Zilog

BASIC Interpreter

Preliminary User's Manual

Zilog

ZILOG

BASIC INTERPRETER

PRELIMINARY
USER'S MANUAL

Sections I - IX from 77348
X - App J " 78116

TABLE OF CONTENTS

SECTION I: INTRODUCTION TO BASIC 1

1.1 SPECIAL KEYS 2
1.2 PROMPT CHARACTERS 3
1.3 STARTING AND STOPPING A BASIC SESSION 4
1.4 CORRECTING TYPOGRAPHICAL ERRORS 5
1.5 BASIC COMMANDS AND STATEMENTS 6

1.5.1 COMMANDS 6
1.5.2 STATEMENTS 6
1.5.3 ERROR MESSAGES 7
1.5.4 CHANGING OR DELETING A STATEMENT . . . 7

1.6 BASIC PROGRAMS 8
1.7 USER'S WORK AREA 9
1.8 LISTING A PROGRAM 10
1.9 RUNNING A PROGRAM 11
1.10 DELETING A PROGRAM 12
1.11 DOCUMENTING A PROGRAM 13

SECTION II: EXPRESSIONS 14

2.1 CONSTANTS 15

2.1.1 NUMERIC CONSTANTS 15
2.1.2 LITERAL STRINGS 16

2.2 VARIABLES 17
2.3 FUNCTIONS 19
2.4 OPERATORS 20
2.5 EVALUATING EXPRESSIONS 22

SECTION III: STATEMENTS 24

3.1 ASSIGNMENT STATEMENT: LET 25
3.2 END/STOP STATEMENTS 27
3.3 LOOPING STATEMENTS: FOR...NEXT 29
3.4 GOTO/ON...GOTO STATEMENTS 33
3.5 GOSUB...RETURN STATEMENTS 35
3.6 CONDITIONAL STATEMENTS: IF...THEN 38
3.7 INPUT STATEMENT 43

3.8 PRINT STATEMENT 45

3.8.1 TAB FUNCTION 46

3.9 READ/DATA/RESTORE STATEMENTS 48
3.10 COMMENTS: REM STATEMENT AND "\" 51
3.11 RANDOMIZE STATEMENT 52
3.12 SYSTEM STATEMENT 53
3.13 TRAP STATEMENT 56

3.13.1 ERR FUNCTION 56
3.13.2 ESC FUNCTION 56

SECTION IV: COMMANDS 57

4.1 PROGRAM EXECUTION COMMANDS 59

4.1.1 RUN 59
4.1.2 XEQ 59
4.1.3 RESUMING PROGRAM EXECUTION:
CONTINUE, STEP, RUN 60
4.1.4 QUIT 61

4.2 EDITING COMMANDS 62

4.2.1 LIST 62
4.2.2 NEW 63
4.2.3 DELETTE 63
4.2.4 RENUMBER 64
4.2.5 SIZE 64
4.2.6 CLEAR 65

4.3 DISK RELATED COMMANDS 67

4.3.1 SAVE 67
4.3.2 ASAVE 68
4.3.3 RSAVE 68
4.3.4 GET 69
4.3.5 XEQ 69
4.3.6 APPEND 69
4.3.7 OBTAINING A LIST OF BASIC PROGRAMS . . 70
4.3.8 DELETING FILES 70

SECTION V: KEYBOARD EXECUTABLE STATEMENTS 72

SECTION VI: NUMERIC TYPES 74

6.1 TYPE SPECIFICATION 75
6.2 NUMERIC CONSTANT FORMS 76

6.2.1 INTEGER FORM 76
6.2.2 FLOATING POINT FORM 76

6.3 NUMERIC EXPRESSIONS 79
6.4 CONDITIONAL NUMERIC EXPRESSIONS 80
6.5 NUMERIC ASSIGNMENT 81
6.6 INPUTTING NUMERIC DATA 82
6.7 OUTPUTTING NUMERIC DATA 83
6.8 NUMERIC FUNCTIONS 84

6.8.1 ABS 84
6.8.2 ATN 84
6.8.3 COS 84
6.8.4 EXP 84
6.8.5 INT 84
6.8.6 LOG 85
6.8.7 RND 85
6.8.8 SGN 85
6.8.9 SIN 85
6.8.10 SQR 85
6.8.11 TAN 85

SECTION VII: ARRAYS 86

7.1 DIM STATEMENTS 88
7.2 STORING DATA IN ARRAYS 90
7.3 PRINTING DATA FROM ARRAYS 91

SECTION VIII: STRINGS 92

8.1 LITERAL STRINGS 93
8.2 STRING VARIABLES 95
8.3 DIM STATEMENT WITH STRINGS 98
8.4 STRING EXPRESSIONS 100

8.5 STRING-RELATED FUNCTIONS 102

8.5.1 CHR$ FUNCTION 102
8.5.2 ASC FUNCTION 102
8.5.3 LEN FUNCTION 102
8.5.4 POS FUNCTION 103
8.5.5 VAL FUNCTION 103
8.5.6 STR$ FUNCTION 103
8.5.7 LEFT$ FUNCTION 104
8.5.8 RIGHT$ FUNCTION 104
8.5.9 SEG$ FUNCTION 104

8.6 COMPARING STRINGS 105
8.7 STRING ASSIGNMENT 107
8.8 STRING INPUT STATEMENT 110
8.9 STRING LINPUT STATEMENT 111
8.10 STRING PRINT STATEMENT 112
8.11 STRING READ/DATA/RESTORE STATEMENT 113

SECTION IX: USER-DEFINED FUNCTIONS 114

9.1 ONE-LINE FUNCTIONS 115
9.2 MULTI-LINE FUNCTIONS 117
9.3 CALLING A USER-DEFINED FUNCTION 119

SECTION X: FILES 122

10.1 FILE TYPES AND ATTRIBUTES 123

10.1.1 ASCII FILES 123
10.1.2 BINARY FILES 123
10.1.3 FILE NAMES 123
10.1.4 FILE ATTRIBUTES AND STRUCTURE 123

10.2 OPENING FILES: FILE STATEMENT 125
10.3 CLOSING FILES: CLOSE STATEMENT 129
10.4 DELETING FILES: ERASE STATEMENT 130
10.5 TRUNCATE STATEMENT 132
10.6 SPACE STATEMENT 133
10.7 FILE ACCESS 135

10.7.1 SEQUENTIAL FILE READ, INPUT AND LINPUT 135
10.7.2 SEQUENTIAL FILE PRINT AND WRITE 137
10.7.3 FILE RESTORE STATEMENT 138
10.7.4 RANDOM FILE READ, INPUT, AND LINPUT . . 139
10.7.5 RANDOM FILE PRINT AND WRITE 140

10.8 FILE RELATED FUNCTIONS 142

10.8.1 EOF FUNCTION 142

SECTION XI: FORMATTED OUTPUT 143

11.1 PRINT USING STATEMENT 144
11.2 FORMAT STRINGS 145

SECTION XII: TRAPPING 148

12.1 TRAP STATEMENT 149

12.1.1 KEYS 150
12.1.2 EXT 151
12.1.3 ESC 151
12.1.4 ERR 152
12.1.5 EOF 152
12.1.6 ENVIRONMENTS AND TRAPS 152

12.2 TRAP RELATED FUNCTIONS 154

12.2.1 TRP 154
12.2.2 ESC 154
12.2.3 KEYS$ 154
12.2.4 ERR 154

SECTION XIII: SEGMENTATION 155

13.1 CHAIN STATEMENT 156
13.2 COM STATEMENT 158

SECTION XIV: COMMUNICATION WITH NON-BASIC PROGRAMS . . 159

14.1 CALL STATEMENT 160

APPENDIX A:	ASCII CHARACTER SET	161
APPENDIX B:	SUMMARY OF ZILOG BASIC STATEMENTS	164
APPENDIX C:	SUMMARY OF ZILOG BASIC COMMANDS	168
APPENDIX D:	BUILT-IN FUNCTIONS	170
APPENDIX E:	ERROR MESSAGES	173
APPENDIX F:	BASIC, PLZ AND ASSEMBLY LANGUAGE LINKAGE	180
APPENDIX G:	EXAMPLE: ASSEMBLY LANGUAGE CALL-SYSTEM STATEMENT PROCESSORS	191
APPENDIX H:	EXAMPLE: A USER PROCEDURE CALL	195
APPENDIX I:	A PROCEDURE FOR PRINTING TO A PRINTER	204
APPENDIX J:	EXTERNAL INTERRUPT LINKAGE	205

SECTION I

INTRODUCTION TO BASIC

Zilog BASIC is a programming language designed for use at a keyboard terminal. It consists of statements for writing programs and commands for controlling program operation.

There are two versions of Zilog BASIC. The difference between them is the math package. BINBASIC includes a binary math package with seven significant bits of precision. BASIC includes a BCD math package with 13 significant digits. Section 6.2.2 describes the two floating point representations. The examples shown in this user's manual will primarily reflect the BASIC BCD math package. The use of the word BASIC in this manual will refer to both BASIC and BINBASIC.

This section describes how to begin and end a BASIC session, how to enter commands and statements and make corrections. A few simple programs are used for illustration. The actual programming language is described in following sections.

This manual assumes that the user knows how to connect his terminal, and is familiar with his terminal keyboard. Special keys with particular functions in Zilog BASIC are described in this section.

1.1 SPECIAL KEYS

RETURN	Must be pressed after every command and statement. It terminates the line and causes the cursor to return to the first print position.
LINEFEED	Causes insertion of a space in the text and moves the cursor to the beginning of the next line.
CTRL	When pressed simultaneously with another key, converts the key to a control character that is usually non-printing.
CTRL-H	Deletes the previous character in a line. The cursor is moved back one position for each character deleted. (See RIO manual.)
RUBOUT (DEL)	Cancels the line currently being typed.
ESCAPE (ESC)	Cancels the current line if typed during command or statement input; stops the currently executing program if typed during run mode. (May not work on a ZDS system because there is no hardware Universal Asynchronous Receiver Transmitter (UART)).
?	Causes output to pause and continue if typed while terminal is outputting information. (May not work on a ZDS system because there is no hardware UART).

1.2 PROMPT CHARACTERS

Zilog BASIC uses a set of prompting characters to signal to the user that certain input is expected or that certain actions are completed.

> The prompt character for the Zilog BASIC Interpreter; a BASIC command or statement is expected.

% The prompt character for the RIO Operating System; RIO commands such as CAT or BASIC are expected.

? User input is expected during execution of an INPUT statement.

1.3 STARTING AND STOPPING A BASIC SESSION

STARTING A SESSION

Once the terminal is connected and ready, the user presses the carriage return. RIO responds with a percent sign (%) at the beginning of the line. The user may now begin.

ENTERING BASIC

The RIO Operating System signals it is ready for the next command by printing:	%
To enter BASIC, type:	%BASIC
BASIC signals that it has control by printing:	ZILOG BASIC version date BCD ARITHMETIC
followed by the prompt character:	>
To enter BINBASIC, type:	%BINBASIC
BINBASIC signals that it has control by printing:	ZILOG BASIC version date BINARY ARITHMETIC
followed by the prompt character:	>

BASIC commands and statements can now be entered. Each command or statement is prompted by the greater-than-sign at the start of a new line.

ENDING A BASIC SESSION

When the user is through, he or she returns control to RIO with the QUIT command.

To leave BASIC, type:	>QUIT
RIO signals that it has resumed control by printing:	%

1.4 CORRECTING TYPOGRAPHICAL ERRORS

Corrections can be made while the line is being entered if the error is noticed before RETURN is pressed. The control character CTRL-H can be used to correct a few characters just typed, or the character RUBOUT (DEL) can be used to cancel the line and start fresh.

Suppose the user misspells the command, RUN	`>RUM`
CTRL-H will delete the last character	`>RU` (Note: CTRL-H was pressed once)
The user retypes the character correctly and finishes the line. When RETURN is pressed, the line is entered correctly.	`>RUN`

If several characters have been typed after the error, CTRL-H must be typed for each character to be deleted.

```
>10 PXINT
>10 P            (Note: CTRL-H was pressed 4 times)
>10 PRINT
```

In this case four characters were deleted.

Another method is to use RUBOUT (DEL) to cancel the line. RUBOUT (DEL) must be typed before return is pressed.

To cancel the line, type RUBOUT (DEL). The user retypes the line	`>10 PRXNT`
	`>` (Note: RUBOUT (DEL) was pressed)
	`>10 PRINT`

1.5 BASIC COMMANDS AND STATEMENTS

1.5.1 Commands

Zilog BASIC commands instruct the Zilog BASIC Interpreter to perform certain control functions. Commands differ from the statements used to write a program in the Zilog BASIC language. A command instructs the interpreter to perform some action immediately, while a statement is an instruction that normally performs an action only when a program is run. Similar to commands, some statements can be executed immediately.

Any Zilog BASIC command can be entered following the BASIC prompt character ">". Each command is a single word. If misspelled, the computer will give an error message. Some commands have parameters to further define command operation.

For instance, QUIT is a command that signals completion of a BASIC programming session and return to the operating system. It has no parameters. Another command, LIST, prints the program currently being entered. It may have parameters to specify that only part of the program is to be listed.

1.5.2 Statements

Statements are used to write a Zilog BASIC program that will subsequently be executed. Each statement performs a particular function. Every statement entered becomes part of the current program and is kept until explicitly deleted or the user exits from BASIC with QUIT.

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

Following each statement, RETURN must be pressed to inform the interpreter that the statement is complete. The interpreter generates a LINEFEED and prints the prompt character ">" on the next line to signal that the statement is accepted. If the entered statement is in error, the computer prints an error message.

Zilog BASIC statements have a free format. This means that blanks are ignored.

For instance, all these statements are equivalent.

```
>20 LET B7=25
>20LETB7=25
>20 L E T B 7 = 2 5
>20   LET  B7  =  25
```

1.5.3 Error Messages

If an error is made in a line and the line is entered with RETURN, the interpreter types a message. The message consists of the word ERR followed by a number indicating the nature of the error. See Appendix E for a list of the error and warning numbers and their meaning.

1.5.4 Changing or Deleting a Statement

If an error is made before RETURN is pressed, the error can be corrected with CTRL-H or the line may be cancelled with RUBOUT (DEL), (See section 1.4). After RETURN is pressed, the error can be corrected by deleting or changing the statement.

To change a statement, simply type the statement number followed by the correct statement.

To change this statement:	>20 LET B7=25
retype it as:	>20 LET B7=37

A change such as this can be made any time before the program is run.

To delete a statement, type the statement number followed by return.

Statement 20 is deleted:	>20

The DELETE command, described in section 4.2.3 is useful to delete a group of statements.

1.6 BASIC PROGRAMS

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

A program can have as few as one statement.

This is an example of a program with only one statement.

```
>100 PRINT "5 * 10 = ";5*10
```

100 is the statement number. PRINT is the key word or instruction that tells the interpreter the kind of action to perform. In this case, it prints the string expression "5 + 10 =" and the result of the expression that follows. 5*10 is an arithmetic expression that is evaluated by the interpreter. When the program is run, the result is printed.

The statement 100 PRINT "5 * 10 = ";5*10 is a complete program since it can run with no other statements and produce a result. Usually a program contains more than one statement.

These four statements are a program:

```
>10 INPUT A,B
>20 LET C=A+B
>30 PRINT
>40 PRINT A;" +";B;" =";C
```

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

This program runs exactly like the program above.

```
>20 LET C=A+B
>10 INPUT A,B
>30 PRINT
>40 PRINT A;" +";B;" =";C
```

It is generally a good idea to number statements in increments of 10. This allows room to intersperse additional statements as needed.

1.7 USER'S WORK AREA

When statements are typed at the terminal, they become part of the user's work area. All statements in the user's work area constitute the current program.

Any statement in the user's work area can be edited or corrected; the resulting statement will then replace the previous version in the user's work area.

When the user exits from BASIC with the QUIT command, the work area is cleared.

1.8 LISTING A PROGRAM

At any time while a program is being entered, the LIST command can be used to produce a listing of the statements that have been accepted by the computer. LIST causes the computer to print a listing of the current program at the terminal.

After deleting or changing a line, LIST can be used to check that the deletion or correction has been made.

A correction is made while entering a program:

```
>10 INPUT A,B
>20 LET C=A+G
>20 LET C=A+B
>30 PRINT
>40 PRINT A;" +";B;" =";C
```

To check the correction, list the program:

```
>LIST
  10 INPUT A,B
  20 LET C=A+B
  30 PRINT
  40 PRINT A;" +";B;" =";C
>
```

Note that the greater-than sign prompt character is not printed in the listing, but is printed when the list is complete to signal that BASIC is ready for the next command or statement.

Should the statements have been entered out of order, the LIST command will cause them to be printed in ascending order by statement number.

For instance, the program is entered in this order:

```
>40 PRINT A;" +";B;" =";C
>20 LET C=A+B
>30 PRINT
>10 INPUT A,B
```

The list is in correct numeric statement order for execution:

```
>LIST
  10 INPUT A,B
  20 LET C=A+B
  30 PRINT
  40 PRINT A;" +";B;" =";C
>
```

1.9 RUNNING A PROGRAM

After the program is entered and, if desired, checked with LIST, it can be executed with the RUN command. RUN will be illustrated with two sample programs.

The first program has one line

```
>100 PRINT "5 * 10 =";5*10
```

When run, the result of the expression 5*10 is printed:

```
>RUN
5 * 10 = 50

READY
>
```

Because the program contains a PRINT statement, the result is printed when the program is run.

The second sample program adds two numbers. The numbers must be input by the user:

```
>10 INPUT A,B
>20 LET C=A+B
>30 PRINT
>40 PRINT A;" +";B;" =";C
```

The two letters following the word INPUT and separated by a comma name variables that will contain a value input by the user from the terminal. When the program is run, the interpreter signals that input is expected by printing a question mark. The user enters the values following the question mark. They are entered with a comma between each successive value.

The statement LET C=A+B assigns the value of the expression to the right of the equal sign to the variable C on the left of the equal sign. The expression adds the values of variables A and B together. The result is the value of C.

When the program is run, the user enters input values and the computer prints the result

```
>RUN
?1078,5.3
1078 + 5.3 = 1083.3

>
```

1.10 DELETING A PROGRAM

If a program that has been entered and run is no longer needed, it can be deleted with the NEW command. Typing NEW deletes whatever program has been entered by the user during the current session.

The first program entered was 100 PRINT "5 * 10 =";5*10. After it has been run, it should be deleted before entering the next program. Otherwise both programs will run when RUN is typed. They will run in the order of their statement numbers. For instance, if both programs are currently in the user's work area, the program with numbers 10 through 40 executes before line 100.

Both programs will run when RUN is typed

```
>100 PRINT "5 * 10 =";5*10
>10 INPUT A,B
>20 LET C=A+B
>30 PRINT
>40 PRINT A;" +";B;" =";C
>RUN
?1078,5.3
1078 + 5.3 = 1083.3
5 * 10 = 50
```

To avoid confusing results, a program that has been entered and run can be deleted with NEW:

After entering and and running: the program is deleted:

```
>100 PRINT "5 * 10 =";5*10
>RUN
 5 * 10 = 50
>NEW
```

The user's work area is now cleared and another program can be entered.

The second program is entered:

```
>10 INPUT A,B
>20 LET C=A+B
>30 PRINT A;" +";B;" =";C
>RUN
?343,275
 343 + 275 = 618
```

Unless this program is to be run again, it can now be deleted and a third program entered.

1.11 DOCUMENTING A PROGRAM

Remarks that explain or comment can be inserted in a program with the REM statement. Any remarks typed after REM will be printed in the program listing but will not affect program execution. As many REM statements can be entered as are needed.

The sample program to add 2 numbers can be documented with several remarks:

```
> 5 REM...THIS PROGRAM ADDS
> 7 REM 2 NUMBERS
>15 REM...2 VALUES MUST BE INPUT
>35 REM C CONTAINS THE SUM
```

The statement numbers determine the position of the remarks within the existing program. A list will show them in order:

List of sample program including remarks:

```
>LIST
   5 REM...THIS PROGRAM ADDS
   7 REM...2 NUMBERS
  10 INPUT A,B
  15 REM...2 VALUES MUST BE INPUT
  20 LET C=A+B
  30 PRINT
  35 REM...C CONTAINS THE SUM
  40 PRINT A;" +";B;" =";C
```

When run, the program will execute exactly as it did before the remarks were entered.

Comments may also appear on the same line as a statement. This is done by preceding the comment with the character "\" (backslash). Characters after the backslash are not processed as part of the statement but are stored along with the program statement.

Comments that follow a statement in this manner cannot be used on the same line as a DATA statement (Section 3.9).

Sample program with comments following statements:

```
>10 INPUT A,B \INPUT 2 NUMBERS
>20 LET C=A+B \FIND SUM OF A AND B
>30 PRINT \CARRIAGE RETURN AND LINE FEED
>40 PRINT A;" +";B;" =";C \PRINT SUM,
```

SECTION II

EXPRESSIONS

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

Some examples of expressions are:

(A*3)-(B+10)	A and B are variables that must have been previously assigned a value. 3 and 10 are constants. Parentheses group those portions of the expression evaluated first. If A=6 wand B=4, it is an integer expression with the value 4.
(X*(Y-2))+Z	X, Y, and Z must all have been assigned values. *, + and - are the multiply, add and subtract operators. The innermost parentheses enclose the part evaluated first. If X=7, Y=4, and Z=3, the value of the integer expression is 17.

2.1 CONSTANTS

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

2.1.1 NUMERIC CONSTANTS

A numeric constant is a positive or negative decimal number including zero. When using BINBASIC (binary math package), a numeric constant consists of seven significant digits. When using BASIC (BCD math package), a numeric constant consists of 13 significant digits. It may be written as an integer, a fixed point number, or a floating point number. See Section 6.2.2 for a description of floating point representation for each BASIC.

Integers are a series of digits with no decimal point.

BINBASIC Integers	BASIC (BCD) Integers
1234567	1234567890123
-7321465	-1234567890123
0	0
60	-56789

Floating point numbers are a number followed by the letter E and an optionally signed integer. In the floating point notation, the number preceding E is a magnitude that is multiplied by some power of 10. The integer after E is the exponent; that is, it is the power of 10 by which the magnitude is multipled.

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

1E+35 = 100000000000000000000000000000000000

1E-35 = .000000000000000000000000000000000001

Examples of Floating Point Numbers:

1E+23 = 1×10^{23} = 100000000000000000000000

1.0E23 (same as above)

.001E26 (same as above)

1.02E+7 = 1.02×10^{7} =10200000

1.02E-7 = .000000102

Within the computer, all these constants are represented as floating point real numbers whose size is between 1E-128 and 1E+127. The precision is determined by the type of BASIC math package.

BINBASIC floating point numbers	BASIC (BCD) floating point numbers
123.4567E+35	123456.789012E+20
1234567E-36	-1234567890123E+5
-.012E+20	123456.0789E-5

2.1.2 LITERAL STRINGS

A literal string consists of a sequence of characters in the ASCII character set enclosed within quotes. The quote itself and the character "<" are the only characters excluded from the character string. Blank spaces are significant within a string.

```
"ABC"

"!!WHAT A DAY!!"

" X Y  Z  "

""       (a null, empty, or zero length string)

"  "     (a string with two blanks)
```

2.2 VARIABLES

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

Real variables are a single letter (from A to Z) or a letter immediately followed by a digit (from 0 to 9).

```
A       A0

P       P5

X       X9
```

A variable of this type always contains a numeric value that is represented in the computer by a real floating point number.

Variables can also hold values internally represented as 16-bit integers. Names of such variables are similar to those above except that their names have a suffix of "%":

```
A%      B5%

X%      X3%
```

Variables may also contain a string of characters. This type of variable is identified by a variable name similar to those above except that their names have a suffix of "$":

```
A$      A0$

P$      P5$
```

The value of a string variable is always a string of characters, possibly null or zero length. String variables cannot be used without being declared with a DIM statement (see Section 8.3).

If a variable names an array (see Arrays, Section VII), it may be subscripted. When a variable is subscripted, the variable name is followed by one or two subscript values enclosed in parentheses. If there are two subscripts, they are separated by a comma. A subscript may be an integer constant, a variable, or any expression which is rounded to an integer value:

```
A(1)    A0(N%,M%)

P(1,1)  P5%(Q5,N/2)

X(N+1)  X9(10,10)
```

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

Simple numeric variables can be used without being declared. Subscripted numeric variables must be declared with a DIM statement (see Section 7.1) if the array dimensions are greater than 10 rows, or 10 rows and 10 columns. The first subscript is always the row number, the second the column number. The rounded subscript expressions must result in a value between 1 and the maximum number of rows and columns.

String arrays differ from numeric arrays in that they have only one dimension, and hence only one subscript. Also, the name of a string array and a simple string variable may not be the same (see String Arrays in Section VIII). Examples of subscripted string array names are:

```
A$(1)   A0$(N)   B5$(Z%)
```

2.3 FUNCTIONS

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

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

Examples of common functions:

INT(X) where X is a numeric expression. When called, it returns the largest integer less than or equal to X. For instance, INT(8.35)=8.

SGN(X) where X is a numeric expression. When called, it returns 1 for X>0, 0 for X=0 and 1 for X<0. For instance, SGN(4*-3)=-1.

Zilog BASIC provides many built-in functions that perform common operations such as finding the sine, taking the square root, or finding the absolute value of a number. The available numeric functions are listed in Appendix D and described in Section 6.8. In addition, the user may define and name functions if there is a need to repeat a particular operation. How to write functions is described in Section IX, User-Defined Functions.

The functions described so far are numeric functions that result in a numeric value. Functions resulting in string values are also available. These are identified by a multi-letter name followed by a "$". String functions are described with user-defined functions in Section IX. Available built-in string functions are listed in Appendix D and described in Section 8.6.

