1 February 1977

NATIONAL BASIC (PACE) PROGRAMMER REFERENCE MANUAL

Table of contents

Table of contents

Table of contents

1. INTRODUCTION

NATIONAL BASIC is an interpreter for an enhanced version of the BASIC computer programming language. It is highly interactive, allowing users to modify, test, and run programs without requiring intermediate storage of the program on paper tape or other media. BASIC is intended for applications requiring a simple high-level language but not requiring high speed or small amounts of memory. Since BASIC is an interpretive language, no object code is generated; the program statements are kept in an internal form and interpreted each time the statement is executed.

This manual is a reference manual for the details of the NATIONAL BASIC language; it is not intended as a primer. Familiarity with high level computer programming languages is assumed; familiarity with BASIC is desirable but not required. There are many excellent introductory textbooks on BASIC which should be consulted if a more tutorial approach is desired.

1.1. System Requirements

The BASIC interpreter described in this manual runs on a PACE microcomputer system with a minimum of 8K words of memory (0000-1FFF) and the standard I/O firmware ROMs. A Teletype or Teletype-like terminal is also required. Other peripherals supported are a high speed printer (Centronics 306), and a high speed paper tape reader and/or punch (via the firmware). BASIC uses about 6.5K words of memory for itself, leaving about 1.5K words for the user program and variable storage in an 8K system.

The PACE Disk Operating System (DOS) may be used for storage of of BASIC programs. If disk files are used, at least 12K words of memory are required (although BASIC itself will load in 8K, there will be very little user space in an 8K system when disk files are needed).

1.2. Features

NATIONAL BASIC includes the following features:

Statements are checked for proper syntax as they are entered;

Leading blanks are allowed in statements;

String and numeric arrays of one or two dimensions are provided;

Strings may be manipulated with substring functions or concatenation;

Boolean operators (AND, OR, XOR, NOT) are available for expressions and bit manipulations;

Assembly language subroutines may be called with the CALL statement and USR function;

TRACE ON and TRACE OFF may be used for logging program flow when debugging programs;

Peripherals may be directly accessed with the INP function and OUT statement;

Memory may be examined or altered with the PEEK function and POKE statement;

Hexadecimal constants and the HEX$ conversion function make memory and I/O use more convenient;

An extensive set of arithmetic operators includes MIN, MAX, and MOD, as well as boolean and relational operators which may be used in any expression;

FOR/NEXT loops are skipped if the terminating condition is not met in the initial execution.

2. STARTING THE INTERPRETER

When BASIC is loaded, it goes through an initialization sequence to set up internal variables needed for program execution. A message is printed identifying the version of the program, and the amount of available memory is determined. BASIC will then ask two questions regarding the system in use. The standard response for each is simply a carriage return.

The first question is "Disk Files Needed?". NATIONAL BASIC provides the capability to store programs on disk; this question determines whether disk files are required by the user. The question is asked only if the DOS monitor is present; if no monitor is present, no disk files are possible and the question is skipped. Valid responses are 'Y' (yes) or 'N' (no) or carriage return (yes). If no disk files are needed, the disk commands become illegal and the disk routine area will become part of the edit buffer, allowing larger programs to be used.

The second question is "Memory Size?". The response may be a number indicating the highest address to be used by BASIC (to allow room at the top of read/write memory for assembly language subroutines); the address may be specified in decimal or in hexadecimal ($xxxx). If a blank line is entered, BASIC will use all available contiguous read/write memory. If an improper response is received, or if the number is too small, the user will be asked for the information again. If the number entered is larger than the calculated memory size, the calculated value will be used instead.

When BASIC has initialized itself, it prints the message

READY

to indicate that the interpreter is now ready to receive commands or program lines. The "READY" message is also printed following the execution of a program, after certain system commands have been processed, and after execution time errors. "READY" indicates that the interpreter is now back in command mode. BASIC will then prompt for an input from the user by printing a question mark and a space ("? ").

3. PROGRAMS

BASIC is a line-oriented language. A program is composed of a sequence of lines ordered by line numbers. Program lines are executed in sequential order, starting with the first line, until

(1) some other action is dictated by a control statement,
(2) a fatal error condition occurs,
(3) the user interrupts execution of the program,
(4) a STOP statement or END statement is executed, or
(5) the last line of the program is executed without transferring control.

3.1. Line Numbers

Each line begins with a line number which does not contain any spaces. Leading spaces and leading zeroes will be ignored. If the value of the integer represented by the line number is zero, or if the line number is omitted, the statement is interpreted as a direct command rather than a program statement. The range of allowable line numbers is 1 through 9999.

3.2. Program Lines

A program line consists of a line number followed by one or more statements. Any number of spaces may be used to separate the line number and the first statement on the line; these spaces may be used to indent FOR-blocks to improve program readability.

Program lines may be entered in any order; they are sorted into ascending line-number order by the editor. Entry of a line with a line number equal to a previously existing line number will cause the previous line to be replaced by the new line. Entry of a line number with no statement following it will delete any existing line with that line number.

The maximum length of a program line is determined by the WIDTH command. Initially, the maximum length is set to 72 characters (not including the carriage return). The maximum line length that can be set by the WIDTH command is 132 characters, and the minimum width is 15 characters.

3.3. Statements

Each statement in BASIC begins with a "keyword" which identifies the statement type. Other keywords in BASIC are the command names and function names. These keywords may be abbreviated to the first three characters when the program is being entered; the keyword is replaced by a one-byte code by the editor. When the keyword is printed, the word is completely spelled out, regardless of how it was entered.

The statements supported by NATIONAL BASIC are described in later chapters of this manual.

3.3.1. Multiple Statements On A Single Line

A program line may consist of more than one statement; statements are separated by a colon (":"). Only the first statement of a program line contains a line number. An example of a program line containing three statements is:

```
100 A=4 : PRINT J*5 : GOTO 999
```

Any statement may be used anywhere within a program line, with one restriction:

> Any statement that may transfer control, or is non-executable, may not be followed by another statement on the same program line. These statements are DATA, DEF, DIM, END, FOR, GOTO, GOSUB, ON, NEXT, RETURN, STOP, and all system commands.

The initial release of NATIONAL BASIC also requires that DATA and NEXT statements be the only statement on a program line.

3.4. Program Entry and Editing

BASIC is a conversational language which allows the user to communicate with the language interpreter by typing on the terminal keyboard. All inputs from the terminal are solicited with a question mark and space ("? ") prompt sequence. The prompt indicates that BASIC is ready to accept a line from the keyboard. This section describes those characters that have special significance to BASIC.

3.4.1. Carriage Return

The carriage return is used to terminate the line being entered and cause it to be processed by BASIC.

3.4.2. Backspace

Either the ASCII backspace character (Control/H) or the underline (left-arrow) may be used to backspace and correct a character which was entered incorrectly. For each backspace typed in, one previously entered character is deleted. As many backspaces may be typed as needed to correct the mistake.

3.4.3. Null, Rubout, Line Feed, Escape

Four characters are ignored on input, primarily to facilitate the reading of paper tapes. These characters are NULL, RUBOUT, LINE FEED, and ESCAPE. Escape sequences consist of the ESCAPE character followed by any ASCII character; both characters of the escape sequence are ignored. The ignored characters are echoed to the terminal (except in paper tape mode) but will not be included in statement text.

