EG3200 Genie III Computer System

BASIC MANUAL

Eaca®

Disclaimer

EACA Computer Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, EACA Computer Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of EACA Computer Ltd. to notify any person of such revision or changes.

EG3200 Genie III Computer System

BASIC MANUAL

eaca®

Disclaimer

EACA Computer Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, EACA Computer Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of EACA Computer Ltd. to notify any person of such revision or changes.

EG3200

GENIE III

BASIC MANUAL

Eaca®
1982

77-2005101-01

CONTENTS

INTRODUCTION:-

This manual is a reference for GENIE III BASIC language. The manual is directed to those who have previously acquired some familiarity with computer programming and who wish to use this as a guide for the language BASIC.

This manual consists of four chapters plus an appendix. The first is an introductory chapter provides an understanding and information about GENIE III BASIC language. Chapter two describes the commands and statements that are available in this system. Chapter three contains all the functions that can be used by this system. Each function is discussed, providing the relative merits of various techniques and usage.

CHAPTER ONE

GENERAL INFORMATION ABOUT GENIE III

BASIC is chosen as the fundamental high level porgramming language of our GENIE III system. Most of the BASIC features come from the DISK BASIC of NEWDOS/80 version 2 which is our operating system. However, some modifications to the BASIC are required to suit our display format, 80 x 24 and 64 x 16 format.

If the GENIE III's system diskette loaded is 64-mode, 64 x 16 display format is assumed. Similarly, 80 x 24 display format is adopted of the system diskette loaded is 80-mode.

There are two kinds of BASIC that can be used in this system; Microsoft Extended Level II Basic and NEWDOS BASIC.

Procedure of entering BASIC:

(1) LEVEL II BASIC:- LEVEL II BASIC can be entered by pressing the BREAK key first and switching ON the power of the system, provided that the system diskette has been placed in drive 0. Another way is to press both BREAK and RESET keys at the same time after the system has entered the DOS level.

(2) NEWDOS BASIC:- NEWDOS BASIC can be entered by the typing in the command "BASIC" to the system in DOS level.

Table 1-A (a)-(b) lists all the commands and statements that are available in GENIE III BASIC.

NOTE: All of them can be used by NEWDOS BASIC, however, only those with "*" are available in Level II BASIC.

TABLE 1-A: BASIC COMMANDS AND STATEMENTS

Table 1-A (a): Active Commands

ACTIVE COMMANDS:

*	AUTO	*	CLEAR		CLOSE
	CMD"C"		CMD"E"		CMD"S"
	CMD"doscmd"		CONT	*	DELETE
*	EDIT		END		ERROR
*	LIST	*	LLIST		LOAD
	MERGE	*	NEW		REF
	RENEW		RENUM	*	RUN
	RUN"program"		SAVE		
*	TROFF	*	TRON		KILL

Table 1-A (b): Programming commands and statements.

PROGRAMMING COMMANDS AND STATEMENTS:

*	CLEAR		CMD"F=POPS"		CMD"F=POPR"
	CMD"F=POPN"		CMD"F=SASZ"		CMD"F=ERASE"
	CMD"F=KEEP"		CMD"F",DELETE		CMD"F=SWAP"
	CMD"F=SS"		CMD"J"		CMD"O"
*	DATA	*	DEFDBL		DEF FN
*	DEFINT	*	DEFSNG	*	DEFSTR
	DEFUSR	*	DIM	*	END
*	FIELD	*	FOR-NEXT	*	GET
*	GO SUB	*	GO TO	*	IF
*	IF-THEN	*	IN-THEN-ELSE	*	INKEY$
*	INPUT	*	INPUT#	*	LET
	LINE INPUT		LINE INPUT#	*	LPRINT
	LSET		MID$	*	ON-ERROR-GOTO
*	ON-GOTO	*	ON-GOSUB		OPEN
*	PRINT	*	PRINT TAB	*	PRINT USING
*	PRINT @	*	PRINT #		PUT
*	READ	*	REM	*	RESUME
*	RESTORE	*	RETURN		RESET
*	STOP				

TABLE 1-B (a)-(d) lists all the functions that are avail-able in GENIE III BASIC.
NOTE: All of them can be used by NEWDOS BASIC, however only those with an "*" are available in Level II BASIC.

Table 1-B (a) String Functions:

*	ASC	*	CHR$		CVD
	CVI		CVS		INSTR
*	LEFT$		LEN	*	MID$
	MKD$		MKI$		MKS$
*	RIGHT$	*	STR$	*	STRING$
*	VAL		VARPTR		

Table 1-B (b): Arithematic Functions

*	ABS	*	ATN	*	CDBL
*	CINT	*	COS	*	CSNG
*	EXP	*	FIX	*	INT
*	LOG	*	RANDOM	*	RND
*	SGN	*	SIN	*	SQR
*	TAN				

Table 1-B (c): Graphic Functions

* CLS	* POINT	* RESET
* SET		

Table 1-B (d): Special Function :

&H	&O	EOF
* INP	LOC	LOF
* MEM	* OUT	* PEEK
* POKE	* POS	TIME$
* USR		

CHAPTER TWO

GENIE III COMMANDS AND STATEMENTS

This chapter describes all the commmands and functions available in GENIE III System.

In the syntax notation, all items in lower case letters enclosed in angle brackets (<>) are to be provided by the user. All punctuations (except angle brackets) and capital letters must be input exactly as shown.

2.1 AUTO

SYNTAX: (a) AUTO
(b) AUTO <line-number>
(c) AUTO <line-number>,<increment>

EXPLANATION: This command automatically sets the line-numbers before each source lines is entered. If the user only types in AUTO followed by the NEW LINE key, the beginning line number will be set at 10, with each increment of 10. The option permits the user to specify the beginning line number as well as the increment desired between lines.
The user may enter his program statement right after the line number.

REMARK: Everytime the user hits the NEW LINE key, the computer will increase the line number. The AUTO command will remain in operation until the BREAK key is hit. Note that whenever AUTO brings up a line that has been used previously, there will be as asterisk appear right next to the line number.
If the user does not want to alter that line, hit the BREAK key to turn off the AUTO function.

EXAMPLE: AUTO
Then the line numbers 10, 20, 30, ... will be generated.

AUTO 2,2
It generates line numbers 2,4,6,.....