2.4 OPERATORS

An operator performs a mathematical or logical operation on one or two values resulting in a single value. Generally, an operator is placed between two values, but there are unary operators that precede a single value. For instance, the minus sign in A - B is a binary operator that results in subtraction of the values; the minus sign in -A is a unary operator indicating that A is to be negated.

The combination of one or two operands with an operator forms an expression. The operands that appear in an expression can be constants, variables, functions, or other expressions.

Operators may be divided into two types depending on the kind of operation performed. The main types are arithmetic, relational, and logical (or Boolean) operators.

The arithmetic operators are:

+	Add (or if unary, positive)	A + B or +A
-	Subtract (or if unary, negative)	A - B or -A
*	Multiply	A x B
/	Divide	A / B
^	Exponentiate	A ^ B

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

The relational operators are:

=	Equal	A=B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B
<>	Not equal	A<>B

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

Logical or Boolean operators are:

&	Logical "and"	A&B
!	Logical "or"	A!B
~	Logical complement	~A

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

Logical operators are evaluated as follows:

```
A&B =   1 (true) if A<>0 and B<>0
        0 (false) if A=0 or B=0

A!B =   1 (true) if A<>0 or B<>0
        0 (false) if A=0 and B=0

~A  =   1 (true) if A=0
        0 (false) if A<>0
```

A string operator is available for combining two string expressions into one:

+	Concatenation	A$+B$

The values of A$ and B$ are joined to form a single string; the characters in B$ immediately follow the last character in A$. If A$ contains "ABC" and B$ contains "DEF", then A$+B$="ABCDEF" (see Strings, Section VIII).

2.5 EVALUATING EXPRESSIONS

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

Highest

unary +, unary -, ~

^

*,/

binary +, binary -

Relational (=, <, >, <=, >=, <>)

& !

Lowest

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

For instance: 5+6*7 is evaluated as 5+(6*7)=47

7/14*2/5 is evaluated as ((7/14)*2)/5=.2

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

then: A+B*C is evaluated as A+(B*C)=7

A*B+C is evaluated as (A*B)+C=5

A+B-C is evaluated as (A+B)-C=0

(A+B)*C is evaluated as (A+B)*C=9

In a relation, the relational operator determines whether the relation is equal to 1 (true) or 0 (false). If A, B and C have the values given above:

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

In a logical expression, other operators are evaluated first for values of zero (false) or non-zero (true). The logical operators determine whether the entire expression is equal to 0 (false) or 1 (true). If A, B, C, D and E have the values given above:

E&A-C/3	is evaluated as 0 (false) since both terms in the expression are equal to zero (false).
A+B&A*B	is evaluated as 1 (true) since both terms in the expression are non-zero (true).
A=B!C=SIN(D)	is evaluated as 0 (false) since both expressions are false (0).
A!E	is evaluated as 1 (true) since one term of the expression (A) is not equal to zero.
~E	is evaluated as 1 (true) since E=0.

For rules governing the evaluation of expressions using strings, see Comparing Strings in Section 8.7.

SECTION III

STATEMENTS

Statements essential to writing a program in BASIC are described here. A general description of statements is given in Section 1.5.2. It should be recalled that all statements in a program must be preceded by a statement number and are terminated by pressing the RETURN key. These statements are not executed until the program is executed with a RUN command. Some statements may also be executed immediately and are useful for debugging (see Section V).

3.1 ASSIGNMENT STATEMENT

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

Format

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

```
variable=expresson

LET variable=expression
```

Several assignments can be made in one statement if they are separated by commas:

```
variable=expression,...,variable=expression

LET variable=expression,...,variable=expression
```

Note that the word LET is an optional part of the assignment statement.

Description

In this statement, the equal sign is an assignment operator. It does not indicate equality, but is a signal that the value on the right of the assignment operator be assigned to the variable on the left. When a variable to be assigned a value contains subscripts, these are evaluated first from left to right, then the expression is evaluated and the resulting value moved to the variable.

Examples:

```
10 LET Z1=34.567
20 Z1=34.567
```

The variable Z1 is assigned the value 34.567. Statements 10 and 20 have the same result.

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

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

```
 80 A=10.5,B=7.5
 90 B$="ABC",C$=B$
100 D%=5,E1%=10
```

The real variable A is set to 10.5, then B is set to 7.5. The string variable B$ is assigned the value ABC, then C$ is assigned the value of B$ (or ABC). The integer variable D% is assigned the value 5, then E1% is assigned the value 10. Strings and string assignments are described in Section VIII.

3.2 END/STOP STATEMENTS

The END and STOP statements are used to terminate execution of a program. Either may be used, neither is required. An END is assumed following the last line entered in the current program.

Format

```
END

STOP
```

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

Description

Both END and STOP terminate program execution. END has a different function from STOP in that END causes all files to be closed and the message "READY" to be printed. STOP causes the message "STOP AT nnnn" to be printed where nnnn is the statement label of the STOP statement. After a STOP, program execution can be resumed (see Section 4.1.3).

Examples

These three programs are effectively the same:

```
  10 DIM A$[5], B$[15], C$[15]
  20 LET A$="HELLO", B$"THERE"
  30 C$=A$+" "+B$
  40 PRINT C$
>RUN
HELLO THERE

READY
>
```

```
  10 DIM A$[5], B$[15], C$[15]
  20 LET A$="HELLO", B$="THERE"
  30 C$=A$+" "+B$
  40 PRINT C$
  50 END
>RUN
HELLO THERE

READY
>
```

```
  10 DIM A$[15], B$[15], C$[15]
  20 LET A$="HELLO", B$="THERE"
  30 C$=A$+" "+B$
  40 PRINT C$
  50 STOP
>RUN
HELLO THERE

STOP AT   50
>
```

When sequence is direct and the last statement in the current program is the last statement to be executed, END or STOP are optional. The message "READY" prints as with END, but open files will remain open. END and STOP have a use, however, when sequence is not direct and the last statement in the program is not the last statement to be executed:

```
  100 INPUT X
  110 PRINT
  120 GOSUB 140
  130 END
  140 IF X>0 THEN PRINT "X > 0"
  150 ELSE PRINT "X <60>= 0"
  160 RETURN
>RUN
?-356
X <= 0
```

The subroutine at line 140 follows the END statement.

3.3 LOOPING STATEMENTS: FOR...NEXT

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

Format

```
FOR variable=expression TO expression

FOR variable=expression TO expression STEP expression
```

The variable may be either a real or integer variable. It is initially set to the value resulting from the expression after the equal sign. When the value of the variable passes the value of the expression following TO, the looping stops. If STEP is specified, the variable is incremented by the value resulting from the STEP expression each time the group of statements is repeated. This value can be positive or negative, but should not be zero. If a STEP expression is not specified, the variable is incremented by 1.

The NEXT statement terminates the loop:

```
NEXT variable
```

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

Description

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

1. The value of the FOR variable is compared to the final value; if it is greater than the final value (or is less than the final value when the STEP value is negative), control skips to the statement following NEXT. Otherwise, processing continues with the statement immediately following the FOR statement.

2. All statements between the FOR statement and the NEXT statement are executed.

3. The FOR variable is then incremented by 1, or, if specified, by the STEP value.

4. Return to step 1.

Each time a FOR loop is begun, BASIC checks to see if there are already any active FOR loops with the same FOR variable. If so, all active loops within and including the duplicated entry are deactivated and processing proceeds as described above.

The user should not execute statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce unpredictable results.

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

Examples

Each time the FOR statement executes, a smaller fraction is printed.

```
>10 FOR A=1 TO 16
>20   PRINT 1/(10^A)
>30 NEXT A
>RUN
 .1
 .01
 .001
 .0001
 .00001
 .000001
 .0000001
 .00000001
 .000000001
 .0000000001
 .00000000001
 .000000000001
 .0000000000001
 1.000000000000E-014
 1.000000000000E-015
 1.000000000000E-016
```

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

```
   10 FOR X=0 TO -5 STEP -1
   20  PRINT X
   30 NEXT X
>RUN
 0
-1
-2
-3
-4
-5
```

The first X elements of the array P(N) are assigned values. When N=X, the loop terminates. In this case, the value of X is input as:

```
>10 INPUT X
>20 PRINT
>30 FOR N=1 TO X
>40   LET P(N)=N*10
>50   PRINT P(N)
>60 NEXT N
>RUN
?6
 10
 20
 30
 40
 50
 60
```

The examples below show legal and illegal nesting. A diagnostic is printed when an attempt is made to run the second example:

```
10 REM..THIS EXAMPLE IS LEGAL
20 DIM Y[7,16]
30 FOR A=1 TO 7 STEP 2
40  FOR B=1 TO 16 STEP 2
50    LET Y(A,B)=-1
60  NEXT B
70 NEXT A
```

```
  10 REM..THIS EXAMPLE IS ILLEGAL
  20 DIM Y[7,16]
  30 FOR A=1 TO 7 STEP 2
  40   FOR B=1 TO 16 STEP 2
  50    LET Y(A,B)=-1
  60  NEXT A
  70 NEXT B
>RUN

ERR:60 AT   70
```

3.4 GOTO/ON...GOTO STATEMENTS

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

Format

GOTO statement label

ON integer expression GOTO statement label, statement label...

GOTO may have a single statement label, while ON...GOTO may be multi-branched with more than one statement label.

If the multi-branch ON...GOTO is used, the value of the integer expression determines the label in the list to which control transfers.

Description

If the GOTO transfers to a statement that cannot be executed (such as REM), control passes to the next sequential statement after that statement. GOTO cannot transfer into or out of a function definition (see Section IX). If it should transfer to the DEF statement, control passes to the line following the function definition. (The function would be redefined in this case -- see DEF statement, Section IX.)

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

If the expression is not an integer, it is rounded to the nearest integer and that value is used to select a label.

Examples

The example below shows a simple GOTO in lines 45, 55, and 65 and a multi-branch GOTO in line 30.

```
10 LET I=0
20 LET I=I+1
30 ON I GO TO 40,50,60,70
40 PRINT "THE VALUE OF I IS 1"
45 GOTO 20
50 PRINT "THE VALUE OF I IS 2"
55 GOTO 20
60 PRINT "THE VALUE OF I IS 3"
65 GOTO 20
70 PRINT "THE VALUE OF I IS 4"
75 END

>RUN
THE VALUE OF I IS 1
THE VALUE OF I IS 2
THE VALUE OF I IS 3
THE VALUE OF I IS 4
```

When run, the program prints the value of I for each ON...GOTO.

3.5 GOSUB...RETURN STATEMENTS

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

Format

GOSUB statement label

ON integer expression GOSUB statement label, statement label,...

RETURN

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

Description

A single branch GOSUB transfers control to the statement indicated by the label. A multi-branch ON...GOSUB transfers to the statement label determined by the value of the integer expression. As in a multi-branch ON...GOTO, if the value of the expression is less than 1 or greater than the length of the list, no transfer takes place. A GOSUB must not transfer into or out of a function definition (see Section IX).

When the sequence of control within the subroutine reaches a RETURN statement, control returns to the statement following the GOSUB statement.

Within a subroutine, another subroutine can be called. This is known as nesting. When a RETURN is executed, control transfers back to the statement following the last GOSUB executed.

If the expression in an ON...GOSUB statement is not an integer then it is rounded to the nearest integer and that value is used to select a label.

Examples

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

```
  10 LET B=70
  20 GOSUB 50
  30 PRINT "SINE OF B IS ";A
  40 GOTO 80
  50 REM:  THIS IS THE START OF THE SUBROUTINE
  60 LET A=SIN(B)
  70 RETURN
  80 REM:  PROGRAM CONTINUES WITH NEXT STATEMENT
>RUN
SINE OF B IS  .7738906815526
```

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

```
  10 LET B=70
  20 GOTO 100
  30 REM:  THIS IS THE START OF THE SUBROUTINE
  40 LET A=SIN(B)
  50 RETURN
  60 REM:  OTHER STATEMENTS CAN APPEAR HERE
  70 REM:  THEY WILL NOT BE EXECUTED
  80 LET A=24, B=50
  90 PRINT "A= ";A,"B= ";B
 100 GOSUB 30
 110 PRINT "THE SINE OF B IS ";A
 120 REM:  A SHOULD EQUAL SIN(B)
 130 PRINT "B=";B
 140 REM:  B SHOULD EQUAL 70
>RUN
THE SINE OF B IS .77389068155526
B= 70
```

This example shows a multi-branch GOSUB in line 20. The third subroutine executed has a nested subroutine.

```
 10 FOR A=1 TO 3
 20 ON A GOSUB 50,80,110
 30 NEXT A
 40 END
 50 REM:  FIRST SUBROUTINE IN MULTIBRANCH GOSUB
 60 PRINT "FIRST SUBROUTINE CALL"
 70 RETURN
 80 REM:  SECOND SUBROUTINE IN MULTIBRANCH GOSUB
 90 PRINT "SECOND SUBROUTINE CALL"
100 RETURN
110 REM:  THIRD SUBROUTINE IN MULTIBRANCH GOSUB
120 REM:  IT CONTAINS A NESTED SUBROUTINE
130 PRINT "THIRD SUBROUTINE CALL"
140 GOSUB 170
145 PRINT "END THIRD SUBROUTINE CALL"
150 RETURN
160 REM:  STATEMENT 150 RETURNS CONTROL TO STATEMENT 30
170 REM:  FIRST STATEMENT IN NESTED SUBROUTINE CALL
180 PRINT "    NESTED SUBROUTINE CALL"
190 RETURN
200 REM:  STATEMENT 190 RETURNS CONTROL TO STATEMENT 150
>RUN
FIRST SUBROUTINE CALL
SECOND SUBROUTINE CALL
THIRD SUBROUTINE CALL
    NESTED SUBROUTINE CALL
END THIRD SUBROUTINE CALL
```

3.6 CONDITIONAL STATEMENTS: IF...THEN

Conditional statements are used to test for specific conditions and specify program action depending on the test result. The condition tested is a numeric expression that is considered true if the value is not zero, false if the value is zero. Conditional statements are always introduced by an IF statement; an ELSE statement may follow the IF statement. Both IF and ELSE statements may be followed by a series of statements enclosed by DO and DOEND.

Format

```
IF expression THEN label

IF expression THEN statement

IF expression THEN DO

statement
    .
    .
    .

DOEND
```

An IF...THEN statement can be followed by an ELSE statement to specify action in case the value of the expression is false. Like IF, ELSE can be followed by a statement, a statement label, or a series of statements enclosed by DO...DOEND.

```
ELSE label

ELSE statement

ELSE DO

statement
    .
    .
    .

DOEND
```

ELSE STATEMENTS never appear in a program unless preceded immediately by an IF...THEN or an IF...THEN DO...DOEND statement. DO...DOEND statements may follow only an IF...THEN or an ELSE statement.

The four diagrams below show all possible combinations of conditional statements. Items enclosed by [ ] are optional; one of the items enclosed by { } must be chosen. Statements immediately following THEN and ELSE are not labelled; all other statements must be labeled.

```
1)                                   ( label     )
    label IF expression THEN        {             }
                                     ( statement )

   [                        ( label     )  ]
   [ label ELSE            {             } ]
   [                        ( statement )  ]

2)  label IF expression THEN DO

    label statement
        .
        .
    label DOEND

   [                        ( label     )  ]
   [ label ELSE            {             } ]
   [                        ( statement )  ]

3)                                   ( label     )
    label IF expression THEN        {             }
                                     ( statement )

    label ELSE DO

    label statement
        .
        .
    label DOEND


4)  label IF expression THEN DO

    label statement
        .
        .
    label DOEND


    label ELSE DO

    label statement
        .
        .
    label DOEND
```

Description

If the expression following IF is true when evaluated, the program transfers control to the label following THEN or executes the statement following THEN. An expression is considered true if it is numeric and non-zero or string and non-null. If DO follows THEN, the program executes the series of labeled statements terminated by DOEND. The program then continues. If the expression is false, control transfers immediately to the next statement or to the statement following DOEND if THEN DO was specified.

When an ELSE statement follows the IF...THEN statement, it determines the specific action should the IF expression be false. When the expression is true, the ELSE statement or the group of ELSE statements enclosed by DO...DOEND is skipped, and the program continues with the next statement after ELSE or DOEND.

A FOR statement can be specified in a DO...DOEND group; if so, the corresponding NEXT must be within the same DO...DOEND group (see FOR...NEXT statement, Section 3.3).

IF statements are nested when an IF statement occurs within the DO...DOEND group of another IF statement. In such a case, each ELSE is matched with the closest preceding IF that is not itself part of another DO...DOEND group.

Examples

The various types of IF statements are illustrated with the following examples:

```
10 IF E=F THEN 30
20 LET E=F*5
30 PRINT E,F
```

If E equals F, the program skips to line 30, otherwise it sets E equal F*5 in line 20 and continues. In either case, line 30 is executed.

```
10 IF X<Y THEN PRINT X
20 ELSE PRINT Y
```

If X is less than Y, the value of X is printed, otherwise the value of Y is printed. The program then continues.

```
10 IF A<B THEN 100
20 ELSE 200
```

Program control transfers to 100 if A is less than B, to line 200 if A is not less than B.

```
10 IF K<L THEN LET K=K+1
20 ELSE DO
30   LET L=L/3
40   LET K=L*K
50 DOEND
60 PRINT K,L
```

If K is less than L, then K is increased by 1 and control skips to line 60. When K is greater than or equal to L, L is equal to L/3, K is set equal to L*K and control passes to line 60.

```
  5 INPUT A
 10 IF A<100 THEN DO
 20   LET A=A+1
 30   GOTO 110
 40 DOEND
 50 ELSE DO
 60   GOSUB 130
 70  LET A=0
 80 DOEND
 90 PRINT "A>=100"
100 END
110 PRINT "A=";A
120 END
130 REM...BEGINNING OF SUBROUTINE
140 PRINT "A=";A
150 RETURN
```

If A is less than 100, it is increased by 1 and control goes to line 110. If A is egual to or greater than 100, the subroutine at line 130 is executed. The subroutine returns control to line 70, still within the DO...DOEND.

If a value less than 100 is input for A, it is incremented by one, line 110 is executed and the program ends:

```
>RUN
?75
A= 76
```

If a value greater than 100 is input for A, the subroutine is executed, then line 100 is executed and the program terminates:

```
>RUN
?150
A= 150
A>=100
```

The examples below illustrate nested IF...THEN statements.

```
  10 INPUT P,Q,R
  15 PRINT
  20 IF (P+10)=(Q+5) THEN DO
  30   LET P=Q
  40   IF P>R THEN LET P=P-R
  50   ELSE LET P=P+R
  60 DOEND
  70 PRINT P,Q,R
>RUN
?20,25,40
20            25             40
```

```
  10 INPUT A,B,C
  15 PRINT
  20 IF A>B THEN DO
  30   IF B>C THEN DO
  40     IF C=10 THEN DO
  50       LET A=A+1
  60       GOTO 200
  70     DOEND
  80     ELSE GOTO 220
  90   DOEND
 100   ELSE DO
 110     IF C=10 THEN LET B=C+A
 120     ELSE LET C=B-A
 130      GOTO 180
 140   DOEND
 150 DOEND
 160 PRINT "A<60>B,A=";A
 170 GOTO 230
 180 PRINT "A>B,B<60>C,B=";B
 190 GOTO 230
 200 PRINT "A>B>C,C=10"
 210 GOTO 230
 220 PRINT "A>B>C,C<60>>10,C=";C
 230 END
>RUN
?10,15,20
A<B,A=10

>RUN
?15,5,10
A>B,B<C,B= 25

>RUN
?20,15,5
A>B>C,C<>10,C= 5
```

So that nested IF statements may be easier to follow, the LIST command indents them as shown in the above examples.

3.7 INPUT STATEMENT

The INPUT statement allows the user to input data to the program from the terminal. INPUT has options that allow the user to print prompting strings before input.

Format

```
INPUT item list

INPUT string constant, item list
```

The items in the item list must be variables, optionally preceded by a string constant. Items are separated by commas.

Description

When an INPUT statement is executed, a question mark (?) is printed at the terminal and the program waits for the user to type the input. The input is in the form of constants separated by commas. If an insufficient number of constants is typed, the program responds with a message requesting retyping of the input. The type of data item, numeric or string, must match the type of variable it is destined for.

> Numeric Constants. Numeric constants aways begin with the first non-blank character preceding the comma or the end of the line.
>
> String Constants. A string may be quoted or unquoted. If unquoted, any leading or trailing blanks are removed and the input item terminates on a comma or the end of line.

The INPUT statement can be requested to print a string constant instead of a question mark by placing the string constant before the input list. When the value for the variable is needed, the string is printed instead of the usual question mark.

Examples

```
  10  DIM C$[25]
  20  INPUT A,B,C$
  25 PRINT
  30  X=A*B^2
  40  PRINT C$;X
>RUN
?4,7,"X=A TIMES B SQUARED, X="
X=A TIMES B SQUARED, X= 196
```

```
  10  INPUT "INPUT VALUE OF RADIUS  ",R
  20  X=3.14*R^2
  30  PRINT "AREA OF X =",X
>RUN
INPUT VALUE OF RADIUS  25
AREA OF X =      1962.5
```

3.8 PRINT STATEMENT AND ":"

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

Format

```
PRINT
:

PRINT print list
: print list
```

The print list consists of items separated by commas or semicolons. The list may be followed by a comma or a semicolon. If the list is omitted, PRINT causes a skip to the next line. Items in the list may be numeric or string expressions or the special print function for tabbing. The character ":" may be used in place of the keyword PRINT.

Description

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

The output line is divided into five consecutive fields: each of 14 characters for a total of 70 characters.* When a comma separates items, each item is printed starting at the beginning of a field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

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

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

*NOTE: The default output line length may be changed by a SYSTEM "LINELEN" call (see Section 3.12)

Numeric values are left justified in a field whose width is determined by the magnitude of the number. The width includes a position at the left of the number for a possible sign. (Thus, a blank will print in the first position if the number is non-negative.)

Examples

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

```
  10  LET A=1,B=2,C=3,D=4,E=5
  20  PRINT A,C*D,E-B*B
  30  PRINT
  40  PRINT "A/(B-C)=";A/(B-C)
  50  PRINT "E+D=";E+D
>RUN
 1            12           1

A/(B-C)=-1    E+D= 9
```

3.8.1 TAB FUNCTION

TAB (n)

TAB moves the cursor to column n MOD (70). If the current cursor position is greater than n MOD (70), the cursor will move to column n MOD (70) on the next line.

Note: If the default line length has been changed using SYSTEM "LINELEN" (Section 3.12), the TAB will be relative to n MOD (line length).

Example

```
  10 PRINT "123456789";TAB(4);"ABCD"
>RUN
123456789
    ABCD
```

The cursor position is greater than four, therefore the cursor is moved to position four on the next line. The string "ABCD" begins in position five.

3.9 READ/DATA/RESTORE STATEMENTS

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

Format

```
READ item list
```

The items in the item list are variables. Items are separated by commas.

```
DATA constant,constant
```

The constants are either numeric or string. Constants in the DATA statement are assigned to variables in the READ statement according to their order: the first constant to the first variable, the second to the second and so forth. A comment (Section 3.10) may NOT be placed on a DATA statement line.

```
RESTORE

RESTORE label

ON integer-value RESTORE label,label,...,label
```

The label identifies a DATA statement.

Description

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

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

If a variable is numeric, the next item in the data list must be numeric; it a variable is a string, the next item in the data list can be of any form.

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

When the RESTORE statement specifies a label, the pointer is moved to the first following DATA statement. When no label is specified, the pointer is restored to the first constant of the first DATA statement in the program.

One of many labels can be selected by the ON...RESTORE statement. The expression is rounded to the nearest integer and a label selected as described previously under the ON...GOTO statement (Section 3.4). The pointer is moved to the first DATA statement following the specified statement. If the expression could not select a label from the list, the pointer is not moved.

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

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

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

```
  5  DIM A$[3],C$[3],D$[3],E$[3],B$[3]
 10  DATA 3,5,7
 20  READ A,B,C
 30  READ A$,B$
 40  DATA ABC,DEF
 50  RESTORE 30
 60  READ C$,D$
 70  RESTORE
 80  READ D,E,F,E$
 90  PRINT A,B,C
100  PRINT A$+B$,C$+D$
110  PRINT D,E,F,E$

>RUN
 3        5        7
ABCDEF   ABCDEF
 3        5        7        ABC
```

An ON...RESTORE is used in the following example, combined with an ON...GOTO.

```
 10 DIM A$(10),B$(10),C$(10),D$(10)
 20 DATA 1111
 30 DATA 2222
 40 DATA 3333
 50 DATA 4444
 60 LET I=0
 70 IF I=4 THEN END
 90 PRINT
100 ON I RESTORE 30,40,50
110 ON I GOTO 130,140,150
120 READ A$
125 PRINT A$;
130 READ B$
135 PRINT B$;
140 READ C$
145 PRINT C$;
150 READ D$
155 PRINT D$;
160 LET I=I+1
165 GOTO 70
>RUN
1111222233334444
222233334444
33334444
4444
```

3.10 COMMENTS: REM STATEMENT AND "\"

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

Format

```
REM any characters
```

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

Description

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

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

Comments may also be placed after any statement by putting backslash "\" between the statement and the comment.
A comment may NOT be put on a DATA statement (Section 3.9).

Examples

```
10 REM: THIS IS AN EXAMPLE
20 REM  OF REM STATEMENTS
30 REM -- ANY CHARACTERS MAY FOLLOW REM: "//**!!&&&,ETC.
40 REM...REM STATEMENTS ARE NOT EXECUTED
50 PRINT A+B \ HERE IS A COMMENT FOLLOWING A PRINT
```

3.11 RANDOMIZE STATEMENT

The RANDOMIZE statement is used to change the seed used by the function RND (Section 6.8.7) to generate a pseudo-random number.