3.4.4. Line Abort

Any control character not otherwise used (backspace, line feed, tab, carriage return) will cause the line being entered to be aborted; the user may then try entering the line again. The characters commonly used are Control/Q and Control/C. Control/C echoes as ^C , and other control characters will echo as two backslashes (\\).

3.4.5. Tab

The tab character (Control/I) may be used to cause a tabulation in the listing. Tab stops are eight columns apart, at columns 9, 17, 25, 33, etc. The tab is echoed as itself when entering the line, but is always printed as the proper number of space characters when the line is listed. Use of a tab within a string will cause blanks to be printed when the program is listed, but the string will contain the tab character when the program is executed. Runtime printing of a string with a tab character will print the tab character rather than the blanks.

3.4.6. Lower Case

Lower case characters are accepted by the editor and are automatically converted to upper case where necessary. Lower case is preserved only within strings, remarks, DATA statements, and INPUT responses.

3.4.7. ETX (Control/C)

Typing the ASCII End-of-text (Control/C) character in response to an input prompt will not only abort the line being typed in, but will also terminate the current input mode. ETX will terminate paper tape input (if desired to quit reading the tape before the END line) and AUTO mode. An ETX in response to an INPUT solicitation stops the program and returns to command mode; a CONTINUE would then retry with the INPUT. The ETX is echoed to the terminal as ^C .

4. DATA

4.1. Constants and Strings

4.1.1. Numeric Constants

All numbers are stored internally in floating point representation with one sign bit, 23 mantissa bits, and eight exponent bits (including sign). Floating point numbers may thus be in the range of 1E-38 to 1E+38.

Numeric constants are expressed in scientific notation with an optional sign and an optional exponent. There are four general forms of numeric constants:

```
sd...d
sd...drd...d
sd...drd...dEsdd
sd...dEsdd
```

where

- d is a decimal digit,
- r is a period,
- s is an optional sign (+ or -), and
- E is the explicit character E.

Examples of numeric constants include:

```
2.   1   500   -21.   .255   1E10
5E-1   .4E3   123.456E7
```

Numeric constants may contain an arbitrary number of digits (with a minimum of one), but only six digits of significance are maintained. Non-zero constants with exponents outside the range -38 to 38 will cause an underflow or overflow warning, but will otherwise be accepted.

4.1.2. Hexadecimal Constants

A hexadecimal constant consists of a dollar-sign ("$") followed by one to four hexadecimal digits (0-9, A-F). A hexadecimal constant may be used wherever a floating point number is allowed. The hexadecimal value is converted from binary (integer) to floating point when it is used.

4.1.3. Quoted Strings

The quotation mark (") denotes the start and ending of a string. Any printable ASCII character (plus space and tab) may be included in the string constant. A quotation mark may be included in the string by the use of two adjacent quotation marks. For example,

```
100 PRINT """"
```

will print one quotation mark.

4.1.4. Unquoted Strings

Strings in DATA statements and INPUT responses do not need to be included in quotation marks if the string does not contain any leading blanks or any of the terminating characters. The characters which will terminate an unquoted string are: comma (,), apostrophe ('), ampersand (&), and exclamation point (!). These characters are specified in the ANSI standard because they may have special significance to certain implementations.

4.2. Variables

A variable is a data item that contains a value that may be changed under program control. The LET, INPUT, and READ statements are used to assign values to variables.

BASIC allows two types of variables (simple or subscripted) in two modes (numeric or string). A numeric variable contains a number as its value; a string variable contains a character string as its value; the character string may be of 0 to 72 characters in length. Subscripted variables contain one or more values, depending upon the number of dimensions to the array. Each element of a subscripted variable is a number (for numeric arrays) or a complete string (for string arrays).

4.2.1. Variable Names

A variable name consists of a single letter or a letter and a digit. A variable name may also consist of the first two letters in a sequence of letters, with the restriction that no keyword be embedded within the variable name, since keywords are replaced by an internal code regardless of the characters surrounding the keyword in the statement. Blanks are never allowed within a name.

A string variable is denoted by a variable name followed by a dollar sign ("$"). A subscripted variable consists of a variable name followed by a subscript list in parentheses.

Each of the four types of variables is independent, even though the names may have the same root. For example, the simple numeric variable A , the array A(5) , the string variable A$, and the string array A$(2,3) may all exist simultaneously without conflict.

Some variable name examples are:

```
Simple Numeric:     A   C4   Z9
Numeric Array:      A(10)   Q1(12,5)
Simple String:      A$   Q7$
String Array:       A$(3)   Z7$(5)   XX$(12,4)
Illegal:            ATO   C77   7FX
```

4.3. Expressions

Expressions may contain any numeric function, simple variable, array variable, or constant, using any mathematical, logical or relational operator. Parentheses may be used to group subexpressions.

4.3.1. Mathematical Operators

The mathematical operators are:

+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
^ or **	Exponentiation
MOD	Modulus (remainder) function
MAX	Maximum function
MIN	Mininum function

All mathematical operators have a binary format such as A*B. The + and - operators also have a unary form, where the operator is not preceded by a value or expression.

4.3.2. Relational Operators

The relational operators are used for comparing the values of numbers or strings. The relational operators return the result of -1 if the relation is true and 0 if the relation is false. The operators are:

<	Less than
<= or =<	Less than or equal
=	Equal
<> or ><	Not equal
>= or =>	Greater than or equal
>	Greater than

The alternative forms of the relational operators (>< , =< , =>) are automatically translated to the standard notation for compatibility.

Comparisons of strings are done in alphabetical order using the collating sequence of the ASCII code. If the strings are of different lengths, the shorter string will be padded on the right with nulls for the comparison. Therefore, the string "Y " is greater than "Y". A string comparison may be used in a PRINT statement only if it is enclosed in parentheses.

4.3.3. Logical Operators

The logical operators are:

- **AND** Logical product: A AND B is true only if both A and B are true.
- **OR** Logical sum: A OR B is true if either (or both) of A or B is true.
- **XOR** Logical difference: A XOR B is true if either A or B is true, but not both.
- **NOT** Logical negation (unary): NOT A is true if A is false; NOT A is false if A is true.

All of the logical operators convert their operands to integer format, perform the logical function, and then convert the result back to floating point format. The logical operators may therefore be used for bit manipulations as well as for logical testing. A value is true if it is non-zero, and false if it is zero.

Examples:

```
A AND $FF          returns the lower 8 bits
                   of the value in A

$5555 XOR $0FF0    returns the value $5AA5
```

4.3.4. Operator Precedence

```
Highest:    String comparisons  < <= = <> >= >

            NOT  -  +   (unary operators)
            MOD
            *  /
            +  -
            MAX  MIN
            <  <=  =  <>  >=  >
            AND
Lowest:     OR  XOR
```

Special case: unary operators which occur immediately following another operator have the highest precedence and are always grouped with the expression immediately following the operator.

Parentheses may be used to force different grouping of operations than would occur normally. Approximately ten levels of parentheses may be used (depending upon the operators involved and the complexity of the expression).