2.2 CLEAR (active command)

SYNTAX: (a) CLEAR
(b) CLEAR <number-of-bytes>

EXPLANATION: This active command clears a specific number of bytes for string storage. If the option (a) is used the computer will reset all numeric varibles to zero, and all string variables to null. When the option (b) is taken, it perfroms similarly as option (a), in addition, a specified number of bytes is cleared for string storage.

NOTE: When the user turns on the computer, a CLEAR 50 command is performed automatically.

EXAMPLE: CLEAR 100
Reset all numeric varibles to zero, and all string variables to null. Then clear 100 bytes of memory for string storage.

2.3 CLEAR (programming command)

SYNTAX: CLEAR n

EXPLANATION: This statement sets all varibles to zero. If number n is specified, the computer sets n bytes of space for string storage.

REMARK: The CLEAR statement becomes critical during program execution, because an OUT-of-string-Space error will occur, if the amount of string storage cleared is less than the greater number of characters stored in string variables.

EXAMPLE: 10 CLEAR 1000
Clear 1000 bytes of memory space for string storage.

2.4 CLOSE

SYNTAX: CLOSE <num>,<num>,.....

EXPLANATION: This statement terminates the file assignment to a particular file through the specified buffer(s). <num> is the number, 1 to 15, refers to the files' buffer number under which the file was OPENED.
A close with no argument closes all open files.

EXAMPLE:
```
CLOSE 2,4,8
or
CLOSE
```

2.5 CMD"C"

SYNTAX: CMD"C"

EXPLANATION: This command with no argument is used to compress out all spaces from the program text, except for those within strings, and deletes all remarks from the text, including those lines which are entirely remarks.

EXAMPLE: The statement

```
:
40 IF Y=X GO TO 60
:
```

After the active command CMD"C", the text will become

```
:
40 IFY=XGOTO60
:
```

2.6 CMD"E"

SYNTAX: CMD"E"

EXPLANATION: This command displays the DOS error messages associated the latest DOS error encountered by BASIC.

2.7 CMD"F=POPS"

SYNTAX: CMD"F=POPS"

EXPLANATION: This statement purges all returns and FOR-NEXT controls.

REMARK: The purpose of this statement is to allow the program to "escape" of complex coding and return to BASIC's first level.

2.8 CMD"F=POPR"

SYNTAX: CMD"F=POPR"

EXPLANATION: This statement purges the current GOSUB level with any remaining FOR-NEXTS for that level. The control will pass to the statement following the CMD"F=POPR" statement.

2.9 CMD"F=POPN"

SYNTAX: CMD"F=POPN"
or
CMD"F=POPN"<variable-name>

EXPLANATION: The CMD"F=POPN" purges the most recently located FOR=NEXT's control data. This is the same as "NEXT" where the loop limit is passed. If the <variable-name> option is used, the FOR=NEXT loop joined with <variable-name> is purged along with any other FOR-NEXT loops established while <variable-name>'s loop was outstanding.

REMARK: Execution continues with the statement following the CMD"F=POPN" or CMD"F=POPN" variable-name> ststement.

2.10 CMD"F=SASZ"

SYNTAX: CMD"F=SASZ",<expression>

EXPLANATION: This statement changes the string area size without affecting or clearing the variables.

REMARK: <expression> must be a value large enough for the string.
An error occurs if <expression> is too small or too large.

EXAMPLE: CMD"F=SASZ",300

2.11 CMD"F=ERASE"

SYNTAX: CMD"F=ERASE",<list-of-variables>

EXPLANATION: This statement clears the specified variables.

REMARK: If a specified variable is within an array the entire array is cleared. However the size remain the same.

EXAMPLE: 5Ø CMD"F=ERASE",ABB,BBC

2.12 CMD"F=KEEP"

SYNTAX: CMD"F=KEEP",<list-of-variables>

EXPLANATION: This statement clears all variables except:
(1) those specified in the <list-of-variablea>;
(2) specially defined variables, for example, those defined by a DEFFN statement.

REMARK: The size of string area remains the same. If <list-of-variables> is omitted, all variables are cleared; except the special one.

The entire array will not clear if a specified variable name is within that array.

EXAMPLE: 2ØØ CMD"F=KEEP",AB$,CCY,A#

2.13 CMD"F",DELETE

SYNTAX: CMD"F",DELETE <line-1>-<line-2>

EXPLANATION: This statement deletes the text lines from <line-1> to <line-2> during program execution.

REMARK: The DEFFN variables for DEFFN statement in the delete range are cleared, however, all other variables are kept as before.

NOTE: CMD"F",DELETE must not be
(a) a direct statement
(b) include in a DEFFN statement
(c) include in a subroutine or a FOR-NEXT loop.

CMD"F",DELETE must be
(a) last statement on its text line
(b) followed by the text line where execution will continue after the delete.

EXAMPLE:
```
100 CMD"F",DELETE 1000-1500
```

2.14 CMD"F=SWAP"

SYNTAX: CMD"F=SWAP",<variable-1>,<variable-2>

EXPLANATION: The CMD"F=SWAP" is used to exchange the value of <variable-1> with that of <variable-2>.

REMARK: <variable-1> and <variable-2> must be the same type.

EXAMPLE: 200 CMD"F=SWAP",A$,B$

2.15 CMD"F=SS"

SYNTAX: (a) CMD"F=SS"
(b) CMD"F=SS",<line-number>
(c) CMD"F=SS",N

EXPLANATION: The CMD"F=SS" is used to perform single step through program execution. The (a) and (b) formats set BASIC into single step mode, while (c) format turns off the single step mode. (b) format specifies the line to start single step.

REMARK: When the program is executed in the single step mode, a "@" displays at the upper right top corner of the screen. For @ is the line number for the next line to be executed. In order to get to next step, press the NEW LINE key.

The single stepping turns off as it comes to a CMD"F=SS",N statement.

2.16 CMD"J"

SYNTAX: CMD"J",<date-1>,<date-2>

EXPLANATION: The CMD"J" is a Calendar Date Conversion statement. This is used to convert the expression <date-1> to the appropriate format and puts the result in the string variable <date-2>.

Two conditions :

(a) If <date-1> is mm/dd/yy then <date-2> is stored in ddd format;

(b) if <date-1> is yy/ddd then <date-2> is stored in mm/dd/yy format.

where

mm is the month of the year from Ø1 to 12
dd is the day of the month from Ø1 to 31
ddd is the day-of-the-year from Ø1 to 366
yy is the year of the 2Ø-century from ØØ to 99.

