLRD bypasses the clearing of the array and the original data is multiplied by scale. In either case, the PUTD pointers are reset to point to the beginning of the array. Data is entered into the array through the PUTD function; however, a CLRD must be issued before data is initially transferred to the array.

A CLRD must be issued at least once before issuing the DIS, FSH, or DXY functions.

Example:

Set up the array C to be used as a display buffer having 256 points.

```
10 DIM C(128)
20 CALL "USE"(C)
30 CALL "CLRD"(C,256,0)
```

I.7.2 "PUTD"(BUF,Y)

Put data point Y into BUF in sequential order, where 65535 =Y =0. This function does not initiate a display, but rather just enters data into the specified array.

Example:

Remove 100 data points from the specified digital sampling buffer D, and transfer them to the display buffer Z.

```
80  DIM D(50),Z(50)
90  CALL "USE"(D,Z)
100 FOR I=1 TO 100
110 CALL "RDB"(D,X)
120 CALL "PUTD"(Z,X)
130 NEXT I
```

Example:

Input 100 BCD data points from the digital input register and display them from the same array.

```
100 DIM D(50)
110 CALL "USE"(D)
120 CALL "CLRD"(D,100,0)
130 CALL "DIS"(D,1,1)
140 CALL "DRS"(D,0,100,1,N)
150 STOP
```

I.7.3 "DIS"(BUF,n,i)

Display data from BUF whenever BASIC is idle. The points displayed start with the nth point in the buffer and proceed in increments of i. If i=1, consecutive points starting with the nth one are displayed. If i=2, every other point is displayed, etc.

Example:

Display data from buffer E beginning at the 12th data point and displaying every 3rd point.

```
10 CALL "DIS"(E,12,3)
```

I.7.4 "FSH"(BUF,n,i)

The FSH command is identical to DIS except that the data points in BUF are completely displayed only once. The next BASIC statement is then executed.

Example:

Using the previous example, display 100 cycles of the array E.

```
100 FOR I=1 TO 100
110 CALL "FSH"(E,12,3)
120 NEXT I
```

I.7.5 "DXY"(BUF1,BUF2,n,i)

Display data from BUF1 and BUF2. BUF1 and BUF2 have the x and y values respectively. No delta x is used from the CLRD, otherwise, DXY is identical to DIS and FSH. A CLRD, however, must be issued for BUF2. A CLRD to BUF1 is optional but convenient since it can initialize all values in the array to non-displayable.

Example:

Generate fiducial marks on the display screen of a 256-point display every 16 points. Marks will be 10 points in height. Data will be generated into the arrays X and Y.

```
100 DIM X(128),Y(128)
110 CALL "USE"(X,Y)
120 CALL "CLRD"(X,256,0)
130 CALL "CLRD"(Y,256,0)
140 FOR I=16 TO 256 STEP 16
150 FOR J=1 TO 16
160 IF J > 10 GO TO 200
170 CALL "PUTD"(X,I)
180 CALL "PUTD"(Y,J)
190 NEXT J
200 NEXT I
210 CALL "DXY"(X,Y,1,1)
220 STOP
```

I.8 BUILDING A LOAD MODULE

The Laboratory Peripheral System (LPS) support for BASIC is supplied in six binary relocatable files (on DECpack disk, DECtape, paper tape, or cassette).

LPS0.OBJ	LPS kernel module	Required
LPS1.OBJ	Analog to digital conversion	Optional
LPS2.OBJ	Real-time clock (60hz line frequency)	One is optional
LPS2C.OBJ	Real-time clock (50hz line frequency)	
LPS3.OBJ	Digital input/output	Optional
LPS4.OBJ	Display	Optional

There are also the following files which are provided in source form in all kits:

FTBL.MAC	Function Table Module
PERVEC.MAC	Vector Definition Module
BASINT.MAC	Interface Module
RTINT.MAC	Interface Module for BASIC/RT11 V01
PTSINT.MAC	Interface Module for BASIC/PTS V01
PERPAR.MAC	Parameter file

Software for BASIC/RT11 with LPS support provided on DECpack disk, DECtape, and cassette also contains a running version of BASIC with LPS support:

BASLPS.SAV

BASLPS.SAV is a non-overlaying version of BASIC/RT11 that includes all four optional LPS modules. It may be run by the following RT-11 Monitor Command:

.R BASLPS

At this point the standard BASIC initial dialogue will occur. See Chapter 1 of the BASIC/RT11 Language Reference Manual for a description of the initial dialogue. As part of the initial dialogue BASIC will print:

USER FNS LOADED

This message will occur whenever BASIC has been linked with LPS support.

NOTE

BASIC with LPS support requires a PDP-11 with 16K or more of core.

To build a load module BASLPS.SAV (BASIC with LPS support) first transfer all LPS and BASIC files to the system device with PIP or PIPC. The parameter file PERPAR.MAC is then edited and assembled with FTBL.MAC, PERVEC.MAC and the appropriate interface module. The three object modules produced are then linked with the LPS and BASIC object modules to produce a load module. The specific instructions that are given to the system programs (EDIT, MACRO, and LINK) are given in the examples that follow the general description of load module building.

NOTE

All of the procedures in this section assume that an unaltered PERPAR.MAC is being edited. It is recommended that a copy of the original PERPAR.MAC be made and saved for future use.

The BASINT.MAC interface module should be used with all versions of BASIC except BASIC/RT11 V01 which should have RTINT.MAC used in the place of BASINT. If the display module is not included in the LPS support to be linked, another background routine may be linked with

BASIC but it must be defined in this module. See Section 8.8.1 of the BASIC/RT11 Language Reference Manual for instructions to define the background routine.

For the LPS routines to be accessible from the BASIC CALL statement, the routine must be defined in a System Function Table as described in Section 8.2 of the BASIC/RT11 Language Reference Manual. FTBL.MAC is a function table in source form. If any user written assembly language routines are also linked with BASIC the routines must be defined in this function table. See Section 8.2.1 of the BASIC/RT11 Language Reference Manual for instructions to add the assembly language routine definitions to the Function Table.

PERVEC.MAC is the vector definition module. It defines the hardware addresses of the status registers and the interrupt vectors. The standard hardware address for the LPS interrupt vector is 340 (octal). In PDP-11E10 machines with LPS support, however, the interrupt vector is location 300 (octal). To assemble PERVEC with the interrupt vector at 300 (octal) it is necessary to delete the semicolon before the $V=0 definition in PERPAR.MAC. If the interrupt locations are at another location in core then correct the interrupt addresses by using the system editor to define $V in PERPAR equal to the interrupt address minus 300 (octal). For example. if the LPS interrupt vectors start at 320 (octal) define $V=20 (octal). A listing of PERVEC.MAC is printed at the end of section I.8.1.

PERPAR.MAC is a parameter file, a listing follows:

```
.TITLE PERPAR -- PERIPHERAL SUPPORT PACKAGE PARAMETER MODULE.
;
; DEC-11-LBPAA-A-LA     BASIC KERNEL V02-01
;
; COPYRIGHT (C) 1974
;
; DIGITAL EQUIPMENT CORPORATION
; MAYNARD, MASSACHUSETTS 01754
;
; THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO
; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
; DEC ASSUMES NO RESPONSIBILITY FOR ANY ERRORS THAT
; MAY APPEAR IN THIS DOCUMENT.
;
; THIS SOFTWARE IS FURNISHED TO PURCHASER UNDER A
; LICENSE FOR USE ON A SINGLE COMPUTER SYSTEM AND
; CAN BE COPIED (WITH INCLUSION OF DEC'S COPYRIGHT
; NOTICE) ONLY FOR USE IN SUCH SYSTEM, EXCEPT AS MAY
; OTHERWISE BE PROVIDED IN WRITING BY DEC.
;
```

```
; DEC ASSUMES NO RESPONSIBILITY FOR THE USE
; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
; WHICH IS NOT SUPPLIED BY DEC.

; THE CONDITIONALS CONTAINED IN THIS MODULE AFFECT THE ASSEMBLY
; OF THE FUNCTION TABLE MODULE "FTBL.MAC".
; TO OBTAIN THE DESIRED CONDITIONAL DEFINITION(S),
; REMOVE (USING AN EDITOR) THE
; SEMI-COLON APPEARING BEFORE THE CONDITIONAL.
;$DISK=0                 ;DEFINE FOR RT-11
        .IFNDF $DISK
$STRNG=0                ;DO NOT DEFINE FOR PTS BASIC WITHOUT
                        ;STRINGS,- DEFINED FOR PTS V01 WITH STRINGS
        .ENDC           ;

;$LPS=0                 ;DEFINE FOR LPS
        .IFDF $LPS
;$V=0                   ;DEFINE FOR LPS WITH VECTORS STARTING
;                       ;    AT 300.  DEFAULT SETTING IS VECTORS AT
;                       ;    340. SET $V = ANY OTHER DISPLACEMENT IF
;                       ;       VECTORS START AT DISPLACEMENTS
;                       ;       OTHER THAN 0 OR 40 FROM
;                       ;       VECTOR 300

$ADC=0                  ;INCLUDE A/D ROUTINES.
$CLK=0                  ;INCLUDE CLOCK ROUTINES.
$DIO=0                  ;INCLUDE DIGITAL IO ROUTINES
$DIS=0                  ;INCLUDE DISPLAY ROUTINES.
        .ENDC   ;$LPS
;
;
;$VT11=0                        ;FOR GT40 (GT44)
;
;


        .IFDF   $VT11
$CLOCK=0                        ;FOR SYSTEM CLOCK (KW11L)
        .ENDC

        .EOT
```

To link the LPS module with BASIC it is necessary to delete the semicolons (;) before the $DISK (for RT-11) and the $LPS=0 statements. If any of the four optional modules are not to be included, a semicolon (;) must be inserted before the appropriate conditional.

Parameter	Insert ; before parameter if
$ADC=0	module LPS1 is not to be included.
$CLK=0	module LPS2 (LPS2C) is not to be included.
$DIO=0	module LPS3 is not to be included.
$DIS=0	module LPS4 is not to be included.

Using the system assembler, the sources are assembled in the following combinations to produce the needed LPS object modules:

Object File	Source Files
FTBL	PERPAR,FTBL
PERVEC	PERPAR, PERVEC
BASINT or	PERPAR, BASINT
RTINT	PERPAR, RTINT

After these modules have been reassembled, the LPS support may be linked with the BASIC object modules with only the desired optional LPS modules included in the LINK command strings.

For extremely long programs that do not use string variables, the LPS support may be linked with the no string object modules: BASNSR, BASNSX, and BASNSE. This no string version of BASIC with LPS support will have more core free for program and array storage.

After BASLPS has been linked it may be loaded by the following monitor command:

```
.R BASLPS
```

At this point the standard BASIC initial dialogue will occur. See Chapter 1 of the BASIC/RT11 Language Reference Manual for a description of the initial dialogue.

Example Load Buildings

When editing PERPAR.MAC $DISK=0 should always be enabled for BASIC/RT11, $LPS=0 should be enabled for BASIC with any LPS support, and $ADC=0, $CLK=0, $D10=0, and $DIS=0 should be disabled whenever the appropriate optional LPS module is not to be included. In addition $V=0 should be enabled for any PDP-11 with LPS hardware interrupt located at 300 (octal) instead of 340 (octal). Most PDP-11E10 with LPS require the defining of the $V=0 assembly parameter. For hardware addresses other than 300 or 340 define $V as described in paragraph about PERVEC.MAC.

The procedures for building the following load modules:

BASIC/RT11 with complete LPS support

BASIC/RT11 with complete LPS support and LPS interrupt vectors at location 300 (octal)

BASIC/RT11 with only the ADC and DIS optional display modules.

are described in this section. Linking instructions for both overlaying and non-overlaying versions are given for each. Instructions for linking both LPS and GT support with BASIC are given in Appendix J. Since all editing instructions assume an original PERPAR, PERPAR.BAK, the edit back-up file, is renamed PERPAR.MAC to allow any future load modules to be built from an un-edited PERPAR.MAC.

(S) represents the ALTMODE Key.

Complete Configuration

To build a load module BASLPS.SAV under RT-11 including LPS support and all four optional modules, enter the following command strings:

```
.R EDIT
*EBPERPAR.MAC$R$$
*F;$DISK=0$$0AD$$
*F;$LPS=0$$0AD$$
*EX$$

.R MACRO
*FTBL=PERPAR,FTBL
ERRORS DETECTED: 0
FREE CORE: 15397. WORDS

*PERVEC=PERPAR,PERVEC
ERRORS DETECTED: 0
FREE CORE: 15411. WORDS

*RTINT=PERPAR,RTINT
ERRORS DETECTED: 0
FREE CORE: 15480. WORDS

*^C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*^C
```

```
.R LINK
*BASLPS,BASLPS=BASICR,FPMP,FTBL,PERVEC,RTINT/B:400/T/C
TRANSFER ADDRESS =
GO
*LPS0,LPS1,LPS2,LPS3,LPS4/C
*BASICE/O:1/C
*BASICX/O:1/C
*BASICH/O:2


*↑C

.
```

These instructions will create a BASLPS.SAV with overlaying which has the maximum usable core area. To link a non-overlaying BASLPS.SAV which will have increased execution speed the following commands should be given to Link:

```
.R LINK
*BASLPS,BASLPS=BASICR,FPMP,BASICE,BASICX/B:400/C
*FTBL,PERVEC,RTINT/C
*LPS0,LPS1,LPS2,LPS3,LPS4,BASICH


*↑C

.
```

Complete Configuration-
Interrupt vectors at location 300 (octal)

These instructions are the same as the preceding instructions except that a $V=0 parameter definition in PERPAR.MAC will be enabled.

```
.R EDIT
*EBPERPAR.MAC$R$$
*F;$DISK=0$$0AD$$
*F;$LPS=0$$0AD$$
*F;$V=0$$0AD$$
*EX$$

.R MACRO
*FTBL=PERPAR,FTBL
ERRORS DETECTED: 0
FREE CORE: 15119. WORDS

*PERVEC=PERPAR,PERVEC
ERRORS DETECTED: 0
FREE CORE: 15137. WORDS

*RTINT=PERPAR,RTINT
ERRORS DETECTED: 0
FREE CORE: 15202. WORDS

*↑C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*↑C

.R LINK
*BASLPS,BASLPS=BASICR,FPMP,FTBL,PERVEC,RTINT/B:400/T/C
TRANSFER ADDRESS =
GO
*LPS0,LPS1,LPS2,LPS3,LPS4/C
*BASICE/O:1/C
*BASICX/O:1/C
*BASICH/O:2


*↑C

.
```

This procedure will create the overlaying version. To create the non-overlaying version the following link commands should be given:

```
.R LINK
*BASLPS,BASLPS=BASICR,FPMP,BASICE,BASICX/B:400/C
*FTBL,PERVEC,RTINT/C
*LPS0,LPS1,LPS2,LPS3,LPS4,BASICH


*↑C

.
```

Partial Configuration

To build a load module BASLPS.SAV under RT-11 FOR BASIC/RT11 V01 which includes only the ADC and display routines, enter the following command strings:

```
.R EDIT
*EBPERPAR.MAC$R$$
*F;$DISK=0$$0AD$$
*F;$LPS=0$$0AD$$
*F$CLK=0$$0AI;$$
*F$DIO=0$$0AI;$$
*EX$$

.R MACRO
*FTBL=PERPAR,FTBL
ERRORS DETECTED: 0
FREE CORE: 15437. WORDS

*PERVEC=PERPAR,PERVEC
ERRORS DETECTED: 0
FREE CORE: 15419. WORDS

*RTINT=PERPAR,RTINT
ERRORS DETECTED: 0
FREE CORE: 15488. WORDS

*↑C


.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*↑C

.R LINK
*BASLPS,BASLPS=BASICR,FPMP,FTBL,PERVEC,RTINT/B:400/T/C
TRANSFER ADDRESS =
GO
*LPS0,LPS1,LPS4/C
*BASICE/O:1/C
*BASICX/O:1/C
*BASICH/O:2


*↑C

.
```

This procedure will create an overlaying version of BASLPS.SAV. Or the following command strings may be used to link a non-overlaid version of BASIC with equivalent LPS support:

```
.R LINK
*BASLPS,BASLPS=BASICR,FPMP,BASICE,BASICX/B:400/C
*FTBL,PERVEC,RTINT/C
*LPS0,LPS1,LPS4,BASICH


*↑C

.
```

I.8.1 LPS in Source Form

The Laboratory Peripheral System support may also be purchased in source form. The following eleven source files are provided.