Format

```
RANDOMIZE
```

Description

There are 128 possible seeds. The RANDOMIZE statement may be placed anywhere in a program. It will restart the RND function with a new seed each time it is executed.

3.12 SYSTEM STATEMENT

The SYSTEM statement is used to perform miscellaneous functions not appropriate for machine-independent BASIC. For example, these operations might include manipulating operating system parameters and terminal characteristics.

Format

```
SYSTEM operation-name[,parameter-list][;return-parameter-list]
```

Operation-name is a string expression whose value is one of the system operations listed below. The parameter lists are structured like those of the CALL statement (Section 13.1)

Description

The available operations control warning message output, listing indentation, ASAVE file line indentation, output line length, and automatic carriage return. The default value of each operation is:

warning message output is ON
list indentation is ON
ASAVE file line indentation is OFF
output line length is 70
automatic carriage return is ON

The allowed operations and the parameters that they require are listed below.

Operation:	"WARNOFF"
Parameters:	None
Function:	Suppress the output of any warning messages generated during a BASIC session.
Example:	SYSTEM "WARNOFF"

Operation	"WARNON"
Parameters:	None
Function:	Enable the printing of warning messages.
Example:	SYSTEM "WARNON"

Operation:	"INDENTOFF"
Parameters:	None
Function:	Turn off the identation normally performed during a LIST of a program.
Example:	SYSTEM "INDENTOFF"

Operation:	"INDENTON"
Parameters:	None
Function:	Turn on indentation feature of LIST.
Example:	SYSTEM "INDENTON"

Operation:	"ASINOFF"
Parameters:	None
Function:	Turn off ASAVE file line indentation
Example:	SYSTEM "ASINOFF"

Operation:	"ASINON"
Parameters:	None
Function:	Turn on ASAVE file line indentation
Example:	SYSTEM "ASINON"

Operation:	"LINELEN"
Parameters:	line-length (integer: 1 to 255)
Function:	Set the line length of the terminal device. Print items that do not fit between the current cursor position and the end of the line are output on the next line. The default line length is 70.
Example:	SYSTEM "LINELEN",132 The line length is set for 132-character wide paper.

Operation:	"AUTOCROFF"
Parameters:	None
Function:	Turn off the automatic begin-new-line feature of terminal output.
Example:	SYSTEM "AUTOCROFF"

Operation:	"AUTOCRON"
Parameters:	None
Function:	Turn the automatic begin-new-line feature on.
Example:	SYSTEM "AUTOCRON"

3.13 TRAP STATEMENT

(TO BE SUPPLIED LATER.)

3.13.1 ERR Function

ERR(x)

(TO BE SUPPLIED LATER.)

3.13.2 ESC Function

ESC(0)

(TO BE SUPPLIED LATER.)

SECTION IV

COMMANDS

So far we have used the LIST, RUN and NEW commands for simple program manipulation. Both LIST and RUN have parameters and functions other than were illustrated. The full capability of commands used to run a program, edit a program, and to save a program on the disk are:

RUN
STEP
CONTINUE

The Editing Commands:

LIST
NEW
DELETE
RENUMBER
SIZE
CLEAR

Disk-Related Commands:

SAVE
RSAVE
ASAVE
GET
XEQ
APPEND
KILL
CAT

A general description of commands is given in Section I. It should be recalled here that commands do not have labels: they are entered directly after the ">" prompt character and are executed immediately. All commands may be abbreviated by their first three letters.

Certain conventions are used in the command description:

UPPER-CASE	Key words that must be spelled correctly
lower-case	Words defined by the user
[ ]	Enclose optional items
{ }	Enclose required items
¦	Separates alternatives, one of which must be chosen
...	Indicate the preceding item may be repeated

4.1 PROGRAM EXECUTION COMMANDS

The program execution commands facilitate the debugging of a program. Execution of the program can be interrupted either manually or under program control. Variables can be examined and/or altered, parts of the program may be displayed or changed, and execution can be resumed.

4.1.1 RUN

The RUN command executes a Zilog BASIC program; the format is

```
RUN[-label]
```

If a label is not specified, execution begins with the first executable statement. If a STOP (Section 3.2) or ESCape is executed within the program, the program may be continued by inputting CONTINUE (Section 4.1.3). If a RUN-label is specified, execution starts at the first executable statement at or after the label number. The starting statement must not be within a function definition.

4.1.2 XEQ

The XEQ command loads and runs a Zilog BASIC program. It is equivalent to a GET (Section 4.3.4) followed by RUN.

```
XEQ-programname
```

The program programname.BP is loaded into the user's workingspace and run. (See Section 4.3 for a description of filename conventions.)

Example

```
>XEQ-BAGELS
```

The program BAGELS.BP is loaded into the user's working space and run.

4.1.3 RESUMING PROGRAM EXECUTION: CONTINUE, STEP, RUN

Once a program has been interrupted by a STOP or ESC (Section 3.2), the Zilog BASIC user can cause execution to continue in one of several ways.

Format

CONTINUE

STEP

RUN-statement number

Each of these commands causes program execution to continue without alteration of any program variables.

Description

The CONTINUE command causes execution to continue with the next statement to be executed (based on when the program stopped). This command can be issued anytime the program is in a stopped state.

The STEP command causes execution to proceed to the beginning of the next outer level statement (i.e., one not part of a multi-line function). This command can be used to step through the program one line at a time executing entire statements at the outer level and not stepping through multi-line functions. STEP can be issued anytime the program is in a stopped state.

"RUN-statement number" causes execution to resume starting with the specified statement. This command cannot cause execution to begin in the middle of a multi-line function or a new DO-DOEND block without undesirable side-effects.

When a program is stopped, the following commands can be executed without preventing resumption of the program:

Any keyboard executable statement (see Section V)
LIST
SAVE
ASAVE
RSAVE
SIZE

Program execution cannot be resumed after any of the following:

Alteration of the program
RENUMBER
NEW
QUIT
GET
XEQ
CLEAR

4.1.4 QUIT

The QUIT command is used to exit BASIC. All open files are closed and the user work area is cleared. Control returns to RIO.

```
QUIT
```

4.2 EDITING COMMANDS

The editing commands always affect the current program; that is, the program that is currently being entered at the terminal.

4.2.1 LIST

The LIST command lists all or part of the current program; the form is

```
LIST[-range]
```

where range specifies the range of statements to be listed. If no range is specified, the entire program is listed. The format and effect of a range is as follows:

LIS-n	only statement n is listed
LIS-n,	statements n to the end of the program are listed
LIS-,n	the first statement and statements up to n (inclusive) are listed
LIS-n,m	statements n through m (inclusive) are listed

Examples

```
>LIST
```

The entire current program is listed at the terminal.

```
>LIST-1,100
```

Statements 1 through 100 of the current program are listed.

Note that a listing can be stopped by pressing the ESCape key. The user is returned to BASIC control. The listing can also be stopped by pressing the "?" key. Pressing the "?" again will continue the listing.

4.2.2 NEW

The NEW command deletes the entire current program; the form is:

```
NEW

NEW-#buffers
```

If the number of buffers (0-15) to be allocated is not specified, two buffers are allocated. A buffer is required for each open file. Each buffer consists of 512 bytes. One buffer is required for file commands.

Example

```
>NEW-7
```

The current program is deleted, seven buffers are allocated, and a new current program can be entered in the user's work area. The SIZ command (Section 4.2.5) shows the number of available/allocated buffers.

```
>SIZ:  AVAIL=9213 PROG=0 VAR=0 BUF=7/7
```

4.2.3 DELETE

The DELETE command deletes one or more specified statements; the form is

```
DELETE-range
```

where range is described below; the statements specified by the parameters are deleted from the program. The range specifies a range of statements which are to be deleted.

DEL-n	deletes line n
DEL-,n	deletes all lines from the beginning of the program up to line n (inclusive)
DEL-n,	deletes lines n to the end of the program
DEL-n,m	deletes lines n through m (inclusive)

Example

```
>DEL-37,43
```

All statements from 37 through 43 inclusive are deleted from the user's current program.

4.2.4 RENUMBER

The RENUMBER command allows the user to renumber any of the statements in the current program; the form is

```
RENUMBER-[newfirst[,delta[,oldfirst[,oldlast]]]]
```

oldfirst and oldlast specify the range of original statements to be renumbered (defaults are 1,9999). If only oldfirst is specified, the default for oldlast is 9999. The first of these statements is assigned the number newfirst (default is 10) and each of the remainder is assigned a statement number delta greater than its predecessor (default for delta is 10). Any statement in the program which references a renumbered statement is changed as required for consistency.

Example

```
>RENUMBER
```

The statements in the current program are renumbered in increments of 10 starting with statement number 10.

```
>REN-3,7,50,250
```

The old statement numbers 50 through 250 are renumbered starting with 3 and increasing by 7.

4.2.5 SIZE

The SIZE command reports the status of the current program.

Format

```
SIZE
```

Example

```
>SIZ: AVAIL=9460 PROG=36 VAR=24 BUF=0/2
```

AVAIL indicates the number of available bytes. PROG indicates the number of bytes occupied by the program. VAR indicates the number of bytes occupied by program variables. BUF indicates the number of available buffers/number of allocated buffers (see NEW, Section 4.2.2, for an explanation of buffers).

4.2.6 CLEAR COMMAND

The CLEAR command causes all variables to become undefined (the space they occupied being deallocated). All function calls, GOSUB's and FOR's are also reset, and all files are closed. The form is:

```
CLEAR
```

CLEAR frees all space allocated during the execution of a program. A CLEAR is automatically performed when the RUN command is issued.

Example

```
>SIZ:  AVAIL=9460 PROG=36 VAR=24 BUF=1/2
>CLEAR
>SIZ:  AVAIL=9520 PROG=0  VAR=0  BUF=2/2
```

Examples Using Editing Commands

The user inputs a program; a mistake is made in line 30, so the line is re-entered.

```
>10  INPUGNT A,B,C,D,E
>20  REM..INPUT 5 VALUES
>30  LET F=(A+B)/5
>40  REM..S=AVERAGE OF 5 INPUT VALUES
>50  PRINT S
>30  LET S=(A+B+C+D+E)/5
```

LIST correctly lists the program:

```
>LIST
  10  INPUT A,B,C,D,E
  20  REM INPUT 5 VALUES
  30  LET S=(A+B+C+D+E)/5
  40  REM..S=AVERAGE OF 5 INPUT VALUES
  50  PRINT S
```

SIZE gives the length in bytes:

```
>SIZ:  AVAIL=11648 PROG=125 VAR=0 BUF=2/2
```

The remark lines are deleted and the program is listed:

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

>SIZ:  AVAIL=11710 PROG=63 VAR=0 BUF=2/2
```

Next, the program is renumbered and listed again:

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

The program is deleted. When LIST is now specified, there is no current program; the computer returns a ">" to prompt for further entries:

```
>NEW
>LIST
>
```

4.3 DISK-RELATED COMMANDS

When a current program is complete, and is to be used again, it should be saved on the disk. A copy of the current program is not affected; it remains the current program until the user ends the BASIC session or until it is deleted with the NEW command.

When a program is saved, it must be given a name with the SAVE or ASAVE command. The program name is used to get, to append, or to kill a program from the disk. The name must be unique among names on a particular disk, but it may be duplicated on other disks. A catalog of the programs and files contained in the user's library may be requested with the RIO CAT command.

All program files created using BASIC are appended with the suffix ".BP". The suffix is not used when manipulating program files within BASIC. The suffix must be used when manipulating program files outside of BASIC.

SAVed files created by BINBASIC are tagged with SUBTYPE=2. SAVed files created by BASIC are tagged with SUBTYPE=3. The files are not compatible. ASAVed files are compatible.

4.3.1 SAVE

The SAVE command stores a copy of the current program on the user's disk; the form is

```
SAVE-programname
```

If there is no file with the same name on the user's disk, a new file is created and a copy of the current program stored on it. If a file with the same name already exists on the disk, the SAVE command is rejected.

The file created contains a copy of the BASIC program in "compiled" form. This form is more compact than the ASCII source form and is faster to retrieve than the source form. It can, however, only be read meaningfully by BASIC as a program. The ASAVE command (Section 4.3.2) produces a saved form of the BASIC program in ASCII source form. This form may be edited with the RIO Editor and processed by other subsystems which deal with ASCII files.

NOTE: The suffix ".BP" is appended to the programname.

Example

```
>SAVE-PROGX
```

The name PROGX.BP is assigned to the copy of the current program that is saved on the user's disk.

4.3.2 ASAVE

The ASAVE command stores a copy of the current program on the user's disk. The form of the file is an ASCII source representation of the program (that is, the characters and lines of the program itself). This file may be edited with the RIO editor and processed by other subsystems that deal with ASCII files.

```
ASAVE-programname
```

NOTE: The suffix ".BP" is appended to the programname.

If there is no file with the same name on the user's disk, a new file is created and a copy of the current program stored on it. If a file with the same name already exists on the disk, the ASAVE command is rejected.

The program is normally ASAVed with line indentation off. A SYSTEM "ASINON" call (Section 3.12) will cause the program to be ASAVed with line indentation.

Example

```
>ASA-MY.NEW.PROGRAM
```

The name MY.NEW.PROGRAM.BP is assigned to the copy of the current program that is saved on the user's disk.

4.3.3 RSAVE

The RSAVE command is similar to SAVE except that the specified program name must already exist. The current program is written to the file.

```
RSA-programname
```

If there is no file with the name programname.BP an error is given and the command rejected. Otherwise, the current program is copied to that file. The form [SAVE or ASAVE] is the same as the file's current contents.

Example

```
>RSA-PROGX
```

The current program is copied to existing file PROGX.BP.

4.3.4 GET

The GET command loads a specific Zilog BASIC program into the user's working space; the form is

```
GET-programname
```

where programname.BP is the name of a program to replace the current program.

Example

```
>GET-SEARCH
```

SEARCH.BP is a program saved on the disk. It is now also available in the user's work area replacing any previous program in that area.

4.3.5 XEQ

The XEQ command loads and runs a Zilog BASIC program. It is equivalent to the sequence GET followed by RUN. See Section 4.1.2 for details.

4.3.6 APPEND

The APPEND command appends a specified program to the user's current program; the form is

```
APPEND-programname
```

The program programname.BP is appended to the end of the current program. Only programs which have been ASAVed (Section 4.3.2) may be appended. Programs which have been saved in pseudo-compiled form (see SAVE command, Section 4.3.1) may not be appended. Line numbers that already exist in the workspace will be replaced by duplicate line numbers from the APPEND file.

Example

```
>APPEND-PROGX
```

PROGX.BP is a program ASAVed on the disk. It is appended to the program currently in the user's work area.

4.3.7 OBTAINING A LIST OF BASIC PROGRAMS

A list of BASIC programs may be obtained using the RIO CAT command, specifying all files ending in ".BP".

NOTE: BASIC must be exited using a QUIT command for RIO commands to be recognized.

Example

```
%CAT *.BP
```

4.3.8 DELETING FILES

All files, including Zilog BASIC programs (filenames ending in ".BP") may be deleted by using the RIO DELETE command or the BASIC ERASE statement (Section 10.4).

Examples Using Disk Commands

A program is input and saved on the disk. The program is then deleted.

```
>100  INPUT A,B,C,D,E
>120  LET S=(A+B+C+D+E)/5
>130  PRINT S

>ASAV-AVERAGE
>NEW
```

A second program is entered and saved. The first program is then appended to this program to make a third program. It too is saved:

```
>10  INPUT R
>20  P=3.14
>30  A=P*R^2
>40  PRINT A
>SAVE-AREA
>APPEND-AVERAGE
>SAVE-CALC
```

Any of these programs may now be brought back as the current program with GET. To illustrate, each is retrieved and then listed:

```
>GET-AVERAGE
>LIST
  100  INPUT A,B,C,D,E
  120  LET S=(A+B+C+D+E)/5
  130  PRINT S

>GET-AREA
>LIST
  10  INPUT
  20  LET P=3.14
  30  LET A=P*R^2
  40  PRINT A

>GET-CALC
>LIST
  10  INPUT R
  20  LET P=3.14
  30  LET A=P*R^2
  40  PRINT
 100  INPUT A,B,C,D,E
 120  LET S=(A+B+C+D+E)/5
 130  PRINT S
```

To determine whether a particular program is on the user's disk, he can use the RIO CAT command followed by the program name (with .BP appended to the programname). If there are not too many files on the disk, he can simply type CAT or CAT *.BP (to RIO) to get a list of all the files currently saved.

SECTION V

KEYBOARD EXECUTABLE STATEMENTS

In general, all statements must be preceded by a statement number and can only be executed as part of a program, some statements however, can be executed directly from the terminal keyboard. This mode of operation can be very useful for debugging or for performing simple calculations (e.g., values of variables or expressions can be printed). The following statements can be executed directly from the keyboard:

STATEMENT	REFERENCE
CALL	13.1
DIM	7.1, 8.3
REM	3.10
PRINT	3.8, 7.3, 8.10
READ	3.9, 8.11, 10.7.1, 10.7.4
WRITE	10.7.2, 10.7.5
RESTORE	3.9, 8.11, 10.7.3
FILE	10.2
RANDOMIZE	3.11
LET	3.1, 6.5, 8.7
ERASE	10.4
SYSTEM	3.12

Description

When one of the above statements is typed without a preceding statement number, the statement is executed immediately. If the statement is a PRINT statement, the output is printed at the terminal.

If program execution was stopped by pressing ESCape or by a STOP statement, values of variables printed will reflect the current value of the variable in the context in which program execution ceased. This means that if the program is stopped while in a multi-line function, the formal parameter variables will have values as used in the function even if there is a global variable with the same name.

Traps (Section 3.13) are disabled during keyboard statement execution.

Example

```
>10  LET A=1
>20  STOP
>30  PRINT A
>RUN
STOP AT 20
PRINT A
 1
>LET A=4
>CONTINUE
4

READY
>
```

SECTION VI

NUMERIC VARIABLE TYPES

Zilog BASIC allows floating point real and integer numeric types. These types apply to variables, arrays, constants, expressions, assignments, and functions.

6.1 TYPE SPECIFICATION

Numeric variables and arrays have a specific data type. A suffix character appended to the variable name determines this type. Variables with the same name but different types are distinct from each other.

Description

Variables with no suffix character are of type REAL. See Section 6.2.2 for a description of the representation of floating point numbers in BASIC and BINBASIC.

Variables with a suffix character "%" are of type INTEGER. The range of integers is -32768 to 32767.

Variables with the suffix character "$" are used to hold strings (see Section VIII).

6.2 NUMERIC CONSTANT FORMS

When constants are used in an expression, DATA statement, or during execution of an INPUT statement, they are represented in one of three forms: integer, fixed-point or floating point. Fixed and floating-point numbers are type REAL.

6.2.1 INTEGER FORM

An integer is a series of digits without a decimal point. Examples of the integer form:

```
 10  LET A%=47,B%=-375,C%=607,D%=0
 20  PRINT A%,B%,C%,D%
>RUN
 47          -375           607           0
```

An unsigned integer constant less than 256 is represented internally as type INTEGER. All other numeric constants are represented as type REAL.

When arithmetic operations are performed on expressions containing only integer constants or variables, the results are integers. However, when any operand is type REAL, the result is type REAL.

6.2.2 FLOATING POINT FORM

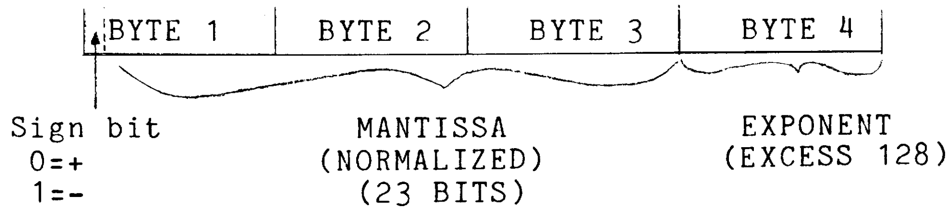
A floating point number is a number that is stored in the computer as a fraction (called the mantissa) and a power of either 2 or 10 (called the exponent). For example,

.3E-11 equals .3*10^(-11).

The floating point representation of binary and decimal numbers follows.

BINARY FLOATING POINT REPRESENTATION:

24 bit sign-magnitude normalized* fraction
8 bit excess-128** exponent (base 2)
sign bit replaces most significant fraction bit (implied "1")
exponent field of zero implies value is zero

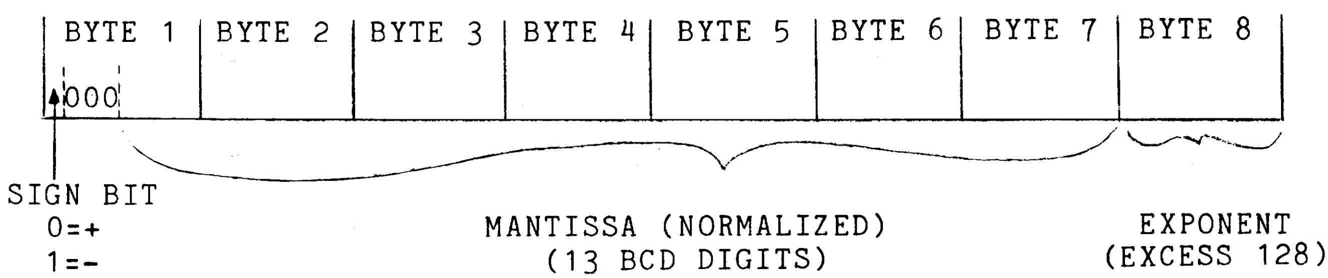



```
7F FF FF FF = 16777215/16777216*2^127 = 1.7014117*10^38
00 00 00 81 = 1/2*2/^1 = 1.0000000
00 00 00 01 = 1/2*2(-127) = 2.9387359*10^(-39)
XX XX XX 00 = 0
FF FF FF FF = -16777215/16777216*2^127 = -1.7014117*10^38
```

all results are rounded to 23 bit fractions

DECIMAL FLOATING POINT REPRESENTATION:

13 BCD digit sign-magnitude normalized* fraction
8 bit excess-128** exponent (base 10)
sign bit is most significant digit
exponent field of zero implies value is zero




```
09 99 99 99 99 99 99 FF = .99999 99999 999 * 10^127
01 00 00 00 00 00 00 81 = 1.0
01 00 00 00 00 00 00 01 = .1 * 10^(-127)
XX XX XX XX XX XX XX 00 = 0
89 99 99 99 99 99 99 FF = -.99999 99999 99 * 10^127
```

all results are truncated to 13 digit fractions

*Normalized means that the exponent is adjusted such that the most significant digit of the mantissa is non-zero or the mantissa is zero.

**Excess 128 means that the power to raise the base is equal to the exponent field minus 128.

This example assigns values to and prints two real variables.

```
 10  LET I=2795348.6,J=2.79E-3
 20  PRINT I,J
>RUN
 2795348.6    .00279
```

6.3 NUMERIC EXPRESSIONS

Variables of all data types and numbers of all data forms can be used in numeric expressions. Zilog BASIC provides the arithmetic operations for both data types as well as automatic conversion when two operands are not of the same type. The following table summarizes the results of combining arithmetic elements with any operator (except &, !, ~, /, ^, and relationals):

TABLE 6-1

Second Element Data Type

		INTEGER	REAL
First Element Data Type	INTEGER	INTEGER	REAL
	REAL	REAL	REAL

When the operators &, !, ~, =, <, >, <=, >=, and <> are used the result is always type REAL (0 for false, 1 for true). When the operators / and ^ are used, the result is always type REAL.

Examples

An integer combined with a real type in an expression results in a real number; two integers result in an integer:

```
 10 LET I%=25,I1%=50,R=2.75
 20 PRINT I%+I1%
 30 PRINT I%+R
>RUN
 75
 27.75
```

6.4 CONDITIONAL NUMERIC EXPRESSIONS

The numeric expression used to make a branching decision in a conditional statement (Section 3.6) can contain any numeric data type. The result will be type REAL. The expression is considered false if equal to 0, true otherwise.

6.5 NUMERIC ASSIGNMENT

When the result of a numeric expression is assigned to a variable, it is converted to the type of that variable

The method of conversion used in assigning values to variables of differing data types is summarized in this table:

TABLE 6-2

Variable Type	Value Type	Conversion Method
INTEGER	REAL	Round
REAL	INTEGER	Float

Note that this table applies wherever values are assigned to variables (INPUT, READ, etc.).

6.6 INPUTTING NUMERIC DATA

Constants of all data forms can be entered using READ and INPUT statements. Once entered they are converted to the type of the receiving variable according to Table 6-2.

6.7 OUTPUTTING NUMERIC DATA

Numbers of all data types can be output with controlled format with the PRINT USING statement (see Section XI). Numbers of all data types can also be written onto mass storage data files. This process is described fully in Section X.

6.8 NUMERIC FUNCTIONS

Most built-in functions which return numeric results return values of type REAL. (User-defined numeric functions are described in Section IX.) These values, when used in expressions or assignments are converted as described in Table 6-2.

Numeric arguments to functions may be of either type REAL or INTEGER and are converted to the type required by the function according to Table 6-2.

6.8.1 ABS FUNCTION

```
ABS(expression)
```

ABS returns the absolute value of the expression.

6.8.2 ATN FUNCTION

```
ATN(expression)
```

ATN returns the arctangent of the expression. The result is in radians. The range is -pi/2 to pi/2.

6.8.3 COS FUNCTION

```
COS(radians)
```

COS returns the cosine of radians MOD 2pi.

6.8.4 EXP FUNCTION

```
EXP(expression)
```

EXP returns e^expression, where e is the Napierian constant, 2.718281828.

6.8.5 INT FUNCTION

```
INT(expression)
```

INT returns the largest integer less than or equal to the expression.

6.8.6 LOG FUNCTION

LOG(expression)

LOG returns the natural logarithm of the expression. The expression must be greater than zero.

6.8.7 RND FUNCTION

RND

RND returns a pseudo-random number greater than or equal to zero and less than one. The seed for the RND function may be changed by the RANDOMIZE statement (Section 3.11).

6.8.8 SGN FUNCTION

SGN(expression)

SGN returns the sign of the expression. If expression>0, SGN returns 1. If expression=0, SGN returns 0. If expression<0, SGN returns -1.

6.8.9 SIN FUNCTION

SIN(radians)

SIN returns the sine of radians MOD 2pi.

6.8.10 SQR FUNCTION

SQR(expression)

SQR returns the square root of the expression. The expression must be greater than or equal to zero.

6.8.11 TAN FUNCTION

TAN(radians)

TAN returns the tangent of radians MOD 2pi.

SECTION VII

ARRAYS

An array (or matrix) is a set of variables which is known by one name. The individual elements of an array are specified by the addition of a subscript to the array name: for example, A[7] is the seventh element of array A.

Arrays have either one or two dimensions. A one-dimensional array consists of a single column of many rows. The elements are specified by a single subscript, indicating the row desired. Rows and columns are numbered starting with 1. A two-dimensional array consists of a specified number of rows and a specified number of columns organized into a table. For example, an array A of four rows and three columns can be represented as follows:

		Columns		
		1	2	3
Rows	1	A[1,1]	A[1,2]	A[1,3]
	2	A[2,1]	A[2,2]	A[2,3]
	3	A[3,1]	A[3,2]	A[3,3]
	4	A[4,1]	A[4,2]	A[4,3]

Each element of the array is specified by a pair of subscripts separated by commas; the first indicates the row and the second indicates the column.

Every array in a Zilog BASIC program is defined in one of two ways:

> Through a DIM statement that specifies the array name, and the number of rows and columns.
>
> Through usage - numeric arrays that are used but are not explicitly defined in a DIM or type statement have 10 rows if one-dimensional or 10 rows and 10 columns if two-dimensional.

The physical size of an array is the total number of elements originally allocated to it; the logical size is the current number of rows times the current number of columns. The physical size of an array cannot be changed during execution, but the logical size (that is, the number of rows and columns) can be changed with a DIM statement so long as the physical size is not exceeded.

Zilog BASIC permits arrays of all numeric data types as well as one-dimensional string arrays. Remarks in this section refer to numeric arrays, unless otherwise noted. String arrays are described in Section VIII.

This section describes DIM as used for numeric arrays.

Parentheses and square brackets ('[' ']') are equivalent when used for specifying subscripts on array variables.

7.1 DIM STATEMENT

The DIM statement is used to reserve storage for arrays and to set upper bounds on the number of elements in arrays. DIM statements may also be used with strings (see Section VIII). This section only refers to numeric arrays.

Format

```
DIM variable[integer],variable[integer],...
```

where the variable is the array name, and the integer specifies the number of rows in a one-dimensional array.