2.17 CMD"O"

SYNTAX: (a) CMD"O",<n>,<av-1>,<av-2>.....
or
(b) CMD"O",<n>,*<iav-1>,<av-2>,<av-3>.....

EXPLANATION: The CMD"O" command is used to sort array in the main memory. <n> specifies the number of element in each of the array participating in the sort. A maximum of 9 arrays may be specified.

Format (a): It is a direct sort meaning that the elements of all 1 to 9 arrays are moved around to adjust to the desired sort order. <av-1>, the arrray-variable-1, must be specified while <av-2>, the array-variable-2, and up are optional.
For each array, the resulting order is the same. For example, if the 4th element of array 1 (<av-1>) is sorted into the 6th element slot, then for each of the other arrays, if any, the 4th element is also placed into the 6th element slot.

Format (b): It is an indirect sort. <iav-1> must be an integer array variable. <iav-1> and <av-2> must be specified while <av-3> and up are optional. Format (b) is different from Format (a), is that, only the n element of array <iav-1> are sorted; the other arrays are in same order as before. In other words, during sorting only array <iav-1> is altered but <av-2> and up are not altered.

EXAMPLE:

```
:
10 DIM A$(20), AMT(30), BB(30)
20 x=10
:
90 CMD"O",X<A$(0),AMT(0)
:
150 CMD"O",X,*BB(2),AMT(3),A$(5)
:
```

2.18 CMD"S"

SYNTAX: CMD"S"

EXPLANATION: This statement with no argument allows BASIC to exist DISK BASIC and return to DOS READY state.

REMARK: If it is in the form of CMD"S=<doscmd>",where <doscmd> is the DOS command, then the following steps occur:

1. It is moved into the DOS command buffer and BASIC released.
2. The <doscmd> placed into the DOS buffer is executed immediately without "DOS READY" displays on the screen.
3. After the completion of the command, the control returns to DOS READY state.

2.19 CMD"doscmd"

SYNTAX: CMD"<DOS-command>"

EXPLANATION: The statement allows BASIC to carry out DOS's command. BASIC moves the command to DOS's command buffer. When the specified command is completed, control returns to BASIC mode.

EXAMPLE: CMD"DIR Ø"
will list a directory on disk drive Ø

CMD"COPY ABC:Ø :1"
will copy ABC in disk drive Ø onto disk drive 1.

2.20 CONT

SYNTAX: CONT

EXPLANATION: This command continues the program execution, at the point where the execution has been stopped by the BREAK key or a STOP statement within the program.

2.21 DATA

SYNTAX: DATA <list-of-items>

EXPLANATION: This statement defines data items to be read by READ statement.
The item list will be accessed by the computer sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement.

REMARK: The items can be string or numeric value.
The order of values in a DATA statement must match up with the variable types in the READ statement.

EXAMPLE:
```
30 READ A,B,C$
40 DATA 11,56,"SENTENCE"
```

2.22 DEFDBL

SYNTAX: DEFDBL <letter-range>

EXPLANATION: A DEFDBL statement declares (treat and store) that the variable names beginning with the letter(s) specified will be double precision.

REMARK: However, a type declaration character can over-ride this type definition. Double precision allows 17 digits of precision, while only 16 digits are displayed when a double precision variable is print.

EXAMPLE:
```
10 DEFDBL M-P,G
```
Causes variables beginning with one of the letters M through P, or G to become double precision.

2.23 DEF FN

SYNTAX: DEF FN name <argument>=<expression>

EXPLANATION: A user-created implicit function is defined with the name FN name. The defined implicit function will automatically be performed when the name is called. The function must be composed of the letters FN followed by a valid level II variable name.

EXAMPLE:

```
    20 DEF FNBB (Y) = Y/2
```

This statement defines a function FNBB(Y) which divides Y by 2.
Thus the statement

```
    40 X=FNBB(4)
```

would set X to the value 2.

2.24 DEFINT

SYNTAX: DEFINT <letter-range>

EXPLANATION: Variable names that begin with letters specified within the letter range, will be treated and stored as integers.

REMARK: Defining a variable name as an integer not only saves memory space, but also saves computer time, because integer calculation is faster than single or double precision calculation.

NOTE: Integers can only take on values between -32768 and +32767 inclusive.

EXAMPLE:

```
10 DEFINT X,Y,Z
```

After the computer has executed line 10, all variables beginning with the letters X,Y, or Z will be treated as integers. Therefore, X2, X3, YA, YB, ZI, ZJ will become integer variables. Except that X1#, X2#, will be still double precision variables, because type declarations always over-ride DEF statements.

```
50 DEFINT A - D
```

This statement defines variables beginning with letter A,B,C, or D to be integer variables.

2.25 DEFSNG

SYNTAX: DEFSNG <letter-range>

EXPLANATION: This statement declares that the <letter-range> to be a single precision variable.
Variable name that begin with those letters specified within the letter range, will be treated and stored as single precision variables. However, a type declaration character can over-ride this type definition.

REMARK: Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. All numeric variables are assumed to be single precision unless otherwise specified. The DEFSNG statement is primarily used to re-define variables which have previously been defined as double precision or integer.

EXAMPLE:
```
10 DEFSNG A-D,Y
```
Defines variables beginning with the letter A through D, or Y to become single precision. However, A# would still be a double precision variable and Y% still be an integer variable.

2.26 DEFSTR

SYNTAX: DEFSTR <letter-range>

EXPLANATION: This is a string declare statement. Variables that begin with those letters specified within the letter range, will be treated and stored as string.

REMARK: However, a type declaration character can over-ride this type definition. Each string can store up to 255 characters, if there is enough string storage space cleared.

EXAMPLE:
```
10 DEFSTR A-D
```
Causes variables beginning with any letter A through D to be string variables.

2.27 DEFUSR

SYNTAX: DEFUSR <n> = <mexp>

EXPLANATION: This statement defines entry address for USR routine. <n> can be of any one of the digits 0, 1, ...,9. If <n> is excluded, 0 is assumed. <mexp> is the entry address to a machine-language routine.

EXAMPLE:
```
10 DEFUSR2 = &H7D00
```
This statement assigns the entry point 7D00 hex (32000 decimal), to the USR2 call. Hence the control branches to subroutine beginning at hex 7D00 when USR2 is called.

2.28 DELETE

SYNTAX: DELETE <line-number>
or
DELETE <line-number>-<line-number>

EXPLANATION: This command clears the memory location that contains the specified line(s).