Example	Equivalent formula
A+-B	A+(-B)
A^+B	A^(+B)
A-B*C	A-(B*C)
A AND B OR C	(A AND B) OR C
-A^B	-(A^B)
-A*B	(-A)*B
A>B AND C OR D	((A>B) AND C) OR D
A*B>C	(A*B)>C
A*B$>C$	A*(B$>C$)

5. REMARKS AND COMMENTS

5.1. REM Statement

The remark statement has the general form:

REM remark-string

The remark-string may contain any printable ASCII characters, including blank and tab, and is terminated by the end of the line (carriage return).

The sole function of the remark is to provide a means of placing explanatory information into the program listing; no action is performed when a remark is executed. A control statement may branch to a remark statement.

5.2. Comments

The apostrophe (') is a special remark character; it may be used even on statements that do not normally allow another statement (e.g., after a GOTO). For example,

```
 .
100 IF A<B THEN 999      ' TEST IF DONE
120 GO TO 10             ' COMPUTE AGAIN
 .
```

The comment beginning with the apostrophe is treated just like a REM statement; its advantage is that it may be used where a REM is not allowed. For economy of storage and readability of the listing, it is recommended that the TAB character be used to precede the comment.

6. ASSIGNMENTS: LET STATEMENT

The LET statement provides for the assignment of the value of an expression to a variable. Numeric values may be assigned to numeric variables, and string values may be assigned to string variables. The general form of the LET statement is:

```
LET variable = expression
```

The expression is evaluated and its value is assigned to the variable to the left of the equals sign. The keyword LET is optional.

String variables may be assigned string values; the string expression may have the form

```
string + string + ... + string
```

The "+" operator, if used, causes two strings to be concatenated (pieced together end-to-end to form a longer string). Each string may be a string constant, a string variable, or a string function. The length of the string variable following the assignment will be the sum of the lengths of the strings, with a maximum of 72 characters. If the string length would exceed 72, the excessive characters on the right will be lost. For example,

```
LET A$="THIS IS A"+" STRING"
```

will assign A$ the value "THIS IS A STRING".

String concatenation is done to a temporary string variable before the final assignment is made; therefore the following program will assign the value "DEFABC" to A$:

```
10 A$="ABCDEF"
20 A$=RIGHT$(A$,3)+LEFT$(A$,3)
```

7. CONTROL STATEMENTS

Control statements allow for the interruption of the normal sequence of execution of statements by causing execution to continue at a specified line, rather than at the one with the next higher line number.

7.1. GOTO

The GOTO statement

```
GOTO line-number
```

allows for an unconditional transfer. The next statement to be executed will be the first statement on the designated line.

The keyword GOTO may be spelled as two words, as in

```
GO TO line-number
```

The split form of the GOTO may be used wherever the GOTO keyword is specified, including the IF and ON statements.

> Note: the "GO TO" form takes at least three bytes of storage while the "GOTO" takes only one byte.

No additional statement may appear following a GOTO on the same program line (except for a comment beginning with ').

7.2. ON ... GOTO

The ON - GOTO statement

```
ON expression GOTO line-number, line-number, ...
ON expression GO TO line-number, line-number, ...
```

allows control to be transferred to a selected line. The expression is evaluated and rounded to an integer whose value is then used to select a line-number from the list following the GOTO. The line numbers in the list are indexed from left to right, starting with 1. Execution of the program continues at the line number selected by the expression index.

An error will occur and program execution will be stopped if the value of the expression is less than one or greater than the number of line numbers in the list.

7.3. IF ... THEN

The IF statement has three forms:

```
IF expression THEN line-number
IF expression GOTO line-number
IF expression THEN statement
```

The expression is evaluated and tested. If the value is zero (false) then the rest of the IF statement is ignored and the program line with the next higher line number is executed. If the value of the expression is not zero, the expression is "true" and the second part of the IF statement is processed.

The first two forms of the IF statement are equivalent; execution is transferred to the line with the designated line number if the expression is true (non-zero).

The third form of the IF statement is very useful for writing readable programs with a minimum number of statements. The statement or statements specified will be executed if the IF-expression is true. Any executable BASIC statement may be used on the IF line except DATA, DEF, DIM, FOR, NEXT, or REM.

7.4. FOR, NEXT

The FOR and NEXT statements provide for the construction of program loops. The general forms of these statements are

```
FOR var = initial-value TO limit STEP increment
NEXT var
```

The sequence of statements beginning with a FOR statement and continuing up to and including the first NEXT statement with the same control variable is termed a "For-block". For-blocks may be nested, i.e., one may contain another (providing that they use different control variables), but they may not be interleaved.

In the absence of a STEP clause in a FOR statement, the increment is assumed to be +1.

Execution of the FOR statement causes the limit-value and increment-value expressions to be evaluated and saved in a special area reserved for the for-block control variable. The initial value is assigned to the control variable, and the test is then made to determine whether the loop is to be processed. If the initial variable value is greater than the limit (for a positive increment), or less than the limit (for a negative increment), then the loop is skipped and execution of the program resumes at the statement following the NEXT statement which closes the for-block.

A for-block becomes active upon execution of its FOR statement and remains active until it is exited via its NEXT statement or until control is transferred to a FOR statement (which may or may not be the one associated with that for-block) having the same control variable.

Nesting of the for-blocks is checked during pass one of a RUN command; an error will occur if blocks are nested improperly or more than ten deep.

For proper operation, the NEXT statement must be the only statement on its program line. The syntax analyzer will give an error only if it is not the last statement on the line.

The value of the control variable upon exit from the for-block via its NEXT statement is the first value not used; if exit is via a control statement, the control variable retains its current value and the for-block remains active.

Transferring into an inactive for-block will cause an error when the NEXT statement is encountered. Transferring into an active for-block causes the NEXT statement to use the values last associated with the specified control variable.

7.5: STOP, END

The STOP and END statements provide means for terminating a program. The major difference between the two statements is that the STOP statement will print a message

STOP IN nnnn

where nnnn is the line number of the line containing the STOP statement. The END statement also terminates the program, but without any message as to the line number of the line containing the END statement. Control is transferred to command mode and the "READY" message indicates that the program is finished and BASIC is ready to accept a command or new program lines.

The END statement does not need to be the last statement of the program, nor does the program need to end with an END statement. The END statement does, however, cause program entry from paper tape to be terminated; therefore, the END statement should not be used in the middle of a program (use a GOTO or STOP instead).

8. INPUT/OUTPUT

8.1. DATA, READ, RESTORE

The DATA statement provides for the creation of a sequence of representations for use by the READ statement. The general syntactic form of the DATA statement is

```
DATA datum, ..., datum
```

where each datum is either a numeric constant, a string constant, or an unquoted string constant.

The READ statement provides for the assignment of values to variables from a sequence of data created from DATA statements. The RESTORE statement allows the data from DATA statements in the program to be reread. The general syntactic forms of the READ and RESTORE statements are

```
        READ variable, ..., variable
    and RESTORE
```

Data from the DATA statements in the program are read from a single data sequence. The order in which data appear textually determines the order of the data in the data sequence. An unquoted string which is a valid numeric representation may be assigned to either a string variable or a numeric variable by a READ statement.

If the execution of a program reaches a line containing a DATA statement, it proceeds to the next line with no other effect.

The DATA statement must be the only statement on its program line. The syntax analyzer will give an error message only if the DATA statement has improper syntax.