LPS0.MAC
LPS1.MAC
LPS2.MAC
LPS3.MAC
LPS4.MAC
FTBL.MAC
PERVEC.MAC
BASINT.MAC
RTINT.MAC
PERPAR.MAC

The following chart lists the assembly parameters for each module.

Source File	Conditionals	Define for Systems with:
LPS0.MAC	None	
LPS1.MAC	None	
LPS2.MAC	CYC50	50 Hz line frequency (60 Hz is default)
LPS3.MAC	None	
LPS4.MAC	None	
FTBL.MAC	$ADC	LPS1
	$CLK	LPS2
	$DIO	LPS3
	$DIS	LPS4
	$LPS	LPS0 (all systems with LPS support)
	$VT11	GT40 or GT44 support
	$DISK	RT-11
PERVEC.MAC	$LPS	LPS0
	$V	LPS interrupts not at location 340 (octal)
	$VT11	GT40 or GT44 support
BASINT.MAC or RTINT.MAC	$LPS	LPS0
	$DIS	LPS4

To assemble the LPS from the sources all the LPS files should be transferred to the system device using PIP or PIPC, and then the following command should be given to the RT-11 MACRO assembler:

```
.R MACRO
*LPS0=LPS0
ERRORS DETECTED: 0
FREE CORE: 15080. WORDS

*LPS1=LPS1
ERRORS DETECTED: 0
FREE CORE: 14997. WORDS

*LPS2=LPS2
ERRORS DETECTED: 0
FREE CORE: 15105. WORDS

*LPS3=LPS3
ERRORS DETECTED: 0
FREE CORE: 15100. WORDS

*LPS4=LPS4
ERRORS DETECTED: 0
FREE CORE: 14928. WORDS

*^C

.
```

or if the line current is 50Hz the following commands should be used

```
.R EDIT
*EWPARAM.MAC$$
*ICYC50=0$$
*EX$$

.R MACRO
*LPS0=LPS0
ERRORS DETECTED: 0
FREE CORE: 15080. WORDS

*LPS1=LPS1
ERRORS DETECTED: 0
FREE CORE: 14997. WORDS

*LPS2C=PARAM,LPS2
ERRORS DETECTED: 0
FREE CORE: 15105. WORDS

*LPS3=LPS3
ERRORS DETECTED: 0
FREE CORE: 15100. WORDS

*LPS4=LPS4
ERRORS DETECTED: 0
FREE CORE: 14928. WORDS

*^C

.
```

Either of these procedures will produce five object modules: LPS0.OBJ, LPS1.OBJ, LPS2.OBJ (or LPS2C.OBJ), LPS3.OBJ, LPS4.OBJ. The instructions to assemble the other LPS files and the instructions to link the LPS object modules with the BASIC object modules are given in the preceding section. Following is a listing of PERVEC.MAC which contains the interrupt vector location for the LPS and GT40(44) hardware:

```
.TITLE PERVEC VECTOR DEFINITION MODULE FOR BASIC SUPPORT PACKAGES.
;
; DEC-11-LBPVA-A-LA     BASIC KERNEL V02-01
;
; COPYRIGHT (C) 1974
;
; DIGITAL EQUIPMENT CORPORATION
; MAYNARD, MASSACHUSETTS 01754
;
; THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO
; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
; DEC ASSUMES NO RESPONSIBILITY FOR ANY ERRORS THAT
; MAY APPEAR IN THIS DOCUMENT.
;
; THIS SOFTWARE IS FURNISHED TO PURCHASER UNDER A
; LICENSE FOR USE ON A SINGLE COMPUTER SYSTEM AND
; CAN BE COPIED (WITH INCLUSION OF DEC'S COPYRIGHT
; NOTICE) ONLY FOR USE IN SUCH SYSTEM, EXCEPT AS MAY
; OTHERWISE BE PROVIDED IN WRITING BY DEC.
;
; DEC ASSUMES NO RESPONSIBILITY FOR THE USE
; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
; WHICH IS NOT SUPPLIED BY DEC.


; THIS MODULE DEFINES THE HARDWARE ADRESSES USED BY
; SUCH HARDWARE AS THE "LPS", THE "VT11"(GT40) AND THE "LV11".
; IF THE VECTORS FOR THESE DEVICES SHOULD CHANGE
; THIS MODULE MUST BE EDITED TO REFLECT THE CHANGE.


        .IFDF $LPS
        .IFNDF  $V
        $V=40
        .ENDC
        .GLOBL LPSAD,LPSADB,LPSDR,LPSDMA
        .GLOBL LPSCKS,LPSPB,LPSDRS,LPSDIB
        .GLOBL LPSDOR
        .GLOBL LPDISS,LPDISX,LPDISY
        .GLOBL CKLIVA,CKLIP,DRSIVA,DRSIP,LPSIVA,LPSIP


; DEVICE EQUATES:

LPSAD   =       170400  ;LPS A/D STATUS REG.
LPSADB  =       170402  ;LPS A/D BUFFER LED REG.
LPSCKS  =       170404  ;LPS CLOCK STATUS REG.
LPSPB   =       170406  ;LPS CLOCK BUFFER PRESET REG.
LPSDR   =       170410  ;LPS DIGITAL I/O STATUS REG.
LPSDRS  =       LPSDR
LPSDIB  =       170412  ;LPS DIGITAL INPUT REG.
LPSDOR  =       170414  ;LPS DIGITAL OUTPUT REG.
LPDISS  =       170416  ;LPS DISPLAY STATUS REG.
LPDISX  =       170420  ;LPS DISPLAY REG. X
LPDISY  =       170422  ;LPS DISPLAY REG. Y
LPSDMA  =       170436  ;LPS DMA REGG.


; INTERRUPT VECTOR PAIRS:
```

```
CKLIVA  =       304+$V  ;ADR. OF CLOCK INTERRUPT VECTOR
CKLIP   =       306+$V  ;ADR. OF CLOCK INT.     PRIORITY

DRSIVA  =       310+$V  ;ADR. OF DRS INT. VECTOR
DRSIP   =       312+$V  ;ADR. OF DRS INT. PRIORITY.

LPSIVA  =       300+$V  ;ADR. OF THE A/D INT. VECTOR.
LPSIP   =       302+$V  ;ADR. OF THE INT.PRIORITY.

        .ENDC   ;$LPS
        .IFDF   $VT11   ;GT40
        .GLOBL  DPC,DSR,DISX,DISY,GTVECT

DPC     =       172000  ;VT11 DISPLAY PC
DSR     =       DPC+2   ;VT11 DISPLAY STATUS REG
DISX    =       DSR+2   ;VT11 X STATUS REG
DISY    =       DISX+2  ;VT11 Y STATUS REG
GTVECT  =       320     ;ADR. OF VT11 [GT40 (GT44)] INTERRUPT
                        ;VECTOR LIST. REDEFINING GTVECT
                        ;REDEFINES THE ENTIRE SET
                        ;OF DISPLAY PROCESSOR INT. VECTORS.
;GTVECT:                ;DISPLAY STOP VECTOR
;GTVECT+4:              ;LIGHT PEN HIT VECTOR
;GTVECT+10:             ;DISPLAY TIME OUT VECTOR
        .ENDC   ;$VT11


        .END
```

I.9 SUMMARY OF LPS COMMANDS

```
CALL "USE"(A,B,C...)
CALL "ACC"(BUF)
CALL "RDB"(BUF,var)

CALL "ADC"(chan,var)
CALL "RTS"(BUF,sc,nsc,npts,mode)
CALL "LED"(var)

CALL "SETR"(rate,mode,preset)
CALL "SETC"(rate,time)
CALL "HIST"(BUF,npts)
CALL "WAIT"(n)

CALL "DIR"(n,var,NEWCSR)
CALL "DOR"(m,n,NEWDOR)
CALL "DRS"(BUF,mode,npts,M,NEWCSR)
CALL "REL"(s,dir)

CALL "CLRD"(BUF,size,scale)
CALL "PUTD"(BUF,Y)
CALL "DIS"(BUF,n,i)
CALL "FSH"(BUF,n,i)
CALL "DXY"(BUF1,BUF2,n,i)
```

I.10 HARDWARE REQUIRED FOR LPS COMMANDS

The following summary describes the necessary options required in order to fully utilize the LPS system.

COMMAND	HARDWARE REQUIRED
USE	None
ACC	None
RDB	None
ADC	LPSAD-12, LPSAM (for additional 8 channels)
RTS	LPSAD-12, LPSAD-NP (for DMA operations), LPSAM (for additional 8 channels), LPSSH (for dual sample and hold), LPSKW (for real-time clocking and Schmitt triggers).
LED	LPSAD-12
SETR	LPSKW
SETC	LPSKW
HIST	LPSKW
WAIT	LPSKW
DIR	LPSDR
DOR	LPSDR
DRS	LPSDR
REL	LPSDR
CLRD	None
PUTD	None
DIS	LPSVC
FSH	LPSVC
DXY	LPSVC

I.11 LPS ERROR MESSAGES

In addition to the normal BASIC error messages, the following messages concerning LPS commands may also occur:

?NOR	Number out of range.
?BUF	Buffer name given in LPS command has not been previously defined in a USE statement.
?ADC	Cannot issue ADC command while an RTS operation is underway.

I.12 EXAMPLE PROGRAMS

Example Program #1

```
01 REM - EXAMPLE #1 - "LED"
05 REM - PROGRAM TO INPUT NUMBERS FROM THE
10 REM - KEYBOARD AND DISPLAY THEM ON THE
15 REM - LED'S.
20 REM
25 INPUT X
30 CALL "LED"(X)
35 GO TO 25
40 END
```

Example Program #2

```
01 REM - EXAMPLE #2 - "ADC AND LED"
05 REM - PROGRAM TO REQUEST A/D CHANNEL NUMBER
10 REM - FROM KEYBOARD AND THEN USE IT TO GET
15 REM - A CONVERSON FROM THE ADC.  RESULT IS
20 REM - DISPLAYED ON THE LED'S.
25 REM
30 PRINT "CHAN ";
35 INPUT C
40 CALL "ADC"(C,X)
45 CALL "LED"(X)
50 GO TO 30
55 END
```

Example Program #3

```
1 REM - EXAMPLE #3 - "ADC AND LED"
5 REM - PROGRAM TO INPUT CHANNEL NUMBER FROM KEYBOARD
10 REM - AND CONTINUOUSLY DISPLAY THE VALUE OF THE
15 REM - SPECIFIED ADC IN THE LEDS.
20 REM
25 PRINT "CHAN ";
30 INPUT C
35 CALL "ADC"(C,X)
40 CALL "LED"(X)
45 GO TO 35
50 END
```

Example Program #4

```
1 REM - EXAMPLE #4 - "SETC AND WAIT"
5 REM - PROGRAM TO DELAY FOR 5 SECONDS, 10 SECONDS
10 REM - AND 15 SECONDS FOR CLOCK RATES 4, 5, AND 7.
20 REM
25 FOR R=4 TO 7
30 IF R<>6GO TO 40
35 R=7
40 PRINT "RATE ";R
45 FOR I=5 TO 15 STEP 5
50 CALL "SETC"(R,I)
55 CALL "WAIT"(0)
60 PRINT "DELAYED ";I;" SECONDS"
```

```
65 NEXT I
70 PRINT
75 NEXT R
80 END
```

Output for Example Program #4

```
RATE  4
DELAYED  5 SECONDS
DELAYED 10 SECONDS
DELAYED 15 SECONDS

RATE 5
DELAYED  5 SECONDS
DELAYED 10 SECONDS
DELAYED 15 SECONDS

RATE 7
DELAYED  5 SECONDS
DELAYED 10 SECONDS
DELAYED 15 SECONDS
```

Example Program #5

```
1 REM - EXAMPLE #5 - "USE, RTS AND RDB"
5 REM - READ ADC FOR EACH OF 5 OCCURRENCES OF
10 REM - SCHMITT TRIGGER 1.  CHANNELS 0 THRU 3 ARE TO BE
15 REM - SAMPLED WITH RESULTS PRINTED OUT ON THE
20 REM - LINE PRINTER.
25 OPEN "LP:" FOR OUTPUT AS FILE #2
30 DIM A(100)
35 CALL "USE"(A)
40 CALL "RTS"(A,0,4,10*2/4,0)
45 FOR I=1 TO 5
50 CALL "RDB"(A,Z)
55 IF Z<0GO TO 50
60 A1(I)=Z
65 CALL "RDB"(A,Z)
70 A2(I)=Z
75 CALL "RDB"(A,Z)
80 A3(I)=Z
85 CALL "RDB"(A,Z)
90 A4(I)=Z
95 NEXT I
100 FOR I=1 TO 5
105 PRINT # 2:A1(I);A2(I);A3(I);A4(I)
110 NEXT I
115 END
```

Output for Example Program #5

```
 1962   354   532   1962
 2793   354   532   1963
 2795   354   532   1962
 2794   353   532   1962
 2794   353   532   1962
```

Example Program #6

```
01 REM - EXAMPLE #6 - "USE, RDB, PUTD AND ACC"
05 REM - PROGRAM TO ILLUSTRATE PARTITIONING AND
10 REM - EQUIVALENCES.
15 OPEN "LP:" FOR OUTPUT AS FILE #2
20 DIM A(10)
25 CALL "USE"(A(1),A(6),A(11),A(16),A)
30 FOR I=1 TO 5
35 CALL "PUTD"(A(1),I)
40 CALL "PUTD"(A(6),I*10)
45 CALL "PUTD"(A(11),I*100)
50 CALL "PUTD"(A(16),I*1000)
55 NEXT I
60 FOR I=1 TO 6
65 CALL "RDB"(A(1),A1)
70 CALL "RDB"(A(6),A2)
75 CALL "RDB"(A(11),A3)
80 CALL "RDB"(A(16),A4)
85 PRINT #2: A1;A2;A3;A4
90 NEXT I
95 CALL "ACC"(A)
100 FOR I=1 TO 5
105 FOR J=1 TO 5
110 CALL "RDB"(A,X)
115 PRINT #2:X;
120 NEXT J
125 PRINT #2:
130 NEXT I
135 CALL "USE"(A(1),A(6),A(11),A(16),A)
140 FOR I=1 TO 20
145 CALL "PUTD"(A,20-I+1)
150 NEXT I
155 CALL "ACC"(A(1))
160 CALL "ACC"(A(6))
165 CALL "ACC"(A(11))
170 CALL "ACC"(A(16))
175 FOR I=1 TO 5
180 CALL "RDB"(A(1),A1)
185 CALL "RDB"(A(6),A2)
190 CALL "RDB"(A(11),A3)
195 CALL "RDB"(A(16),A4)
200 PRINT #2:  A1;A2;A3;A4
205 NEXT I
210 END
```