NOTE: D <line-number> is same as DELETE <line-number>.

EXAMPLE:
```
    DELETE 5
Clear line 5

    DELETE 30-60
Clear line 30 through 60.
```

2.29 DIM

SYNTAX: DIM <name> (<dim-1>,<dim-2>,.....)

EXPLANATION: This statement defines the variable name to be an array or list of arrays. The number of elements in each dimension may be specified through dim-1, dim-2,..,etc.

REMARK: If <dim> is not specified, 11 elements in each dimension is assumed in each array. The number of dimension(s) is/are limited only by the memory size available.

EXAMPLE:
```
    10 DIM A(5), B(3,4)
```
This statement defines the one dimensional array, A(5), with 6 element (from 0 to 5) and the two dimensional array, B(3,4), with 20 elements (4*5).

2.30 EDIT

EXPLANATION: This command will cause the computer to shift from the Active Command level to the Editing Level.

THERE ARE TWO BASIC EDITORS IN GENIE III SYSTEM; LEVEL II BASIC EDITOR AND SCREEN EDITOR. REFER TO CHAPTER 4 IN THIS MANUAL FOR DETAIL DESCRIPTION AND INFORMATION.

2.31 END

SYNTAX: END

EXPLANATION: This statement with no argument terminates the execution of a program.
The END statement is primarily used to cause execution to terminate at some point other than the logical end of the program.

REMARK: The CONT command can be used to resume execution at the statement following the END statement.

EXAMPLE:

```
10 A=B+C
:
:
200 End
```

2.32 ERROR

SYNTAX: ERROR <code>

EXPLANATION: This command displays the error message corresponding to specified error code.

EXAMPLE: When type in

```
ERROR 4
```

the DISK BASIC respones and displays with the message

```
OUT OF DATA
```

2.33 FIELD

SYNTAX: FIELD <num>, <len-1> AS <var-1>,<len-2> AS ....

EXPLANATION: This statement addresses field sizes and names to a random file buffer. <num> is the file number under which the file was OPENed (random access mode). <len-1> is the length of the first field to be arranged into <var-1> (variable name for the first field). <len-2> is the second field and <var-2> is second variable name and so on.

EXAMPLE:

```
20 FIELD 2, 10 AS A$, 20 AS B$, 30 AS C$
```

This statement assigns the first 10 bytes of bytes of buffer to the variable name (buffer name) A$, the next 20 bytes to B$, and the next 30 to C$.

2.34 FOR....NEXT

SYNTAX: FOR <name>=<initial> TO <stop> STEP <increment>
:
:
NEXT <name>

EXPLANATION: These statement form an iterative loop so that a sequence of program statements may be executed over a specified number of times.
<name> must be a numeric variable.
<exp> is an expression.
In the FOR statement, initial value, final value and increment can be constants, variables or numeric expressions.
When the FOR statement is encountered during execution, <name> is set to the value of "start". Then the statements between the FOR and NEXT statements are executed. When the NEXT statement is encountered, the value of <increment> is added to <name>. If <increment> is positive and <name> is greater than <stop>, or, if <increment> is negative and <name> is less than <stop>, execution continues with the statement following the NEXT statement. Otherwise, the statements between the FOR and NEXT statement are repeated. The step <increment> part of the statement can be omitted, in this case, <increment> is assumed to be 1.

EXAMPLE:

```
:
20 FOR K=0 to 20 STEP 2
:
:     (statements)
:
 130 NEXT K
:
```

2.35 GET

SYNTAX: GET <file-number>,<record-number>

EXPLANATION: This statement causes a data record <record-number> read from a random disk file into a specified buffer <file-number>. The disk file must have been opened with OPEN statement before using GET. If <record-number> is omitted, the current record is read into the buffer.

EXAMPLE:

```
:
20 OPEN "A",1,"ABC/BAS"
:
100 GET 1,50
:
```

2.36 GOSUB

SYNTAX: GOSUB <statement-number>

EXPLANATION: This command transfers program control to the specified line number where a subroutine starts. Only if the computer encounters a RETURN statement, it will then jump back to the statement that immediately follows the GOSUB. Just like GOTO, GOSUB may be preceded by a test statement, such as:

```
IF A=B THEN GOSUB 100
```

EXAMPLE:

```
10 PRINT "MAIN PROGRAM"
20 GOSUB 50
30 PRINT "END OF PROGRAM"
40 END
50 PRINT "SUBROUTINE"
60 RETURN
```

2.37 GOTO

SYNTAX: GOTO <line-number>

EXPLANATION: This statement transfers program control to the specified line number. If used independently, an unconditional branch will result.
However, test statements may precede the GOTO statement to create a conditional branch.

EXAMPLE:

```
:
:
30 GOTO  100
:
:
100 PRINT "AAAAA"
:
```

2.38 IF

SYNTAX: IF <expression-action-clause>

EXPLANATION: This statement instructs the computer to test a logical or relational expression. If the expression is TRUE, control will proceed to the "action" clause immediately following the expression. If the expression is FALSE, control will jump to the matching ELSE statement (if there is one) or down to the next program line.
In numerical term, if the expression, has a non-zero value, it is always equivalent to a logical true.

EXAMPLE:

```
:
:
60 IF A>20 GOTO 150
:
```

2.39 IF-THEN

SYNTAX: IF <condition> THEN <statement-or-line-number>

EXPLANATION: This statement is the action cause after the IF test. If the "test" is true, the <statement-or-line-number> will perform.

EXAMPLE:

```
:
50 IF A>B THEN PRINT " A>B "
```

2.40 IF-THEN-ELSE

SYNTAX: IF <condition> THEN <state-line> ELSE <state-line>

EXPLANATION: This ELSE statement must be used after the IF statement, and acts as an alternative action in case the IF test fails.
<exp> is the expression-action-cause.
<state-line> is the statement or line number.

REMARK: IF-THEN-ELSE statements may be nested, but the number of IFs and ELSEs must match with each other.

EXAMPLE:

```
    :
    20 IF A=1 THEN 60 ELSE 40
```

In this example, if A=1 then control branches to line 40. If the ELSE clause is not used and A is not equal to 1, the computer will go to the next statement instead of branching to line 40.