The READ statement causes variables in the variable list to be assigned values, in order, from the data sequence. A pointer is associated with the data sequence& At the initiation of execution of a program, this pointer points to the first datum in the data sequence. Each time a READ statement is executed, variables in the variable list are assigned values from the data sequence beginning with the datum indicated by the pointer, and the pointer is advanced to point behind the data used.

The RESTORE statement resets the pointer for the data sequence to the beginning of the sequence, so that the next READ statement executed will read data from the beginning of the sequence once again.

The type of a datum in the data sequence must correspond to the type of the variable to which it is to be assigned; i.e., numeric variables require numeric constants as data, and string variables require quoted strings or unquoted strings as data.

Subscript expressions in the variable list are evaluated after values have been assigned to the variables preceding them (i.e., to the left of them) in the list.

There is no provision for null data items, i.e., for two adjacent commas in a DATA statement. If the null string is to be included as a datum, it must be enclosed in quotation marks.

Errors resulting from underflows or overflows in the conversion of numeric data will cause a warning and the standard action for expressions to be taken; i.e., an underflow warning will use zero as the value, and an overflow warning will use the maximum representable value. The program will proceed normally in case such errors occur.

8.2. PRINT

The PRINT statement is designed for simple generation of labeled and unlabeled output or of output in a consistent tabular format. The general syntactic form of the PRINT statement is

```
PRINT item p item p ... p item
```

where each item is either a string, an expression, a print function, or null; and each punctuation mark p is either a comma, a semicolon, or null.

The execution of a PRINT statement generate a string of characters for transmission to an external device (normally the console). This string of characters is determined by the successive evaluation of each print-item and print-separator in the print-list.

Numeric expressions are evaluated to produce a string of characters consisting of a leading space if the number is positive, or a leading minus sign if the number is negative, followed by the decimal representation of the number and a trailing space.

Each number that can be represented exactly as an integer with 6 or fewer decimal digits will be printed using the standard integer representation without a decimal point.

Numbers which are not integers, but have values in the range 0.01 to 999999.5 , are represented in fixed point format with a maximum of 6 significant digits and a decimal point; trailing zeroes in the fractional part will be omitted.

All other numbers will be represented in scientific notation; for example,

sign significand E sign two-digit-integer

where the value x of the significand is in the range 1 <= x < 10 and is to be represented with six digits of precision. If the first sign is a plus sign, it is replaced by a space.

String constants, string variables, and string functions are evaluated (if necessary), generating a string of characters exactly corresponding to the string itself.

The evaluation of the semicolon separator (or an internal null separator) generates a null string, i.e., a string of zero length.

The evaluation of a comma separator depends upon the string of characters already generated by the current or previous print statements. The "current line" is the (possibly empty) string of characters generated since the last end-of-print-line was generated. The "margin" is the number of characters that may be printed on one line; the "columnar position" of the current line is the print position that will be occupied by the next character printed on that line. Print positions are numbered consecutively from the left, starting with position one.

Each print line is divided into a fixed number of print zones. Each print zone is fourteen (14) characters wide; the number of print zones on a line depends upon the current width of the line. The last print zone on the line may be longer than fourteen characters if the line width is not a multiple of fourteen.

The evaluation of the comma separator generates enough spaces to fill out the current print zone, unless this is the last print zone on the line, in which case a carriage return - line feed sequence is printed such that the next character to be printed will be in column one of the next print line.

If the print list does not end in a print separator (comma or semicolon), the print line is ended with a carriage return and line feed, and the next PRINT statement to be executed will then print on a new line.

If the evaluation of any print item in a print list would cause the columnar position of a nonempty line to exceed the margin, a new print line is started before the characters generated by that print-item. If a string is printed that is longer than the number of characters in a print line, a new print line is generated each time the columnar position of the current line exceeds the margin.

A completely empty print-list will generate an end-of-print-line (carriage return, line feed), thereby completing the current line of output. If this line contained no characters, a blank line results.

Null print-items, as in PRINT ,,X, are allowed as a means of tabulating over several print zones.

8.3. Print Functions

The print functions may be used only in PRINT statements.

TAB(I) Sets the columnar position of the current line to the specified value prior to printing the next print item. The argument I is rounded to the nearest integer to determine the new print column. Columns are numbered starting at column 1 on the left. If I is greater than the line width m, then the value I is reduced by an integral multiple of m such that 1 <= I <= m . If the columnar position is less than or equal to I, enough spaces are generated to move the columnar position to I. If the columnar position is greater than I, a new print line is initiated and spaces generated to move to column I.

SPC(I) The argument is rounded to an integer and the corresponding number of spaces is printed.

LIN(I) The argument is rounded to an integer and the corresponding number of line feeds is printed.

8.4. INPUT

INPUT statements provide for interaction with a running program by allowing variables to be assigned values that are supplied from a source external to the program. The input statement enables the entry of mixed string and numeric data, with data items being separated by commas. The general syntactic form of the INPUT statement is

```
INPUT variable, ..., variable
```

The INPUT statement causes the user to be prompted at the terminal to supply a data list. Once the data has been typed, the INPUT statement will cause the variables in the variable-list to be assigned, in order, the values from the input-reply.

The type of each datum in the input-reply must correspond to the type of the variable to which it is to be assigned; i.e., numeric constants must be supplied as input for numeric variables, and either quoted strings or unquoted strings must be supplied as input for string variables. If the response to an input for a string variable is an unquoted string, leading blanks are ignored.

Subscript expressions in the variable list are evaluated after values have been assigned to the variables preceding them (i.e., to the left of them) in the variable list.

If the data supplied on the input-reply is insufficient to fill the variables in the variable list, the user is prompted by a double question mark ("??") for more input. If more data is supplied than is required for the variable list, the extra data is ignored and no error message is printed.

If the data supplied is not the correct type for the corresponding variable, or if a syntax error is detected in the input-reply, the message "ERROR: RETYPE LINE" is printed and the user is prompted to retype the entire input-reply.

A prompt string may be supplied on the INPUT statement; the general form is

```
INPUT "string" p variable, ..., variable
```

where the string may be any legal quoted string constant, and p is a print delimiter (either a comma or a semicolon). The string and the print delimiter are printed in the same manner as they would be printed by a PRINT statement, then the prompt is printed for the input reply.

9. SUBROUTINES

A subroutine is a section of code performing some operation required at more than one place in the program. The GOSUB statement is used to transfer control to the subroutine and the RETURN statement is used to return control to the place from which the subroutine was called.

9.1. GOSUB

The GOSUB statement has the general syntactic form

GOSUB line-number

The GOSUB statement is processed in the same manner as the GOTO statement, except that the address of the next line in the program (after the GOSUB) is saved on a stack to be retrieved by a RETURN statement. GOSUB may be spelled as either one word or two words ("GO SUB"). "GOSUB" will occupy one byte of program storage, while "GO SUB" will occupy at least three bytes of storage.

9.2. ON ... GOSUB

The ON-GOSUB statement is similar to the ON-GOTO statement except that the action taken upon selection of one of the line numbers in the list is a subroutine jump (GOSUB statement) rather than an unconditional jump (GOTO statement). An error will occur if the expression has a value less than one or greater than the number of line numbers in the list.