Output for Example Program #6

```
  1      10      100     1000
  2      20      200     2000
  3      30      300     3000
  4      40      400     4000
  5      50      500     5000
 -2      -2      -2      -2
  0      1       2       3       4
  5      10      20      30      40
  50     100     200     300     400
  500    1000    2000    3000    4000
  5000   -2      -2      -2      -2
  19     14      9       4
  18     13      8       3
  17     12      7       2
  16     11      6       1
  15     10      5       5000
```

Example Program #7

```
1 REM - EXAMPLE #7 - "USE, PUTD, CLRD AND FSH"
5 REM - PROGRAM TO DISPLAY A PIRAMID ON THE
10 REM - SCOPE AND ALTER ITS SCALE VALUE
15 REM - AFTER EACH 25 SWEEPS ON THE SCREEN.
20 REM
25 DIM A(100)
30 CALL "USE"(A)
35 FOR I=1 TO 100
40 CALL "PUTD"(A,I)
45 NEXT I
50 FOR I=100 TO 200
55 CALL "PUTD"(A,200-I)
60 NEXT I
65 FOR S=1 TO 7
70 CALL "CLRD"(A,200,SQR(S))
75 FOR I=1 TO 25
80 CALL "FSH"(A,1,1)
85 NEXT I
90 NEXT S
95 FOR S=1 TO 5
100 CALL "CLRD"(A,200,1/S)
105 FOR I=1 TO 25
110 CALL "FSH"(A,1,1)
115 NEXT I
120 NEXT S
125 GO TO 65
130 END
```

Example Program #8

```
1 REM - EXAMPLE #8 - "USE, HIST, SETR AND RDB"
5 REM - COLLECT A TIMED HISTOGRAM BETWEEN
10 REM - EXTERNAL EVENTS (SCHMITT TRIGGER 2) AND STORE
15 REM - THE RESULTS IN ARRAY T.  WHENEVER
20 REM - DATA IS READY IN ARRAY T, REMOVE IT
25 REM - IMMEDIATELY AND TRANSFER IT TO ANOTHER
30 REM - ARRAY NAMED Z.  LIST RESULTS ON THE
35 REM - LINE PRINTER WHEN ALL POINTS HAVE BEEN
```

```
40 REM - PROCESSED.
45 OPEN "LP:" FOR OUTPUT AS FILE #2
50 DIM T(5)
55 CALL "USE"(T)
60 CALL "HIST"(T,10)
65 CALL "SETR"(4,2,1)
70 FOR I=1 TO 10
75 CALL "RDB"(T,X)
76 IF X<0GO TO 75
80 Z(I)=X
85 NEXT I
90 FOR I=1 TO 10
95 PRINT #2:Z(I)
100 NEXT I
105 END
```

Output for Example Program #8

```
 4371
 5149
 5150
 5893
 6559
 6560
 7211
 7933
 8664
 10648
```

Example Program #9

```
1 REM - EXAMPLE #9 - "USE, CLRD, PUTD AND DXY"
5 REM - PROGRAM TO DISPLAY A CIRCLE
10 REM
15 DIM X(16),Y(16)
20 CALL "USE"(X,Y)
25 CALL "CLRD"(X,16,0)
30 CALL "CLRD"(Y,16,0)
35 RESTORE
40 FOR I=1 TO 16
45 READ X1,Y1
50 CALL "PUTD"(X,X1)
55 CALL "PUTD"(Y,Y1)
60 NEXT I
65 CALL "DXY"(X,Y,1,1)
70 END
75 DATA 890,2048
80 DATA 1000,2560
85 DATA 1240,2950
90 DATA 1630,3175
95 DATA 2093,3276
100 DATA 2560,3175
105 DATA 2900,2950
110 DATA 3125,2560
115 DATA 3225,2048
120 DATA 3125,1630
125 DATA 2900,1220
130 DATA 2560,920
```

```
135 DATA 2093,820
140 DATA 1630,920
145 DATA 1220,1220
150 DATA 1000,1630
155 END
```

Example Program #10

```
1 REM - EXAMPLE #10 - "USE, RDB, DIR, DOR AND DRS"
5 REM - PROGRAM TO ILLUSTRATE THE DIGITAL I/O.
10 REM - DIGITAL I/O CABLE CONNECTING INPUTS AND
15 REM - OUTPUTS MUST BE INSTALLED FOR PROPER PROGRAM
20 REM - OPERATION.
25 OPEN "LP:" FOR OUTPUT AS FILE #2
30 DIM X(30)
35 CALL "USE"(X)
40 CALL "DOR"(1,65535,N)
45 CALL "DIR"(1,Y,N1)
50 PRINT #2:Y
55 PRINT #2:
56 O=1
60 FOR I=1 TO 16
65 CALL "DOR"(0,O,N)
70 CALL "DIR"(1,Y,N1)
75 PRINT #2:N;Y
80 O=O*2
85 NEXT I
90 PRINT #2:
95 CALL "DOR"(1,65535,N)
100 CALL "DIR"(1,Y,N1)
105 PRINT #2:N;Y
110 PRINT #2:
115 O=0
120 FOR I=1 TO 10
125 CALL "DOR"(1,65535,N)
130 CALL "DOR"(0,O,N)
135 CALL "DIR"(0,Y,N1)
140 PRINT #2:N;Y
145 O=O+4096+256+16+1
150 NEXT I
155 PRINT #2:
160 CALL "DOR"(1,65535,N)
165 CALL "DRS"(X,1,30,0,N)
170 CALL "SETR"(5,1,100)
175 M=0
180 O=0
185 FOR I=1 TO 30
190 CALL "DOR"(1,65535,N)
195 CALL "DOR"(M,O,N)
200 O=O+4096+256+16+1
205 CALL "WAIT"(0)
210 NEXT I
215 GOSUB 300
220 CALL "DRS"(X,0,0,0,N)
225 CALL "DOR"(1,65535,N)
230 CALL "DRS"(X,0,10,1,N)
235 O=1
240 FOR I=1 TO 10
245 CALL "DOR"(0,O,N)
250 O=O*2
255 NEXT I
```

```
260 GOSUB 300
265 END
300 FOR J=1 TO 30
305 CALL "RDB"(X,Y)
310 PRINT #2:J,Y
315 NEXT J
320 RETURN
325 END
```

Output for Example Program #10

```
0
1         1
3         3
7         7
15        15
31        31
63        63
127       127
255       255
511       511
1023      1023
2047      2047
4095      4095
8191      8191
16383     16383
32767     32767
65536     65535
0         0
0         0
4369      1111
8738      2222
13107     3333
17476     4444
21845     5555
26214     6666
30583     7777
34952     8888
39321     9999
1                   0
2                   4369
3                   8738
4                   13107
5                   17476
6                   21845
7                   26214
8                   30583
9                   34952
10                  39321
11                  43690
12                  48059
13                  52428
14                  56797
15                  61166
16                  65535
17                  65535
18                  65535
19                  65535
20                  65535
21                  65535
22                  65535
```

```
23              65535
24              65535
25              65535
26              65535
27              65535
28              65535
29              65535
30              65535
1               1
2               3
3               7
4               15
5               25
6               45
7               85
8               165
9               265
10              465
11             -2
12             -2
13             -2
14             -2
15             -2
16             -2
17             -2
18             -2
19             -2
20             -2
21             -2
22             -2
23             -2
24             -2
25             -2
26             -2
27             -2
28             -2
29             -2
30             -2
```

Example Program #11

```
1 REM - EXAMPLE #11 - "USE, CLRD, PUTD AND DXY"
5 REM - PROGRAM TO GENERATE AND DISPLAY A CIRCLE
10 REM - OF A GIVEN RADIUS WITH ITS CENTER IN THE
15 REM - MIDDLE OF THE SCOPE
20 REM
25 DIM X(128),Y(128)
30 CALL "USE"(X,Y)
32 PRINT "RADIUS";
34 INPUT R
35 CALL "CLRD"(X,256,1)
40 CALL "CLRD"(Y,256,1)
45 CALL "DXY"(X,Y,1,1)
60 T=0
65 D = 2*3.14159/256.
70 FOR I=1 TO 256
75 CALL "PUTD"(X,2048+R*COS(T))
80 CALL "PUTD"(Y,2048+R*SIN(T))
85 T=T+D
90 NEXT I
95 GO TO 32
100 END
```

APPENDIX J

GT GRAPHICS SUPPORT

J.1 INTRODUCTION

BASIC is provided with GT Graphics support for the GT44 and GT40 Display Processors. The support consists of a collection of routines accessible by the CALL statement. These routines allow BASIC programs to have complete control of the display processor.

Points, vectors, text, and graph data may all be combined through simple CALL statements. The screen may easily be scaled to any coordinates. Portions of the display may be controlled independently through use of the subpicture feature. Special graphic routines allow the display of an entire array of data by one call statement. The area of core that is allocated to the display buffer may be dynamically controlled.

When operating in the RT-11 environment, any display may be saved as a file on a mass storage device with the exception of graph arrays. This file may later be restored which will cause the original display to appear on the screen without the BASIC program originally needed to create it.

Support is provided for a real-time clock. The graphics support package will link with and support the Laboratory Peripheral System support that is also provided with BASIC.

The hardware required for use of the BASIC GT Graphics support is a GT40 or GT44 processor, a VT11 display screen, 16K or more of core memory, and a user's terminal. In addition to the peripheral input/output device needed to support the BASIC system (disk, DECtape, cassette, or paper tape), the calls to TIME and TIMR require a real-time clock. The core required for the Graphics support itself is approximately 2.5K in a core resident form and 2.1K in an overlay form.

The documentation for BASIC with Graphics support is provided in two parts the BASIC Manual (BASIC/RT11 Language Reference Manual) and this appendix. All information concerning BASIC arithmetic, strings, operations, functions, statements, and commands may be found in the BASIC Manual. This appendix describes the use of the BASIC calls to the GT Graphic routines. A general description of the CALL statement may be found in section 8.1 of the BASIC/RT11 Language Reference Manual.

The GT Support is supplied in the BASIC kit in the following files:

GTB.OBJ	Main GT object module
GTC.OBJ	GT object module that may be linked in an overlay (otherwise it is linked in core)
PERVEC.MAC	Vector definition source file

FTBL.MAC	Function table
BASINT.MAC	Interface Module
RTINT.MAC	Interface Module for BASIC/RT11 V01
PTSINT.MAC	Interface Module for BASIC/PTS V01
PERPAR.MAC	Parameter file
GTNLPS.OBJ	Module linked with GT when LPS support is not also linked

For instructions to build a load module of BASIC with GT support see Section J.3. Software for BASIC/RT11 with GT support that is provided on DECtape, cassette and DECpack disk also contain two running versions of BASIC:

BASGT.SAV	BASIC with GT support
BGTLPS.SAV	BASIC with GT and LPS support

BASGT.SAV is an non-overlaying version of BASIC with GT support. BGTLPS.SAV is a non-overlaying version of BASIC with GT and LPS support. BASGT.SAV is loaded by the following RT-11 monitor command:

```
.R BASGT
```

Or to load a version of BASIC with GT and LPS support the following command should be given:

```
.R BGTLPS
```

At this point the standard BASIC initial dialogue will occur. See Chapter 1 of the BASIC/RT11 Language Reference Manual for a description of the initial dialogue. As part of the initial dialogue BASIC will print:

```
USER FNS LOADED
```

This message will be printed whenever BASIC has had GT support linked with it. BASIC will terminate the initial dialogue by printing:

```
READY
```

NOTE

BASIC with GT support should not be run by the RT-11 monitor after GTON, a program supplied with RT-11, has been run. GTON causes RT-11 to print all information on the graphic display screen and any attempt by BASIC with GT support to use the display screen causes the computer to halt. If this happens, the monitor must be rebooted. To avoid this, when GTON has been run, do not run BASIC until the monitor has been rebooted by either a hardware bootstrap or the PIP reboot command. See

Section 4.13 of the RT-11 System Reference Manual for a description of the PIP command.

J.1.1 Documentation Conventions

The following chart describes the documentation conventions used in the description of the GT calls in this Appendix.

CONVENTION	MEANING
Square Brackets [ ]	Optional arguments are enclosed.
Lower case letter or lower case letter followed by a digit (a,b,x0,y1)	Value to be supplied by user -- may be any valid arithmetic expression.
Lower case letter followed by a dollar sign, (a$,x$)	String to be supplied by user may be string constant (enclosed in quotes) or variable (A$).
Upper case letter (A,B,X,Y)	Numeric variable whose value will be determined by call or an array name.
Y axis	The vertical axis
X axis	The horizontal axis

J.2 DISPLAY PROCESSOR CONTROL ROUTINES - CALL SUMMARY

BASIC programs can control the GT44 display processor by the use of the twenty-nine routines that are supplied with GT support for BASIC. A complete description of the BASIC call statement may be found in section 8.1 of the BASIC/RT11 Language Reference Manual.

The format of the CALL statement is:

```
CALL "name" (argument list)
        or
LET A$="name"
CALL (A$) (argument list)
```

The following chart summarizes the names, argument lists, and effects of the graphic calls supplied with the GT graphic support.

Call	Argument List	Effect
AGET	(A(i), Z)	Unscaled element i of the graphic array A and stores in Z.

Call	Argument List	Effect
APNT	(x,y [,l,i,f,t])	Positions beam at point represented by (x,y) after scaling. Optional changing of l,i,f, and t parameters. l is light pen sensitivity, i is intensity, f is flash, and t is line type.
APUT	(A (i), b)	Assigns element i of the graphic array A the scaled value of b. Dynamically changes display of array A.
DCNT		Restores to the screen display stopped by call to DSTP.
DFIX (n)		Eliminates old display buffer if it exists and creates a display buffer of n words. Closes all open BASIC files.
DON (t)		Turns on subpicture with tag t that had been turned off with a call to OFF.
DSAV	[("[dev:]filname[.ext]")]	Compacts the display file by eliminating references to erased subpictures and graphics arrays and if a file is specified creates a copy of the graphic display on a file on DECtape or disk. Display may then be restored to the screen at any time by a call to RSTR. DK: is the default device. DPY is the default extension.
DSTP		Stops display of the entire display buffer. Display may be restored by a call to DCNT.
ERAS	[(t)]	Erases subpicture with the tag. If t is not specified this call erases the tracking object created by a call to TRAK.
ESUB		Terminates subpicture created by a call to SUBP (with one argument).
FIGR	(A[,l,i,f,t])	Creates vectors from array A to form figure. See section J.2.8 concerning the cautions required when using graphic array calls. Optional changing of l,i,f, and t parameters.