```
DIM variable[integer,integer],variable[integer,integer],...
```

where the variable names a two-dimensional array, the first integer specifies the number of rows in the array, and the second integer the number of columns.

Rows and columns are numbered starting with 1. The overall size is the number of elements. In a one-dimensional array it is identical to the number of rows; in a two-dimensional array it is the product of the rows and columns.

Both one- and two-dimensional numerical arrays and string variables can be named in the same DIM statement; they are separated by commas. Each element in the numeric arrays is set to zero.

Description

The elements of an array are specified by subscripted variables. The values of the elements are zero after the DIM statement is executed. The number of elements in the array is defined by a DIM statement or by usage. The DIM statement can appear anywhere in a program and is executed. Thus, the DIM statement must be executed before the array is referenced.

Example

```
10 DIM A[17],A7[6,8],B[2,5]
20 REM   A HAS 17 ROWS, ONE COLUMN
30 REM   A7 AND B ARE TWO-DIMENSIONAL ARRAYS
40 REM   A7 HAS 6 ROWS, 8 COLUMNS;B HAS 2 ROWS, 5 COLUMNS
50 DIM C[5],C1[5,1],C2[1,5]
60 REM   C AND C1 HAVE THE SAME DIMENSIONS:  5 ROWS, 1 COLUMN
70 REM   C2 HAS 1 ROW, 5 COLUMNS
```

Note that the DIM statement for C1 in line 50 would be different if it included C1[5] since array elements must be referenced with the same number of subscripts as in the DIM statement.

The DIM statement can be used to change the number of rows and columns in an existing array.

When using DIM to redimension an array, the number of rows and columns can be changed as desired provided these three conditions are met:

1) The number of dimensions must not be changed

2) The total number of elements (rows times columns) must not be increased beyond the physical size (original dimensions) of the array.

3) The array is numeric. String arrays may not be redimensioned.

NOTE: Any data in the array is lost as the array is initialized to zero when redimensioned.

7.2 STORING DATA IN ARRAYS

There are several methods of assigning values to arrays. Individual elements can be assigned using the assignment statement:

```
10  LET A[5]=26
20 A7[1,6]=N*4.5
```

In addition, individual elements can appear in INPUT and READ statements:

```
10 INPUT A[1],A[2],A[3]
20 READ A7[3,2]
```

FOR loops can be used to fill entire arrays element by element:

```
10  DIM A[17],A7[6,8]
20  FOR N=1 TO 17
30    INPUT A[N]
40  NEXT N
50  FOR N=1 TO 6
60    FOR M=1 TO 8
70      READ B[N,M]
80    NEXT M
90  NEXT N
```

7.3 PRINTING DATA FROM ARRAYS

The mechanisms for printing data from arrays are parallel to those used for filling arrays. Individual elements can be printed using PRINT:

```
100 PRINT A[1],A[2],A[3]
```

FOR loops can be used to print entire arrays element by element:

```
 90  DIM A[17],A7[6,8]
100  FOR N=1 TO 17
110    PRINT A[N]
120  NEXT N
130  FOR N=1 TO 6
140    FOR M=1 TO 8
150      PRINT B[N,M]
160    NEXT M
170  NEXT N
```

SECTION VIII

STRINGS

Zilog BASIC allows the programmer to manipulate character strings through the use of string literals, variables, arrays, functions, operators, assignment, statements, and input/output statements. Many of the uses of strings are enhancements to statements that have already been described, such as READ and PRINT.

8.1 LITERAL STRINGS

A literal string is a sequence of up to 255 characters. Each character is represented internally by a number between zero and 255 as defined in the standard ASCII character set (see Appendix A). Some of these characters have graphic representations (they can be printed - A,B,d,%), while others do not (they are nonprinting - return, linefeed). Both types of characters can be included in a literal string, but each is handled differently.

Format

A literal string consists of a series of graphic characters surrounded by quote marks:

```
"character string"
```

The quote mark (") and the left angle bracket (<) cannot be included as a character in the character string.

The quote mark, left angle bracket, and nonprinting characters can, however, be included in a literal string by using the integer numeric equivalent of the character enclosed in angle brackets.

```
<integer>
```

The integer is the ASCII code of the desired character and may be in the range 0-255, but it is good practice to restrict this form to nonprinting characters, the quote mark (34) and the left angle bracket (60). Nonprinting characters can be combined with quoted strings in a literal string.

Description

Literal strings can include both upper case and lower case letters. When a literal string is printed, each character value is printed literally on the output device. However, when a program is listed, literal strings are listed with graphic characters (except the quote mark and left angle bracket) in quotes and non-graphic characters represented in the angle bracket form. Characters represented in a literal string graphically have their higher order bit equal to zero, i.e., they are all represented by ASCII values less than 128.

Examples

""	A null string (a string of zero length)
"BASIC"	
"B "	
"<13><10>"	Carriage return, line feed
"LINE 1<13><10>LINE 2"	The literal prints on two lines
"A<124>B"	The literal is A vertical line B
"<34>"	The quote mark

8.2 STRING VARIABLES

A string variable (simple or subscripted) is used to hold a series of ASCII characters. The declared size of a string variable is called its physical length. The maximum length of any string variable is 32767 characters. String variables are further constrained in that they must fit inside available main memory.

During execution, each string variable contains strings whose length cannot exceed the variable's physical size. This dynamic length is called the logical length of the variable and is initialized to zero (i.e., the null string) at the beginning of program execution.

Simple and subscripted string variables must be dimensioned in a DIM statement (section 8.3).

Format

A simple string variable is referenced by its name and an optional substring designator in parentheses or brackets.

```
string name

string name[first character]

string name[first character,last character]
```

The string name is a letter followed by a "$" or a letter and a digit followed by a "$".

The substring designator consists of one or two numeric expressions, separated by a comma. The first expression always specifies the first character position of the substring. The second expression specifies the last character position. If there is only one expression, the ending character position is the last character of the string.

A string array variable is referenced by the string name followed, in parentheses or brackets, by a subscript and an optional substring designator separated by a comma.

```
string name[subscript]

string name[subscript,first character]

string name[subscript,first character,last character]
```

The subscript is an integer expression that specifies the element of the array to be selected. Since a string array may have only one dimension, there may be only one subscript value.

The substring designator and the string name are specified in the same way for string array variables as for simple string variables.

NOTE: Unlike numeric array variables, a string array variable must not have the same name as a simple string variable.

Description

Any string variable, simple or subscripted, can be qualified by a substring designator, which is used to select a part of the string to be extracted.

If the substring is specified by a single expression, the substring equals the rest of the string taken from the position indicated by the expression.

If two expressions are separated by a comma, the substring consists of the characters from the position specified by the first expression to the position specified by the second expression. (Note: the second expression can be less than the first; this specifies the null string.)

If A$ is a simple variable:

A$(3,5)	is the 3rd through 5th character of the string
A$(3,2)	is the null string
A$	every character in the string is selected

If B$ is an array variable:

B$(3)	is the entire 3rd string in the string array
B$(2,3,5)	is the 3rd through 5th characters in the second string of the string array

A string array variable must always be subscripted.

The subscript and substring designator expressions may be any integer expressions. Suppose the variables I and J are used, with I equal to 5 and J equal to 10:

C$(I)	is the 5th character to the end of the string if C$ is a simple string variable; it is the entire 5th string element if C$ is a string array variable.
C$(I,J)	is the 5th through 10th character if C$ is a simple string variable; it is the 10th character to the end of the string of the 5th string if C$ is a string array variable.

If a substring extends beyond the logical length of a string variable, only the characters that exist in the string are returned and a warning message is issued.

Examples

```
 10 DIM A$[10]
 20 A$="ABCDEFGHIJ"
 30 PRINT "STRING A$=";A$
 40 PRINT "SUBSTRING A$(3)=";A$(3)
 50 PRINT "SUBSTRING A$(4,7)=";A$(4,7)
 60 PRINT "A$(7,5)";A$(7,5);"=NULL STRING"
>RUN
STRING A$=ABCDEFGHIJ
SUBSTRING A$(3)=CDEFGHIJ
SUBSTRING A$(4,7)=DEFG
A$(7,5)=NULL STRING
```

8.3 DIM STATEMENT WITH STRINGS

Literal strings can be contained in string variables, simple or subscripted. Simple string variables and array string variables must be dimensioned in a DIM statement. The purpose of the DIM statement is to reserve storage for strings and arrays and to establish their names and maximum size.

Format

The DIM statement consists of the word DIM followed by a list of variable and array definitions separated by commas.

```
DIM variable[string size],variable[string size],...
```

where variable is the name of a simple string variable specified as a letter followed by a "$" or a letter and a digit followed by a "$". The string size is an integer expression that specifies the maximum number of characters the string can contain.

```
DIM variable[array size,string size],variable[array size,
  string size],...
```

The array size specifies the total number of elements in the array; the string size specifies the maximum number of characters in each element. Only one-dimensional string arrays are allowed. Both array size and string size are integer expressions.

If more than one variable is included in a single DIM statement, they must be separated by commas. Simple string variables, subscripted string variables and numeric arrays (Section 7.1) may be dimensioned in the same DIM statement.

Description

String arrays must be declared in DIM; there is no implicit size for string arrays as there is for numeric arrays. String variables and elements of string arrays are initialized to the null string.

NOTE: String variables and arrays may not be redimensioned.

Example

```
 10 DIM A$[20],B$[5,35],C$[5,3]
 20 LET A$="TITLE OF SECTION IS "
 30 FOR K=1 TO 5
 40   READ B$[K]
 50 NEXT K
 60 FOR K=1 TO 5
 70   READ C$[K]
 80 NEXT K
 90 DATA "INTRODUCTION TO BASIC","EXPRESSIONS","STATEMENTS"
100 DATA "COMMANDS","KEYBOARD EXECUTABLE STATEMENTS"
110 DATA "  I"," II","III"," IV","  V"
120 FOR K=1 TO 5
130    PRINT A$[1,17];C$[K];A$[17,20];B$[K]
140 NEXT K
>RUN

TITLE OF SECTION   I IS INTRODUCTION TO BASIC
TITLE OF SECTION  II IS EXPRESSIONS
TITLE OF SECTION III IS STATEMENTS
TITLE OF SECTION  IV IS COMMANDS
TITLE OF SECTION   V IS KEYBOARD EXECUTABLE STATEMENTS
```

A substring of A$ is printed, followed by the kth element of C$, another substring of A$, and the kth element of B$. This example lists the titles of the first five sections of this manual.

8.4 STRING EXPRESSIONS

String expressions consist of one or more source strings (literal strings, string variables, string valued functions) combined from left to right with the concatenate operator (+) to form a single new string value. String expressions can be assigned to string variables or compared with other string expressions to form a numeric expression.

Format

The format is a list of source strings separated by "+"

```
string

string + string...
```

Each source string can be either a literal string, a string variable, or a string function.

Description

A source string is any entity from which a string value is extracted. The value of the source string is as defined under "String Literals", "String Variables", and "String Functions". An example of a literal string is "BASIC" or "<10>"; of a string variable is A$, C5$(2), B$(2,3), or A1$(5,3,10); of a string function is CHR$(208).

The "+" character, when used between two source listings, is the concatenate operator. The concatenation of two strings produces a temporary string whose characters are those of the first string immediately followed by those of the second. This temporary string can be used in further concatenation operations, in string comparisons, or it can be assigned to a string variable.

The maximum length of any temporary string is 32767 characters. The original operands are unaffected by concatenation.

Legal string expressions:

```
A$+B$(2)+"<10><13>ABCD"+C$(3,1,2)

"BASIC"+C5$(2)

"BASIC"

C5$(2)
```

Example

```
 10 DIM A$[5],B$[10,10]
 20 LET A$="CON",B$[1]="CATENATION"
 30 PRINT A$+B$[1,1,7]+B$[1,4,4]
>RUN
CONCATENATE
```

8.5 STRING-RELATED FUNCTIONS

There are a number of predefined functions in Zilog BASIC that accept string values as parameters and/or return a string value as their result. (User-defined string functions are described in Section IX.)

8.5.1 CHR$ Function

```
CHR$(integer expression)
```

where integer expression results in a value in the range 0 to 255 inclusive. The value of CHR$ is the string character that corresponds to the value of the expression in the standard character set (see Appendix A). For example,

```
 10 PRINT CHR$(65)
>RUN
A
```

8.5.2 ASC FUNCTION

```
ASC(string expression)
```

ASC returns the numeric value of the first character of the string in the expression according to the standard character code in Appendix A. For example,

```
 10 PRINT ASC("A")
>RUN
 65
```

8.5.3 LEN Function

LEN returns the logical length of the string expression. For example,

```
 10 DIM A$[20]
 20 LET A$="ABCDEFG$"
 30 PRINT LEN(A$)
>RUN
 8
```

8.5.4 POS FUNCTION

POS(stringA,stringB)

where stringA and stringB are any string expressions. POS returns the smallest integer that represents the starting position of a substring in stringA that exactly equals stringB. If stringB is not a substring of stringA, then POS equals zero. If stringB is null, then POS equals one. For example,

```
 10  PRINT POS("12ABC34","C3")
>RUN
 5
```

8.5.5 VAL FUNCTION

VAL(string expression)

VAL returns the numeric value represented by the characters in the string expression. An error occurs if no legal number is found at the beginning of the string expression. The number is considered to begin at the first character of the string expression and end on the first character that is not legal in a number. Blanks are ignored and E-notation may be used. For example,

```
 10  DIM A$[30]
 20  LET A$="123X4EZ"
 30  PRINT VAL(A$)
 40  PRINT VAL(A$[5,6]+"9")
>RUN
 123
 4000000000
```

8.5.6 STR$ FUNCTION

STR$(numeric expression)

The STR$ function returns a string representing its single numeric argument. The string is in the form that would be produced by the PRINT statement except that all blanks are removed. For example,

```
 10  DIM A$[4761]
 20  LET A$=STR$(2.36*4)
 30  PRINT A$
 40  PRINT VAL(A$)
 50  PRINT STR$(VAL(A$))
>RUN
9.44
 9.44
9.44
```

8.5.7 LEFT$ FUNCTION

LEFT$(string expression,integer expression)

LEFT$ returns the n leftmost characters of the string expression. The integer expression gives the position of the last character to be returned. For example,

```
 10  PRINT LEFT$("ABCDE",3)
>RUN
ABC
```

8.5.8 RIGHT$ FUNCTION

RIGHT$(string expression,integer expression)

RIGHT$ returns the rightmost characters of the string expression, from the nth character to the end. The nth character is indicated by the integer expression. For example,

```
 10  PRINT RIGHT$("ABCDEFGHIJ",4)
>RUN
DEFGHIJ
```

8.5.9 SEG$ FUNCTION

SEG$(string expression,integer expression,integer expression)

SEG$ returns the substring of characters of the string expression from the character specified by the first integer expression to the character specified by the second integer expression. For example,

```
 10  PRINT SEG$("ABCDEFG",3,6)
>RUN
CDEF
```

8.6 COMPARING STRINGS

String expressions can be compared with relational operators to produce a result of true (numeric 1) if the relation holds or false (numeric 0) if the relation does not hold. The relational operators are:

=	Equal
<>	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal
>=	Greater Than or Equal

Two strings are equal only if they have the same logical length and each character matches. A string is less than another if its first character that does not match the other is numerically less (according to the standard character code in Appendix A) or it is an initial proper subset of the other (e.g., "AB"<"ABC" but "BA>"ABC").

A string comparison can appear within a numeric expression, since the result is a number. The string relational operators have the same position in the hierarchy of operators as do the numeric relations. For example, these are string comparisons:

```
A$=B$

A$=B$!C$>=D$

(A$<>"BOB")+5
```

See Section 2.5 for the meaning and hierarchy of relational operators.

A common use of string comparisons is in IF statements.

Examples

```
 10  DIM A$[10],B$[10]
 15  FOR K=1 TO 4
 20  READ A$,B$
 30  IF A$<B$ THEN PRINT A$;"<60>";B$
 40  ELSE DO
 50    IF A$=B$ THEN PRINT A$;"=";B$
 60    ELSE PRINT A$;">";B$
 70  DOEND
 80  NEXT K
 90  DATA "ABC","ABCD","ABC","B"
100  DATA "ABC","ABC","C",""
>RUN
ABC<ABCD
ABC<B
ABC=ABC
C>
```

8.7 STRING ASSIGNMENT

The assignment operator (=) can be used to assign a string value (defined by a string expression) to one or more string variables (or substrings of string variables). Several different assignments can appear in one LET statement.

Format

The formats of LET are

```
LET variable=expression

LET variable=expression,variable=expression,...
```

The word LET is entirely optional and can be left out. The variable is an entire string variable (simple or subscripted) or part of a string variable (indicated by a substring designator) into which a string value is to be copied. Numeric assignments as described in Section 3.1 can be mixed with string assignments in the same LET statement.

Description

The execution of a LET statement proceeds as follows. The subscripts of variables to be assigned values are evaluated from left to right. The expression is then evaluated and assigned to the variable. The manner in which each assignment occurs depends upon the number of substring subscripts specified for the destination variable.

If there is no substring designator, the entire variable is replaced by the string value. If the new value will fit entirely into the variable, the logical length of the variable is set to the length of the new value. If the variable is too small, a warning message is printed, the value is truncated on the right and the logical length of the string is made equal to the physical length.

If there is one substring subscript, it specifies the starting position for the assignment. The entire string value is copied into the variable starting with the indicated position and continuing to the physical end of the variable or the end of the string value, whichever comes first. The part of the variable preceding the subscript is unchanged. The starting subscript must be no more than one greater than the current logical length of the variable (i.e., there can be no undefined character positions in the middle of a string variable). If the variable is too small, the value is truncated on the right and a warning message is printed.

If two substring subscripts are specified, they define a field within the variable into which the string value is stored. If necessary, the value will be truncated on the right and a warning message printed, or padded out with blanks to fit exactly the substring specified. The substring for the destination must not extend beyond the physical length of the string variable and all previously mentioned rules must be followed also. The new logical length of the variable is the larger of the old logical length or the last position of the substring. Any characters from the old value to the left or right of the substring are unchanged.

Example

```
 10  DIM A$[10]
 20  LET A$="1234567890"
 30  PRINT A$
 40  LET A$[5]="ABCDEF"
 50  PRINT A$
 60  LET A$[7,9]="1234"
 70  PRINT A$
 80  LET A$[6,8]="X"
 90  PRINT A$
100  LET A$=A$[1,4]+"567890"
110  PRINT A$
>RUN
1234567890
1234ABCDEF

WARNING 146 AT   60
1234AB123F
1234AX  3F
1234567890
```

Note that the literal "1234" in line 60 is truncated to fit in substring A$(7,9).

In line 80, substring A$(6,8) is blank filled since "X" is only one character. The final value of A$ is the same as its original value assigned in line 20.

The example below illustrates variations on assignments to substrings of array elements:

```
 10  DIM A$[3,5]
 20  A$[1]="ABCDE",A$[2]="ABCDE",A$[3]="ABCDE"
 30  LET A$[1,3]=A$[2]
 40  PRINT A$[1],A$[2],A$[3]
 50  LET A$[2,4,5]=A$[3]
 60  PRINT A$[1],A$[2],A$[3]
 70  LET A$[2]=A$[1,1,1],A$[3,2,3]=A$[1,1,1]
 80  PRINT A$[1],A$[2],A$[3]
>RUN

WARNING 146 AT   30
ABABC           ABCDE           ABCDE

WARNING 146 AT   50
ABABC           ABCAB           ABCDE
ABABC           A               AA DE
```

8.8 STRING INPUT STATEMENT

The INPUT statement can be used to assign string constants to string variables from the terminal.

Strings may be quoted or unquoted. If unquoted, leading and trailing blanks are removed and the input item ends on a comma (,) or return.

The rules used to assign the value to the variable are those described under "String Assignment" (Section 8.7).

Examples

```
   10  DIM A$[16],B$[2,5],C$[40]
   20  INPUT A$,B$[1],B$[2],C$
   25  PRINT
   30  PRINT A$;B$[1];B$[2];C$
>RUN
?"THE VALUE OF B$=","1234 "," 2X5 ", "X5=ABC"
THE VALUE OF B$=1234 2X5 X5=ABC
```

8.9 STRING LINPUT STATEMENT

The LINPUT statement accepts all the characters that a user types in at the terminal and assigns them as a string to a specified string variable.

Format

```
LINPUT string variable

LINPUT string literal,string variable
```

where the string variable is the destination of the input. The variable may be simple or subscripted. In the second form, the string literal replaces the standard question mark prompt.

Description

All characters are accepted including quotes, commas and blanks. Input is terminated by a carriage return.

Example

```
  10  DIM A$[20]
  20  PRINT "TYPE 20 CHARACTERS:"
  30  LINPUT "",A$
  35  PRINT
  40  PRINT A$
  50  LINPUT "TYPE 5 CHARACTERS:";A$
  55  PRINT
  60  PRINT A$
>RUN
TYPE 20 CHARACTERS:
"ANY CHARACTERS" O.K.
"ANY CHARACTERS" O.K
TYPE 5 CHARACTERS:E"+"*
E"+"*
```

Because more than 20 characters (the size of A$) were input by the user, the final period in the first input is truncated. In the second input, quotes are entered as part of the string.

8.10 STRING PRINT STATEMENT

Any string expression can be output to the list device (e.g., the terminal) using the PRINT statement. The size of the output field is the number of printed characters in the string value. If the string expression is preceded by a comma, it is printed starting in the next division. Each print line is divided into five divisions, each with a width of 14 characters (see PRINT statement, Section 3.8). If the string expression is preceded by a semicolon, it is printed immediately following the preceding output.

Strings can be output to the terminal with special formats through the PRINT USING statement (see Section XI, Formatted Output). Strings can be output to files as described in Section X.

Example

```
 10 DIM C$[10],N5$[3,5]
 20 LET C$="XK9-753-20",A=2.5,B=1E-19,N5$[1]="ABCDE"
 30 PRINT A,B,C$
 40 PRINT "BOB"+C$,N5$[1]
 50 PRINT C$+"BOB";N5$[1]
 60 PRINT "<10><34>LINE<34><10><13>-1"
>RUN
 2.5            1.00000E-19          XK9-753-20
BOBXK9-753-20  ABCDE
XK9-753-20BOBABCDE

"LINE"
-1
```

In line 60, the <10> (linefeed) causes a linefeed and the <13> (carriage return) causes a carriage return when the line is printed. The <34> (quote) causes a quote to be printed. The actual quote (") before and after the string LINE in the PRINT statement is not printed.

8.11 STRING READ/DATA/RESTORE STATEMENTS

The READ, DATA, and RESTORE statements can be used with string variables that are simple or subscripted, with or without substrings. The string variable is listed in the READ statement and a corresponding string constant must appear in the DATA statement. A RESTORE statement can be used if the DATA statement is to be read again by a subsequent READ statement. For a full description of READ/DATA/RESTORE statements, see Section 3.9.

String variables can be mixed with numeric variables in READ, but the corresponding constant for each numeric variable must be a numeric value. The string constant is assigned to the variable according to the rules defined in String Assignment, this section. Either numeric or string data elements can be read into string variables. The character representation as it appears in the DATA statement is used in either case.

Strings can also be read from files as described in Section X.

Example

```
 10  DIM A$[20],B$[20]
 20  DATA "BOB","<10>JONES"
 30  READ A$[1,3]
 40  READ A$[4,9]
 50  LET B$="HI"
 60  PRINT B$,A$
>RUN
HI         BOB
              JONES
```

When the PRINT statement is executed, the character for linefeed <10> is printed and causes a linefeed.

SECTION IX

USER-DEFINED FUNCTIONS

A user-defined function is one that is defined within the user program and is called within that program in the same way that a built-in function is called. Function names consist of the letters "FN" followed by a single letter or letter-digit pair followed by an optional type character ("$" or "%"). If no type character is specified, then the function returns a REAL value, otherwise, it returns a value of the specified type: "%" for integer, "$" for string.

A function is called within an expression by referring to its name and an optional list of parameter values enclosed in parentheses. The value returned by the function takes its place in the expression.

There are two levels of complexity in the definition of a Zilog BASIC function. At the simple level, a one-line function simply relates a function name and list of parameters to any expression which may use the parameters to calculate the result value. The multi-line function is a more complex entity; it can consist of many statements. It returns its result value with a RETURN statement.

For a discussion of Zilog BASIC built-in functions, see Functions in Section 2.3. A complete list of the built-in functions available to the Zilog BASIC user is contained in Appendix D.

9.1 ONE-LINE FUNCTION

A one-line function is defined completely in one line, using the function DEF statement; its result is calculated by an expression.

Format

The formats for one-line function definitions are:

DEF function-name(formal parameter list)=expression

DEF function-name=expression

DEF string-function-name(formal parameter list)=string expression

DEF string-function-name=string expression

The optional formal parameter list includes

Real parameters (i.e., no type suffix)

Typed parameters (i.e., variable name with a type suffix)

The expression can be any legal numeric or string expression, and can make use of both parameters and program variables.

Description

The parameters in a function definition are formal parameters; when the function is called, they are replaced by the actual parameters which are passed to the function. All variables used as formal parameters are local to the function; that is, they are unrelated to any program variables having the same name. The formal and actual parameters are matched according to their position in the list.

The DEF statement is executable although the function it defines can be entered only by referring to the function name within an expression. The DEF statement defining a function must be executed prior to a reference to the function itself.

Subsequent DEF statements with the same function name redefine that function. The previous definition is forgotten.

Examples

```
10   DEF FNZ(C,D)=C*(D+10)-6
```

The function FNZ is type real. The formal parameters C and D are also type real. When called, the actual parameters will give values to C and D, then the expression C*(D+10)-6 will be evaluated, and the result will replace the function name where it appears in an expression.