9.3. RETURN

The RETURN statement is used to terminate a subroutine which was initiated by a GOSUB statement. The RETURN statement mupt be the last statement on its program line. Each time a RETURN is executed, the address on top of the gosub-stack is removed from the stack and execution of the program is continued at the line located at the indicated address; i.e., the return is to the program line following the last unterminated GOSUB statement.

Programs can execute up to ten GOSUB statements without an intervening RETURN statement. It is not necessary that equal numbers of GOSUB statements and RETURN statements be executed before termination of the program.

9.4. CALL

The CALL statement provides a simple method of linking to external machine code subroutines (normally written in assembly language). The general syntactic form of the CALL statement is

CALL address

where the address is a numeric expression which is rounded to an integer (in the range -32768 to +32767). The integer value is then used as the entry point address of the machine code subroutine. No parameters are passed to the subroutine (the register contents are undefined), and no results are returned. If a parameter or returned value is required, the USR function should be used instead.

The machine code subroutine will have free use of all of the machine registers (except that the stack interrupt enable and master interrupt enable should not be changed), and at least fifteen stack levels.

10. FUNCTIONS

Many mathematical operations are built into BASIC as intrinsic functions. In general, a function takes one value, known as the argument, performs the defined operation, and returns a single value as the result of the function. The function result may then be processed by any other operators that are present in the expression in which the function is used.

10.1. Intrinsic Functions

The processing of the function arguments is denoted as follows: an argument (X) denotes a function that uses a floating point value; an argument (I) denotes a function that rounds its argument to an integer in the range -32768 to 32767 before processing it; and an argument (S$) denotes a string argument.

10.1.1. Standard Numeric Functions

ABS(X) Returns the absolute value of X; i.e., X if X>=0, or -X if X<0.

ATN(X) Returns the arctangent of the argument X. The result will be in the range of -PI/2 to PI/2 radians.

COS(X) Returns the cosine of the argument X (in radians).

EXP(X) Exponential: returns the constant "e" (2.71828) raised to the power X.

INT(X) Returns the greatest integer which is less than or equal to the value of the argument X. For example, INT(2.7) is 2, and INT(-5.1) is -6.

LOG(X) Returns the natural (base e) logarithm of the argument X.

RND Returns a uniformly distributed pseudo-random number in the range 0 <= result < 1 . An argument may be provided [for example, RND(X)], but it will be ignored. The RANDOMIZE statement is used to alter the number sequence generated by successive RND function calls.

SGN(X) Returns the sign function of X; -1 if X<Ø, Ø if X=Ø, or +1 if X>Ø.

SIN(X) Returns the sine of the argument X (in radians).

SQR(X) Returns the square root of the argument X; X must be >= Ø.

TAN(X) Returns the tangent of the argument X (in radians).

1Ø.1.2. Advanced Numeric Functions

BRK(I) This function enables the user to enable or disable the capability to interrupt execution by striking a key on the keyboard. The function returns as its value the previous state of the break capability (Ø or 1). An argument less than zero leaves the capability unchanged; an argument of zero disables the break capability; and an argument greater than zero turns the break capability on.

INP(I) Read peripheral: a read operation is done with the device address specified by the argument I. Bit 15 of the address is always set for this operation.

PEEK(I) Read memory: a read operation is done with the memory address specified by the argument I. Since both peripherals and memory use the same instructions, this function may be used to access either memory or peripherals; the only difference between INP and PEEK is that PEEK does not automatically set address bit 15.

POS(I) Returns the current cursor column on the output terminal. The leftmost column is column 1.

USR(I) The user supplied (assembly language) function whose address has been placed into location ØØ1Ø is called. If location ØØ1Ø has not been preset by the user (with a POKE statement), a USR ERROR will occur. The argument is rounded to an integer and passed to the user function in the register ACØ. The function result should be returned as an integer in ACØ.

10.1.3. String to Numeric Conversion Functions

ASC(S$) Returns the ASCII value of the first character of the string. For example, ASC("AB") returns 65, which is the decimal value of the character "A".

LEN(S$) Returns the current length (in characters) of the argument string.

VAL(S$) Returns the decoded decimal value of the string. If the string does not contain a proper numeric representation, the result will be 0.

10.1.4. Numeric to String Conversion Functions

CHR$(I) Converts the decimal number I to an ASCII character with the corresponding binary value. Only the least significant eight bits of the argument I are used to form the character. For example, CHR$(65) returns the character string "A".

HEX$(I) Returns a string consisting of the hexadecimal representation of the argument I. The string will be four characters long, in the form "XXXX".

STR$(X) Returns a string consisting of the decimal representation of the argument X. The length of the string returned will depend upon the value of the number. The representation is the same as would be printed by a PRINT statement.

10.1.5. Substring Functions

The substring functions provide access to specified groups of characters within a complete string. In the following descriptions, I1 and I2 denote the numeric values specified as the function arguments, and I3 denotes the number of characters contained in the string represented by the argument S$.

MID$(S$,I1,I2) or MID$(S$,I1)
Returns a substring of the argument S$. The first numeric argument (I1) denotes the character number at which the substring is to begin. The second numeric argument (I2) specifies the maximum number of characters in the substring. If I2 is not

specified, the substring will consist of the remainder of the original string; i.e., it will contain I3-I1+1 characters. In no case will the substring contain more characters than the remainder of the string; the length of the substring is the minimum of I2 (if specified) and I3-I1+1. If I1 is greater than I3, the substring will be null (zero length).

LEFT$(S$,I1)
: Returns the leftmost I1 characters of the string S$. If the string S$ contains fewer than I1 characters, the entire string will be returned.

RIGHT$(S$,I1)
: Returns the rightmost I1 characters of the string S$. If the string contains fewer than I1 characters, the entire string will be returned.

NOTE: The substring functions are not recursive; the string argument may be any string except another substring function.

10.2. RANDOMIZE

The RANDOMIZE statement overrides the predefined sequence of pseudo-random numbers as values for the RND function, allowing different (and unpredictable) sequences each time a given program is executed.

In the absence of a RANDOMIZE statement, the RND function will generate the same sequence of pseudo-random numbers each time a program is run. This convention is chosen so that programs employing pseudo-random numbers can be executed several times with the same result - a desirable feature if one is trying to debug a program.

10.3. User-defined Functions

In addition to the intrinsic functions provided for the convenience of the programmer, BASIC allows the programmer to define new functions within a program.

The general form of statements for defining functions is

```
DEF FNx (parameter) = expression
```

where x is a single letter and a parameter is a simple numeric variable.

A function definition specifies the means of evaluating the function in terms of the value of an expression which may involve the parameter and other variables or constants. When the function is referenced, i.e., when an expression involving the function is evaluated, the expression in the argument list for the function is evaluated and its value is assigned to the parameter in the parameter-list for the function definition. The expression in the function definition is evaluated then, and this value is assigned as the value of the function.

The parameter appearing in the parameter-list of a function definition is local to that definition, i.e., it is distinct from any variable with the same name outside of the function definition. Variables which do not appear in the parameter-list are the variables of the same name outside the function definition.

The function definition must appear only once in a program, but need not appear before the location of the first reference to it. The expression in a DEF statement is not evaluated (except to check proper syntax) unless the defined function is referenced. If control is passed to a DEF statement, the statement on the program line immediately following the DEF statement will be the next one executed.