Call	Argument List	Effect
FPUT	(A(i),b)	Assigns element i of graphic array A the scaled value of b and compensates the i + 2 element of A to leave following points of figure at same absolute location. Dynamically changes display of array A.
FREE		Eliminates display buffer, clears screen, and closes all open BASIC files.
INIT		Initializes display buffer and clears screen.
LPEN	(H,T[,X,Y])	Records a light pen hit (in H), the tag of a subpicture in which the hit occurred (in T), and, optionally the X and Y coordinates of the hit.
NOSC		Eliminates scaling factor for subsequent graphics.
OFF(t)		Turns off the subpicture with the tag t.
RDOT	(x,y,[,l,i,f,t])	Positions a beam originally at (x0,y0) scaled to (x0+x, y0+y) scaled. Optional changing of l,i,f, and t parameters.
RSTR	("[dev]filnam[.ext]")	Restores the display of the file created by a call to DSAV. The default extension is DK: and the default extension is DPY.
SCAL	(x0, y0, xl, yl [, X,Y])	Scales the x axis to vary from x0 to xl and the y axis to vary from y0 to yl and optionally returns the x and y scaling factors.
STAT	(s [,p])	Enables or disables the italic mode and optionally enables or disables visual intensification of light pen hit.
SUBP	(a [,b])	When there is only one argument - begins definition of a subpicture with a tag a. When there are two arguments - copies subpicture with tag b as a new subpicture with tag a.
TEXT	(list)	Outputs text to screen. List contains text to be printed "carriage return" information,

Call	Argument List	Effect
		and information to convert text to the GT40 (44) special shift-out character set.
TIME (z)		Sets timer based on real-time clock to a value of z ticks where each tick represents 1/60 second (for 60 Hz line current).
TIMR(E)		Returns the current value of the timer in variable E.
TRAK	(X,Y)	Puts a tracking object on the screen at the location (X,Y), the initial values of the variables, and centers the object on any light pen hit within its area and updates value of X and Y to the new location.
VECT	(x,y[,l,i,f,t])	Draws a line segment from the current beam position (x0,y0) to the point (x0+x, y0+y). Optional changing of l,i,f, and t parameters.
XGRA	(y,A[l,i,f,t])	Plots array A as series of points with the X position determined by the value of the element of the array and the Y position starting at the current beam location and being incremented by y for each point. See section J.2.8 concerning the caution required when using graphic array calls. Optionally changes l,i,f, and t parameters.
YGRA	(x,A[l,i,f,t])	Plots array A as a series of points with the Y position determined by the value of the element of the array and the X position starting at the current beam location and being incremented by x for each point. See section J.2.8 concerning the caution required when using graphic array calls. Optionally changes l,i,f, and t parameters.

NOTE

To be compatible with previous versions. DFIX, DSTP, DCNT, DON, and DSAV may also be called as FIX, STOP, CONT, ON, and SAVE respectively.

J.2.1 Display Buffer Control (DFIX FREE, INIT, DSTP, and DCNT)

The display processor is similar to the central processor in that it executes instructions from memory. Normally, it is not necessary to allocate the amount of core to be used by the display processor. In this case, the first call to a graphics routine will automatically cause one-half of the available space to be allocated for the display buffer and a message telling the buffer length will be printed on the console terminal. In a machine with 16K core, about 3000 words will be allocated with no user program or arrays laid out. This space would allow the display to draw about 1500 vectors. However, it is possible to specifically request a certain number of words for the size of the display buffer by the following call:

```
CALL "DFIX"(n)
```

For example, to use only 1000 words the following call should be made:

```
CALL "DFIX" (1000)
```

If the number of words asked for is greater than the space available, a message is printed indicating that there is not enough room for the display buffer. If the number of words requested is less than or equal to zero, an error messag5 results. This routine may be called at any time to change the size of the display buffer, but any picture on the screen at the time of the call is lost.

The display buffer may be allocated either in a program (explicitly or implicitly) or in immediate mode. Once allocated, the buffer need not be changed even between or during the execution of several programs. It may only be changed by calls to DFIX. Once the buffer has been created any call to a graphics routine will cause display processor instructions to be put into the display buffer. If the buffer becomes filled a ?DFO (Display File Overflow) error message is printed and BASIC returns to the READY message.

After an explicit or implicit call to DFIX one of the following messages may be printed

Messages	Meaning
xxxx WORDS FOR DISPLAY FILE	xxxx words have been allocated for display buffer. Printed after an implicit call to DFIX.
?NER - FOR DISPLAY BUFFER	The number of words requested in explicit call to DFIX exceeded the amount of available core.

NOTE

A call to either DFIX or FREE will automatically close all open files,

since both routines must internally rearrange a large part of BASIC.

If the user desires to run a program that does not use the display and does need all the available space for arrays (etc.), any space currently allocated for the display buffer may be relinquished by:

```
CALL "FREE"
```

This routine may be called at any time and the screen will be cleared. To guarantee that all dimensioned arrays will have the best chance to fit into core, it would be reasonable to call "FREE" in immediate mode. The first graphics call following such a call will result in an implicit call to "DFIX".

It is possible to "erase" and initialize the screen at any time by the following call:

```
CALL "INIT"
```

This routine clears the screen and initializes the display buffer. If a call to "FIX" has not been made previous to a call to "INIT", an implicit call to "FIX" is made.

Normally, the first graphics call following a call to "FIX" will implicitly call "INIT".

There are two further control routines which allow the screen to be turned "on" and "off". They are:

```
CALL "DCNT"

CALL "DSTP"
```

A call to DSTP will clear the screen and a call to DCNT will restore to the screen the display that existed before the call to DSTP. If a display is stopped, then a call to "DSTP" will not have any effect. Likewise, if the display is running, a call to "DCNT" will have no effect.

J.2.2 Scaling Instructions (SCAL and NOSC)

Unless a call to SCAL specifies otherwise, the screen has a coordinate system with the lower left hand corner of (0,0) and the upper right corner of (1023,1023). (Those screens set up in rectangular format allow a maximum visible y-value of 768). Often it is desirable to work in another coordinate system. To set the lower left hand corner of the screen at (x0,y0) and the upper right hand corner at (x1,y1), then the following call should be made:

```
CALL "SCAL"(x0,y0,x1,y1[X,Y])
```

Every graphics call after a call to "SCAL" will automatically assume that the lengths and coordinates specified are in terms of the "window" given in the call to "SCAL". It is possible to obtain the X and Y direction scale factors by providing an extra one or two parameters in the call to the scale routine. If a fifth parameter is given, the X scale factor will be returned as the value of that parameter (in the above example,X). If a sixth parameter is given, the Y scale factor will also be returned.

At any time, another call to the scale routine will change the scale to reflect the new request. The new scale factors will only effect graphics calls following the scale request.

Consider the following call:

```
CALL "SCAL"(0,0,200,200)
```

If a vector of length 200 in the X and 200 in the Y directions is drawn from the point x=0,y=0, it will go completely across the screen. Although the software does not allow the drawing of a single vector longer than the length of the screen, several vectors may be drawn consecutively to move the "beam" off the screen. If in the call to SCAL x0=x1 or y0=y1 the ?DV0 (DiVision by 0) error message will be printed and BASIC will return to the READY message.

NOTE

> In the call to scale, there is no requirement that x0 be less than x1 or y0 be less than y1; hence, pictures may be drawn upside down, backwards, etc. although text always will appear left to right.

The scale factoring may be eliminated once it is set by

```
CALL "NOSC"
```

This turns the scaling routine off and the default coordinate system will apply. If there has been no previous call to "SCAL" a call to "NOSC" will have no effect. Equivalently, one may establish a unit scale by

```
CALL "SCAL"(0,0,1023,1023)
```

but this is much less efficient since it invokes a scale with the X and Y factors equal to one at every graphics call.

EXAMPLE

Start a program by initializing the display file, scaling the screen and much later changing the scale and restarting the program.

```
100 A=100
110 CALL "INIT"
120 CALL "SCAL"(-A,-2*A,A,2*A,X,Y)
130 PRINT "X SCALE FACTOR IS"X
140 PRINT "Y SCALE FACTOR IS"Y
150 REM CREATE GRAPHICS IN FOLLOWING STATEMENTS
300 REM RESCALE HERE AND RESTART
310 PRINT "ENTER NEW RELATIVE SCALE";
320 INPUT Z
330 IF Z<=0 THEN 310
340 A=A/Z
350 GO TO 110
500 END
RUNNH
X SCALE FACTOR IS 5.12
Y SCALE FACTOR IS 2.56
ENTER NEW RELATIVE SCALE?5
X SCALE FACTOR IS 25.6
Y SCALE FACTOR IS 12.8
ENTER NEW RELATIVE SCALE?↑C

.REE

READY
```

J.2.3 Positioning the Beam (APNT and RDOT)

To position the beam at a point (X,Y) on the screen, the following CALL should be made:

```
CALL "APNT"(x, y [,l [,i [,f [,t]]]])
```

This will set the beam at the position specified by (x,y). The optional parameters l, i, f, and t are parameters whose values determine properties of the current and following graphics functions. Since the call may include 1, 2, 3, or all 4 of these parameters, all following specifications will show these parameters as

```
...[l, i, f, t]...
```

If this call has been preceded by a call to SCAL, the X and Y values will be scaled to their proper place on the screen. At no time can the scaled X or Y position the beam at a negative value or a value exceeding 1023, that is x and y must be within the range of x and y defined in the call SCAL.

The following chart describes the effects of the parameters.

Parameter	Effect
l	Light pen interaction. If l is given and is positive, then current and subsequent graphic output will be light pen sensitive (causing a light pen interrupt when pointed to on the screen). If l is zero or not given, there is no change in the parameter from its previous setting. If l is negative, the light pen will not cause an interrupt.
i	Intensity. Points, vectors and text may be put on the screen in one of 8 (eight) intensities. If i is given and is positive (between 1 and 8), the intensity will

Parameter	Effect
	be set to that value (8 being the brightest on the screen). If i is zero or not given then no change is made from the previous setting. If i has a value between -8 and -1, the current graphics output (except text) is invisible and the intensity is changed to the absolute value of i for the next graphics call only.
f	Flash. If f is given and is positive, then the blink mode will be enabled starting with the current graphic output. If f is zero or not given, no change is made in the blink mode. A negative value for f will disable the blink mode.
t	Type of Line. If t equals 1, 2, 3, or 4 the line type will be enabled to solid, long-dashed, short-dashed, or dot-dashed line, respectively. If t is given and is either greater than 4 or non-positive, no change in the line type will be made. If t is not given, then no change is made from the current setting.

These parameters may also be changed in calls to RDOT and VECT. The parameters may be changed at any time without effect on the display by a call to RDOT with the x and y arguments equal to zero and a negative intensity or by a call to VECT with the x and y arguments equal to zero (see Section J.2.4).

NOTE

Any change in the parameters will not effect any graphics previously drawn.

At the start of the display file, the default conditions are:

No scaling
Beam at lower left hand corner
Light pen interaction is disabled
Intensity is set to 5
Flash is turned off
The solid line type is enabled

Dots on the screen relative to the current beam position may be inserted via

```
CALL "RDOT"(x,y [,l,i,f,t])
```

This will set a beam originally at (x0,y0) to the position (x0+x,y0+y). As in the case of "APNT", any current scaling will be done automatically.

J.2.4 Drawing Vectors (VECT)

Vectors may be drawn by using the absolute value of

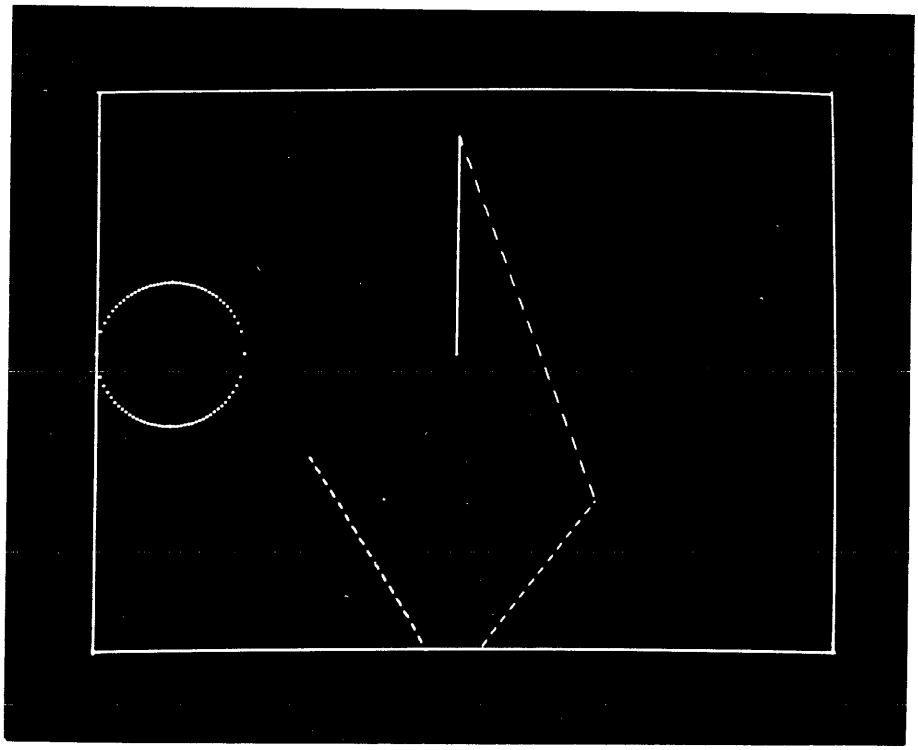
```
CALL "VECT"(x,y [,l,i,f,t])
```

This routine will cause a line segment to be drawn from the current beam position, (x0,y0), to the point (X0+x,Y0+y). The absolute values of x and y must be less than 1024 after scaling. If x and y both have the value 0 then no output will be visible on the screen (regardless of the value of i).