```
20   DEF FNG$(K$,L$)=K$+L$+K$
```

The function FNG$ is a string function. The formal parameters K$ and L$ are string variables that will be assigned values according to the matching actual parameters in the function call. When called, the literal string resulting from the concatenation of the values K$, L$, and K$ will replace the function name in the expression where it appears.

```
30   DEF FNB%(A%,X2%)=A%*X2%+(A%+X2%)
```

The function FNB% is an integer function that results in an integer value when called. The computations will be performed in integer arithmetic because both A% and X2% are integers.

9.2 MULTI-LINE FUNCTIONS

A multi-line function is written as several contiguous statements beginning with a DEF statement and ending with an FNEND statement. Execution of the function ends when a function RETURN statement is encountered; this sends the result value back to the place of call.

Format

A multi-line function definition has three parts; the function head, the function body, and the function end.

The function head appears as

```
DEF function-name(formal parameter list)

DEF function-name
```

All parts of these function definitions are the same as described for one-line functions.

The function body consists of a sequence of statements, including at least one function RETURN statement:

```
RETURN expression
```

The expression is numeric or string depending on whether the function is numeric or string. For numeric functions, the RETURN expression is converted to the type of the function.

The function end consists of a one-word statement;

```
FNEND
```

This statement must always be the last statement in the function definition.

Description

The body of a function can contain any Zilog BASIC statements with the following restrictions:

1) A function definition cannot appear within a function body, but function calls are allowed, including calls to the same function.

2) The function body must be self-contained; FOR loops and DO-blocks must be completed within the body and branches must not occur into or out of the body.

The formal parameters in a multi-line function head are specified in the same way as those in the one-line function definition. The formal parameters may be altered in the body of the function. The value of the actual parameter, however, is never affected by the change to the formal parameter.

The following multi-line function returns a string value; its formal parameter is a string variable:

```
10  DEF FNR$(A$)
20    REM..FNR$ RETURNS THE REVERSE OF A$
30    IF LEN(A$)<=1 THEN RETURN A$
40    RETURN FNR$(A$[2])+A$[1,1]
50  FNEND
```

9.3 CALLING A USER-DEFINED FUNCTION

A user-defined function is called by referring within an expression to the function name followed by a list of actual parameters in parentheses. The function call is replaced by the value returned by the function.

Format

A function call has the form:

```
function-name(actual parameter list)

function-name
```

The optional parameter list contains one or more actual parameters separated by commas. An actual parameter may be a numeric expression or a string expression.

Description

Actual parameters may be used to pass only single values to a function, usually to be used within the function although this is not required.

The number of actual parameters in the function call must be the same as the number of formal parameters in the function definition. The names of corresponding parameters need not be the same. Actual and formal parameters correspond according to their positions in the two lists. For instance, the third actual parameter in a function call corresponds to the third formal parameter in a DEF statement.

If the formal parameter is a simple numeric value (V) then the actual parameter can be a numeric expression resulting in a single value, or a simple or subscripted numeric variable (2*V,V,5*7,V(5)). If the variables are different types or the actual parameter is an expression, any necessary conversion is performed as described in Section 6.5, Numeric Assignment.

If the formal parameter is a simple string variable, the corresponding actual parameter must be a string expression.

Examples

To call the one-line function:

```
10  DEF FNZ(C,D)=C*(D+10)-6
```

the actual parameters are numeric variables of the same type:

```
 500 LET C=5,D=2
 510 PRINT FNZ(C,D)
>RUN
 54
```

The actual parameters might also be numeric expressions:

```
 520 PRINT FNZ(5,2)
>RUN
 54
```

To call the string function:

```
20  DEF FNG$(K$,L$)=K$+L$+K$
```

The actual parameters can be string variables:

```
 530 K$="ABC",L$="123"
 540 PRINT FNG$(K$,L$)
>RUN
ABC123ABC
```

or string expressions:

```
 550 PRINT FNG$("ABC","123")
>RUN
ABC123ABC
```

To call the function FNB returning an integer value:

```
30 DEF FNB%(A%,X2%)=A%*X2%+(A%+X2%)
```

the actual parameters can be variables:

```
 500 LET X%=4,Y%=2
 510 PRINT FNB%(X%,Y%)
>RUN
 14
```

or numeric expressions:

```
520 PRINT FNB%(4,2)
>RUN
14
```

Each of the above examples is a one-line function for which a single value is returned. The formal parameters are not affected by execution of the function. In a multi-line function, the formal parameters may be altered in the body of the function. The value of the actual parameter, however, is never affected by the change to the formal parameter.

The multi-line function below returns a string value that is the reverse of the string value input as the actual parameter:

```
10  DEF FNR$(A$)
20    REM..FNR$ RETURNS THE REVERSE OF A$
30    IF LEN(A$)<=1 THEN RETURN A$
40    RETURN FNR$(A$[2])+A$[1,1]
50  FNEND
```

To call this function, the actual parameter may be a string literal:

```
70  PRINT FNR$("ABCDE")
>RUN
EDCBA
```

The actual parameter may also be a string variable:

```
60  DIM X$[5]
70  X$="12345"
80  PRINT "FNR$ RETURNS:";FNR$(X$)
>RUN
FNR$ RETURNS:54321
```

SECTION X

FILES

For problems that require permanent data storage external to a particular program, Zilog BASIC provides a data file capability. This capability allows flexible, direct manipulation of large volumes of data stored on files.

10.1 FILE TYPES AND ATTRIBUTES

There are two types of files used in Zilog BASIC: binary files and ASCII files.

A catalog of BASIC ASCII and binary files, as well as any other non-BASIC files on the disk, can be requested with the RIO CAT command (see Commands, Section 4.3).

10.1.1 ASCII FILES

ASCII files are created either through the RIO operating system or by the BASIC interpreter itself and are treated by Zilog BASIC as terminal-like devices. They can be actual terminals. Output to them is formatted according to the rules for the PRINT statement (see Section 3.8). Input from ASCII files is analyzed according to the rules of the INPUT statement (Section 3.7).

10.1.2 BINARY FILES

Binary files are unformatted files created through the RIO Operating System or by BASIC. Data items are stored in binary files as binary words without type information. When data is read from a binary file, it is assumed to be the type of the variable into which it is being read.

10.1.3 FILE NAMES

When any file is created, whether it is ASCII or binary, it is assigned a file name by the user who creates the file. The file name may contain up to 32 alphanumeric characters, the first of which must be a letter. The file name may be fully qualified as described in the RIO manual.

10.1.4 FILE ATTRIBUTES AND STRUCTURE

A file consists logically of a contiguous string of 8-bit bytes. The file also has a name and set of attributes. BASIC files are also divided into fixed-size logical groups of bytes called logical records. This division has no effect on file access or structure except in its use with random access. The first record on the file is called record 0.

When a file is accessed, BASIC maintains a cursor or pointer into the file which specifies the next byte to be read or written. The cursor is advanced through the file as sequential input or output transfers are performed. New bytes are never inserted into the middle of a file; it is only lengthened or shortened by appending or removing bytes at the end of the file. A write to a file, performed when the file cursor is in the middle of the file, overwrites whatever information was previously there. Surrounding data are unaffected.

Files are created automatically when first accessed in a BASIC program. They are, however, only deleted by an explicit command (see Section 10.4).

The following table summarizes the statements that are used to access files.

Function	Statement(s) Used	Section
Creating and Opening files	FILE	10.2
Closing files	CLOSE, END	10.3
ASCII input from files	INPUT, LINPUT	10.7.1
Binary input from files	READ	10.7.1
ASCII output to files	PRINT	10.7.2
Binary output to files	WRITE	10.7.2
Rewinding files	RESTORE	10.7.3
Moving the file cursor	SPACE	10.6
Random access	RESTORE, INPUT, LINPUT, READ, PRINT, WRITE	10.7.3, 10.7.4 10.7.5
Shortening files	TRUNCATE	10.5
Deleting files	ERASE	10.4
Detecting the end-of-file	EOF Function	10.8

10.2 OPENING FILES: FILE STATEMENT

In order for a program to access a file, the file must be open. A buffer is required for each open file. The NEW command (Section 4.2.2) is used to allocate 512 byte blocks for buffers. The NEW command may allocate from 0 to 15 blocks. By default, NEW allocates two blocks. Each open file requires one buffer. The buffer size is determined by the record length of the file. The default record size is 128 bytes. For every file that is to be opened, an association is established between the file number used in access statements and the file name. The file number is an integer between 1 and 15.

The linkage between file name and file number is accomplished by the FILE statement. FILE causes a file number to be assigned to a file name. If another file was associated with the file number, that file is closed.

Format

The formats for FILE are

```
FILE #filenumber;name options string

FILE #filenumber;name options string,return variable
```

The filenumber is a number between 1 and 15. The name options string is a string expression. It includes a filename (see Section 10.1.3) followed by optional parameters, all delimited by semicolons. The options (ACC, RL, REC) are described in Table 10-1. The second form of FILE includes a numeric return variable. It is used to return the status of the FILE statement's execution (see Table 10-2).

Description

The file name is associated with the file number. The file is then created (if necessary) and opened. Subsequent accesses to the specified file number affect the named file. The FILE statement allows many options to be listed in the name options string. They and their effects are described in Table 10-1.

TABLE 10-1
FILE OPTIONS PARAMETERS

Parameter	Effect
REC=constant	Set the logical record size of the file to the specified constant which must be between 1 and 32767 inclusive. (Default: REC=128)
RL=constant	For a new file only. Set the physical record length to the specified constant. See the RIO manual for a discussion of acceptable values and their implications. (Default: RL=128)
ACC=option	Option must be one of: IN — File must already exist OUT — FILE may exist but is emptied of data NEW — FILE must not already exist UPD — Pointer placed at beginning of file (default) An error message is issued if the restrictions on the file's existence are not met. If there are no restrictions, the file is created if nonexistent.

If the numeric return variable is not present, any errors encountered during processing are handled in the normal manner by printing a message on the user's terminal (or trapping to a selected line if an appropriate TRAP statement, Section XII, has been executed). If the variable is present, no errors are generated and the variable is set to a value indicating the success or cause of failure of the FILE statement. These values are given in Table 10-2.

TABLE 10-2
RETURN STATUS OF FILE EXECUTION

Return Value	Status
0	Everything is O.K.
1	The file already exists and "ACC=NEW" was specified
2	The file does not exist and "ACC=IN" was specified
3	The file name was illegal
4	An option was encountered which was illegal
5	The record size specified by "RL=" or "REC=" was illegal
6	There was insufficient memory for buffers

Examples

```
FILE #1;"AFILE;ACC=IN"

FILE #2;"BFILE;ACC=NEW",C

FILE #3;"CFILE"

FILE #12;"SCRATCH;ACC=NEW;RL=1024"
```

AFILE is opened and must exist. BFILE is opened, must not have existed previously and C contains the FILE execution return status. CFILE is opened with the pointer at the beginning of the file. If CFILE does not exist, it is created. In the fourth example, a file, SCRATCH, is created (it must not already exist) with a physical record size of 1024 bytes.

```
FILE #K+1;"XXX.BP;ACC=IN;REC=1",R%
```

In this example, the file XXXX.BP (which must already exist) is opened. The logical record (used for random access) is set to 1. This allows random access to particular single bytes in the file. The integer variable R% is set to indicate the status of the open.

10.3 CLOSING FILES: CLOSE STATEMENT

All files are closed automatically upon program termination. A file may be closed during program execution with the CLOSE statement. This should be done wherever practical to release buffer space for other files.

The CLOSE statement breaks the name-file number linkage established by the file statement and releases resources that were needed to access the file.

Format

```
CLOSE   #file number

CLOSE   #file number,#file number,...

CLOSE
```

The specified file numbers are closed.

Description

If a file number was not associated with a file by a previous FILE statement, no action is taken and no error is issued.

In the third form, all files are closed.

Examples

```
 105  CLOSE #4
1210  CLOSE #1,#2,#3,#4,#5
1401  CLOSE #2*K+6
7090  CLOSE
```

10.4 DELETING FILES: ERASE STATEMENT

A file can be deleted from the system with an ERASE statement.

Format

The formats for the ERASE statement are

```
ERASE file name

ERASE file name,return variable
```

The file name is a string expression. The return variable will contain a result following execution of the ERASE statement.

NOTE: The complete filename must be specified as it would appear in a RIO CAT list. Thus, if a BASIC program file is to be deleted, ".BP" must be appended.

Description

The file specified in the statement is deleted and is not recoverable.

The numeric variable in the statement returns a result or status of the ERASE operation:

0	successful delete
1	file is being accessed and cannot be deleted
2	user is not permitted to delete this file
3	there is no such file

Examples

```
 10  ERASE "BFILE",N
 20  PRINT N
>RUN
 0
```

An ERASE statement is used to delete BFILE. The result of deleting BFILE is printed. Since it was a successful deletion, the result is zero.

```
10  DIM A$(96)
20  INPUT "PROGRAM TO DELETE",A$
30  ERASE A$+".BP"
```

The above example deletes a BASIC program file, appending the ".BP" suffix to the programname.

10.5 TRUNCATE STATEMENT

The TRUNCATE statement is used to specify that bytes beyond the current file cursor of a specified file are to be removed from the file and the disk space they occupied freed for reuse.

Format

```
TRUNCATE #filenumber
```

Description

Any bytes beyond the current cursor position are removed from the file. The byte before the cursor becomes the last byte in the file.

Example

```
709 TRUNCATE #2
```

10.6 SPACE STATEMENT

The SPACE statement is used to alter the position of the cursor in a file. The cursor can be moved forward or backwards (toward the beginning of the file) by a specified number of bytes or until a specified character is encountered.

Format

```
SPACE #filenumber,movecount

SPACE #filenumber;movecount,delim-string

SPACE #filenumber;movecount,delim-string,
      return variable
```

Description

In the first form, the cursor for the file specified by filenumber is moved by the number of bytes specified by movecount. If movecount is positive, the cursor is moved toward the end of the file; if negative, toward the beginning of the file; and if zero, the statement has no effect. The movement of the cursor proceeds until the cursor has been moved by the given number or until the end of the file (or beginning of the file if movecount is negative) is encountered. The EOF function indicates whether or not the end-of-file was encountered.

In the second form, the cursor movement proceeds toward the end of the file in a manner described above with the addition that the movement stops if the byte specified by delim-string is encountered. Delim-string must be a string expression of length one. The cursor movement stops after movecount bytes have been passed, regardless of whether the delim-string was encountered.

The third form operates in the same manner as the second form, toward the end of the file. The return variable, which must be a simple or subscripted variable, is given the value of the number of bytes that the cursor was actually moved.

Note: SPACEing toward the beginning of the file may not be done using the second and third forms described above.

Examples

```
 10 DIM A$(96)
 20 INPUT "FILE:",A$
 30 FILE #1;A$+";ACC=IN"
 40 C%=0
 50 SPACE #1;32767,"<13>"
 60 IF ~EOF(1) THEN DO
 70     C%=C%+1
 80     GOTO 50
 90 DOEND
100 PRINT "<13><10>NUMBER OF LINES IN ";A$;" IS ";C%
>ASA-COUNT.LINES
>RUN
FILE: COUNT.LINES.BP
NUMBER OF LINES IN COUNT.LINES.BP IS 10
```

SPACE is used in the above example to count the number of lines in file COUNT.LINES.BP.

```
 10 FILE #1;"TESTFILE"
 20 DIM A$(30)
 30 PRINT #1;"0123456789ABCDEF"
 40 RESTORE #1    \ MOVE POINTER TO BEGINNING
 50 LINPUT #1;A$ \ READ AND PRINT THE LINPUT STRING
 60 PRINT A$
 70 RESTORE #1    \MOVE POINTER TO BEGINNING
 80 SPACE #1;5    \MOVE POINTER FORWARD 5 BYTES
 90 LET A$=" "
100 READ #1;A$(1,6)    \READ AND PRINT THE NEXT 6 BYTES
110 PRINT A$
120 SPACE #1;50,"D",C  \ MOVE CURSOR PAST THE NEXT "D"
130 LINPUT #1;A$
140 PRINT A$,C     \PRINT STRING AND RETURN VARIABLE
150 SPACE #1;-10    \MOVE CURSOR BACK 10 BYTES(FROM END)
155 REM RECALL THAT THE CR COUNTS AS 1 OF THE 10 CHARACTERS
160 LINPUT #1;A$
170 PRINT A$
180 CLOSE #1
190 ERASE "TESTFILE"
>RUN
0123456789ABCDEF
56789A
EF             3
789ABCDEF
```

10.7 FILE ACCESS

There are two types of access to a file: sequential and random. For sequential access, the items read or written immediately follow the previous access. A pointer associated with each open file always points to the next item in the file to be accessed.

For random access, a particular record is specified at which the access begins. In this case, the pointer is first moved to the beginning of this record.

In Zilog BASIC files, random and sequential access can be combined in the same file. It is possible, for instance, to position the pointer to the beginning of a record with an appropriate statement, and then to access the file sequentially from that point.

Files may be accessed randomly only if they are disk files under ZDOS/RIO. Otherwise sequential access must be used.

10.7.1 SEQUENTIAL FILE READ, INPUT AND LINPUT

The Sequential File READ, INPUT and LINPUT statements read items from a file specified by file number into numeric or string variables. The first item read is the item following the current position of the pointer, that is, immediately following the last item accessed. As with sequential PRINT (Section 10.7.2), record boundaries are ignored and the list of read items can start in the middle of one record and end in the middle of another.

Format

The format of Sequential File Reads are:

```
INPUT #file number;read item list

LINPUT #file number;string variable

READ #file number;read item list
```

The read item list is a series of variables separated by commas. The rules governing this list are the same as those described for the READ statement in Section 3.9.

Description

The Sequential File Input statement reads ASCII data from a file in much the same manner that the INPUT statement reads ASCII data from the terminal.

Each item in the specified file is read into a variable in the read item list, the first item into the first variable, the second into the second, and so forth.

The destination for a string value must be a string variable; the destination for a numeric value must be a numeric variable. Otherwise, an error occurs. If the numeric value is not the same data type as the variable, conversion is performed as described in Section 6.5, Table 6-2.

LINPUT # reads into a string variable up to a carriage return character. Items read with an INPUT # statement must be separated by commas.

The sequential file READ statement transfers binary data from the specified file to variables in the read item list. The number of bytes transferred is exactly that required to fill each input variable. No conversion or type checking is performed. The number of bytes transferred is shown in Table 10-3.

When an EOF condition occurs, the variables remain unchanged and the EOF function (Section 10.8.1) becomes true.

Reading Strings

When a string is read from a binary file, the number of characters read depends on the form of the variable. For instance, if A$ is a simple string variable:

READ#1;A$	reads the physical length of A$
READ#1;A$(I)	reads the physical length of the substring starting at I
READ #1;A$(I,J)	reads J-I+1 characters into the substring starting at I

When INPUTting strings, if the string variable is not large enough to hold the entire item, the extra characters are discarded.

TABLE 10-3

Data Type	Number of Bytes on File
INTEGER	2
REAL (Binary)	4
REAL (BCD)	8
STRING	actual number of characters supplied: logical* size of string if unsubscripted or specified size if subscripted

*physical size if READ

10.7.2 SEQUENTIAL FILE PRINT AND WRITE

The Sequential File PRINT and WRITE statements write data items on a file, starting at the current position of the pointer. The items may be numeric or string expressions.

Format

The forms of a Sequential File Print statement are:

```
PRINT #file number;print list

PRINT #file number

WRITE #file number;print list
```

The print list is a series of numeric and/or string expressions. The rules for specifying the list are the same as those described for the PRINT statement in Section 3.8.

If the print list is omitted, the statement is ignored unless the file is an ASCII file, in which case a line is skipped as in a PRINT statement (i.e., a return is written to the file).

Description

Each item in the print list is written on the file in the order it appears in the Sequential File Print statement. The items are written starting at the position where the pointer currently appears, overlaying whatever data may be in that position in the file. Record boundaries are ignored; a sequential Print can start in the middle of one record and end in the middle of another.

The data written by PRINT are exactly those ASCII characters that would appear on the terminal if an equivalent PRINT statement had been executed. It should be noted that a File INPUT statement cannot read back the equivalent data produced by a PRINT statement unless commas are interspersed between print items and quote marks surround string items as needed.

A sequential FILE WRITE statement transfers binary data from items in the print list to the specified file. The amount of data written (in bytes) depends on the size of the data item (see Table 10-3). No conversion is performed and it is not possible to determine the structure or type of data on the file using information on the file alone. Data so written are read using the file READ statement.

10.7.3 FILE RESTORE STATEMENT

The File RESTORE statement repositions the file pointer to the start of the file. The statement can be used for any file. The random file RESTORE statement positions the file pointer to the beginning of any particular record.

Format

```
RESTORE #file number

RESTORE #file number,record number
```

The file number identifies a file that is currently open.

In the second form, record number specifies the record at which the file pointer is to point.

Description

When File RESTORE is executed, the file pointer is set to point to the beginning of the first record in the file.

When a random File RESTORE is executed, the pointer is set to point to the beginning of the specified record.

Example

```
 5  FILE #1;"AFILE"
10  FILE #2;"BFILE"
20  PRINT #1;123.4
30  WRITE #2;567.8
40  RESTORE #2
50  RESTORE #1
60  INPUT #1;C
70  READ #2;D
80  PRINT C,D
>RUN
 123.4          567.8
```

When the File RESTORE statements are executed, the pointer in file number 2 is moved back to the start of that file. Then the pointer in file number 1 is moved to the start of that file. File number 1 is accessed as an ASCII file using INPUT and PRINT statements. File number 2 is accessed as a binary file using READ and WRITE statements.

10.7.4 RANDOM FILE READ, INPUT, AND LINPUT

The Random File READ, INPUT, and LINPUT statements read data values starting at a specified record of a specified file and assign them to variables.

Format

The forms of the Random File READ, INPUT, and LINPUT statements are:

READ #file number,record number;read item list

INPUT #file number,record number;read item list

LINPUT #file number,record number;string variable

The file number and record number are integer expressions. The read item list is the same form as in a READ statement (section 3.9).

Description

Data values are read from the specified record and assigned to the variables in the item list. If a record number is specified outside the range of the file, an end-of-file condition occurs.

To move the file pointer to the beginning of a specified record, but not read any data, use the Random File RESTORE statement (section 10.7.3).

10.7.5 RANDOM FILE PRINT AND WRITE

The Random File PRINT and WRITE statements write a list of data items onto the specified file. Printing begins at a particular record specified in the PRINT statement. Data that precedes or follows the specified area is not changed.

Format

The forms of a Random File PRINT and WRITE are:

```
PRINT #file number,record number;print list
WRITE #file number,record number;print list
```

Both the file number and record number are integer expressions. The print list has the same format as a Sequential File PRINT. It is not, however, optional.

Description

The Random File PRINT and WRITE statements position the pointer at the beginning of the specified record and then write the contents of the print list.

The first record of the file is record number 0.

Sequential and Random PRINT statements can be used to write on the same file as can Sequential and Random WRITE statements. A sequential PRINT following a random PRINT will write its data items immediately following the previous items.

Example Program Using Random Files

The following program uses the random access feature of BASIC files. Two files are created and written into. Lines of each file are then swapped into the other file, causing a "jumbled" file. The original files are then recreated and compared. A "successful" message is output if the files are equal. The two files are then closed and deleted.