A function definition may refer to other defined functions, up to a nesting limit of ten. If a function references itself, it will always result in a nesting error.

11. HARDWARE ORIENTED STATEMENTS

11.1. POKE, OUT

The POKE and OUT statements are used to write integer data to memory or peripherals. The general syntactic forms of these statements are

```
POKE address, data
OUT  address, data
```

where address and data are both expressions which are evalutated and rounded to an integer value in the range -32768 to 32767. Since the PACE microprocessor uses memory reference instructions for input/output, the major difference between these statements is that the OUT statement forces address bit 15 to be set regardless of its specification in the address expression. This is to conform to the normal use of upper memory addresses for peripherals. For example,

```
     POKE $8Ø48,1
and  OUT $48,1
```

perform exactly the same operation. Since POKE is intended for memory alteration, and OUT for peripheral output, POKE will verify that the POKE operation was performed successfully (resulting in a POKE WARNING if not), while OUT sends its data to the peripheral without verification.

11.2. WAIT

The WAIT statement is used to read data (usually a status word) from an input port, and wait until a particular bit configuration is obtained. The general syntactic form is

```
WAIT address, and-mask, xor-mask
```

where the xor-mask is optional. Each of the expressions in the WAIT statement is evaluated and rounded to an integer. The address specified is then read, exclusive-OR'ed with the xor-mask (if supplied), AND'ed with the and-mask, and then checked for the resulting value. If the value is zero, the process is repeated and the program will wait until the series of operations result in a non-zero value. When a non-zero value is obtained, control is then allowed to proceed to the next statement.

11.3. TWAIT

The TWAIT statement provides a timed wait capability. The general form of the statement is

```
TWAIT expression
```

The expression is evaluated, rounded to an integer, and a delay of that number of milliseconds takes place. The maximum delay possible is 32767 milliseconds (32.8 seconds). The delay time does not include the time required to interpret the statement or to evaluate the expression.

12. ADVANCED CONTROL STATEMENTS

The SIZE, TRACE, and WIDTH statements are provided to aid in the debugging of programs and to allow different terminal widths.

12.1. SIZE

The SIZE statement prints out a summary table of the address ranges used for the program, for array and string storage, and for the symbol table. This information may be used to determine the amount of unused memory still available.

12.2. TRACE

The TRACE statement will cause each program line to be printed as it is executed. The printout occurs before interpretation of the statements on the line takes place. The listing of the program line is prefixed by a "->" character sequence. The program trace may be turned on and off at will to print only those statements in a particular line range. The statement forms are

```
TRACE
TRACE ON
TRACE OFF
```

Either of the first two forms turns on the program trace, and the third form disables the trace. The TRACE statement may be used either in command mode or as a statement in a program; trace mode is changed only by the TRACE statement and is not changed by the RUN command.

12.3. WIDTH

The WIDTH statement is used to accommodate BASIC to terminals with different line lengths. The statement form is

    WIDTH expression

where the expression is evaluated and rounded to an integer in the range of 15 to 132, representing the number of print columns to be used on the user's terminal. Any expression value less than 15 will cause a length of 15 to be used (one print zone), and any value greater than 132 will cause a width of 132 to be used.

The terminal width is initialized to 72 when BASIC is loaded; the WIDTH statement may be used in either command mode or program mode to change it at any time. The terminal width affects both the length of a line that can be accepted for editing, and controls the number of print zones which will be printed on a line.

13. ARRAYS

Arrays and string arrays may have either one or two dimensions. The current implementation of NATIONAL BASIC allocates strings and arrays when the RUN command is issued; no array or string may be used in immediate mode which has not been previously allocated by running a program. There are two methods of specifying the storage needed by an array: explicit declaration with the DIM statement, or implicitly by the appearance of the array in the program.

13.1. Declaration: DIM

The DIM statement is used to reserve space for arrays. Unless declared in a DIM statement, all array subscripts have a lower bound of zero and an upper bound of ten. Thus the default space allocation reserves space for 11 elements in one-dimensional arrays and 121 elements in two-dimensional arrays. By use of a DIM statement, the subscript(s) of an array may be declared to have an upper bound other than ten.

The general form of the dimension statement is

```
DIM declaration, ..., declaration
```

where each declaration has the form

```
     array-variable (constant)
or   array-variable (constant,constant)
```

and each array variable may be either a numeric array variable or a string array variable.

Each array declaration occurring in a dimension statement declares the array named to be either one or two dimensional according to whether one or two bounds are listed for the array. In addition, the bounds specify the maximum values that subscript expressions can have. Each array may be dimensioned only once in a program. Arrays that are not declared in any dimension statement are declared implicitly to be one dimensional, and to have an upper bound of ten.

14. DISK FACILITIES

Four commands are available to load and save programs on disk files. These commands are legal only when the "Disk Files Needed?" prompt at initialization was answered with "Y" or a carriage return. The commands are:

LOAD filename	Scratch the current program and load a new file from disk.
MERGE filename	Merge a disk file into the current program.
EXEC filename	Scratch the current program, load a new program from disk, and execute it. This statement may be used in a program.
SAVE filename	Write the current program to a source file on the disk.

The filename specification for these commands is a literal string and may be either quoted or unquoted (as in a DATA statement). The filename should be in the form of

drive:name.modifier

where only the name is required. If the drive number is not specified, drive 1 will be used; if a file modifier is not specified, it will be BAS. Refer to the PACE Disk Operating System Users Manual for additional information on files.

15. COMMANDS

Several system commands are provided to read and list programs from storage media, to edit programs in memory, and to control program execution. The system commands in this list always return to command mode; no additional statement may follow a command on the same line.

The command syntax allows for zero, one, or two line numbers to follow a command. Any command that doesn't need line numbers will just ignore any that are specified.

15.1. Editing Commands

The editing commands direct BASIC to read a program from a device other than the user's terminal, or to control entry or deletion of program lines.

15.1.1. SCRATCH

The SCRATCH command tells BASIC to delete the current program, and prepare to read a new program.

15.1.2. TAPE

The TAPE command tells BASIC to read a program from paper tape. The data is read in the same manner as it would be read from the terminal, except that the program lines are not echoed to the terminal. Syntax error messages, if any, are printed on the terminal, but the lines in error are saved so that they may be listed by the user.

Paper tape input mode is terminated by either of the following:

(1) the END statement, or
(2) typing a control/C.

15.1.3. AUTO

Command format: AUTO [start [,increment]]

The AUTO command causes program lines to be solicited from the terminal with line numbers supplied by the system. The starting line number and the increment may be supplied on the command; the defaults are 10 for each. If an increment is specified then the starting number must also be specified.

For example:

```
? AUTO 100
100 ? REM THIS IS THE LINE ENTERED
110 ? REM THIS LINE IS ALSO ENTERED
120 ? ^C
?
```

AUTO mode is terminated by typing a blank line to the prompt, or by aborting the line input with a control/C. If the line is aborted (with a control character other than control/C) or found to be in error, the user will be prompted for the same line again. The line number may be overridden by supplying a new line number following the prompt; the next line prompted will be the line number entered plus the increment; for example,

```
? AUTO 100
100 ? REM THIS IS EXAMPLE TWO
110 ? 200 REM LINE 200
210 ? REM LINE 210
220 ?
?
```

will result in lines 100, 200, and 210 being entered.