Another example:

```
    20 IF A<B THEN GOTO 60 ELSE PRINT "A>B"
```

2.41 INKEY$

SYNTAX: <variable-name>=INKEY$

EXPLANATION: This statement returns a one-character string determined by an instantaneous input from the keyboard. If no key is pressed during the execution of this statement, a null string is returned.
Characters typed to an INKEY$ are not automatically displayed on the screen.

EXAMPLE:

```
:
50 A$=INKEY$: IF A$="G" THEN 90 ELSE 120
:
```

2.42 INPUT

SYNTAX: INPUT <item-list>
or
INPUT "prompt-statement";<item-list>

EXPLANATION: This statement causes the computer to suspend execution of a program and wait until the user has input the specified number and type of values through the keyboard. Input values can be string or numeric according to the variable type. The items (if more than one) in the list must be separated by commas.

REMARK: When the computer executes this statement, it dispalys a "?" and waits for the "user" to type in the data.

EXAMPLE:

```
:
60 INPUT A$,G$,C,J
```

this statement read two strings and 2 numbers

```
:
100 INPUT "ENTER ONE NUMBER";A
```

this example issues a prompt, reads a number.

2.43 INPUT#

SYNTAX: INPUT#<file-number>, <variable-list>

EXPLANATION: This statement causes data to be transferred from the sequential disk file, converting the data to number or string as specified by the type of the "variables", <variable-list>. <file-number> is the number used when the file was OPENed for input. <variable-list> is the list of variable names to contain the data from the file.

EXAMPLE:

```
:
5Ø INPUT#1,A,B,C
```

2.44 KILL

SYNTAX: KILL <file-name>

EXPLANATION: This active command deletes the specified file.

EXAMPLE:

```
KILL ABC/BAS
```

2.45 LET

SYNTAX: LET <variable>=expression

EXPLANATION: This statement is used to assign a value to a variable. The word LET is not required in assignment statements by the Video Genie BASIC interpreter. However, the user may use the word LET in order to make the program compatible with other systems.

EXAMPLE:

```
:
5Ø LET A=2*Y/(AMT+RATE)
:
```

2.46 LINE INPUT

SYNTAX: LINE INPUT <"prompt-message">,<name>

EXPLANATION: This statement is used to input a line of characters from keyboard. The <"prompt-message"> is a string that is printed at the terminal before input(<name>) is received. <name> is the assigned name for the line to be typed in.

NOTE: This statement inputs a line from the keyboard without question mark indicated from the screen. The only way to terminate the string input is to press NEWLINE.

EXAMPLE:

```
:
50 LINE INPUT "IDENTITY NUMBER, NAME", A$
:
```

This statement is used to display the prompt message (IDENTITY NUMBER, NAME) and waits for input data.

2.47 LINE INPUT#

SYNTAX: LINE INPUT#<file-number>,<string-variable>

EXPLANATION: This statement reads a line of string data from a sequential disk data file into a string variable (<string-variable>). <file-number> is the number under which the file was OPENed. <string-variable> is the variable name to which the line will be assigned.

EXAMPLE:

```
:
60 LINE INPUT#1, A$
```

2.48 LIST

SYNTAX: (a) LIST
(b) LIST <line-number>
(c) LIST <line-number>-<line-number>

EXPLANATION: This active command will inform the computer to display any specified program lines stored in the main memory. If the option is not used, the computer will scroll the entire program onto the display. In order to pause and examine the text, the user should hit the SHIFT and CONTROL keys simultaneously. The scrolling will continue by hitting any key.

EXAMPLE:

```
    LIST
```
This command displays all lines in the momory

```
    LIST 50
```
This command displays line 50

```
    LIST 20-60
```
This command displays from line 20 to line 60

2.49 LLIST

SYNTAX: (a) LLIST
(b) LLIST <line-number>
(c) LLIST <line-number>-<line-number>

EXPLANATION: This command lists a program onto the printer.This command funtions in a very similar way as the LIST command. The options are perform in the same way as LIST except it is list onto the printer.

REMARK: If the Line printer is not properly connected, the computer will enter a dead loop and waits to print the first character. This situation can only be resolved by turning the print on or hitting the RESET button.

2.50 LOAD

SYNTAX: (a) LOAD <program-name>
(b) LOAD <program-name>,R

EXPLANATION: This command loads a program with the name <program-name> from the disk into main memory. The R option is to RUN the program after it is loaded. The LOAD command erases the resident program, deletes all variables, and closes all open files. If the R option is used with LOAD, all opened files are kept open. Therefore, LOAD with R-option can be used to chain several programs.

EXAMPLE:

```
LOAD "ARST/BAS",R
```

2.51 LPRINT

SYNTAX: LPRINT <item-list>

EXPLANATION: This command (or statement) prints a file onto the printer. LPRINT is function similar to a PRINT except insteasd displays on the screen, LPRINT prints onto the printer.

REMARK: If the line printer is not properly connected, the computer will enter a dead loop and will wait to print the first character. This situation can only be resolved by turning on the printer or hitting the RESET button.

EXAMPLE:

```
:
60 LPRINT "THE VALUE OF AMT. IS";A
:
```

2.52 LSET

SYNTAX: LSET <string-variable>=<string-expression>

EXPLANATION: This statement moves character-string data to a random buffer field and adds blanks on the right to fill field. <string-variable> is the field name. <string-expression> is the character-string data to be placed into <string-variable>.

EXAMPLE: Let ID$ be a field name for a random file buffer with a length of 10 characters.
Then the statement:

```
50 LSET ID$="S-073"
```

will put the data in the buffer as follows:

```
S-073#####
```

where "#" is blank space.

NOTE: If the string is too long for the field, characters are dropped from the right.

2.53 MERGE

SYNTAX: MERGE <filename>

EXPLANATION: This command is to merge a specified disk file with the resident program (program currently in memory.)

MODIFICATION: MERGE has been modified to used as a programming statement. If it is used as a programming statement, then:

(a) MERGE statement must not be
- part of a DEFFN statement;
- part of a sub-routine;
- part kof a FOR-NEXT loop.

(b) MERGE statement must be
- last statement of the text line.

(c) the merge file must not contain a line whose number is the same as the number of a text line existing at the start of the execution of the merge.

(d) execution will continue after the MERGE statement.

REMARK: MERGE protects all variables and closes all files. The user must make sure that there is enough momery space for the "merge".