```
 10 FILE #1;"RANDOM.TEST.1"
 20 FILE #2;"RANDOM.TEST.2"
 30 FOR I=1 TO 100
 40   PRINT #1;I,I,I
 60 NEXT I
 70 DIM A$(128),B$(128)
 72 FOR I=0 TO 24
 74   READ #1,I;A$
 76   WRITE #2;A$
 78 NEXT I
 80 FOR I=1 TO 2
 90   FOR J=0 TO 10
100     READ #1,10-J;A$
110     READ #1,10+J;B$
120     WRITE #1,10-J;B$
130     WRITE #1,10+J;A$
140   NEXT J
160 NEXT I
165 RESTORE #1
170 FILE #1;"RANDOM.TEST.1"
180 FILE #2;"RANDOM.TEST.2"
190 FOR J=20 TO 0 STEP -1
200   READ #1,J;A$
210   READ #2,J;B$
220   IF A$<>B$ THEN  400
230 NEXT J
240 PRINT "TEST SUCCESSFUL"
250 GOTO  410
400 PRINT "TEST FAILED"
410 CLOSE #1
420 CLOSE #2
430 ERASE "RANDOM.TEST.1"
440 ERASE "RANDOM.TEST.2"
```

10.8 FILE RELATED FUNCTIONS

The following function may be used in conjunction with files.

10.8.1 EOF Function

EOF (x)

EOF indicates whether an end-of-file condition exists with file number x. The function returns a "1" if EOF=true and a "0" if EOF=false. The file number is a number between 1 and 15.

SECTION XI

FORMATTED OUTPUT

The USING clause can be included in the print list of a PRINT statement to control the format of numbers output. The number of digits to appear to the left and right of the decimal point can be specified. The exact position of the sign can be controlled. Asterisk-fill or a floating dollar sign can be specified.

Formatting is controlled by specifying a prototype "picture" of what the number output should look like. The prototype is specified as a string where each character in the string corresponds to a single character in the final output. Characters in the format string must be from a set of legal characters. In this set are characters to specify the number of digits to be printed, the position of sign and decimal point, the nature and content of any "fill" characters, and the position of commas.

No formatting facility is supplied for strings as sufficient string handling functions are available in BASIC to make such formatting redundant. Also, no provision is made in general to mix alphabetic data with numbers as it is possible to do this using combinations of existing BASIC constructs.

11.1 PRINT STATEMENT WITH FORMAT CONTROL

The PRINT statement with one or more USING clauses, allows the user to output a list of items according to a customized format.

Format

The form of PRINT with USING is:

    PRINT print using list

The print using list is a list of expressions and functions from which items are printed. The clause "USING string expression" may be interspersed with other print items. In other respects, the print using list is like a print list (see PRINT statement, Section 3.8).

The string expression following "USING" evaluates to a format string which controls the output of subsequent print items.

Description

A format string describes the form in which items in the print using list are to be printed. The full description of format strings is contained under Format Strings, Section 11.2.

When a USING clause is encountered, the format string is evaluated. Several formats, each for a single number, may be included in the format string if they are separated by one or more blanks. One blank is printed following each number except the last. Any strings or calls to the TAB function (Section 3.8.1) in the print list are printed directly without any format control intervention.

The USING clause remains in control until the last format string has been used to format a number. Then, subsequent print items are printed using the normal formatting rules. A USING clause also ceases to affect PRINT output when the last print item of the current statement is output or another USING clause is encountered. While output is under USING format control, any commas separating print items serve as separators only. They do not affect spacing as they normally do.

If the number to be printed fails to fit within the format supplied, the entire format field is filled with asterisks and a warning message is issued.

11.2 FORMAT STRINGS

The following are the legal format characters and their function:

Char	Prints	Comments
#	digit or blank or "-"	blank if in leading or trailing zero position; "-" sign counts as one if number is negative
D	digit	prints as zero if in leading or trailing zero position
+	"+" or "-"	sign
-	"-" or blank	blank if number is non-negative
$	"$" or digit	prints " " if in leading or trailing zero position except for leftmost $ in leading or trailing digit position which prints "$" (acts as floating $). Otherwise, digits are substituted for the format "$"s.
*	"*" or digit	prints "*" in leading or trailing zero position; prints as a digit otherwise.
P	"."	also defines the decimal point position of the number
.	"." or blank	prints "." if number is non-integer, blank otherwise; the number is considered non-integer only if there are digits printed to the right of the decimal point. Also defines the decimal point position of the number
	"," or blank	prints blank if only non-digits were printed to the left, otherwise prints ","
^^^^	Esdd	where s is "+" or "-" and dd are digits. Causes number to be printed in exponential form.
^^^^^	Esddd	Same as "^^^^" but there is room for a three digit exponent

Notes:
1) Only one occurrence of "P" and "." is allowed.
2) "," may not appear to the right of a "P" or "," or "^".
3) ^^^^ and ^^^^^ can appear only at the right of a format string.
4) If "+" or "-" appear, "#" will never be used for a sign.
5) If no "+" or "-" appear, then one "#" may be used as a sign.
6) If no "P" or "." appear, the decimal unit position is assumed to be the right-most "#" or "D".

```
                    Field
Format String  Size      Number            Output
-------------  -----  ---------------  ------------------------------------
###.##           6       123                |123   |
                         123.5              |123.5 |
                         123.526            |123.53|
                         -12                |-12   |

###.DD           6       123                |123.00|
                         5.6                |  5.60|

##DD.D           6       124                | 123.0|
                         4                  |  04.0|

+####            5       3                  |+   3|
                         1234               |+1234|
                         -2                 |-   2|
                         23.6               |+  24|

-####            5       4                  |    4|
                         -71                |-  71|
                         0                  |    0|

##-              3       0                  | 0 |
                         -4                 | 4-|
                         -23                |23-|

##.DD+           6       0                  | 0.00+|
                         -2.4               | 2.40-|
                         12.34              |12.34+|

###P##           6       123                |123.  |

$###.DD          7       4.2                |$  4.20|
                         123.45             |$123.45|
                         .5                 |$   .50|

$$$$.DD          7       1.267              |  $1.27|
                         234                |$234.00|
                         9876.5             |9876.50|

****.DD          7       4.2                |***4.20|
                         123.45             |*123.45|
                         .5                 |****.50|

##,###,###.D     12      1234               |     1,234.0|
                         1000000            | 1,000,000.0|

#.###^^^^        9       1                  |1    E+00|
                         234.5              |2.345E+02|

##.DD^^^^^       10      0                  |  .00E+000
                         1                  |10.00E-001|
                         3456               |34.56E+002|
                         -5E+123            |-5.00E+123|
```

```
###,##,,DDD.DD$  15     1                  |        001.00$|
                        12345              |    12,,345.00$|
                        .003               |        000.003|

DD##,##+#--++#   13     1                  |00   ,  +   ++1|
                        -34                |00   ,  -3----4|
```

SECTION XII

TRAPPING

Trapping is a handy tool for recognizing abnormal conditions during the execution of BASIC programs. The conditions for trapping are:

ESCape	entered from console
ERR	a runtime error
EOF	an end-of-file error
KEYS	depression of a console key other than ESC
EXT	a user defined external condition

12.1 TRAP STATEMENT

The TRAP statement is used to enable and disable trap conditions. When a condition is enabled, a line number is specified to which control will transfer when the trapped condition occurs.

Format

```
TRAP condition TO label
TRAP condition OFF
TRAP ESC
```

Condition is one of ESC, ERR, EOF, KEYS, or EXT. Label is a statement label in the current program.

In the first form, a trap for the specified condition is established so that control will go to the specified label if the condition occurs. If a trap for that condition was already in effect, the destination label is changed to the one given in the new TRAP statement.

The second form disables any trap established for the specified condition.

The third form establishes no trap, but instead disables the normal function of the ESCape key (and disables the ESC trap if it was active). ESC will then not terminate the execution of a program. The ESC function can then be used to check if the ESC key has been depressed. The user is cautioned against using this feature since there is no way to exit an infinite loop (other than restarting the system) once "TRAP ESC" has been executed. Execution of TRAP ESC OFF will cause the ESCape key to function normally again.

Description

The TRAP feature of ZILOG BASIC allows the user program to handle five exceptional conditions. When one of these conditions occurs and the corresponding trap has been enabled by use of the TRAP statement, control branches to a specified line number instead of the next sequential statement as is the usual case. The line number of the last statement executed before the trap occurred is available to the programmer by use of the TRP function. Additional information about the condition that caused the trap is available through other functions described below.

The five conditions that can be trapped are:

ESC	Escape (from the console keyboard)
ERR	a runtime error (optionally including warnings)
EOF	the End-of-File error
KEYS	depression of console keys other than ESC
EXT	a user defined external condition

When a program is run, all traps are disabled. At any point in the program, a trap for any of the above conditions can be established by use of the TRAP statement. In the TRAP statement, a line number is specified to which control will transfer should the selected condition occur.

Traps are only initiated between the completion of one statement and the execution of the next. Occurrence of the ERR or EOF condition causes the termination of the execution of the current statement. If a trap is enabled for one of these conditions, the trap will then be initiated.

When a trap is initiated the following events occur. First the trap is disabled. Another "TRAP condition TO label" statement must be executed for another trap with the same condition to occur. (If this was an ESC trap, the ESC key will still not interrupt the program even though the trap is disabled. The only way to enable the function of the ESC key is to execute a "TRAP ESC OFF" statement). The line number of the last line executed is saved and can be determined by the user program by use of the TRP function. Control is then transferred to the line specified for processing whichever condition caused the trap.

A discussion of each of the trap conditions and their characteristics and use follows.

12.1.1 KEYS

Keys pressed on the console terminal when the program is not waiting for input can be read by referencing the KEYS$ function (regardless of whether the KEYS trap is enabled). When a key is pressed, it is held in a buffer until the buffer is read using the KEYS$ (Section 12.2.3) function. Only the most recently pressed key is saved in the buffer.

If the KEYS trap is enabled and the buffer contains a character, then the KEYS trap is initiated as described above. Note that the KEYS function must be referenced before the trap is reenabled. If not, the trap would be re-initiated immediately as the buffer would still contain a character.

12.1.2 EXT

This trap is initiated when an external (to BASIC) program sets a flag within BASIC. The protocol for how an external user program does this is described in Appendix J.

When the trap is initiated, the flag is cleared. The BASIC user program can use the CALL statement to fulfill any necessary interface requirements.

12.1.3 ESC

Both the "TRAP ESC" and the "TRAP ESC TO label" statement disable the ESCape key; however, only the latter establishes a trap. The ESC trap works as the KEYS trap. After the execution of each statement, BASIC checks to see if the ESC key has been pressed. The function ESC (Section 12.2.2) has the value zero if the ESC key has not been pressed, and the value one if it has been pressed. Similar to KEYS$, ESC returns the value one at most once for each time the ESC key is pressed.

Normally, depression of the ESC key stops the BASIC program. When a "TRAP ESC" statement is executed, the ESC key no longer does this. If a "TRAP ESC TO label" statement is executed, a trap is initiated when the ESC key is pressed in the manner described above. Again, similar to KEYS, the ESC function must be referenced before the ESC trap is reenabled. Otherwise, the trap would be reinitiated immediately since the buffer would still contain the ESCape.

12.1.4 ERR

Run-time program execution errors can be trapped by use of the ERR trap. When the trap is invoked, the error number of the error that caused the trap is returned by the ERR function (Section 12.2.4). The line number of the line that caused the error is returned by the TRP function. Any error after the trap is initiated (but before another "TRAP ERR TO label" statement is executed) will cause termination of the program with a normal error message. The effect of error traps on the run-time environment (of multi-line function calls) is significant and is described below in "Environments and Traps".

12.1.5 EOF

Normally, when a READ, INPUT, or LINPUT statement encounters an End-of-File condition, the statement's variable list is not processed and the EOF(n) function (Section 10.8.1) becomes true (non-zero value) for the particular file number n. A subsequent READ, INPUT, or LINPUT statement for the same file number causes an error. When a "TRAP EOF TO label" statement has established an EOF trap, the execution of the first file input statement that detects the EOF condition (the time the EOF(n) function first becomes non-zero) will cause a trap to the specified label.

12.1.6 Environments and Traps

Each time a multi-line function is invoked, a new "environment" is created. Each environment can have its own "local" traps. They are local in the sense that traps can be established or deactivated within the environment and they do not affect traps established in other environments. Traps can be established, triggered and processed, or disabled within an environment, without affecting other environments.

An occurrence of an ERR or EOF condition when a trap does not exist in the current environment but is established in another active environment, does affect execution. In this case, environments are discarded in the reverse order that they were created until an environment is found that has an active trap for the ERR or EOF that occurred. In this environment, the trap is invoked. The line number returned by the TRP function identifies the statement that invoked the multi-line function that ultimately caused the ERR or EOF condition.

Entry to a new environment when a previous one had a TRAP ESC trap (or ESC disabled) affects execution similarly. In this case, the function of the ESC key is still disabled and a record is kept if it is depressed. When control returns to an environment with an active ESC trap, the trap is then invoked. The KEYS condition is similarly preserved and the trap invoked if control returns to an environment with a KEYS trap enabled.

12.2 TRAP RELATED FUNCTIONS

12.2.1 TRP — Returns an integer value representing the label of the last line executed before a trap occurred.

12.2.2 ESC — Has the integer value one if the ESC key was pressed, and zero otherwise. An interlock is associated with ESC so that it returns the value one at most once for each time the ESC key is pressed. Used in conjunction with the TRAP ESC statement.

12.2.3 KEYS$ — Returns a string containing the character (other than ESC) typed during program execution (but not during an INPUT statement). If no keys were pressed, returns the null string. An interlock is associated with KEYS$ so that characters typed can be returned at most once.

12.2.4 ERR — Returns an integer whose value is the error number of the error that caused the last ERR trap. Used in conjunction with the TRAP ERR statement.

SECTION XIII

SEGMENTATION

Because the maximum size of a Zilog BASIC program is necessarily limited by memory resources, Zilog BASIC provides language facilities for segmenting programs into units that can call each other. Each unit must be saved on the disk; from there it may be called by the currently executing program into the user's work area.

The CHAIN statement is used for interprogram transfer. The COM statement allows variables to be used in common by several programs.

13.1 CHAIN STATEMENT

The CHAIN statement terminates the current program and begins execution of another program.

Format

The format of CHAIN is:

```
CHAIN string expression
```

The string expression, when evaluated, is the name of a Zilog BASIC program that is on the user's disk. This may be a fully qualified file name (see Section X, Files). Execution begins at the first executable statement in the called program.

Description

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

Only variables declared in a COM statement in both programs are saved during a CHAIN operation. All variables and arrays of the current program that were not declared in COM are lost when the new program begins execution.

All files opened in the current program remain open.

Examples

```
>10  PRINT "HI FRED"
>20  CHAIN "FRED"
>ASA-MA
>NEW
>10 PRINT "HI MA"
>ASA-FRED
>NEW
>XEQ-MA
HI FRED

HI MA
```

The main program, MA, calls program FRED with a CHAIN statement in line 20. Execution of FRED begins and execution terminates with the last line of FRED. None of the variable values from MA are saved following the execution of CHAIN "FRED".

13.2 COM STATEMENT

The COM statement is used to pass data values between program segments. Variables specified in a COM statement are placed in a common area so that values assigned to these variables in one program will be retained when transferring to another program with CHAIN.

COM statements must precede all other statements in a program except for REM statements. All dimensioning of variables is done within the COM statement, and any variables that appear in a COM statement must not simultaneously appear in a DIM statement in the same program.

Format

The format of the COM statement is:

    COM com item list

The com item list consists of a list of variable declarations. Simple variables are indicated by the variable name; arrays are indicated by the array name and a bounds indicator. The bounds indicator is equivalent to the dimension specification used in a DIM statement.

The type of items in the com item list is assumed to be real unless the variable name contains a "$" suffix to indicate a string variable, or a "%" suffix to indicate an integer.

Arrays and simple variables declared in a COM statement are initialized to zero. Strings declared in a COM statement are set to null. Such common variables must not also be declared in a DIM statement.

COM item lists need NOT be identical in variable name ORDER from program to program. However, variables that are common between programs must be identically named and dimensioned.

SECTION XIV

COMMUNICATION WITH NON-BASIC PROGRAMS

A Zilog BASIC user can access a Zilog PLZ system language procedure or an assembly language subprogram from a BASIC program with the CALL statement.

14.1 CALL STATEMENT

The CALL statement is used to access procedures written in assembly language or PLZ. Values of BASIC variables and expressions can be passed to the user procedure and the user procedure can pass values (real, integer, and string) back to BASIC to be assigned to BASIC variables.

Format

```
CALL procedure-name[,BASIC-procedure-p-list]
        [;procedure-BASIC-p-list]
```

Procedure-name is a string expression whose value is the name of the user procedure to be called. BASIC-procedure-p-list is a list of expressions (separated by commas if there are more than one) whose values are to be passed to the user procedure. Procedure-BASIC-p-list is a list of simple or subscripted variables (separated by commas if there are more than one) to receive values from the user procedure. Each list is optional.

Description

The expressions (if any) in the BASIC-procedure-p-list are evaluated and their values are passed to the named procedure. The techniques for establishing a user procedure are described in Appendix F. When the user procedure returns to BASIC, it can supply values which are assigned to the variables in the procedure-BASIC-p-list.

Examples

```
 10  CALL "INITIALIZE.TESTER"