15.1.4. DELETE

The DELETE command is used to delete a range of lines from the program. If one line number is specified on the command, only that line will be deleted. If two line numbers are specified on the command, all lines in the program with line numbers in the range of line1 through line2, inclusive, will be deleted. If the value of the second line number is less than the value of the first line number, only the line (if any) with the first line number will be deleted.

The command formats are:

```
DELETE  line#        Delete single line.
DELETE  line1,line2  Delete line range (inclusive).
```

15.2. CLEAR

The CLEAR command is used to clear the symbol table created by immediate mode operations; this has the effect of reinitializing all variables to zero and removing the allocation of all arrays.

15.3. Execution Control Commands

15.3.1. RUN

General form: RUN [startline]

The RUN command causes the symbol table to be cleared (so that all variables will start off with zero or null values), and the program to be executed. Normally, the first statement in the program is the first statement to be executed, but that may be overridden by the specification of a starting line number on the RUN command; the effect will be as if a GOTO statement appeared before the first program line. If the specified starting line number does not exist, execution will begin with the next nigher numbered line (if any).

15.3.2. CONTINUE

General form: CONTINUE [startline]

The CONTINUE command is used to resume program execution following a STOP, END, break, or error. The CONTINUE command is valid only if the current program has been run and has not been edited since run termination. Certain errors are considered fatal and will invalidate any use of CONTINUE, but this situation should not occur often.

Program execution is normally resumed at the program line following the last line executed or the designated line if the last statement was a control statement (GOTO, etc.). The normal next line may be changed by specifying a line number on the

CONTINUE command, in which case execution will resume at the designated line (or the line with the next higher line number, if the designated line does not exist).

15.3.3. BYE

The BYE command returns control to the system monitor. This is the only means by which control is relinquished by BASIC (except for the front panel).

15.4. Listing Commands

Listing commands may have one of three forms:

command		List all lines in edit buffer
command	line#	List all lines starting at line#
command	line1,line2	List line range (inclusive)

The commands are:

LIST	List on console
HLIST	List on high speed printer
PUNCH	Punch paper tape

Note: the terminal width should be set to 132 to avoid extra characters which may be printed on long lines.

A break during a list operation will stop the list after the current line has finished printing, and return to command mode. The break capability is always enabled for lists.

An example of the list format is:

```
   1 REM SAMPLE LIST
  10 PRINT "LINE 10"
 100 REM .. LINE 100
9999 END
```

Line numbers are listed with leading spaces and followed by at least one space, so that the program lines will always be lined up and easier to read. This can sometimes cause a line to print more characters than were entered (due to the extra spaces). Since the WIDTH command may be used to change the number of characters that are printed on a line, it is possible that the line as printed will require more characters than may be printed on a single line. In this case, a carriage return and line feed will be printed at the column indicated by the WIDTH setting. This should not adversely affect legibility of program listings, but it does mean that any listing to paper tape may not read properly when reloaded. For this reason, it is recommended that a WIDTH 132 be issued before the PUNCH command is used.

16. IMMEDIATE MODE OPERATION

NATIONAL BASIC may be used in immediate mode, to function like a desk calculator. It is not necessary to write a complete program and run it to obtain information. Most of the statements may be used either in a program or may be given on-line as commands, which are immediately executed by the BASIC interpreter.

Lines entered for later execution and lines entered for immediate execution are differentiated by the presence of a non-zero line number preceding the program line. Statements which begin with a line number are stored; statements without a line number (or a zero line number) are executed immediately upon being entered to the system.

Immediate mode operation is especially useful for program debugging. Once a program has run, the values of the variables may be printed or changed by the user, and the CONTINUE command may then be used to resume program execution. The user may either place STOP statements at strategic places in the program, or merely use the break facility to interrupt execution when desired.

Program lines used in immediate (command) mode may include more than one statement, separated by a colon (":"), as usual. Interpretation of the line will stop if an error occurs (with an error message printed), or a system command is executed (since commands always return to command mode).

The statement summary in the appendix indicates which statements may be used in immediate mode. Any statement which has an optional line number may be used in immediate mode; use of any other statement will result in a "STMT ERROR" (improper statement use).

17. PROGRAM EXECUTION

Execution is in a two-pass mode. During the first pass, all statements are checked for proper syntax and FOR-NEXT nesting, arrays and strings are allocated storage space, and the symbol table is created and initialized. If any errors are encountered, they are listed and analysis continues; execution will not be stopped until all statements have been checked.

After the program has been completely checked, the second pass is started which is the actual program execution. If a starting line number was specified on the RUNP command, the program execution will begin there (if the line exists); otherwise execution will start with the first line in the program.

17.1. Interrupting Program Execution

Any key struck on the keyboard while a run is in progress will "break" or interrupt program execution. The break key is recommended since it is most reliably detected by the hardware. The break will occur after a line has completely finished execution. The line number appearing on the break message is the line number of the next line to be executed. The break capability may be turned off by the BRK(Ø) function and restored by BRK(1).

17.2. Restart

If it is necessary to stop execution immediately, and the break key is not satisfactory (or has been disabled), BASIC may be restarted by depressing the INITIALIZE then the RUN switches on the front panel of the computer. This sequence will restart BASIC in command mode, which then types "READY" and waits for a new command. This procedure is the only way to get out of a WAIT statement that waits indefinitely for some non-existant event; it is also necessary if the break capability has been turned off. Restart of BASIC does not affect the program or the current values of the variables; a CONTINUE command should work properly, although it will probably return to the same program line which was being executed when the switches were depressed.

18. HINTS

18.1. Peripheral Use

BASIC normally does all of its input and output to the terminal serving as the computer console; however, it is possible to use the built-in routines to print on the high speed printer (Centronics 306 or equivalent), or to read paper tapes (without echo or line feeds). The POKE statement is used to change the peripheral device flag that BASIC uses to determine which device is active. The statements are:

```
POKE $1B,2  to set output device to printer;
POKE $1B,0  to set output device to terminal;
POKE $1A,1  to set input device to paper tape;
POKE $1A,0  to set input device to terminal.
```

These commands may be used anywhere in a program. Note that setting the output device to the printer does not change the printout of prompt strings on the INPUT statement; the prompt is always printed on the console. Setting the device flag does, however, affect all PRINT statements.

For best results during command mode, the POKE statement and PRINT statement should be used together on a program line; for example,

```
POKE $1B,2:PRINT CHR$($C);
```

will send a form feed character to the high speed printer.

18.2. POKE Protection

The POKE statement is normally inhibited from altering the BASIC interpreter or any system firmware locations (0000, 0002, 000A-000F, and the top locations in RAM). A protect flag is located at 0011 which may be used disable the protection test and enable the user to poke anywhere. The integrity of the system cannot be guaranteed if protection is turned off and POKE is used indiscriminately; be sure you know what you are doing. The statements which affect POKE protection are:

```
POKE $11,0  to turn off protection;
POKE $11,1  to turn on protection.
```

18.3. Editing BASIC Files

BASIC programs stored as a disk file may be created or modified by the text editor. The default file modifier for these files is .BAS to denote a BASIC file; this modifier will be used unless otherwise specified on the file name. Please note that the BASIC line numbers must be part of the text; the sequential line numbers used by the editor are not part of the text file and cannot be used by BASIC.