EXAMPLE

```
10 REM
20 CALL "INIT"
30 CALL "APNT"(400,200)
35 REM CREATES A VISIBLE POINT
36 REM
40 CALL "RDOT"(100,200)
45 REM POSITIONS BEAM RELATIVELY
46 REM CREATES A VISIBLE POINT
47 REM
50 CALL "VECT"(0,300)
55 REM CREATES A VERTICAL SOLID LINE
56 REM FROM POINT CREATED BY 40
57 REM
60 CALL "VECT"(200,-500,0,0,0,2)
65 REM CREATES A LONG-DASHED LINE
66 REM
70 CALL "VECT"(-200,-250,0,0,0,3)
75 REM CREATES A SHORT-DASHED LINE
76 REM THAT CROSSES BOTTOM EDGE OF SCREEN
77 REM
80 CALL "SCAL"(0,0,100,100)
85 REM RESCALES SCREEN
86 REM
90 CALL "VECT"(-20,30)
92 REM CREATES A 2ND SHORT-DASHED LINE
93 REM UNDER SCALING OF 80
94 REM
95 CALL "NOSC"
97 REM UNSCALES SCREEN
98 REM
100 FOR X=0 TO 100 STEP 5
110 LET Y=SQR(10000-X↑2)
120 CALL "APNT"(100+X,400+Y)
130 CALL "APNT"(100+X,400-Y)
140 CALL "APNT"(100-X,400+Y)
150 CALL "APNT"(100-X,400-Y)
160 NEXT X
165 REM LOOP CREATES POINTS IN CIRCLE
166 REM
200 CALL "SCAL"(0,0,100,134)
210 CALL "APNT"(0,0,0,-5,0,1)
220 CALL "VECT"(100,0)
230 CALL "VECT"(0,100)
240 CALL "VECT"(-100,0)
250 CALL "VECT"(0,-100)
255 REM CREATES BOX AROUND GRAPHICS
256 REM NOTE RECTANGULAR SCREEN USED FOR PICTURE
500 END
```

The previous program creates this output:



Example

Scale the screen for a window -50 to +50 in both directions, draw a diamond centered on the window and finally put a visible dot in the center of the diamond.

```
 40 CALL "INIT"
 100 CALL "SCAL"(-50,-50,50,50)
 105 REM SCALES SCREEN
 106 REM
 110 CALL "APNT"(0,25,-1,-5,-1)
 115 REM POSITIONS BEAM
 116 REM DISABLES LIGHT PEN INTEREACTION
 117 REM POINT CREATED IS INVISIBLE
 118 REM DISABLES FLASH
 119 REM
 120 CALL "VECT"(-25,-25,0,0,0,1)
 125 REM CREATES SOLID LINE
 126 REM
 130 CALL "VECT"(25,-25)
 140 CALL "VECT"(25,25)
 150 CALL "VECT"(-25,25)
 155 REM COMPLETES DIAMOND
 156 REM
 160 CALL "RDOT"(0,-25)
 165 REM CREATES VISIBLE POINT IN CENTER
 166 REM COULD ALSO HAVE BEEN
 167 REM 160 CALL "APNT"(0,0)
 500 END
```

J.2.5 Text Instruction (TEXT and STAT)

Textual and carriage control information may be put on the screen in an extremely easy manner. The form of the text call is

```
CALL "TEXT" (list)
```

where the list may contain

element:	effect:
string variables or constants	text to be printed on screen
[+] m	Carriage Return and m Line Feeds
0	Carriage Return and no Line Feed
-m	Converts following text to shift out characters.

A carriage return causes the next line of text to be output starting at the left hand edge of the screen.

Normal text may be any printable character with an ASCII value greater than or equal to 40(octal). Shift-out characters are the special character set, consisting of 31 symbols and Greek letters, within the display processor. The following characters will be converted to shift-out characters when included in a string following -m:

```
@ABCDEFMHIJKLMNOPQRSTUVWXYZ[\]↑_
```

See the following example for a list of the shift-out characters. Note that the user should not include any character other than those listed above within the shift-out text. The letter O within shift-out text will be converted to a blank space on the display screen.

The location of the text printed may be controlled by positioning the beam with an invisible dot before the call to text. The lower left corner of the first character printed will start at this point.

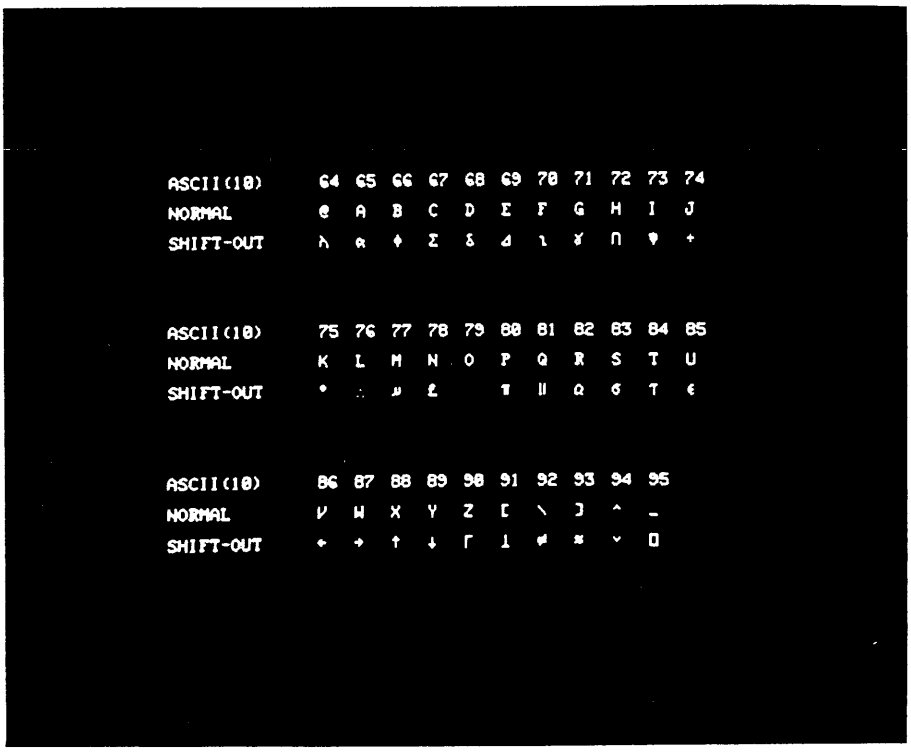
NOTE

The TEXT call will only print strings. To print a number that has been computed by the program, it must first be converted to a string by the STR$ function. See section 6.3 of the BASIC/RT11 Language Reference Manual for information on the STR$ function.

```
10 CALL "INIT"
15 X=300\X1=50
20 K1=64\K2=74
25 Y=600\A=40
26 REM INITIALIZES VARIABLES FOR FIRST RUN THROUGH LOOP
27 REM
30 CALL "APNT"(100,Y,0,-5)
40 CALL "TEXT"("ASCII(10)")
42 CALL "APNT"(100,Y-A,0,-5)
44 CALL "TEXT"("NORMAL")
46 CALL "APNT"(100,Y-2*A,0,-5)
```

```
48 CALL "TEXT"("SHIFT-OUT")
49 REM PRODUCES LABELS AT LEFT
55 FOR K=K1 TO K2\REM BEGINS LOOP
60 CALL "APNT"(X,Y,0,-5)
70 CALL "TEXT"(STR$(K))
75 REM PRINTS DECIMAL ANSII VALUE
80 CALL "APNT"(X,Y-A,0,-5)
90 CALL "TEXT"(CHR$(K))
95 REM PRINTS "NORMAL" TEXT
100 CALL "APNT"(X,Y-2*A,0,-5)
110 CALL "TEXT"(-1,CHR$(K))
115 REM PRINTS "SHIFT-OUT" TEXT
120 X=X+X1
130 NEXT K\REM ENDS LOOP
140 IF K1=86 THEN 250
145 REM TESTS IF FINISHED
150 IF K1=75 THEN 200
155 REM TESTS IF PROGRAM HAS BEEN THROUGH LOOP TWICE
160 K1=75\K2=85\Y=400\X=300
170 GO TO 30
175 REM INITIALIZES VALUES FOR SECOND RUN
176 REM THROUGH LOOP AND JUMPS TO START OF LOOP
200 K1=86\K2=95\Y=200\X=300
210 GO TO 30
215 REM INITIALIZES VALUES FOR THIRD RUN
216 REM THROUGH LOOP AND JUMPS TO START OF LOOP
250 END
RUNNH

READY
```

will produce the following output on the screen.



NOTE

To change any of l, i, f, or t for textual output, the call to "TEXT" must be preceded by

```
CALL "RDOT"(0, 0, 1, -i, f, t)
```

Also, text material is always visible; that is, one can not write "invisible words" on the screen.

Italics may be enabled by a call to the "STAT" routine which allows the user to disable or enable the italics mode, and disable or enable the intensification that occurs at a light pen hit. The CALL is of the form:

```
CALL "STAT"(s [, p])
```

where:

s Font. If s is given and is positive, then the italics font is disabled. If s is zero, then there is no change from the previous setting. If s is negative, the italics mode is enabled.

p Intensification of Light Pen Hit. If p is positive, then the point of a light pen hit will be raised to intensity level 8. If p is zero, there is no change from the previous setting. If p is negative, the point of a light pen hit will not change in intensity. The intensification will only occur if the line hit is light pen sensitive.

The default conditions at the start of the display file are italics disabled and light pen intensification enabled. Do not confuse intensification of a light pen hit with interaction of a light pen (see the section J.2.7).

J.2.6 Subpictures (SUBP, ESUB, DON, OFF, and ERAS)

A series of graphics calls may be defined as a unit by use of the subpicture calls (SUBP and ESUB). This allows the series of graphics calls to be copied, turned off and on, erased, and tested for light pen hits by referencing the subpicture's tag. A series of graphics calls is defined as a subpicture by inserting a call to SUBP with one argument, called the "tag" before the graphics calls and a call to ESUB with no arguments after the graphics calls. No graphics will be displayed after the call to SUBP until the call to ESUB is executed. The form of the calls to SUBP and ESUB are:

```
CALL "SUBP"(a)

CALL "ESUB"
```

Each subpicture has associated with it a unique "tag". The parameter "a", a tag, is a positive integer (less than 32768). Alternatively, a subpicture may be a copy of a previous subpicture. Any subpicture to be copied should not contain any calls to "APNT" but only relative graphic data or else the subpicture will not be completely "movable". To copy a subpicture at any point on the screen, position the beam at the desired point and then make a call to SUBP with two arguments. The form of the call is:

```
CALL "SUBP" (a,b)
```

where b is the tag of the subpicture being copied and a is the tag of the new subpicture being created. This call creates a jump in the display buffer back to the display buffer code for the original subpicture. A call to SUBP with two arguments should not have a corresponding call to ESUB. For every call to SUBP with one argument there should be a corresponding call to ESUB after it. If there is a call to SUBP with one argument without a corresponding call to ESUB the graphics will not appear on the screen until a call to ESUB is made.

EXAMPLE

Define a subpicture that draws a horizontal resistor and then makes two copies of it.

```
 80 CALL "INIT"
 90 CALL "APNT"(500,100,0,-5)
 95 REM CREATES AN INVISIBLE POINT
 96 REM
100 CALL "SUBP"(1)
105 REM SUBPICTURE STARTS
106 REM
110 CALL "VECT"(20,0,0,5,0,1)
120 CALL "VECT"(5,10)
125 REM CREATES FIRST PART OF RESISTOR
126 REM
130 FOR I=1 TO 2
140 CALL "VECT"(10,-20)
150 CALL "VECT"(10,20)
160 NEXT I
165 REM CREATES MIDDLE OF RESISTOR
166 REM
170 CALL "VECT"(10,-20)
180 CALL "VECT"(5,10)
190 CALL "VECT"(20,0)
195 REM CREATES END OF RESISTOR
196 REM
```

```
200 CALL "ESUB"
205 REM ENDS SUBPICTURE, DISPLAY BECOMES VISIBLE
206 REM
210 CALL "APNT"(200,400,0,-5)
215 REM CREATES AN INVISIBLE POINT
216 REM
220 CALL "SUBP"(2,1)
225 REM COPIES FIRST SUBPICTURE
226 REM
230 CALL "SUBP"(3,1)
235 REM COPIES FIRST SUBPICTURE
236 REM NEXT TO COPY CREATED BY 220
237 REM
240 END
```

Note that subpictures may contain calls to other subpictures. The subpicture call depth limit is 10(decimal). A nested subpicture is created by a call to SUBP with one argument followed by a second call to SUBP before any calls to ESUB. Including the first call to SUBP there can be up to ten calls to SUBP before a call to ESUB. The first call to ESUB encountered will end the last subpicture created and no graphics within the nested subpicture will appear on the display screen until one call to ESUB has been executed for every call to SUBP (with only one tag). Nested subpictures would be useful for applications where there are several component graphics that will be frequently copied or tested for light pen hits as a group and individually. If there are more than 10 subpictures nested, and ?INS (Illegally Nested Subpictures) error message will be printed. The legal depth of subpicture nesting can be changed by assembling the BASIC GT support from the sources, which are available separately.

Subpictures may be turned on and off individually by the use of the following calls

```
CALL "DON"(t)

CALL "OFF"(t)
```

where t is the tag of the subpicture. When a subpicture is turned off there will be two effects on the display:

1. The graphics calls included in the subpicture will not appear on the screen and

2. any relative graphics call after the subpicture will appear relative to the beam position before the subpicture instead of after the subpicture.

Consequently any subpicture to be turned off should be followed by a call to APNT unless it is desirable to have the graphics calls moved. The code for the graphics remains in the display buffer and a subpicture may be copied while it is turned off. Once a subpicture has been turned off it may be turned on by making a call to DON. Turning on a subpicture that is already on or turning off a subpicture that is off has no effect.

When a subpicture is no longer needed, it may be erased by

```
CALL "ERAS"(t)
```

where t is the tag of the subpicture. The difference between erasing a subpicture and turning it off is that erasing turns off the subpicture and further, eliminates the code in the display buffer freeing up the corresponding tags for use. Erasing a subpicture does not recapture that portion of the display buffer that was used for the subpicture; however once subpictures have been erased the display buffer may be condensed by a call to SAVE. See Section J.2.10.

J.2.7 Light Pen Interaction (LPEN and TRAK)

If any graphics on the screen has been made light pen sensitive, it is possible to test for a light pen hit on light pen sensitive graphics, determine the x and y coordinates of the light pen hit, and determine which if any subpicture the hit appeared in. The form of the call is:

```
CALL "LPEN"(H,T[,X,Y])
```

where H is a flag that is equal to zero if no hit has occurred and equal to one after a hit has occurred. If H equals one, then the value of T is the tag of the subpicture in which the hit occurred. If the hit occurred other than in a subpicture, T will be set to zero. If X and Y are specified, the scaled X and Y coordinate values of the hit will be stored in these variables.