107  CALL "AND",X%,7;Y%

325  CALL "TIME";A$(2,10)

500  CALL "SET.TIME","3:27"

650  CALL B$+"X",C$;D$
```

APPENDIX A

ASCII CHARACTER SET

Graphic	Decimal Value	Comments
	0	Null
	1	Start of heading
	2	Start of text
	3	End of text
	4	End of transmission
	5	Enquiry
	6	Acknowledge
	7	Bell
	8	Backspace
	9	Horizontal tabulation
	10	Line feed
	11	Vertical tabulation
	12	Form feed
	13	Carriage return
	14	Shift out
	15	Shift in
	16	Data link escape
	17	Device control 1
	18	Device control 2
	19	Device control 3
	20	Device control 4
	21	Negative acknowledge
	22	Synchronous idle
	23	End of transmission block
	24	Cancel
	25	End of medium
	26	Substitute
	27	Escape
	28	File separator
	29	Group separator
	30	Record separator
	31	Unit separator
	32	Space
	33	Exclamation point
	34	Quotation mark

Graphic	Decimal Value	Comments
#	35	Number sign
$	36	Dollar sign
%	37	Percent sign
&	38	Ampersand
'	39	Apostrophe
(	40	Opening parenthesis
)	41	Closing parenthesis
*	42	Asterisk
+	43	Plus
,	44	Comma
-	45	Hyphen (minus)
.	46	Period (decimal point)
/	47	Slant
0	48	Zero
1	49	One
2	50	Two
3	51	Three
4	52	Four
5	53	Five
6	54	Six
7	55	Seven
8	56	Eight
9	57	Nine
:	58	Colon
;	59	Semicolon
<	60	Less than
=	61	Equals
>	62	Greater than
?	63	Question mark
@	64	Commercial at
A	65	Uppercase A
B	66	Uppercase B
C	67	Uppercase C
D	68	Uppercase D
E	69	Uppercase E
F	70	Uppercase F
G	71	Uppercase G
H	72	Uppercase H
I	73	Uppercase I
J	74	Uppercase J
K	75	Uppercase K
L	76	Uppercase L
M	77	Uppercase M
N	78	Uppercase N
O	79	Uppercase O
P	80	Uppercase P
Q	81	Uppercase Q
R	82	Uppercase R

Graphic	Decimal Value	Comments
S	83	Uppercase S
T	84	Uppercase T
U	85	Uppercase U
V	86	Uppercase V
W	87	Uppercase W
X	88	Uppercase X
Y	89	Uppercase Y
Z	90	Uppercase Z
[	91	Opening bracket
\	92	Reverse slant
]	93	Closing bracket
^	94	Circumflex
	95	Underscore
`	96	Grave accent
a	97	Lowercase a
b	98	Lowercase b
c	99	Lowercase c
d	100	Lowercase d
e	101	Lowercase e
f	102	Lowercase f
g	103	Lowercase g
h	104	Lowercase h
i	105	Lowercase i
j	106	Lowercase j
k	107	Lowercase k
l	108	Lowercase l
m	109	Lowercase m
n	110	Lowercase n
o	111	Lowercase o
p	112	Lowercase p
q	113	Lowercase q
r	114	Lowercase r
s	115	Lowercase s
t	116	Lowercase t
u	117	Lowercase u
v	118	Lowercase v
w	119	Lowercase w
x	120	Lowercase x
y	121	Lowercase y
z	122	Lowercase z
{	123	Opening (left) brace
|	124	Vertical line
}	125	Closing (right) brace
~	126	Tilde
	127	Delete

APPENDIX B

SUMMARY OF ZILOG BASIC STATEMENTS

This summary of Zilog BASIC statements provides the statement names in alphabetic order with a brief description and a reference to the section or sections containing a complete statement description.

Statement	Description	Reference
CALL	Calls for execution of a procedure stored in memory, optionally passing parameters to the procedure.	14.1
CHAIN	Terminates the current program and calls for execution of the BASIC program named in the CHAIN statement. Variables are shared between programs if named in COM statements.	13.1
CLOSE	Close all specified files, freeing access resources and breaking the association between file numbers and files.	10.3
COM	Declares the specified variables to be common to more than one program. Effective when one program calls another with CHAIN.	13.2
DATA	Provides data to be read by READ statements.	3.9, 8.11
DEF	Introduces a function definition.	9.1, 9.2
DIM	Reserves storage for arrays and sets the upper bounds on the number of elements. Also redimensions arrays.	7.1, 8.3
DO...DOEND	Used only after IF...THEN or ELSE, they enclose statements to be executed when an IF or ELSE condition is satisfied. (See IF...THEN).	3.6
ELSE	Used only in conjunction with IF... THEN, it introduces a statement to be executed when the IF condition is false. (See IF...THEN).	3.6

Statement	Description	Reference
END	Terminates execution of the current program; may be omitted since last line of program provides an implicit END.	3.2
ERASE	Deletes a specified file from the system.	10.4
FILE	Assigns a file name to a file number and creates and opens the named file. Closes any file previously associated with the specified number.	10.2
FNEND	Terminates a multi-line function definition.	9.2
FOR...NEXT	Allows repetition of a group of statements between FOR and NEXT. The number of repetitions is determined by the initial and final values of a FOR variable and by an optional step specification.	3.3
GOTO	Transfers control to a specified statement label.	3.4
GOSUB	Causes execution of a subroutine beginning at a specified statement label. Following a RETURN statement in the subroutine, control returns to the statement following GOSUB.	3.5
IF...THEN	Evaluates a conditional expression and specifies action to be taken if condition is true. If the conditional expression is a numeric expression it is considered true if its value is nonzero, false if its value is zero. The action may be transferred to a statement label, a single executable statement, or a DO...DOEND group.	3.6
INPUT	Requests user to enter one or more variables by printing a "?" and accepts string or numeric data from the terminal.	3.7, 8.9, 6.6, 7.2

Statement	Description	Reference
INPUT #	Accepts string or numeric data from a file as input, similar to a terminal input.	10.7.1, 10.7.4
LET	Introduces assignment statement that assigns one or more values to a variable or array element. The word LET may be omitted.	3.1, 6.5,
LINPUT	Requests a line of input from the terminal, all of which is assigned to a single string variable.	8.9
LINPUT #	Accepts data from a file as input to a string variable. Data up to a return is read.	10.7.1, 10.7.4
NEXT	Terminates a loop introduced by a FOR statement. Specifies a variable that must match the FOR variable.	3.3
ON...GOSUB	Multi-branch GOSUB executes one of a list of subroutines depending on the value of an integer expression.	3.5
ON...GOTO	Multi-branch GOTO transfers control to one of a list of statement labels depending on the value of an integer expression.	3.4
ON...RESTORE	Multi-case RESTORE sets the data pointer to a label containing a DATA statement, based on the value of an integer expression.	3.9
PRINT	Prints the contents of a list of numeric or string expressions on the list device.	3.8, 8.10, 7.3
PRINT #	Outputs the contents of a list of numeric or string variables to the specified file in ASCII.	10.7.2, 10.7.5
PRINT USING	Prints the contents of a list of numeric or string variables with format controlled by format specifications included in the PRINT USING statement.	11.1

Statement	Description	Reference
RANDOMIZE	Selects a seed for the pseudo random number generator function RND. RND normally produces the same sequence each time a program is run.	3.11
READ	Assigns constants and string literals from one or more DATA statements to the variables specified in READ. Treats contents of all DATA statements as a single data list.	3.9, 8.11
READ #	Reads one or more items from a binary file into specified variables.	10.7.1, 10.7.4
REM	Introduces remarks and comments in the program listing.	3.10
RESTORE	Resets the data pointer to the beginning of the program or to the first DATA statement following a specified label.	3.9, 8.11
RESTORE #	Repositions the file pointer to the start of the file or to a specified record.	10.7.3
RETURN	Returns control from a GOSUB subroutine to the statement following the last GOSUB.	3.5
SPACE	Moves the cursor in a file forward or backward.	10.6
STOP	Terminates execution of the run.	3.2
SYSTEM	Controls the following system dependent functions: warning message output; warning message trap; list indentation; ASAVE line indentation; output line length; automatic carriage return.	3.12
TRAP	Establishes or disables a trap for any of the five conditions: ESCape, terminal KEYs, ERRors, EOF conditions, or EXTernal interrupt.	12.1
TRUNCATE	Sets the End-of-file of a file.	10.5
WRITE #	Outputs the unconverted binary contents of a list of numeric and string variables to a specified file.	10.7.2, 10.7.5

APPENDIX C

COMMAND SUMMARY

Each command is listed by name in alphabetical order followed by a brief description and reference to the section or sections containing a complete description of the command. All commands may be abbreviated by their first three letters.

Command	Description	Reference
APPEND	Appends a specified program (which must be in ASCII form) to the current program.	4.3.6
ASAVE	Stores a copy of the current program on the user's disk in ASCII form.	4.3.1
CLEAR	Deallocates all variable space, closes files and resets function and subroutine calls. Frees space to save a program if there is not enough available.	4.2.6
CONTINUE	Resumes program execution after an interruption by ESCape or a STOP statement.	4.1.3
DELETE	Deletes one or a range of more than one statement from current program.	4.2.3
GET	Gets the specified Zilog BASIC program from the user's library, replacing the current program.	4.3.4
LIST	Lists all or part of the current program at the terminal.	4.2.1
NEW	Deletes entire current program.	4.2.2
QUIT	Terminates the current Zilog BASIC session.	4.1.4

Command	Description	Reference
RENUMBER	Renumbers any group of statements in the current program, optionally from a new first line number with a specified increment. By default, renumbering starts at 10 with increments of 10.	4.2.4
RUN	Executes the current program.	4.1.1, 4.1.3
RSAV	Stores a copy of the current program in a file that already exists.	4.3.3
SAV	Stores a copy of the current program on the user's disk in compiled form.	4.3.2
STEP	Resumes execution, completes an outer level statement (not part of a function) and then stops. Can be used to step through a program one line at a time.	4.1.3
SIZE	Gives status of: space available (bytes); program size (bytes); variable storage size (bytes); number of 512 byte reserved blocks.	4.2.5
XEQ	Gets and runs the specified program.	4.1.2, 4.3.5

APPENDIX D

BUILT-IN FUNCTIONS

A set of built-in (or predefined) functions are included in Zilog BASIC. These functions are listed below in alphabetic order.

Note that an argument for a trigonometric function must be expressed in radians with 1 radian equal to 180/pi or 57.1958 degrees.

Name and Parameters	Meaning	Reference
ABS(x)	Absolute value of x.	6.8.1
ASC(s)	ASCII code for first character of string expressions.	8.5.2
ATN(x)	Arctangent of x; result is in radians.	6.8.2
CHR$(x)	Generates a one-character ASCII string; x is in the range 0-255.	8.5.1
COS(x)	Cosine of x; x must be expressed in radians.	6.8.3
EOF(x)	Indicates whether EOF condition has been encountered in file number x. If so, has value 1; if not, has value 0.	10.8.1
ERR	Returns an integer whose value is the error number of the error that caused the last ERR trap. Used in conjunction with the TRAP ERR statement.	12.2.4
ESC	Returns a string containing the character (other than ESC) typed during program execution (but not during an INPUT statement). If no keys were pressed, the null string is returned. An interlock is associated with KEYS$ so that characters typed can be returned at most once.	12.2.2

Name and Parameters	Meaning	Reference
EXP(x)	e^x	6.8.4
INT(x)	Largest integer less than or equal to x.	6.8.5
KEYS$	Has the integer value one if the ESC key was pressed, and zero otherwise. An interlock is associated with ESC so that it returns the value one only once for each time the ESC key is pressed. Used in conjunction with the TRAP EOF statement.	12.2.3
LEFT$(s,n)	Leftmost n characters of the string s. LEFT(A$,n) is equivalent to A$[1,n].	8.5.8
LEN(s)	Logical length of string s.	8.5.3
LOG(x)	Natural logarithm of x; x must be greater than zero.	6.8.6
POS(s1,s2)	Smallest integer representing starting position in s1 of substring identical to s2. If no such substring, then equals zero.	8.5.4
RIGHT$(s,n)	Rightmost n characters of the string s, from the nth character to the end. RIGHT(A$,n) is equivalent to A$[n].	8.5.9
RND	Pseudo-random number between 0 and 1 but not equal to 1.	6.8.7
SEG$(s,n,m)	Segment of string s from the nth through mth characters. SEG$(A$,n,m) is equivalent to A$[n,m].	8.5.10
SGN(x)	Sign function; equals 1 for x>0, 0 for x=0, and -1 for x<0.	6.8.8
SIN(x)	Sine x; x must be expressed in radians.	6.8.9

Name and Parameters	Meaning	Reference
STR$(x)	String of characters representing the value x.	8.5.7
SQR(x)	Square root of x; x must be >=0.	6.8.10
TAB(x)	Tab to print position x MOD LINELENGTH (next print value will begin at position x+1).	3.8.1
TAN(x)	Tangent x; x must be expressed in radians.	6.8.11
TRP	Returns an integer value representing the last line executed before a trap occurred.	12.2.1
VAL(s)	Has the value which the string s represents as a number. VAL converts from string to numeric.	8.5.5

APPENDIX E

LIST OF ERROR NUMBERS AND EXPLANATIONS

GROUP I: COMPILE TIME ERRORS

ERROR #	EXPLANATION
1	Bad statement number
2	Unrecognizable input
3	Unbalanced parentheses
4	Literal too long
5	Statement illegal in second clause
6	Expression too complex
7	Illegal expression element
8	Missing close quote
9	Illegal user function name
10	Characters after statement's end
11	Missing "#"
12	Illegal file designator expression
13	Missing ";"
14	Missing or illegal file name string
15	Illegal return variable
16	Illegal record number expression
17	Missing or illegal statement number
18	Illegal selector expression
19	Illegal function word
20	Illegal THEN, ELSE clause
21	Illegal assignment object
22	Missing assignment operator
23	Illegal expression
24	Illegal reference variable
25	Illegal list element
26	Illegal formal parameter
27	Non-simple variable used as FOR/NEXT index
28	Illegal "USING" string
29	LINPUT variable must be string
30	Missing "="
31	Missing or illegal initial value
32	Missing "TO"
33	Missing or illegal limit value
34	Missing or illegal STEP value
35	Parse failed
36	Missing "THEN"
37	Illegal function DEFinition
38	Cannot execute in keyboard mode
39	Illegal trap object
40	Command not allowed from file
41	Compiled file in illegal context
42	Dead environment, can't continue
43	Non-BASIC file
44	Illegal parameter

GROUP II: PROGRAM STRUCTURE ERRORS

ERROR #	EXPLANATION
50	Reference to undefined variable, or undimensioned array
51	Reference to nonexistent line number
52	Reference to undefined function
53	Reference to undeclared file number
54	Nested DEF's are illegal
55	Illegal number of buffers
56	Illegal file number
57	Unbalanced DO/DOEND's
58	RETURN without prior GOSUB
59	FNEND without RETURN
60	NEXT without FOR
61	NEXT mismatch (Illegal nesting of FOR/NEXT)
62	NEXT not in same block with FOR
63	INPUT/READ cannot invoke functions
64	Unbalanced user function calls at termination
65	Type mismatch
66	Dimension too large
67	String may not be redimensioned
68	Improper number of arguments/subscripts
69	Improper number of CALL/SYSTEM parameters
70	Reference to undefined procedure
71	Illegal delimiter string
72	Illegal TAB usage
73	Illegal TRAP situation

GROUP III: SYSTEM LIMITS AND FAILURES

ERROR #	EXPLANATION
80	Symbol table full
81	Too many files open
82	Out of storage
83	Runtime stack overflow
84	DO's nested too deep
85	Insufficient RIO resources
86	Feature not implemented
87	Interpreter error (impossible)
88	RIO interface error (impossible)

GROUP IV: BOUNDS, ARRAYS, STRINGS

ERROR #	EXPLANATION
100	Argument out of range
101	Illegal substring designator
102	Subscript out of range
103	Second string subscript out of range
104	Attempt to increase dimension
105	Missing subscript (dimension)

GROUP V: I/O ERRORS

ERROR #	EXPLANATION
120	File does not exist
121	File already exists
122	Attempt to space past beginning-of-file
123	Attempt to access past end-of-file
124	Out of DATA
125	Illegal file name
126	Illegal file type
127	File protection error
128	File already open
129	Unassigned I/O
130	File not open
131	Scratch file created (impossible)
132	Disk error
133	Disk not ready
134	Disk full
135	Invalid operation
136	Input numeric conversion error
137	Insufficient input
138	File structure error

GROUP VI: WARNINGS

ERROR #	EXPLANATION
140	Illegal number
141	Overflow
142	Underflow - Warning
143	Division by zero
144	Square root of negative number
145	LOG of negative or 0
146	String truncated during assignment
147	Format too small to contain number

GROUP VII: PRINT USING ERRORS

ERROR #	EXPLANATION
160	Illegal format character
161	Illegal exponent field
162	Zero field width (no digit positions)
163	Null format string

NOTES ON BASIC ERROR MESSAGES

Error Number	Comments
5	Only certain statements may be in the THEN or ELSE clause of an IF or ELSE statement. You have used one that is not (e.g., COM, DATA, NEXT, FOR, ELSE).
12	A file designator expression is the "#n,m;" or "#n;" part that follows an INPUT, LINPUT, READ, WRITE, or PRINT statement word.
15	The return variable follows the name in an ERASE or FILE statement.
21	The "assignment object" is that part to the left of an "=". It must be a simple or subscripted variable.
22	The "=" in an assignment statement is missing. This message is often given when garbage is typed, since the first letter is assumed to be a variable name and the error given is that the "=" following the variable is missing.
35	Some syntax error occurred. There might be a control character in an expression.
38	You have used a statement that may NOT be executed directly from the keyboard (Section V).
40	This line was ignored. Only BASIC statements may appear in ASCII files that are used with the GET or APP commands or with the CHAIN statement.
42	Execution of a program cannot be resumed after the program has been modified. It must also be started with the RUN (or XEQ) command initially.

43 The file referred to in a GET or XEQ (or CHAIN) command is binary but not the correct subtype. BASIC and BINBASIC SAVed files are NOT compatible. This error is issued if an attempt is made to use a SAVed file from the other BASIC.

50 A variable that has not been previously assigned a value has been used in an expression. Also, strings must be dimensioned before they are used.

54 There cannot be a DEF inside a multi-line function.

58 More RETURN's were executed than GOSUBs.

59 The FNEND of a function definition was executed. Control must return to the function caller via a "RETURN <expression>" statement.

62 In the search for the NEXT of a FOR, a DOEND was encountered. The NEXT must be in the same block as the FOR.

63 The subscript expressions in variables in an INPUT or READ statement cannot invoke functions. This is an implementation restriction.

64 An END (or the physical end of the program) was encountered while a function was still active. The program should terminate at the outer level (all functions having terminated).

68 Too many or too few arguments to a function call appeared in a function reference, or too many or too few subscripts appeared in a variable reference.

70 The procedure name in a CALL or SYSTEM statement was not in the Procedure Name Table.

80 Only 300 variable and function names are allowed and this limit had been reached.

82 Storage may run out in several ways. There may be insufficient space for variables, arrays, program statements, and file buffers. The SIZ command and the statement number of the error are helpful in determining why the error occurred.

83 Functions, FOR/NEXT loops, or GOSUB's have been nested too deep.

85 RIO returned the errors ASSIGN BUFFER FULL or LOGICAL UNIT TABLE FULL.

87 An "impossible" internal condition has happened; please send Zilog enough information to reproduce the error.

88 RIO returned the error INVALID UNIT, MEMORY PROTECT, MISSING OR INVALID OPERANDS, SYSTEM ERROR, NON-EXISTENT COMMAND, PROGRAM ABORT, MISSING OR INVALID PROPERTIES, I/O ERROR, (4DH), DIRECTORY FORMAT ERROR, ATTRIBUTES TRUNCATED, UNIT ALREADY OPEN, INVALID ATTRIBUTE, OR INVALID RENAME.

129 A multi-line function called from an I/O statement has altered the I/O environment (e.g., closed a file) so that the I/O statement cannot be completed.

131 An "impossible" internal condition has happened; please send Zilog enough information to reproduce the error.

132 RIO returned the error SEEK ERROR, DATA TRANSFER ERROR, SECTOR ADDRESS ERROR, or DISK ID ERROR.

138 RIO returned the error POINTER ERROR.

146 An assignment was made where the destination string length was shorter than the source string length. The source string is truncated on the right.

APPENDIX F

BASIC, PLZ, AND ASSEMBLY LANGUAGE LINKAGE

There are two points of interface between BASIC and user procedures. One is the Procedure-Name Interface and the other is the Call-Time Interface. The Call-Time parameter passing sequence is compatible with PLZ procedures. The Procedure-Name Interface is established when BASIC and the user procedures are loaded.

In this appendix, the two points of interface are discussed and then the procedure for linking a set of user procedures with BASIC is described.

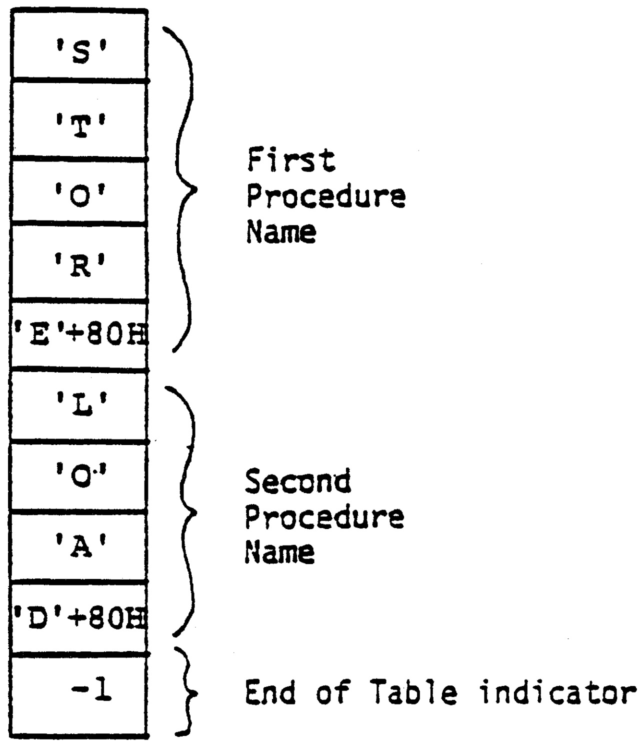
F.1 PROCEDURE-NAME INTERFACE

Two tables establish the procedure-name interface: the Procedure Name Table (PNT) and the Procedure Pointer Table (PPT). The PNT contains a list of names of user procedures and the PPT contains pointers to the Procedure Descriptor (PD) for each procedure. The correspondence between names and procedures is established by relative positions in the two tables.

The Procedure Name Table (PNT) consists of a series of names terminated by a -1 byte. Each name is a series of characters with the high order bit of the last one set (one) and the high order bit of all others reset (zero). Appendix G shows the source for a SYSTEM call (Section 3.12), including a macro for constructing name entries. The SYSTEM "LINELEN" call passes one parameter to the procedure. Appendix H includes macros for interfacing with non-BASIC procedures and an example using them. The following figure shows a Procedure Name Table. The macro WORD in Appendix G can be used to create the PNT.

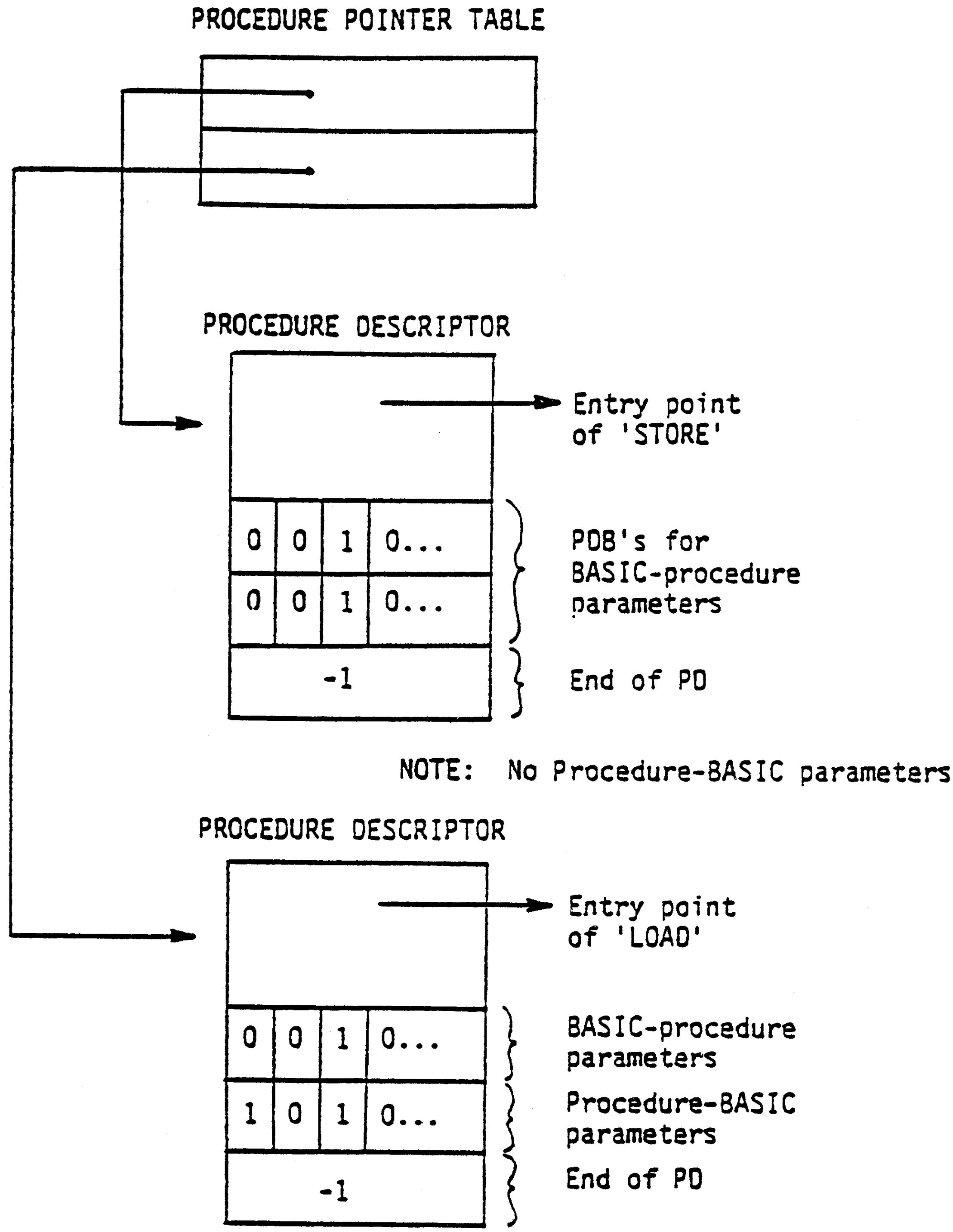
```
.
.
WORD S,T,O,R,E
DEFB -1
```

PROCEDURE NAME TABLE



The Procedure Pointer Table (PPT) is a list of pointers to the Procedure Descriptor for each procedure. The first pointer in the list goes with the first name, and so on. BASIC searches the PNT for the string given in the CALL statement and chooses the corresponding pointer from the PPT.

The Procedure Descriptor (PD) gives the entry point of the procedure and Parameter Descriptor Bytes (PDBs) for each parameter of the procedure. The entry point is simply the address of the first word to be executed. The PDBs are separated into two groups, each of which is optional. The first group describes parameters which are passed from BASIC to the user procedure. The second group describes those parameters returned by the procedure back to BASIC.


PROCEDURE POINTER TABLE
PROCEDURE DESCRIPTOR
Entry point
of 'STORE'
0 0 1 0...
0 0 1 0...
PDB's for
BASIC-procedure
parameters
-1
End of PD
NOTE: No Procedure-BASIC parameters
PROCEDURE DESCRIPTOR
Entry point
of 'LOAD'
0 0 1 0...
BASIC-procedure
parameters
1 0 1 0...
Procedure-BASIC
parameters
-1
End of PD

Structurally, the PD consists of the entry point address followed by the BASIC-to-procedure PDBs, followed by the procedure-to-BASIC PDBs, followed by a -1 byte. The first procedure-to-BASIC PDB (if any) must have its high order bit set (one) and all other PDBs must have their high order bits reset (zero). This is done so that BASIC can distinguish the two groups of PDBs.

Each PDB indicates the data type of a parameter passed to or from BASIC. The bit patterns used to indicate Real, Integer, and String types are indicated in the table below.

Parameter Descriptor Byte Format

x	s	i	0	0	0	0	0

s=0,i=1	INTEGER
s=1,i=0	STRING
s=0,i=0	REAL
s=1,i=1	illegal
x=1	for first procedure-to-BASIC PDB
x=0	for all other PDBs

F.2 CALL-TIME INTERFACE

Parameters are passed to user procedures using a PLZ compatible protocol. Prior to executing a CALL instruction, BASIC pushes 3 16-bit quantities on the stack. The first two must be set by the user procedure prior to a RETURN to BASIC. The top-most word pushed by BASIC is a pointer to a table which contains pointers to all BASIC-to-procedure parameters. The two words the user procedure must set are:

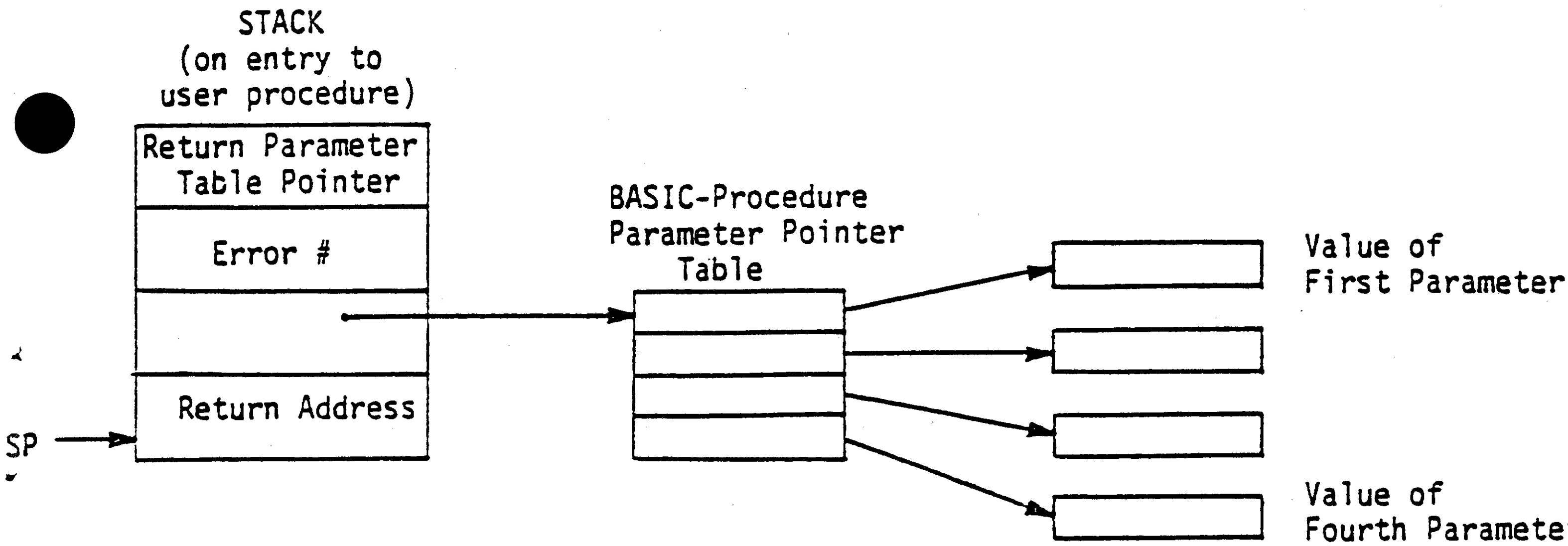
1. a pointer to a table containing pointers to values that are to be assigned to RETURN parameters in the CALL statement

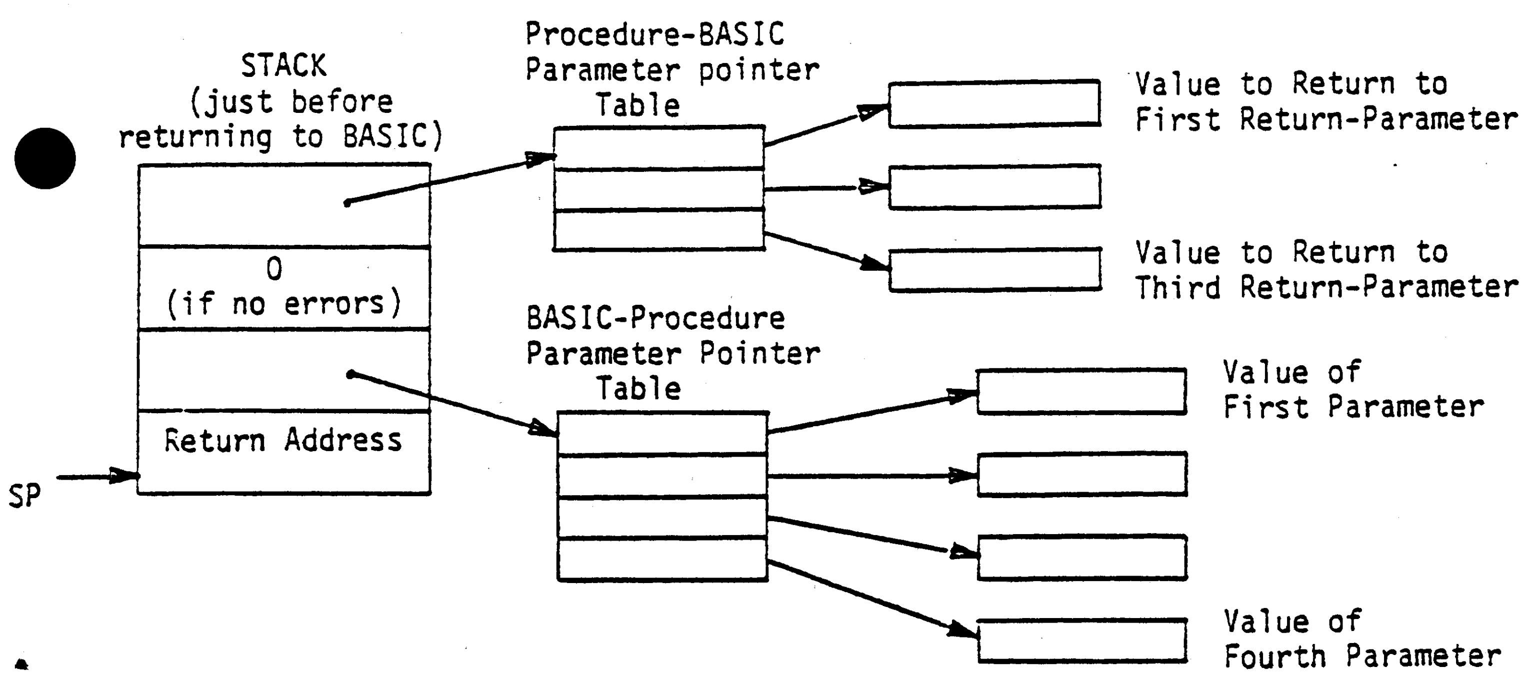
2. an error number which, if nonzero, will be issued (as a standard error message) when the user procedure returns to BASIC

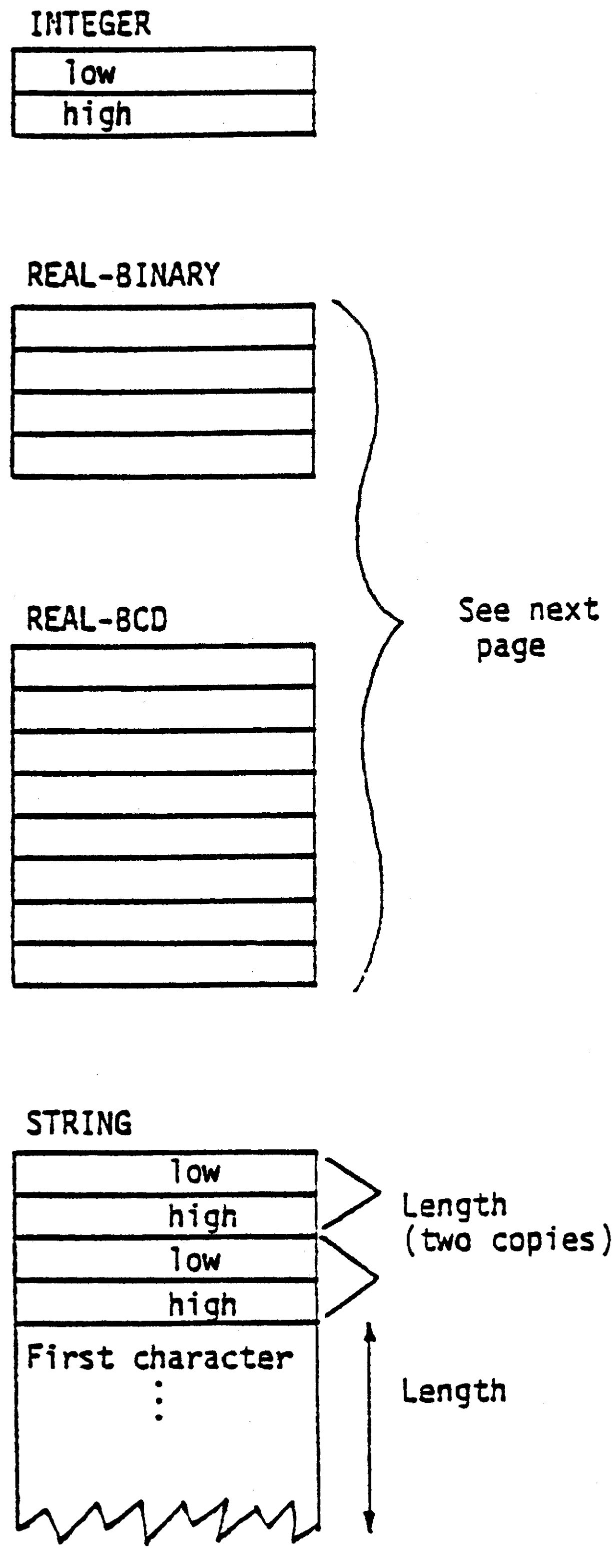
If the error number is non-zero, no assignment of return values will take place. Additionally, no assignment of return values will take place if the pointer described in (1) above is zero.

Each pointer in the BASIC-to-procedure Parameter Pointer Table points to a value of the type indicated by the corresponding Parameter Descriptor Byte (PDB) in the first group of PDBs in the Procedure Descriptor (PD). Likewise, BASIC requires that each pointer in the procedure-to-BASIC Parameter Pointer Table point to a value of the type indicated by the corresponding PDB in the second group of PDBs in the PD.

The proper structure of each data type is shown in the table below. A pictorial diagram of the passing structure is also given.

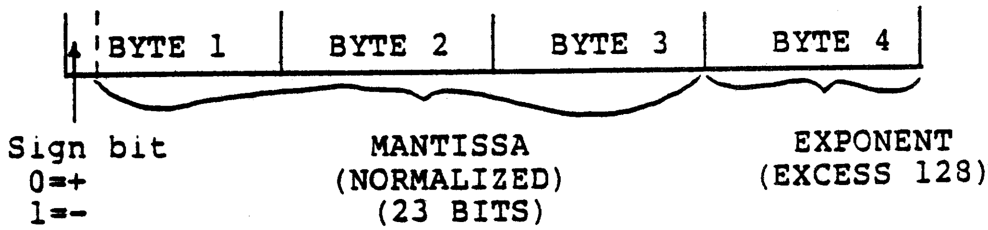

STACK
(on entry to
user procedure)
Return Parameter
Table Pointer
Error #
Return Address
SP
BASIC-Procedure
Parameter Pointer
Table
Value of
First Parameter
Value of
Fourth Parameter


STACK
(just before
returning to BASIC)
0
(if no errors)
Return Address
SP
Procedure-BASIC
Parameter pointer
Table
Value to Return to
First Return-Parameter
Value to Return to
Third Return-Parameter
BASIC-Procedure
Parameter Pointer
Table
Value of
First Parameter
Value of
Fourth Parameter


INTEGER
low
high
REAL-BINARY
REAL-BCD
See next page
STRING
low
high
low
high
Length
(two copies)
First character
Length

Binary Floating Point Representation:

24 bit sign-magnitude normalized* fraction
8 bit excess-128** exponent (base 2)
sign bit replaces most significant fraction bit (implied "1")
exponent field of zero implies value is zero

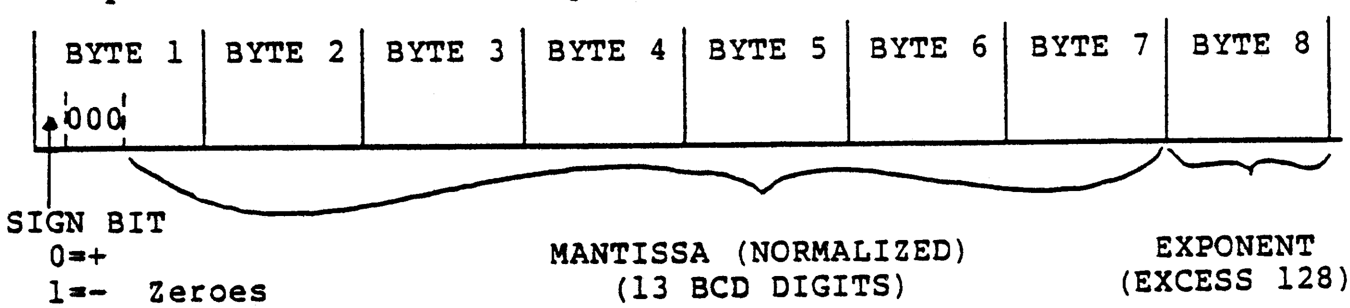