The file created by the editor may contain commands (unnumbered lines); these lines will be executed immediately as the file is loaded by BASIC. The WIDTH and PRINT statements may be useful in certain situations. Unnumbered statements will not be written to the file by a SAVE command; they can be created only by the editor.

18.4. Assembly Language Subroutines

NATIONAL BASIC provides two methods for linking to assembly language subroutines: the CALL statement and the USR function. The CALL statement is used when no parameters are to be passed; the USR function is used when a parameter is to be passed between BASIC and the subroutine (in either or both directions).

The USR function gets the address of the subroutine from memory location 0010; the POKE statement is used to set the address. For example,

```
POKE $10,$E000
A=USR(12)
```

This example calls a subroutine at location 0E000 with the integer value 12 in register AC0; the integer value in AC0 upon return from the subroutine is assigned to the variable A.

If subroutines are to included in main memory together with the BASIC interpreter and program, the "Memory Size?" prompt at initialization time must be answered with an address lower than the starting address of the subroutine area, in order to keep BASIC from using the memory area occupied by the subroutine.

19. APPENDIX

19.1. Command Summary

AUTO [start [,increment]]	Start line number prompt mode.
BYE	Return to monitor.
CLEAR	Clear symbol table and reset flags and pointers.
CONTINUE [startline]	Continue after break, stop or error.
DELETE line#	Delete single line.
DELETE line1,line2	Delete line range (inclusive).
HLIST [start [,end]]	List on high speed printer.
LIST [start [,end]]	List on console.
LOAD filename	Scratch the current program and load a new program from the disk.
MERGE filename	Merge a source file from the disk into the current program.
SAVE filename	Save the current program as a source file on the disk.
SCRATCH	Scratch program: clear the buffer and prepare to read a new program.
PUNCH [start [,end]]	Punch a paper tape of the current program.
RUN [startline]	Run the program.
TAPE	Read paper tape and merge into the current program.

19.2. Statement Summary

n = Line Number

Items enclosed in brackets [...] are optional.

```
[n]  CALL address
 n   DATA constant list
 n   DEF FNid(arg) = expression
 n   DIM array(const), array(const,const),...
[n]  END
[n]  EXEC filename
 n   FOR id = expression TO expression [STEP expression]
 n   GOTO line#
 n   GOSUB line#
 n   IF expression GOTO line#
 n   IF expression THEN line#
[n]  IF expression THEN statement
          NOTE: Any statement except DATA, DEF,
          DIM, FOR, NEXT, or REM may be used.
 n   INPUT ["prompt" delim] variable list
[n]  [LET] variable = expression
[n]  [LET] strvar = string expression
 n   NEXT id
 n   ON expression GOTO line# list
 n   ON expression GOSUB line# list
[n]  OUT address,data
[n]  POKE address,data
[n]  PRINT list
[n]  RANDOMIZE
 n   READ variable list
[n]  REM comment
[n]  RESTORE
 n   RETURN
[n]  SIZE
[n]  STOP
[n]  TRACE [ON]
[n]  TRACE OFF
[n]  TWAIT milliseconds
[n]  WAIT address, andmask [,xor-mask]
[n]  WIDTH expr
```

19.3. Function Summary

ABS(X)	Absolute value of X
ASC(S$)	ASCII value of first character
ATN(X)	Arctangent of X
BRK(I)	Break capability (returns status and resets)
CHR$(I)	Single character having ASCII value of I
COS(X)	Cosine of X
EXP(X)	Exponential (E^X)
HEX$(I)	Converts number to hexadecimal format string
INP(I)	Read peripheral
INT(X)	Greatest integer <= X
LEFT$(S$,I1)	Left justified substring
LEN(S$)	Length of string
LOG(X)	Natural logarithm
MID$(S$,I1,I2)	Substring function (specified length)
MID$(S$,I1)	Substring function (remainder of string)
PEEK(I)	Read memory
POS(I)	Console cursor position
RIGHT$(S$,I1)	Right justified substring
RND	Random number (0 to 1)
SGN(X)	Sign of X (-1, 0, +1)
SIN(X)	Sine of X
SQR(X)	Square root of X
STR$(X)	Converts number to decimal format string
TAN(X)	Tangent of X
USR(I)	Calls user-supplied function
VAL(S$)	Value of string

19.4. Error Messages

Most error messages are of the form

xxxx ERROR [IN nnnn]

where xxxx is the error name (see below), and nnnn is the line number in which it occurred (omitted if in a direct command).

19.4.1. Program Errors

ERROR	MEANING
AREA	Out of memory
ARG	Argument out of range (to math functions)
CHAR	Character after logical end of statement
CONT	No continue possible
DATA	Out of data (READ statement)
DDEF	Duplicate function (FNx) definition
DDIM	Duplicate array dimensioning (DIM statement)
DISK	Disk or file error
END"	No ending quote on string
FOR	FOR without NEXT
NEST	Nesting limit exceeded (expressions, FOR, GOSUB, etc.)
NEXT	NEXT without FOR
NOGO	Line number specified by a control statement (GOTO, etc.) does not exist
RANG	Subscript or parameter out of range
RTRN	RETURN without previous GOSUB
SNTX	Syntax
STMT	Statement type used improperly
TYPE	Type mismatch (numeric or string)
UDEF	Undefined function (FNx)
USR	Undefined USR function address
VALU	Constant format or value

19.4.2. Warnings

DIVØ	Division by zero
IFIX	Integer overflow
OVFL	Floating point overflow
POKE	POKE does not verify, or POKE into BASIC
UNFL	Floating point underflow

Warnings indicate that some action is worthy of a message to the user, but is not sufficiently fatal to cause termination of the program. The action taken when each of these messages occurs is as follows:

DIVØ	Division by zero: plus or minus infinity is returned.
IFIX	Integer overflow: +32767 or -32768 is returned. This message occurs if the numeric value is less than -32767 or greater than +32767 when an integer conversion is attempted.
OVFL	Overflow: plus or minus infinity is returned; the sign depends upon the sign of the original number.
POKE	POKE into reserved locations is ignored. These locations are Ø, 2, ØA-ØF, the BASIC interpreter itself, and the read/write memory used by the firmware. This message also occurs when a POKE into non-existent or ROM memory is attempted.
UNFL	Underflow: Ø is returned.

19.4.3. Internal Errors

SYSTEM ERROR AT nnnn
OPERAND ERROR AT nnnn

These are internal errors in BASIC. The processor will halt if either of these messages appears; the number is the hexadecimal address where the error occurred. Pressing the RUN switch on the computer will return the interpreter to command mode (the halt is to facilitate debugging).

If either of these messages is encountered, there are two possible causes. The more likely case is that part of BASIC has been lost due to a memory failure or other hardware problem, or due to an error in an assembly language subroutine. If the error persists, BASIC should be reloaded. If the error still persists, the diagnostic programs should be run to determine if there has been a system malfunction.

A less likely possibility is an error in the BASIC interpreter itself. If the error persists and the system passes diagnostics, the error should be reported to National Semiconductor, along with as much information as possible about what the program was doing at that point.