Example

```
100 CALL "INIT"
110 CALL "APNT"(200,200,1,-5)
115 REM CREATES AN INVISIBLE POINT
116 REM ENABLES LIGHT PEN SENSITIVITY
117 REM
120 CALL "SUBP"(10)
130 FOR I=1 TO 10
140 CALL "VECT"(50,0)
150 CALL "RDOT"(-50,5,0,-5)
160 NEXT I
170 CALL "ESUB"
175 REM CREATES 10 VERTICAL LINES IN SUBPICTURE 10
176 REM
190 CALL "APNT"(200,400,0,-5)
200 FOR I=1 TO 10
210 CALL "VECT"(0,50)
220 CALL "RDOT"(5,-50,0,-5)
230 NEXT I
235 REM CREATES 10 HORIZONTAL LINE, NOT IN SUBPICTURE
236 REM
240 CALL "APNT"(300,300,0,-5)
250 CALL "SUBP"(1)
260 CALL "TEXT"("WRONG BOX-TRY AGAIN")
270 CALL "ESUB"
272 REM CREATES SUBPICTURE WITH TEXT
275 CALL "OFF"(1)
```

```
276 REM TURNS OFF "WRONG BOX-TRY AGAIN"
277 REM
280 CALL "APNT"(300,300,0,-5,1)
285 CALL "SUBP"(2)
290 CALL "TEXT"("HIT ON SUBPICTURE 10!")
292 REM CREATES SUBPICTURE WITH FLASHING TEXT
295 CALL "ESUB"\CALL "OFF"(2)
296 REM TURNS OFF "HIT ON SUBPICTURE 10!"
300 CALL "LPEN"(H,T,X,Y)
310 IF H=0 THEN 300
315 REM TESTS FOR LIGHT PEN HIT
320 IF T=10 THEN 400
325 REM TESTS FOR LIGHT PEN HIT ON SUBPICTURE 10
330 CALL "DON"(1)
340 CALL "TIME"(2*60)
350 CALL "TIMR"(T)
360 IF T>0 THEN 350
370 CALL "OFF"(1)
380 GO TO 300
385 REM PUTS MESSAGE ON SCREEN FOR 2 SECONDS
400 CALL "DON"(2)
410 CALL "TIME"(2*60)
420 CALL "TIMR"(T)
430 IF T>0 THEN 420
440 CALL "OFF"(2)
450 GO TO 300
455 REM PUTS MESSAGE ON SCREEN FOR 2 SECONDS
500 END
```

In order to fully utilize the light pen capability of the display processor, BASIC with GT support has provision for optical tracking with the light pen. If X and Y are variables (not constants), then

```
CALL "TRAK"(X,Y)
```

puts an object on the screen at scaled (X,Y) that will react to light pen hits in such a way as to always center itself on the last light pen hit on the object. The tracking object will only respond to hits within its area. The diamond shape tracking object is shown in the example at the end of this section. The initial position of the tracking object should be determined by a LET statement before the call to TRAK. At each light pen hit, X and Y are updated to the new center of the tracking object for program use. The object can not be "snapped" off the screen by a fast motion of the light pen; upon reaching the edge, it will reposition itself near the center of the screen.

NOTE

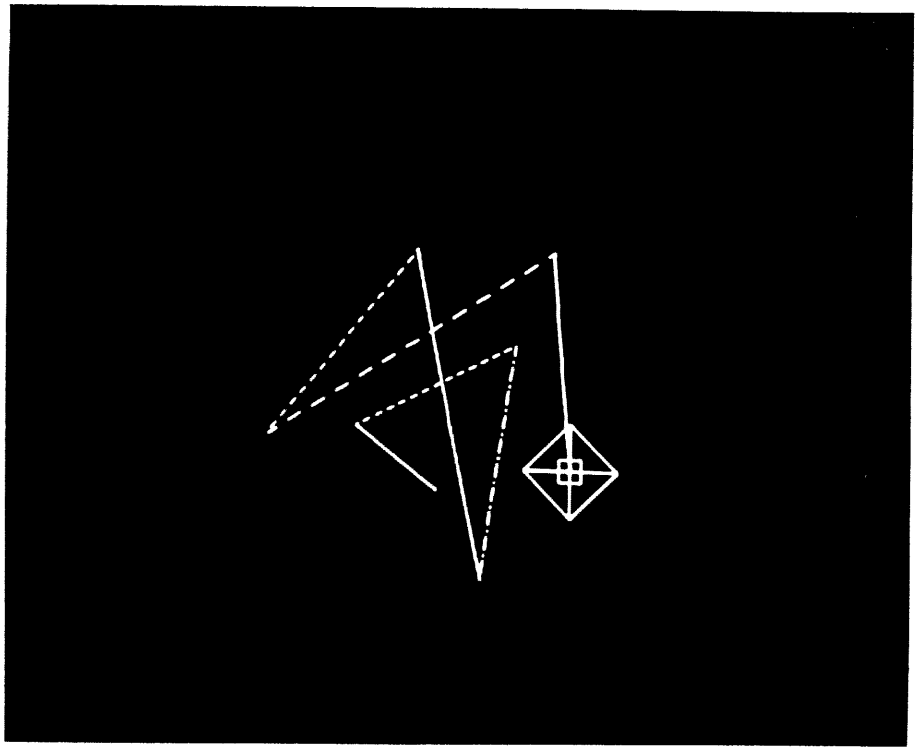
On a rectangular screen, the initial values of X and Y (scaled to between 0 and 1023) may put the tracking object off the "top" of the screen.

A call to "ERAS" with no tag specified will remove the tracking object from the screen:

EXAMPLE

```
LISTNH
100 X=100\Y=100
110 CALL "INIT"
120 CALL "TRAK"(X,Y)
121 PRINT "THIS PROGRAM DRAWS LINES WITH THE LIGHT PEN"
122 PRINT "LIGHT PEN MOVES TRACKING OBJECT"
123 PRINT \PRINT "INPUT EFFECT"\PRINT "VALUE"\PRINT
124 PRINT "-1    EXITS OUT OF PROGRAM"
125 PRINT "0     SAME AS PREVIOUS LINE TYPE"
126 PRINT "1     SOLID LINE"
127 PRINT "2     LONG-DASHED LINE"
128 PRINT "3     SHORT-DASHED LINE"
129 PRINT "4     DOT-DASHED LINE"
130 INPUT Z
135 IF Z=-1 THEN 500
140 IF Z=0 THEN 200
150 IF Z=1 THEN 200
160 IF Z=2 THEN 200
170 IF Z=3 THEN 200
180 IF Z=4 THEN 200
190 GO TO 130
200 REM DRAW LINE HERE
210 IF N=0 THEN 240
220 CALL "VECT"(X-X1,Y-Y1,0,0,0,Z)
230 X1=X\Y1=Y
235 GO TO 130
240 N=1\REM FIRST TIME ONLY
250 CALL "APNT"(X,Y)
260 X1=X\Y1=Y
270 GO TO 130
500 END

READY

RUNNH
THIS PROGRAM DRAWS LINES WITH THE LIGHT PEN
LIGHT PEN MOVES TRACKING OBJECT

INPUT   EFFECT
VALUE

-1      EXITS OUT OF PROGRAM
0       SAME AS PREVIOUS LINE TYPE
1       SOLID LINE
2       LONG-DASHED LINE
3       SHORT-DASHED LINE
4       DOT-DASHED LINE
?0
?0
?3
```

```
?4
?1
?3
?2
?1
?-1

READY
```



J.2.8 Graphic Arrays: Graphs and Figures (YGRA,XGRA,AGET,APUT,FIGR,FPUT)

BASIC with GT support has three graphic array calls (YGRA, XGRA, and FIGR) that allow the display of an entire array through one graphics call. Graphics created by these calls may be changed while they are being displayed by calls to APUT, AGET, and FPUT. Because the graphics created may be dynamically changed, extreme care must be taken in their use. Once a graphic array call (YGRA, XGRA, or FIGR) has been made a call to INIT must be made before the user array is zeroed by the BASIC system. The following BASIC statements and commands zero the user arrays: RUN, RUNNH, SCRATCH, CLEAR, NEW, OLD, and CHAIN. If the user's arrays are zeroed while a graphics array is being displayed the display processor will not be accessible by any program. Any attempt to execute any graphics call will cause the processor to enter a closed loop. If this situation occurs, two CTRL/C's will return control to the monitor and BASIC with GT support may be run by the following command:

```
.R BASGT
```

This will cause a hardware reset of the display processor and complete initialization of the BASIC system. The stored program will be lost. It is usually possible to re-enter BASIC by the RE monitor command before this is done and the BASIC program can then be saved by the BASIC SAVE or REPLACE command. There may be errors in the saving of the BASIC program and it will still be necessary to return to the monitor and run BASIC.

It is possible to plot a graph of points by a series of "APNT" calls. Most often, in the case of Y-graph data, the X coordinates are evenly spaced across the screen. When this is the case, one may place the Y-coordinate data in an array and display directly from the array via

```
CALL "YGRA"(x,A [1, i, f, t])
```

where x is the increment in the X direction and A is the name of a 1-dimensional array. The X position will be incremented by x starting at the current beam position. Note that the Y-array data must be stored as absolute coordinates. If an array is dimensioned 50, then 51 points are displayed, A(0) thru A(50).

Similarly, one may display a graph of X-data by

```
CALL "XGRA"(y,B [,1, i, f, t])
```

where y is the increment in the Y direction and B is the name of a 1-dimensional array.

NOTE

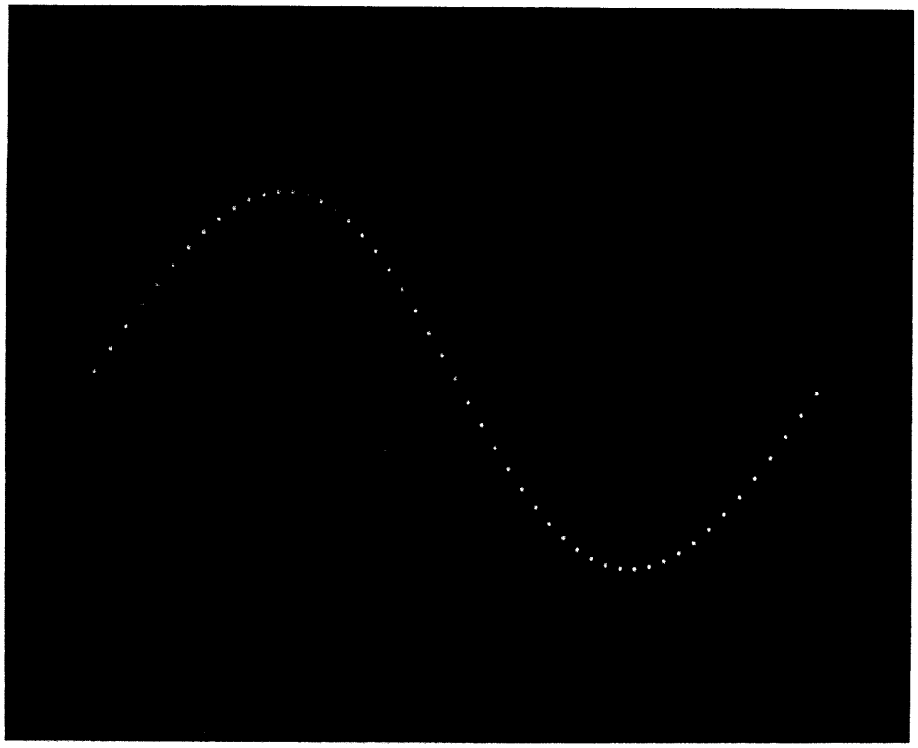
After execution of XGRA, YGRA, or FIGR always make a call to INIT before zeroing the user array by a RUN, RUNNH, SCRATCH, CLEAR, OLD, or NEW command or CHAIN statement. If this is not done the processor will enter a closed loop.

EXAMPLE

Draw a sine wave across the entire screen using 51 points.

```
LISTNH
90 X=128
100 CALL "INIT"
110 DIM A(50)
120 FOR I=0 TO 50
130 A(I)=256*(1+SIN(3.14159*I/25))+X
150 NEXT I
155 REM FILLS ARRAY A WITH VALUES OF SINE FUNCTION
156 REM
160 CALL "APNT"(0,0,0,-5)
170 CALL "YGRA"(20,A)
175 REM CREATES GRAPHIC DISPLAY
176 REM
180 PRINT "INPUT 0 TO END";
190 INPUT Z
200 IF Z=0 THEN 300
210 GO TO 180
300 CALL "INIT"
```

```
305 REM             NOTE
306 REM       CAN NOT RE-RUN PROGRAM UNLESS
307 REM       CALL IS MADE TO "INIT" FIRST
310 END

READY

RUNNH
INPUT Ø TO END?Ø

READY
```



Since the data in the array is scaled and the display processor "executes" the data in the array, every attempt has been made to "lock" the array to prevent the display processor from running wild. Hence, if array A is used as graph data, every entry in the array (except A(0)) is "locked" and an "SOB" (Subscript Out of Bounds) error will result if these entries are accessed through normal subscripting. If the first element of the array (A(0)) is assigned a new value by a LET statement after the call to YGRA or XGRA, the first point will be altered in an unpredictable way.

However, graph data may be accessed via

```
CALL "AGET"(A(i),Z)

CALL "APUT"(A(i),b)
```

where A is the name of a graph data array, i is a valid subscript, and Z is a scalar variable (not a scalar whose name is also a graph data array). "AGET" unscales the A(i) and returns a value in Z. "APUT" scales the value of b and stores the correct value in A(i). The display will be immediately updated.

Note that LPS arrays may be displayed directly using the above routines only if the array, say A, has been named in the "USE" routine as A or A(0). Otherwise, the arrays will not be correctly displayed. The valid subscripts for an LPS array are twice that of the value in the dimension statement. Such subscripts are valid arguments to "AGET" and "APUT". The LPS routines can not access arrays that have been displayed by a call to XGRA or YGRA. The arrays can only be accessed by AGET and APUT.

For further graphics capability, particularly in regards to motion of objects, a figure may be drawn from an array A of pairs of vectors. (A(0),A(1)) is the first X,Y-pair, (A(2),A(3)) is the second X,Y-pair, etc.

The figure is "drawn" from the current beam position. The form of the call is:

```
CALL "FIGR"(A[,l,i,f,t])
```

NOTE

After execution of XGRA, YGRA, or FIGR always make a call to INIT before zeroing the user array by a RUN, RUNNH, SCRATCH, CLEAR, OLD, or NEW command or CHAIN statement. If this is not done the processor will enter a closed loop.

Figure arrays are "locked" just as the other graphic arrays. These arrays may be accessed by the "AGET" and "APUT" routines as described above. A call to APUT will change the appropriate vector and will cause all following vectors to be displaced.

EXAMPLE

The following example uses an "invisible" figure to move a graphics display.

```
LISTNH
90 DIM A(1)
100 CALL "INIT"
110 CALL "FIGR"(A,0,-5)
115 REM NOTE-INVISIBLE FIGURE
116 REM INITIAL X AND Y VALUES ARE ZERO
117 REM
120 CALL "VECT"(100,0)\CALL "VECT"(0,100)
130 CALL "VECT"(-100,0)\CALL "VECT"(0,-100)
135 REM BOX CREATED IS DISPLAYED RELATIVE TO THE FIGURE
136 REM
140 CALL "APNT"(0,0,0,-5)
205 FOR I=0 TO 500
210 CALL "APUT"(A(0),I)
250 NEXT I
255 REM LOOP MOVES BOX TO RIGHT
256 REM
260 FOR J=0 TO 500
270 CALL "APUT"(A(1),J)
300 NEXT J
305 REM LOOP MOVES BOX UP
306 REM
310 FOR I=500 TO 0 STEP -1
320 CALL "APUT"(A(0),I)
350 NEXT I
355 REM LOOP MOVES BOX TO LEFT
356 REM
360 FOR J=500 TO 0 STEP -1
370 CALL "APUT"(A(1),J)
400 NEXT J
405 REM LOOP MOVES BOX DOWN
406 REM
410 GO TO 205
415 REM COMPLETE LOOP MOVES BOX IN A SQUARE PATH
416 REM PROGRAM CAN BE EXITED BY CTRL/C
520 END

READY

RUNNH
↑C
↑C

.
.REE

READY
```

NOTE

Because of the algorithm used for evaluating string expressions (such as the name of the subroutine in a CALL statement), the time that elapses between execution of CALL statements may vary. Specifically, when free storage fills with strings, storage is recovered by the deletion of strings that will not be needed. When a display contains a moving object, the object will stop moving briefly during this operation.