```
7F FF FF FF = 16777215/16777216*2^127 = 1.7014117*10^38
00 00 00 81 = 1/2*2/^1 = 1.0000000
00 00 00 01 = 1/2*2(-127) = 2.9387359*10^(-39)
XX XX XX 00 = 0
FF FF FF FF = -16777215/16777216*2^127 = -1.7014117*10^38
```

all results are rounded to 23 bit fractions

Decimal Floating Point Representation:

13 BCD digit sign-magnitude normalized* fraction
8 bit excess-128** exponent (base 10)
sign bit is most significant digit
exponent field of zero implies value is zero




```
09 99 99 99 99 99 99 FF = .99999 99999 999 * 10^127
01 00 00 00 00 00 00 81 = 1.0
01 00 00 00 00 00 00 01 = .1 * 10^(-127)
XX XX XX XX XX XX XX 00 = 0
89 99 99 99 99 99 99 FF = -.99999 99999 99 * 10^127
```

all results are truncated to 13 digit fractions

*Normalized means that the exponent is adjusted such that the most significant digit of the mantissa is non-zero or the mantissa is zero.

**Excess 128 means that the power to raise the base is equal to the exponent field minus 128.

F.3 LINKING USER PROCEDURES WITH BASIC

BASIC is informed of the existence of a PNT and PPT at start-up time by entering BASIC at the entry point BSTART, with

HL = address of the PNT

DE = address of the PPT

Control does not return as this entry point starts the BASIC interpreter. The stack must be in the same state as when RIO begins a user program, hence the transfer to BSTART should be a JUMP, not a CALL. The current values of BSTART are 4406H and 4006H for the MCZ-1 and ZDS systems, respectively.

The user procedures must be linked at the top of memory. A command sequence to create a BASIC environment including user procedures (in a 48K system) is:

```
        %USERPROCEDURES,          load user code linked at
                                  C000, don't execute

        %BASIC,X C000             load BASIC,start user code
```

The code at B600 (the first few bytes of the USERPROCEDURES module):

```
BSTART  EQU    4406               ;MCZ version
        LD     HL,nametable
        LD     DE,pntrtablel
        JP     BSTART
```

When a user procedure is entered, the stack pointer specifies the internal BASIC stack. The maximum guaranteed size of this stack is 30 bytes. Thus, if more extensive stack usage is required, the user procedure must change to its own stack. However, the BASIC stack MUST be restored prior to return.

APPENDIX G

EXAMPLE: ASSEMBLY LANGUAGE CALL - SYSTEM STATEMENT PROCESSOR

The following example shows the SYSTEM Statement Assembly program. The macro WORD constructs procedure name entries in the PNT. The PPT entries point to the Procedure Descriptors (PDs) for each type of SYSTEM call (Section 3.12). The SYSTEM "LINELEN" call has one BASIC-to-procedure parameter.

```
; Parameter interface for SYSTEM operations


; Procedure name table (PNT).

SYSNT:  WORD    A,U,T,O,C,R,O,F,F
        WORD    A,U,T,O,C,R,O,N
        WORD    I,N,D,E,N,T,O,F,F
        WORD    I,N,D,E,N,T,O,N
        WORD    L,I,N,E,L,E,N
        WORD    W,A,R,N,O,F,F
        WORD    W,A,R,N,O,N
        WORD    A,S,I,N,O,N
        WORD    A,S,I,N,O,F,F
PTEUT:  DEFB    -1


; Parameter Pointer Table (PPT)
; - Points to Procedure Descriptors (PDs)

SYSPT:  DEFW    ACROF
        DEFW    ACRON
        DEFW    INDOF
        DEFW    INDON
        DEFW    LINSET
        DEFW    WAROF
        DEFW    WARON
        DEFW    ASON
        DEFW    ASOFF
```

```
; Individual Procedure Descriptors (PDs)

; Auto CR OFF
ACROF:  DEFW    RACROF              ; -> entry point
        DEFB    -1                  ; no parameters

; Auto CR ON
ACRON:  DEFW    RACRON
        DEFB    -1

; Indent OFF
INDOF:  DEFW    RINDOF
        DEFB    -1

; Indent ON
INDON:  DEFW    RINDON
        DEFB    -1

; Set line length.
LINSET: DEFW    RLINS
        DEFB    20H                 ; single integer parameter
        DEFB    -1

; Warnings OFF
WAROF:  DEFW    RWAROF
        DEFB    -1                  ; no parameters

; Warnings ON
WARON:  DEFW    RWARON
        DEFB    -1

;  ASAVE line indent ON
ASON:   DEFW    RASON
        DEFB    -1

; ASAVE line indent OFF
ASOFF:  DEFW    RASOFF
        DEFB    -1
```

```
; Now, the code for each.

RACROF: LD      HL,FLAGS
        SET     FLACR,(HL)
        RET


RACRON: LD      HL,FLAGS
        RES     FLACR,(HL)
        RET


RINDOF: LD      HL,FLAGS
        SET     FLIND,(HL)
        RET


RINDON: LD      HL,FLAGS
        RES     FLIND,(HL)
        RET


RLINS:  POP     BC                  ; = ret
        POP     HL                  ; -> parm vector
        PUSH    HL
        PUSH    BC                  ; pardon the intrusion.

; HL -> IN parameter vector
        LD      E,(HL)
        INC     HL
        LD      D,(HL)              ; DE -> first parm
        EX      DE,HL
        LD      A,(HL)              ; A = new line len
        LD      (LINELN),A
        RET                         ; that's all...


RWAROF: LD      HL,FLAGS
        SET     FLWRN,(HL)          ; no warning messages
        RET


RWARON: LD      HL,FLAGS
        RES     FLWRN,(HL)
        RET

RASON:  LD      HL,FLAGS
        SET     FLASI,(HL)
        RET

RASOFF: LD      HL,FLAGS
        RES     FLASI,(HL)
        RET
```

```
; MACROS FOR BASIC AND ETC.
;

; WORD TABLE ENTRY
WORD    MACRO   #C1,#C2,#C3,#C4,#C5,#C6,#C7,#C8,#C9
        COND    '#C2'
        DEFB    '#C1'
        WORD    #C2,#C3,#C4,#C5,#C6,#C7,#C8,#C9
        ENDC

        COND    '#C2'=0
        DEFB    '#C1'+80H        ; SET BIT IN LAST BYTE
        ENDC
        ENDM
```

APPENDIX H

EXAMPLE: A USER PROCEDURE CALL

This appendix provides some macros for the BASIC-Assembly Language CALL linkage. Two macros are for use in the Procedure-Name Interface: WORD and PD. Four macros are for use in the Call-Time Interface: BPENTR, BPEXIT, GETP, and PUTP.

The format and explanation of the macro calls are given, followed by an example that uses the macros.

H.1 PROCEDURE-NAME INTERFACE MACRO CALLS

The Procedure-Name Interface is established with two tables: the Procedure Name Table (PNT) and the Procedure Pointer Table (PPT). The macro WORD is used to set up a procedure name in the PNT. The PPT consists of pointers to Procedure Descriptors. The macro PD sets up a Procedure Descriptor.

```
[label] WORD procname
```

Procname is the name of a user procedure. Each character of procname is separated by a blank. WORD puts each character of procname into the PNT.

The following example sets up a PNT for two assembly language procedures called "LOAD" and "STORE". The address of the PNT is NAMTAB. Note the commas between the characters of the procedure names. The PNT is terminated by a -1 byte.

```
NAMETAB:
        WORD L,O,A,D
        WORD S,T,O,R,E
        DEFB -1
```

The PPT contains pointers to the procedure descriptors for each procedure. A PPT with address PTRTAB may be set up as as follows:

```
PTRTAB:
        DEFW PD1
        DEFW PD2
```

PD1 and PD2 point to the Procedure Descriptors for LOAD and STORE. The Procedure Descriptors can be defined using the macro PD.

```
[label] PD addr [di1 di2 ...din] [! do1 do2 ...don]
```

Addr is the entry point address for a user procedure. The parameters di1, di2,...din define the data types of the BASIC-Procedure input parameters. The parameters do1, do2,...don define the data types of the Procedure-BASIC output parameters. The possible data types are IN (integer), ST (string), and RE (real). The character "!" is used to separate the BASIC-Procedure and Procedure-BASIC parameters.

The following example sets up a Procedure Descriptor whose starting address is defined by the label LOAD. The procedure accepts three parameters from BASIC, two integers and a string. The procedure returns two parameters, both integers.

```
PD1     PD      LOAD IN IN ST ! IN IN
```

H.2 CALL-TIME INTERFACE MACROS

The Call-Time Interface macros use the IY and IX registers to point to the BASIC-Procedure and Procedure-BASIC Parameter Pointer Tables, respectively. The index registers are set up at the entry of the assembly language procedure by the BPENTR macro. The BPEXIT macro sets up the return error code if specified. It must be used to exit a procedure that was initiated with the BPENTR macro. The PUTP and GETP macros are provided to move data between the Parameter Pointer tables and the HL register pair.

```
[label] BPENTR [outaddr]
```

The BPENTR macro sets the IY and IX registers to point to the BASIC-Procedure and Procedure-BASIC Parameter Pointer Tables. The parameter outaddr must be included if there are any Procedure-BASIC parameters. Outaddr is the pointer to the Procedure-BASIC Parameter Pointer Table. The pointer to the BASIC-Procedure Parameter Pointer Table is in the stack when the assembly language program is entered. BPENTR places the pointer (outaddr) to the Procedure-BASIC Parameter Pointer Table in the stack.

```
[label] BPEXIT [anychar]
```

The BPEXIT macro is used to put the return error code on the stack. The parameter anychar is any character. If anychar is present, an error code is present in the HL register pair. If anychar is not present, an error code of 0 is returned to the BASIC program. BPEXIT balances the stack and does the return.

```
                    B
[label] GETP parmno P

                    B
[label] PUTP parmno P
```

The GETP macro gets the pointer to a parameter from a Parameter Pointer Table using the index parmno and places it in the HL register pair. The PUTP macro takes the pointer to a parameter from the HL register pair and places it in the appropriate Parameter Pointer Table index parmno. The optional parameter in PUTP and GETP indicates which Parameter Pointer Table to use. The BASIC-Procedure Parameter Pointer Table is indicated by a "B". The Procedure-BASIC Parameter Pointer Table is indicated by a "P". GETP assumes "B" as its default value. PUTP uses "P" as its default value.

Continuing with the example procedure "LOAD":

```
LOAD:                             ;entry for LOAD proc
                                  ;    called from BASIC

          BPENTER RETTAB          ;set up index registers
                                  ;  and return table address

          GETP  1                 ;get the first parameter pointer
                                  ;    and put it in HL reg pair
                                  ;defaults to BASIC-Procedure
                                  ;    Parameter Pointer Table

          .
          .
          .
          JR  Z,OK                ;jump if no error

          LD  HL,ERCODE           ;set error code

          BPEXIT *                ;the parameter indicates that
                                  ;       HL=error code

OK:

          .
          .
          .
          LD  HL,PARM

          PUTP  1                 ;put pointer to return value in
                                  ;    Procedure-BASIC parameter
                                  ;    Pointer Table

          BPEXIT                  ;no parameter on macro call
                                  ;    indicates no error

RETTAB    DEFS  4                 ;allow room for two parameter
                                  ;    pointers
```

NOTE: The PUTP macro need not be used if the Procedure-BASIC parameter pointers are assembled into the Procedure-BASIC Parameter Pointer Table.

Example:

```
        RETTAB
                DEFW PRM1
                DEFW PRM2

        PRM1    DEFS  2        ;storage for an integer
        PRM2    DEFS  2
```

H.3 AN EXAMPLE

The following example procedure takes one integer parameter from BASIC and returns two integers containing the lower and upper byte of the input parameter.

```
;        asm procedure to split an integer into two bytes

*L OFF
*I BASIC_IF.M
*L ON

;        from BASIC do
; CALL "SPLIT",P1;P2%,P3%
;        the integer P1 is split into two bytes,
;        P2 is lsbyte, P3 is msbyte


         ld       hl, namtab
         ld       de, ptrtab
         jp       4006H      ; initialization entry into BASIC

SPLIT

         BPENTR RETTAB

         GETP 1

         ld       e,(hl)
         inc      hl
         ld       d,(hl)    ; retrieve the integer

         ld       hl,rp1
         ld       (hl),e
         ld       hl,rp2
         ld       (hl),d

         BPEXIT

RETTAB
         defw rp1
         defw rp2

rp1      defw 0
rp2      defw 0

namtab
         WORD S,P,L,I,T
         defb -1

ptrtab
         defw PD1

PD1      PD       SPLIT IN ! IN IN
```

H.4 THE MACROS

```
;       a macro to set up Procedure Name Table entries

; WORD TABLE ENTRY
WORD    MACRO   #C1,#C2,#C3,#C4,#C%,#C6,#C7,#C8,#C9
        COND    '#C2'
        DEFB    '#C1'
        WORD    #C2,#C3,#C4,#C5,#C6,#C7,#C8,#C9
        ENDC

        COND    '#C2'=0
        DEFB    '#C1'+80H         ; SET BIT IN LAST BYTE
        ENDC
        ENDM


;       macro to set up parameter specs
;       for Procedure Descriptors


PD  macro   #0 #1 #2 #3 #4 #5 #6 #7 #8 #9
        defw  #0
        defprm #1 #2 #3 #4 #5 #6 #7 #8 #9
        defb  -1
  endm


defprm macro #1 #2 #3 #4 #5 #6 #7 #8 #9

  cond   '#1'='!'
fooxxx  defl 80h
        setxxx  #2
        defprm  #3 #4 #5 #6 #7 #8 #9
  endc

  cond \(('#1'=0)^('#1'='!'))
fooxxx  defl 0
        setxxx  #1
        defprm  #2 #3 #4 #5 #6 #7 #8 #9
  endc

  endm
```

```
setxxx  macro  #1 #2

  cond  '#1'='IN'
  defb  20h+fooxxx
  endc

  cond  '#1'='ST'
  defb  40h+fooxxx
  endc

  cond  '#1'='RE'
  defb  fooxxx
  endc

  endm



        macros to field calls from basic (or PLZ?)


;  registers
; a pointer to the TOS at entry is pushed onto the stack
;       ix=Basic to Procedure Parameter Pointer Table
;       iy=Procedure to Basic PPT


; %^&  B P E N T R  %^*&

; set up iy BtoP
; set up ix PtoB
; parameter is PtoB PPT

BPENTR  macro #1

        push ix
        ld  ix,0
        add ix,sp
        ld h,(ix+3)
        ld l,(ix+2)
        push hl
        pop  iy

        cond '#1'
        ld  hl,#1
        ld (ix+7),h
        ld (ix+6),l
        push hl
        pop  ix
        endc
  endm
```

```
; if param then hl=error number
BPEXIT macro #1
        pop ix
        cond '#1'=0
        ld hl,0
        endc

        ld (ix+5),h
        ld (ix+4),l
        ret
  endm

; *&^%  G E T P  &%^$   get parameter

; GP  #parameter #BorP
; BorP ::= B | P   ;  which parameter list
;                       Basic or Procedures

; GET defaults to BASIC-Procedure Parameter Pointer Table

GETP macro #1 #2
  cond  ('#2'=0)^('B'='#2')
        getreg iy #1
  endc
  cond 'P'='#2'
        getreg ix #1
  endc
  endm

getreg  macro  #r #o
  ld  h,(#r+2*(#o-1)+1)
  ld  l,(#r+2*(#o-1))
  endm


; $#^&%  P U T P  &^%&^%$  put parameter
;  w    PUTP defaults to Procedure -BASIC Parameter Pointer Table

PUTP  macro  #1 #2
  cond ('#2'=0)^('#2'='P')
        putreg ix #1
  endc
  cond '#2'='B'
putreg   iy #1
  endc
  endm


putreg  macro  #r #o
        ld (#r+2*(#o-1)+1),h
        ld (#r+2*(#o-1)),l
  endm
```

APPENDIX I

A PROCEDURE FOR INTERFACING WITH A PRINTER

Activate the printer driver (PRINTER) before entering BASIC.

```
%ACTIVATE $PRINTER
%BASIC
```

To print from a program to PRINTER:

```
>10 FILE #1;"$PRINTER"
>20 PRINT #1;"HELLO"
>30 STOP
>RUN

STOP AT 30
```

To print from a keyboard executable statement to PRINTER:

```
>FILE #1;"PRINTER"
>PRINT #1;"HELLO THERE"
```

To list the current program in the workspace on PRINTER:

```
>ASA-$PRINTER/xx
```

NOTE: An error will occur after the list is done. This will not affect the listing.

APPENDIX J

EXTERNAL INTERRUPT LINKAGE

When the external interrupt flag is set, a trap can be invoked by the BASIC interpreter. The trap is established by the "TRAP EXT TO label" statement. Then, an external user program can cause the trap to be invoked by setting the external interrupt flag to a non-zero value. The flag will be reset to zero when the trap is invoked.

The flag may be accessed as shown in the following diagram:

```
                      |---------|
     BSTART           |         |
                      |---------|
                      |         |
                      |---------|
     BSTART+2         |    L    |>
                      |---------|>   ---->    External Interrupt
                      |    H    |>            Flag
                      |---------|
```

Thus, at location BSTART+2, there is a pointer to the byte that is the External Interrupt Flag.

Zilog

Zilog (U.K.) Limited
Nicholson House
Maidenhead
Berks.
Telephone: 0628 36131
Twx: 848 609

Printed in UK