EXAMPLE:

(a) An active command

```
MERGE "SUB/TXT"
```

(b) A programming command

```
:
40 MERGE "SUB/TXT"
```

2.54 MID$

SYNTAX: MID$(<name-$>,<n>,<m>)=<exp$>

EXPLANATION: This MID$ function replaces any part of a string with a substring. <name-$> is the name of string. <n> is the beginning position for the substitution. <m> is the number of characters to be replaced. <exp$> is the replacement string. If <exp$> is too long to fit in <name-$>, the extra characters at the right of <exp$> are neglected.

EXAMPLE:

Let A$="CDEFGH", then the statement

```
40 MID$ (A$,2,3,)="5678"
```

will set A$ to the string of "C5678GH"

2.55 NEW

SYNTAX: NEW

EXPLANATION: This active command with no argument will clear all program lines; reset numeric variables to zero and string variables to null. It does not change the memory size previously set by the CLEAR command.

2.56 ON ERROR GOTO

SYNTAX: ON ERROR GOTO <line-number>

EXPLANATION: This statement allows the user to set up an error-trapping routine to recover a program from an erorr and to continue, without any break in execution. Without this statement, the computer will stop execution and print out an error message, once it encounters any kind of error in the user's program.

EXAMPLE:

```
:
60 ON ERROR GOTO 450
:
```

2.57 ON-GOTO

SYNTAX: ON <n> GOTO <list-of-line-number>

EXPLANATION: This statement allows multi branching to the line numbers specified according to the value of <n>. <n> is an expression. The value of the expression must be between 0 to 255 inclusive.

REMARK: When ON-GOTO statement is executed, first, the expression is evaluated and the integer portion, that is INT (expression) is obtained. Then the computer assigns this integer to N, and counts over to the Mth element in the line number list, and then branches to the line number specified by that element.

If N is greater than the available line number M, the control fall through to the next statement in the program. If the expression or number is less than zero, an error will occur.

EXAMPLE:

```
:
60 INPUT "ENTER MODE"; C
70 ON C GOTO 200,300,400,500
:
```

2.58 ON-GOSUB

SYNTAX: ON <expression> GOSUB <line-number-list>

EXPLANATION: This statement works like ON-GOTO, except control branches to one of the subroutines specified by the line numbers in the line numbers list.

EXAMPLE:

```
:
60 INPUT "ENTER 1,2, OR 3";N
70 ON N GOSUB 250,400,600
:
```

2.59 OPEN

SYNTAX: OPEN <m>,<file-name>,<file-specify>

EXPALNATION: This statement opens a file for access (allow I/O to a disk file. <m> is string expresion to specify the mode in which the file is to be opened. <file-name> is the number from 1 to 15, the buffer to be assigned. <file-specify> is the file specification.

REMARK: If the mode <m> is -

I	then it is in sequential input mode
O	then it is in sequential output mode
R	then it is in random Input/Output mode

EXAMPLE:

```
:
200 OPEN "I",1,"ABC/BAS"
:
```

this example opens the file "ABC/BAS" for sequential input.

2.60 PRINT

SYNTAX: PRINT <item-list>

EXPLANATION: This statement displays an item or list of items on the screen. The item can be any of the following:
numeric constants, numeric variables, string constants, string variables, or expressions.

REMARK: Items in the item list may be separated by commas or semi-colons. If commas are used, the corsor automatically advances to the next printing zone before printing the next item. If semi-colons are used, no space is inserted between alphabetic items before printing on the display, but one space is inserted before each numeric item.

EXAMPLE:
```
:
PRINT "THE AMOUNT IS"; A
```

2.61 PRINT TAB

SYNTAX: PRINT TAB<expression>

EXPLANATION: This statement allows the user to print at any specified cursor position within a line. More thab one TAB in a PRINT statement is acceptable. However, the value in the expression should be between Ø and 63 inclusive.

NOTE: If it is 8Ø-mode BASIC, then the value of the <expression> must be between Ø to 255 inclusive.

EXAMPLE:
```
:
5Ø PRINT TAB(5) "POS=5"
```

2.62 PRINT USING

SYNTAX: PRINT USING <format>,<item-list>

EXPLANATION This statement allows the user to print the data with a pre-defined format. The data can be numeric or string values.
<format> and <item-list> can be expressed as variables or constants. The statemet prints the item list according to the format specified.

REMARK: The folllowing specifiers may be used in the format field:

(I) Numeric:

(a) The sign # represents the proper position of each digit in the item list.
If the format field is greater than the numeric value (in the <item list>), the unused field positions to the left of the number will be dispalyed as spaces and those to the right of the decimal point will be dispalyed as zeros.

(b) The decimal point, ".", can be placed anywhere in the format field established by the #. Rounding off will take place if the digits to the right of the decimal point are suppressed.

(c) The comma, ",", can be placed at any position between the first digit and the decimal point. The comma will be displayed to the right of every three digits.

(d) The two asterisks, **, placed at the beginning of the format field will cause all unused positiosns to the left of the decimal point to be filled with asterisks.

(e) Two dollar signs , $$, palced at the beginning of the field will act as a floating dollar sign.

(f) Combines the ** and $$ will fill empty spaces with * and $ will occupy the first position before the number.

(g) When a "+" sign is placed at the beginning or at the end of the format field, the computer will print a + sign for a positive number or a - sign for a negative number at the specific position accordingly.

(h) When a "-" sign is placed at the end of the format field , it will causes a negative sign to be printed after any negative number, and will display as a blank for positive numbers.

(II) String

(a) % space % : to define a string field of more than one character. The length of the format field will be 2 plus the number of spaces between the percentage signs. An exclamation mark (!) informs the computer to use only the first character of the current string value.

(b) By using the ! sign, we can also concatenate, or join strings together.

NOTE: In the case of numeric string: the % sign will be automatically printed out if the field is not large enough to contain the number of digits found in the numeric value.

EXAMPLE:

The following tables show some of the format and its correspondent display.

(a) Numeric:

Let the number be 12.34

format	display
##.##	12.34
#.##	%2.34
**##.##	**12.34
$$#.##	$12.34
**$###.##	***$12.34
**$###	***$12
+##.##	+12.34

(b) String

Let the string be "ABCDE"

format	display
% %	AB
% %	ABC

2.63 PRINT@

SYNATX: PRINT@ <location>,<item-list>

EXPALNATION: This statement prints out items in the items list at the screen location specified. The "@" sign must follow PRINT immediately, and the location specified must be a number of value from Ø to 1Ø23.

NOTE: If it is 8Ø-mode BASIC, then the location specified must be a number from Ø to 1919.