Alternatively it may be desirable to alter only one point in a displayed figure, a call to FPUT should be made. The form of the call is:

```
CALL "FPUT" (A(i),b)
```

This call will change the i element in array A to the value of b scaled and will then compensate A (i+2) to keep the following points in the figure in the same location. The call

```
CALL "FPUT" (A(3),100)
```

is equivalent in its effect to the following series of AGET and APUT calls

```
CALL "AGET" (A(3),Y)
CALL "AGET" (A(5),Z)
CALL "APUT" (A(3),100)
CALL "APUT" (A(5),(Z+Y-100))
```

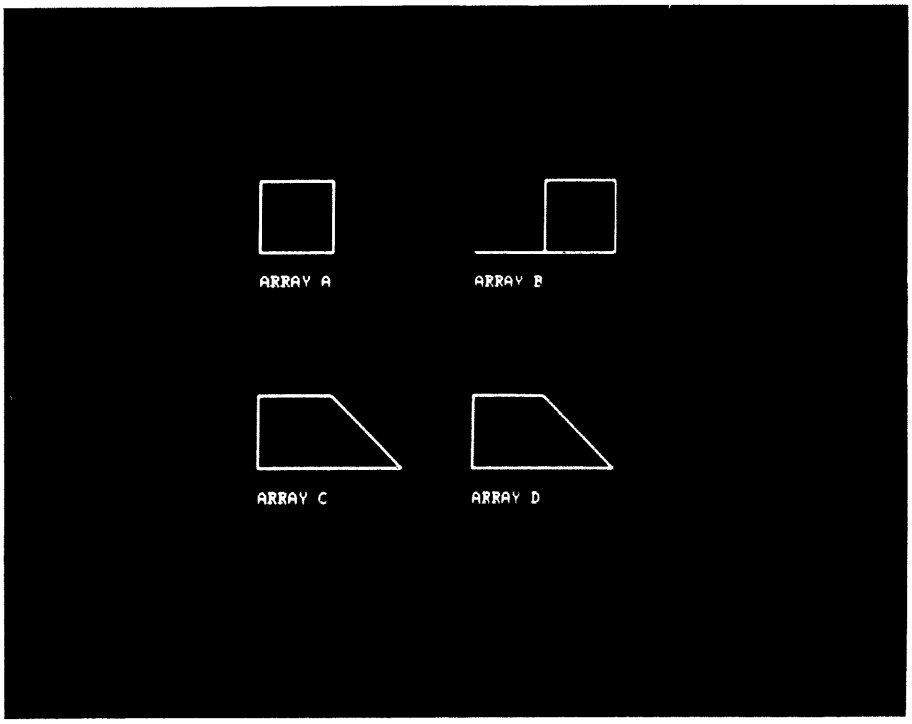
If there is no A(i+2) a call to FPUT (a(i),b) is equivalent to a call to APUT (a(i),b).

Example

This example demonstrates the FIGR, APUT, and FPUT instructions.

```
LISTNH
100 DIM A(7),B(7),C(7),D(7)
110 CALL "INIT"
120 A(0)=100\A(1)=0
130 A(2)=0\A(3)=100
140 A(4)=-100\A(5)=0
150 A(6)=0\A(7)=-100
160 REM THIS HAS FILLED ARRAY A(7)
165 REM
170 FOR I=0 TO 7
180 B(I)=A(I)\C(I)=A(I)\D(I)=A(I)
190 NEXT I\REM THIS HAS FILLED C(7),D(7),AND B(7)
195 REM
200 CALL "APNT"(100,500,0,-5)
210 CALL "FIGR"(A)
215 CALL "APNT"(100,450,0,-5)
216 CALL "TEXT"("ARRAY A")
217 REM CREATES DISPLAY OF ARRAY A
218 REM ARRAY A IS LEFT UNCHANGED
219 REM
220 CALL "APNT"(400,500,0,-5)
230 CALL "FIGR"(B)
235 CALL "APNT"(400,450,0,-5)
236 CALL "TEXT"("ARRAY B")
237 REM CREATES DISPLAY OF ARRAY B
238 REM
240 CALL "APNT"(100,200,0,-5)
250 CALL "FIGR"(C)
255 CALL "APNT"(100,150,0,-5)
256 CALL "TEXT"("ARRAY C")
257 REM CREATES DISPLAY OF ARRAY C
258 REM
260 CALL "APNT"(400,200,0,-5)
270 CALL "FIGR"(D)
275 CALL "APNT"(400,150,0,-5)
276 CALL "TEXT"("ARRAY D")
```

```
277 REM CREATES DISPLAY OF ARRAY D
278 REM
300 CALL "APUT"(B(0),200)
305 REM SINGLE APUT ON ARRAY B
306 REM
310 CALL "APUT"(C(0),200)
320 CALL "APUT"(C(2),-100)
325 REM TWO APUT'S ON ARRAY C
326 REM
330 CALL "FPUT"(D(0),200)
335 REM SINGLE FPUT ON ARRAY D
340 REM C AND D (LEFT BOTTOM AND RIGHT BOTTOM) SHOULD BE SAME
400 PRINT "INPUT ANY NUMBER TO EXIT FROM PROGRAM";
410 INPUT X
420 CALL "INIT"
430 END

READY

RUNNH
INPUT ANY NUMBER TO EXIT FROM PROGRAM?10

READY
```



Note that figure arrays only use an even number of array elements starting at a subscript of zero; hence the dimension of the array will be odd (as in the above example)

```
DIM A(7)
```

All scaling is done automatically and in the above example

```
CALL "FPUT"(A(6),B)

CALL "FPUT"(A(7),B)
```

will act exactly like the corresponding "APUT" calls since there are no further points (X,Y-pairs) in the array.

J.2.9 Timing Routines (TIME,TIMR)

Many graphic routines require the use of the real time clock - especially when pictures must be on the screen for a certain number of seconds and then moved or erased.

The user may set a count-down timer to a value of z ticks by

```
CALL "TIME"(z)
```

If Z is negative or equal to zero an ?ARG error message will be printed. If the line current is 60Hz. (cycles per second), there will be 60 ticks per second. Therefore to set the clock to count down for one minute the CALL to TIME would be

```
CALL "TIME"(60*60)
```

The timer will operate independently of, as well as compatibly with, any other routine which uses the real time clock.

NOTE

> Once the timer has been set it will continue to count down until it reaches zero. It will continue to count down even if BASIC is exited and another program is being used. When the timer reaches zero it will output an interrupt to BASIC with GT support. This may cause the computer to halt if BASIC with GT support is no longer in core. To avoid this possibility, if the timer has been set and may still be running make the following CALL while still in BASIC:
>
> ```
> CALL "TIME" (1)
> ```

The timer will count down to zero and then remain zero. Its value at any time may be obtained by

```
CALL "TIMR"(E)
```

and the current value will be returned in the variable E.

NOTE

> If the hardware does not include a real-time clock TIMR will always return 0.

EXAMPLE

Put a subpicture (with tag 2) on the screen and then erase it 15 seconds later.

```
5 CALL "INIT"
10 CALL "SUBP"(2)
11 REM - DEFINE SUBP 2 HERE
12 REM        .
13 REM        .
14 REM        .
120 CALL "ESUB"
121 REM - ESUB PUTS SUBPICTURE 2 ON THE SCREEN
130 CALL "TIME"(15*60)
131 REM - 15*60 TICKS @ 60 TICKS/SEC  =  15 SECONDS
140 CALL "TIMR"(E)
150 IF E<>0 THEN 140
160 CALL "ERAS"(2)
170 END
```

J.2.10 Display Buffer Condensing, Storage, and Retrieval (DSAV,RSTR)

After the execution of a portion of a graphics program the display buffer may become filled with subpictures that are no longer needed (i.e., they have been erased) or it may become desirable to unlock a graphic display array to allow new values to be input without using the APUT or FPUT calls. A call to DSAV will compact the display buffer by eliminating the code for erased subpictures and the references to graphic arrays. The data in the graphic arrays will be lost and should be saved (if desired) before the call the DSAV. The form of the call is

```
CALL "DSAV"[(file)]
```

where file may be:

a string variable

a literal string in the form
"[dev:]filnam[.ext]"

If no file is specified, then the display buffer is condensed and the graphic arrays are unlocked and the program continues. The device can only be DECtape or disk. The default device is "DK:", the system device. The default extension is ".DPY". If the file is specified in the correct format then one of the following messages is printed on the user terminal.

Message	Meaning
SAVE COMPLETED	File has been stored.
FILE ALREADY EXISTS	File with same name already exists on device specified. Original file is left unchanged, no storage of new file.

Message	Meaning
?DEV ERR - C	The device specified is illegal or read-only (only disk or DECtape are legal devices), an error has been detected while writing the file, or no RT-11 device handler exists for the device.
NER - C	There is not enough free space in memory to load the device handler.

After one of these messages is printed the execution of the program continues.

Saved files may be called in from the disk or DECtape by a call to RSTR. A file may be restored into either an empty buffer, a non-empty buffer, or at the start of the program before a buffer has been allocated. If there is no buffer, one will be created by an implicit call to DFIX. Otherwise the restored file will be placed at the first free location in the display buffer. The form of the call is

```
CALL "RSTR" (file)
```

After the call to RSTR one of the following error messages may be printed.

Message	Meaning
SAVE FILE NOT FOUND	No file of specified name exists on specified device.
NOT ENOUGH DISPLAY BUFFER FOR RESTORE	Restored code will not fit into free area of display buffer. Buffer returned to state before the call to RSTR.

If no error message is printed, the restore has been successful. After the printing of an error message or after the restore is completed, execution of the program continues. To correct a program that has insufficient free buffer area, the call to RSTR may be preceded by a call to DSAV to condense the present buffer or a call to DFIX at the beginning of the program to expand the area of core used by the display buffer.

This feature allows several programs to create pictures and another program to use them without the overhead of the code required to create them. An application of this feature is the creation of a picture library. Subpictures are created (and turned off) and then saved. When the file is restored, those subpictures may be either turned on or copied with the two parameter subpicture calls. In this way, for example, a library of electronic components could be built and called in and used when needed without having to actually redraw them.

NOTE

The file version of the "DSAV" routine as well as the "RSTR" routine are only supported in BASIC/RT11.

When the display file contains data and a file is restored at the end of the current data, the tags that are in the original buffer may duplicate those in the restored file. This means that any reference

to a given tag will always apply to the first instance of that tag! This situation should be avoided by the careful numbering of subpictures. However, if two subpictures do have the same tag, the original subpicture can be copied with a new tag (in the same location if desired) and then erased. Then the restored subpicture will be accessible by its tag and the original subpicture will be accessible by its new tag.

The display processor treats the restored file as if the code required to create it had been entered at that point. Specifically, after the call to RSTR all display parameters (l, i, f, t, s and p) have the values they were assigned in the program that created the restored file.

EXAMPLE

```
70 REM THIS PROGRAM RESTORES A DISPLAY FILE
75 REM AND COPIES SUBPICTURES AT LIGHT PEN HITS
80 X=200\Y=200
90 CALL "INIT"
95 PRINT "WHAT FILE DO YOU WANT RESTORED";
100 INPUT A$
105 PRINT "WHAT IS THE HIGHEST NUMBERED SUBPICTURE";
106 INPUT N
107 Z=N+1
110 CALL "RSTR"(A$)
120 CALL "TRAK"(X,Y)
121 PRINT "POINT THE LIGHT PEN WHERE A SUBPICTURE ";
122 PRINT "SHOULD BE COPIED.  "
123 PRINT "INPUT THE TAG OF SUBPICTURE TO BE COPIED."
124 PRINT "MOVE THE TRACKING OBJECT AND INPUT THE NEXT TAG."
125 PRINT "IF TRAK HAS NOT MOVED SUBPICTURE WILL BE"
126 PRINT "CREATED IMMEDIATELY AFTER PREVIOUS ONE."
127 PRINT "INPUT -1 TO ERAS TRAK AND EXIT."
130 INPUT A
140 IF A=-1GO TO 500
150 IF A<1GO TO 130
160 IF A>NGO TO 130
165 IF X<>X1 THEN 200
170 IF Y<>Y1 THEN 200
171 Z=Z+1
180 GO TO 210
200 CALL "APNT"(X,Y)
210 CALL "SUBP"(Z,A)
220 Z=Z+1\X1=X\Y1=Y
230 GO TO 130
500 CALL "ERAS"
510 END
```

J.3 BUILDING A LOAD MODULE

GT support is provided on 3 object files and 6 MACRO files:

GTNLPS.OBJ	Object module linked when LPS is not also linked
GTB.OBJ	Main Kernel Module
GTC.OBJ	Overlay Module

PERVEC.MAC	Vector Definition Module
FTBL.MAC	Function table
BASINT.MAC	BASIC interface module
RTINT.MAC	Interface module for BASIC/RT11 V01
PTSINT.MAC	Interface Module for BASIC/PTS V01
PERPAR.MAC	Parameter file

Two running versions are also included in the software provided on DECtape, cassette or disk.

BASGT.SAV	Non-overlaid version of BASIC with GT support
BGTLPS.SAV	Non-overlaying version of BASIC with GT and LPS support.

See section J.1 for a description of the load modules provided.

To build a load module BASGT.SAV (BASIC with GT support) first transfer all GT and BASIC files to the system device with PIP or PIPC. The parameter file PERPAR.MAC is then edited and assembled with FTBL.MAC, PERVEC.MAC, and the interface module. The three object modules produced are linked with GTB.OBJ, GTC.OBJ (and GTNLPS if the LPS is not also linked) and the BASIC object modules to produce a load module.

NOTE

All the procedures for editing PERPAR.MAC in this section assume that an unaltered PERPAR.MAC is being used. It is recommended that a copy of the original PERPAR.MAC be made and saved for future use.

The BASINT.MAC interface module should be used with all versions of BASIC except BASIC/RT11 V01 which should have RTINT.MAC used in the place of BASINT.MAC. If a background routine is also linked with BASIC, it must be defined in the interface module. See section 8.8.1 of the BASIC/RT11 Language Reference Manual for instructions to link a background routine and GT Support with BASIC.

For the GT routines to be accessible by a BASIC "CALL" statement the routines must be defined in a System Function Table as described in Section 8.2 of the BASIC/RT11 Language Reference Manual. FTBL.MAC is a function table in source form. If any user written assembly language routines (or LPS support) are also linked with BASIC the routines must be defined in this function table. See section 8.2.1 of the BASIC/RT11 Language Reference Manual for instructions to add assembly language routine definitions to this function table.