REMARK: IF the user constructs a PRINT@ statement to print on the bottom line of the display, there will be an automatic line-feed, causing everything displayed to move up one line. To suppress this action, add a semi-colon at the end of the statement.

EXAMPLE:

```
:
5Ø PRINT @1ØØ "LOC 1ØØ"
```

2.64 PRINT#

SYNTAX: PRINT# <file-number>, <list-of-expression>

EXPALNATION: This statement prints data sequentially to the specified disk file. The file must have been opened with the OPEN statement. <file-number> is the number from 1 to 15, specifies a sequential output file buffer. <list-of-expression> is the list of expression to be transferred and written to the disk.

EXAMPLE:

```
:
30 PRINT#1,A,B
```

This causes the string A and B to be written to the file buffer 1.

2.65 PUT

SYNTAX: PUT <file-number>,<reocrd-number>

EXPLANATION: This statement writes a record from a random buffer to a specified record-place in the file. The file must have been opened with the OPEN statement. <file-number> is the file buffer number from 1 to 15. <record-number> is the record number in the file, it ranges from 1 to 335. If <record-number> is excluded, the current record number will be used.

2.66 READ

SYNTAX: READ <item-list>

EXPLANATION: This statement instructs the computer to read in a value from a DATA statement and assign that value to the specified variable.

REMARK: The values in the DATA statement will be read sequentially by the READ statement. After all the items in the first DATA statement have been read, the next READ statement encountered will access the second DATA statement for the next variable. If there is no more value in the DATA statement available for a READ statement an Out-of-Data error will occur.

EXAMPLE:

```
:
5Ø READ A$
6Ø IF A$ = "EOF" GOTO 1ØØ
```

2.67 REF

SYNTAX: REF<format>

EXPLANATION: This active statement displays where a line number, a variable, a string, an integer, a function code, a packed sequence of characters or an unpacked sequence of characters is referenced.

If the <format> is:

(a) * ,then full reference list for all line number, integers and variables will be displayed.

(b) $, same as * except that it prints on the printer.

(c) <nn>, where <nn> is a variable name with two characters only and without any suffix. It displays all references to the variable named <nn>.

(d) <num>, where <num> is the line number. It displays all references to the line number <num>.

REMARK: In order to terminate the REF command, press UP ARROW key. Also press BREAK key to pause and press NEW LINE to continue.

2.68 REM

SYNTAX: REM <any comment or sentences>

EXPLANATION: REM represents remarks. This statement informs the computer that the rest of the line only consists of comments, and should be ignored. If REM is used in a multi-statement program line, it must be the last statement.

EXAMPLE:

```
:
30 REM  * THIS IS A COMMENT *
```

2.69 RENEW

SYNTAX: RENEW

EXPLANATION: This active command replaces back a program erased by the command NEW. The work space of all program, variables, and data will be reinstated.

REMARK: After the NEW command, if any text line is loaded or created, then the previous text is not reinstated.

2.70 RENUM

SYNTAX:
(a) RENUM,
(b) RENUM <start>,<increment>
(c) RENUM <start>, ,<from>,<end>
(d) RENUM <start>,<increment>,<from>,<end>

EXPLANATION: This active command renumber the BASIC program.

(a) RENUM, : renumbers the whole program, starts with line number 10 and increment of 10.

(b) RENUM <start>,<increment> : renumbers the whole porgram, starts the first line with the number <start> and increment of <increment>.

Example: If the command is
RENUM 100,100
then the program line numbers will be 100, 200, 300 and so on.

(c) RENUM <start>, ,<from>,<end> :
Example:
RENUM 200, ,400,600
this example renumbers the program line, from 400 to 600 inclusively, the first line is set to be 200 and increment by 10.

(d) RENUM <start>,<increment>,<from>,<end>:
Example:
RENUM 100,20,10,200
this example renumbers the line 10 to line 200 inclusively, then start with the line number 100 and increment by 20.

2.71 RESUME

SYNTAX: RESUME <line-number>

EXPLANATION: This statement terminates an error handling routine by specifying where normal execution is to resume.

REMARK: RESUME Ø or RESUME without a line number causes the computer to return to the statement in which the error occured. IF RESUME is followed by a line number, it causes the computer to branch to the line number provided.
RESUME NEXT causes the computer to branch to the statement followed the point at which the error occured.

EXAMPLE:

```
:
15Ø RESUME 5Ø
:
```

2.72 RESTORE

SYNTAX: RESTORE

EXPALNATION: This statement with no argument allows the next READ statement to access the first item in the first DATA statement, and the subsequent items.

EXAMPLE:

```
:
5Ø PRINT A$,A
6Ø RESTORE
7Ø READ B$,B
:
```

2.73 RETURN

SYNTAX: RETURN

EXPLANATION: This statement ends a subroutine and returns control to the statement that immediately follows the GOSUB. An error will occur if RETURN is encountered without execution of a matching GOSUB.

EXAMPLE:

```
:
:
20 GOSUB 100
:
:
100 PRINT "SUBROUTINE"
110 RETURN
:
```

2.74 RSET

SYNTAX: RSET <string-variable>=<string-expression>

EXPLANATION: This statement moves character-string data to a random buffer field and adds blanks on the left to fill field. <string-variable> is the field name. <string-expression> is the character-string data to be placed into <string-variable>.

EXAMPLE: Let ID$ be a field name for a random file buffer with the length of 10 characters.
Then the statement:

```
RSET ID$="S-073"
```

will put the data in the buffer as follows:

```
#####S-073
```

where "#" is the blank.

NOTE: If the string is too long for the field, characters are dropped from the right.

2.75 RUN

SYNTAX: RUN
or
RUN <line-number>

EXPLANATION: This command will instruct the computer to start executing(or RUN) the user's program stored in main memory. If a line number is not specified, the computer will start executing from the lowest line number.
However, if a line number is provided, the computer will execute from the given line number to higher order lines.

NOTE: An error will occur if an invalid line number is used.
Everytime a RUN is executed, a CLEAR command also executed automaticlly before it.

EXAMPLE: RUN

2.76 RUN "program"

SYNTAX: RUN <program-name>
or
RUN <program-name>,R
or
RUN <program-name>,V

EXPLANATION: This command is to load a file from disk and executes it. If R option is used, all open files continue open.
RUN without the R option closes all open files.
If V-option is used, all open files and variables will be preserved; excepting the DEFFN variable.