A listing of PERPAR.MAC follows

```
        .TITLE PERPAR -- PERIPHERAL SUPPORT PACKAGE PARAMETER MODULE.
        ;
        ; DEC-11-LBPAA-A-LA      BASIC KERNEL V02-01
        ;
        ; COPYRIGHT (C) 1974
        ;
        ; DIGITAL EQUIPMENT CORPORATION
        ; MAYNARD, MASSACHUSETTS 01754
        ;
        ; THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO
        ; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
        ; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
        ; DEC ASSUMES NO RESPONSIBILITY FOR ANY ERRORS THAT
        ; MAY APPEAR IN THIS DOCUMENT.
        ;
        ; THIS SOFTWARE IS FURNISHED TO PURCHASER UNDER A
        ; LICENSE FOR USE ON A SINGLE COMPUTER SYSTEM AND
        ; CAN BE COPIED (WITH INCLUSION OF DEC'S COPYRIGHT
        ; NOTICE) ONLY FOR USE IN SUCH SYSTEM, EXCEPT AS MAY
        ; OTHERWISE BE PROVIDED IN WRITING BY DEC.
        ;
        ; DEC ASSUMES NO RESPONSIBILITY FOR THE USE
        ; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
        ; WHICH IS NOT SUPPLIED BY DEC.

        ; THE CONDITIONALS CONTAINED IN THIS MODULE AFFECT THE ASSEMBLY
        ; OF THE FUNCTION TABLE MODULE "FTBL.MAC".
        ; TO OBTAIN THE DESIRED CONDITIONAL DEFINITION(S),
        ; REMOVE (USING AN EDITOR) THE
        ; SEMI-COLON APPEARING BEFORE THE CONDITIONAL.
        ;$DISK=0                 ;DEFINE FOR RT-11
                .IFNDF $DISK
        $STRNG=0                ;DO NOT DEFINE FOR PTS BASIC WITHOUT
                                ;STRINGS,- DEFINED FOR PTS V01 WITH STRINGS
                .ENDC           ;

        ;$LPS=0                 ;DEFINE FOR LPS
                .IFDF $LPS
        ;$V=0                   ;DEFINE FOR LPS WITH VECTORS STARTING
        ;                       ;     AT 300.  DEFAULT SETTING IS VECTORS AT
        ;                       ;     340. SET $V = ANY OTHER DISPLACEMENT IF
        ;                       ;        VECTORS START AT DISPLACEMENTS
        ;                       ;        OTHER THAN 0 OR 40 FROM
        ;                       ;        VECTOR 300

        $ADC=0                  ;INCLUDE A/D ROUTINES.
        $CLK=0                  ;INCLUDE CLOCK ROUTINES.
        $DIO=0                  ;INCLUDE DIGITAL IO ROUTINES
        $DIS=0                  ;INCLUDE DISPLAY ROUTINES.
                .ENDC   ;$LPS
        ;
        ;
        ;$VT11=0                        ;FOR GT40 (GT44)
        ;
        ;


                .IFDF   $VT11
        $CLOCK=0                        ;FOR SYSTEM CLOCK (KW11L)
                .ENDC

                .EOT
```

BASIC with GT support and no LPS support

The instructions for editing PERPAR.MAC and then assembling and linking BASIC with GT and no LPS support in the RT-11 environment follow:

(S) represents the ALTMODE key

```
.R EDIT
*EBPERPAR.MAC(S)R(S)(S)
*F;$DISK=0(S)0AD (S)(S)
*F;$VT11=0(S)0AD(S)(S)
*EX$$

.R MACRO
*FTBL=PERPAR,FTBL
ERRORS DETECTED: 0
FREE CORE: 15373. WORDS

*PERVEC=PERPAR,PERVEC
ERRORS DETECTED: 0
FREE CORE: 15480. WORDS

*RTINT=PERPAR,RTINT
ERRORS DETECTED: 0
FREE CORE: 15496. WORDS

*^C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*^C
.R LINK
*BASGT,BASGT=BASICR,FPMP,FTBL,PERVEC,RTINT/B:400/T/C
TRANSFER ADDRESS =
GO
*GTNLPS,GTB/C
*BASICE/O:1/C
*BASICX/O:1/C
*GTC/O:1/C
*BASICH/O:2


*^C
```

Or a core resident version may be linked to increase the execution speed. In that case, the following sequence is suggested:

```
.R LINK
*BASGT,BASGT=BASICR,FPMP,BASICE,BASICX/B:400/C
*FTBL,PERVEC,RTINT/C
*GTNLPS,GTC,GTB,BASICH


*^C

.
```

The core resident version will execute more quickly than the overlaid version and is recommended for DECtape systems.

BASIC with GT and LPS support

To build a load module including both GT and LPS support (with all the optional modules) the following instructions should be given to the RT-11 editor, assembler, and linker.

```
.R EDIT
*EBPERPAR.MAC$R$$
*F;$DISK=0$0AD$$
*F;$LPS=0$0AD$$
*F;$VT11=0$0AD$$
*EX$$

.R MACRO
*FTBL=PERPAR,FTBL
ERRORS DETECTED: 0
FREE CORE: 15273. WORDS

*PERVEC=PERPAR,PERVEC
ERRORS DETECTED: 0
FREE CORE: 15383. WORDS

*RTINT=PERPAR,RTINT
ERRORS DETECTED: 0
FREE CORE: 15472. WORDS

*^C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*^C
.R LINK
*BGTLPS,BGTLPS=BASICR,FPMP,FTBL,PERVEC,RTINT/B:400/T/C
TRANSFER ADDRESS =
GO
*LPS0,LPS1,LPS2,LPS3,LPS4/C
*GTB/C
*BASICE/O:1/C
*BASICX/O:1/C
*GTC/O:1/C
*BASICH/O:2


*^C

.
```

or for a nonoverlaying version

```
.R LINK
*BGTLPS,BGTLPS=BASICR,FPMP,BASICE,BASICX/B:400/C
*FTBL,PERVEC,RTINT/C
*LPS0,LPS1,LPS2,LPS3,LPS4/C
*GTC,GTB,BASICH


*^C

.
```

J.3.1 Assembling GT Sources

BASIC GT support may also be purchased in source form. The following files are provided in the source kit:

```
GTB.MAC
GTC.MAC
PERVEC.MAC
FTBL.MAC
BASINT.MAC
RTINT.MAC
PTSINT.MAC
PERPAR.MAC
GTNLPS.MAC
```

FTBL.MAC is the function table for the GT support functions. It will always be provided in source form and may be extended by the careful addition of the user's own functions as described in section 8.1.1 of the BASIC/RT11 Language Reference Manual.

GTB.MAC is the main module containing the majority of code of the GT support functions.

GTC.MAC is a module that is normally linked into BASIC as an overlay co-existent with BASICE and BASICX. However it may be linked in a non-overlaid version at the expense of core for the user's program and data space.

PERVEC.MAC is the vector definition module. If the hardware configuration is non-standard the address of the graphic vector should be changed in the source. PERVEC.MAC is listed at the end of section I.8.1.

BASINT.MAC is the interface module that should be used with all versions of BASIC except BASIC/RT11 V01 and BASIC/PTS V01. The interface module for these versions should be RTINT.MAC and PTSINT.MAC respectively.

PERPAR.MAC is a parameter file that must be edited for assembling with all the remaining files. The user must edit the parameter file, PERPAR.MAC, that is used to conditionally assemble the remaining files according to the user's special requirements. Some of the parameters that may be defined are:

$VT11	Always define
$DISK	Define to assemble the RT-11 version.
$LONGER	Define to give longer GT error messages.
$CRASH	Define to cause a halt with an ?AOR error message rather than change data that is too large for the display instructions to use. Otherwise data will be transformed into the nearest legitimate value.
$CLOCK	Always define even when hardware does not include a real-time clock.
$LPS	Always define when assembling GTB and GTC.
$DEPTH	Define to change the default depth of nesting of subpictures. If not defined here, the default is 10 (decimal).

PERPAR.MAC is listed in the preceding section and is edited by the EDIT program. To assemble the GTB.MAC, GTC.MAC, and GTNLPS.MAC sources to duplicate the object modules provided, the following RT-11 Monitor instructions should be typed:

(ALT) represents the ALTMODE key

```
.R EDIT
*EBPERPAR.MAC$R(ALT)(ALT)
*F;$DISK=Ø(ALT)ØAD(ALT)(ALT)
*F;$LPS=Ø(ALT)ØAD(ALT)(ALT)
*F;$VT11=Ø(ALT)ØAD(ALT)(ALT)
*EX(ALT)(ALT)

.R MACRO
*GTB=PERPAR,GTB
ERRORS DETECTED: Ø
FREE CORE: 13646. WORDS

*GTC=PERPAR,GTC
ERRORS DETECTED: Ø
FREE CORE: 13965. WORDS

*GTNLPS=GTNLPS
ERRORS DETECTED: Ø
FREE CORE: 15227. WORDS

*↑C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*↑C
```

Building a load module may now be accomplished by following the instructions given in the preceding section.

GTB.MAC and GTC.MAC may have the optional $LONGER, $CRASH, and $DEPTH parameters defined in PERPAR.MAC. For example to create a BASIC with GT support for RT-11 with longer GT error messages, halting after error messages, and a Depth of nesting of 17(Octal) (decimal=15) allowed, the following instructions should be given to the RT-11 monitor.

```
.R EDIT
*EBPERPAR.MAC(ALT)R(ALT)(ALT)
*I$LONGER=Ø
$CRASH=Ø
$DEPTH=17
(ALT)(ALT)
*F;$DISK=Ø(ALT)ØAD(ALT)(ALT)
*F;$LPS=Ø(ALT)ØAD(ALT)(ALT)
*F;$VT11=Ø(ALT)ØAD(ALT)(ALT)
*EX(ALT)(ALT)

.R MACRO
*GTB=PERPAR,GTB
ERRORS DETECTED: Ø
FREE CORE: 13534. WORDS
```

```
*GTC=PERPAR,GTC
ERRORS DETECTED: Ø
FREE CORE: 13953. WORDS

*GTNLPS=GTNLPS
ERRORS DETECTED: Ø
FREE CORE: 15227. WORDS

*↑C

.R PIP
*PERPAR.MAC=PERPAR.BAK/R
*↑C
```

These GTB and GTC object modules may be linked with FTBL, PERVEC, RTINT, BASIC, and LPS object modules as described in the previous section. The PERPAR.MAC editing instructions should also enable the appropriate LPS parameters when they will be linked and the PERPAR.MAC produced may also be used to assemble FTBL, PERVEC, and RTINT.

J.3.2 Technical Description of Display File Management

The following technical information should be read before any user-written assembly language routines alter the graphic files or before the sources of GT are altered.

1. Overall Structure

   a. Root portion

      i) Tracking object subpicture (with tag -1)
      ii) Default conditions; i.e., beam at (0.0) etc.
      iii) A display jump to the user space

   b. User space

      i) User data starts at address in global ACTST
      ii) Data continues through address in global ACTEND
      iii) A display jump and address of root code is stored in the two words ending at ACTEND
      iv) The top of the display buffer is the address in global DCRASH

2. Subpictures

   a. Call consists of 5 words:

      i) A display stop (code:173400)
      ii) Address of the rest of the file
      iii) Address of the subpicture
      iv) The tag of the subpicture
      v) The pointer to the next tag or zero if the last subpicture.

b. Header of the subpicture consists of one spare word used in the SAVE routine. The address in 2a-ii points to the next word.

c. The end of a subpicture is given by 2 words:

   i) A display stop
   ii) A zero word

d. Subpictures are turned off and on by interchanging the display stop of 2a-i by a display jump respectively.

e. When subpictures are erased:

   i) The display stop of 2a-i is replaced by a display jump.
   ii) The tag of the subpicture and the tags of any sub-pictures contained in it are deleted from the linked list.

f. The first subpicture tag pointer is located at address global TAG1+2.

3. Display stop handler

Assume DPU interrupts with the address X in the display PC.

a. Call to subpicture if the contents of X is greater than X+2. In this case X+2 is stored in a runtime stack (the subpicture tag is at X+4). The address in X (see 2a-iii above) is moved to the DPC to start the display.

b. Addition to display file if the contents of X is less than X+2. This condition arises when the display jump at ACTEND-2 is replaced by a display stop. The words are inserted and the display is started at its beginning in the root code.

c. End of a subpicture if contents of X is zero. The top entry in the runtime stack is popped off into R0 and the display is started at the address contained in R0-2.

4. Light Pen Handler

a. Checks the runtime stack to see if the hit occurred in a subpicture.

b. If a hit occurred in the tracking object, move the object's center to new coordinates weighted by 3 times the old X (Y) plus the new X (respectively Y) divided by 4. If the tracking object is on the edge, move it to the center. Continue at 4d.

c. If a hit occurred in a subpicture, store its tag, otherwise store zero unless a previous hit has not been queried, in which case, go to 4e.

d. Unscale to user coordinates and if the hit was in the tracking objecta update the variables passed in the call to the TRAK routine.

e. Resume the display and exit.

5. Display Time-out

   a. Reset the runtime stack and restart the display.

6. Calls to XGRA, YGRA, FIGR.

   a. The following code is generated for XGRA and YGRA:

      i) Set graphic mode to XGRA or YGRA
      ii) Load Status Register B with the graphplot increment.
      iii) A display jump
      iv) Address of the first word of A(0) where the screen-scaled array data starts.
      v) The actual first subscript of the array. Stored as negative if an LPS array.

   b. The display jump at the "end" of the array points to the word following 6a-v.

   c. For FIGR. 6a-i and 6a-ii are replaced by a single word to set the graphic mode to long vector.

7. File structure as a "Saved File"

   a. The saved file has no references to "GRAPH" arrays since restoring the file into a program without such arrays is meaningless.

   b. The file has a total of n+2 words where the first word is "n" and the second word is a relative offset from the next word to the first subpicture tag if there is one; else it is zero.

   c. The file is stored entirely in relocatable form relative to the third word of the file. Any restoration by the user with his own routines may result in the storage of meaningless information unless care is taken to locate all the addresses properly.

HOW TO OBTAIN SOFTWARE INFORMATION

SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes newsletters and Software Performance Summaries (SPS) for the various Digital products. Newsletters are published monthly, and contain announcements of new and revised software, programming notes, software problems and solutions, and documentation corrections. Software Performance Summaries are a collection of existing problems and solutions for a given software system, and are published periodically. For information on the distribution of these documents and how to get on the software newsletter mailing list, write to:

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

SOFTWARE PROBLEMS

Questions or problems relating to Digital's software should be reported to a Software Support Specialist. A specialist is located in each Digital Sales Office in the United States. In Europe, software problem reporting centers are in the following cities.

Reading, England	Milan, Italy
Paris, France	Solna, Sweden
The Hague, Holland	Geneva, Switzerland
Tel Aviv, Israel	Munich, West Germany

Software Problem Report (SPR) forms are available from the specialists or from the Software Distribution Centers cited below.

PROGRAMS AND MANUALS

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center.

Digital Equipment Corporation	Digital Equipment Corporation
Software Distribution Center	Software Distribution Center
146 Main Street	1400 Terra Bella
Maynard, Massachusetts 01754	Mountain View, California 94043

Outside of the United States, orders should be directed to the nearest Digital Field Sales Office or representative.

USERS SOCIETY

DECUS, Digital Equipment Computer Users Society, maintains a user exchange center for user-written programs and technical application information. A catalog of existing programs is available. The society publishes a periodical, DECUSCOPE, and holds technical seminars in the United States, Canada, Europe, and Australia. For information on the society and membership application forms, write to:

DECUS	DECUS EUROPE
Digital Equipment Corporation	Digital Equipment Corporation (Europe)
146 Main Street	P.O. Box 340
Maynard, Massachusetts 01754	1211 Geneva 26
	Switzerland

BASIC/RT11 Language
Reference Manual
DEC-11-LBACA-D-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

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 do not require a written reply, please check here. ☐

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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 EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS 01754