2.77 SAVE

SYNTAX: SAVE <program-name>
or
SAVE <program-name>,A

EXPLANATION: This command is used to save a program file on disk. If the file name (<program-name>) already exits, its previous content will be lost as the file is re-created. The A option is used to save the program file in ASCII format. Without A option, the program is saved in compressed-format.

EXAMPLE:

```
SAVE "ARTS/BAS"

SAVE "ARTS/BAS",A
```

2.78 STOP

SYNTAX: STOP

EXPLANATION: This statement is essentially a debugging aid. It sets a break point in a program during execution, and allows the user to examine or modify variable values.
A message will be printed out as "BREAK IN line number" once the computer executes the STOP statement. The active Command CONT can then be used to re-start execution at the point where it breaks.

EXAMPLE:

```
:
50 A=B+C
60 STOP
:
```

2.79 TROFF

SYNTAX: TROFF

EXPLANATION: This active command will turn off the Trace function. Usually follows the TRON command.

2.80 TRON

SYNTAX: TRON

EXPLANATION: This command will turn on a Trace function that allows the user to keep track of the program flow for debugging and execution analysis. Everytime the computer executes a new program line, the line number will be displayed inside a pair of brackets.

REMARK: In order to pause execution before its natural end, the SHIFT key and @ key must be pressed simultaneously. To continue, just press any key.
To turn off the Trace function, enter TROFF, TRON and TROFF are available for use within user programs to check if a given line is executed.

EXAMPLE:

```
:
:
90 IF A=B THEN 160
100 TRON
110 A=B+C
120 TROFF
```

CHAPTER THREE

GENIE III BASIC FUNCTION

This chapter is a complete description of all the functions available in the GENIE III System.

In the syntax and description of some functions we use the term X to indicate a "numeric argument". A numeric argument can be a numeric variable, constant, or expression. The term <string> is a string argument. A string argument can be a string variable, constant, or expression.

When using these functions, all arguments must be enclosed in parentheses.

3.1 &H

SYNTAX: &H<hexa>

DESCRIPTION: This hex (base 16) constant returns the decimal equivalent of the hexadecimal value. <hexa> is hexadecimal numerals Ø,1,2,..... 9,A,....F. The following table shows the hex constant and its decimal equivalent.

Hex Constant	Decimal
&H1	1
&H2	2
:	:
:	:
&H7FFF	32767
&HØØØ	-32768
:	:
:	:
&HFFFE	-2
&HFFFF	-1

EXAMPLE:

```
40 PRINT &H7FFF
```

this statement prints the decimal equivalent of the hex constant which is equal to 32767.

3.2 &O

SYNTAX: &O<octal>

DESCRIPTION: This octal (base 8) constant returns the octal equivalent of the octal value. <octal> is octal numerals Ø,1,....7. The following table shows the octals constant and its decimal equivalent.
NOTE that the O from the octal constant can be excluded, so &O <octal> = &<octal>

Octal Constant	Decimal
&1	1
&2	2
:	:
:	:
&77777	32767
&1ØØØØØ	-32768
&1ØØØ1	-32767
:	:
:	:
&177777	-1

EXAMPLE:

```
60 PRINT &51ØØØ
```

this statement prints the decimal equivalent of octal constant which is equal to 2Ø992.

3.3 ABS

SYNTAX: ABS(X)

DESCRIPTION: This function returns the absolute value of the argument X. (It returns a positive number equal to the magnitude of X.)

EXAMPLE:

```
50 A=ABS(-3-4/700)
```

3.4 ASC

SYNTAX: ASC (<string>)

DESCRIPTION: This statement returns the ASCII code (in decimal) for the first character of the specified string. The string specified must be enclosed in parentheses.

NOTE: A null-string will causes an error to occur.

EXAMPLE:

```
300 PRINT ASC("H")
:
```

3.5 ATN

SYNTAX: ATN (X)

DESCRIPTION: This function returns the arctangent function of the argument, where X is expressed in radians. To get the arctangent in degrees, multiply ATN(X) by 57.29578.

EXAMPLE:

```
:
30 DDH=ATN(AMT)
:
```

3.6 CDBL

SYNTAX: CDBL (X)

DESCRIPTION: This function returns a double-precision representation of the argument. The value returned contains 17 digits, however, only the digits contained in the argument will be significant.

EXAMPLE:

```
:
60 A=CDBL(34*AMT)
:
```

3.7 CHR$

SYNTAX: CHR$(<expression>)

DESCRIPTION: This statement works as the inverse of the ASC function, that is to return the character of the specified ASCII, control or graphics code. The argument may be any number from Ø to 255, or any variable expression with a value within that range. The argument must be enclosed in parentheses.

EXAMPLE:

```
:
5Ø PRINT CHR$(33)
:
```

3.8 CINT

SYNTAX: CINT(X)

DESCRIPTION: This function returns the largest integer that is not greater than the argument. The argument must be within the range of -32768 to +32768.

EXAMPLE:

```
     :
     5Ø A=CINT(3.666)
```

this statement sets A equal to 3

For the statement:

```
     5Ø B=CINT(-3.666)
```

this statement sets B equal to -4.

3.9 CLS

SYNTAX: CLS

DESCRIPTION: This function clears the entire display by turning off all the graphics blocks. It also moves the cursor to the upper left corner. This function allows the user to present an outstanding display on the screen, without any symbol previously displayed.

EXAMPLE

```
:
4Ø CLS
```

3.10 COS

SYNTAX: COS(X)

DESCRIPTION: This function returns the cosine function of the argument (in radians). In order to obtain the cosine of X when X is in degrees, use COS(X*Ø.Ø174533).

EXAMPLE:

```
:
5Ø A=3Ø+COS(BBC)
:
```

3.11 CSNG

SYNTAX: CSNG(X)

DESCRIPTION: This function returns a single-precision representation of the argument. It returns a 6 significant digits number with 4/5 rounding for a double precision argument.

EXAMPLE:

```
:
6Ø A=CSNG(AMT)
:
```

3.12 CVD

SYNTAX: CVD (<8-byte-string>)

DESCRIPTION: This function replaces a 8-byte string with a double precision number after it is read from disk.

EXAMPLE:

```
:
A=CVD(NB$)
:
```

3.13 CVI

SYNTAX: CVI (<2-byte-string>)

DESCRIPTION: This function replaces a 2-byte string with an integer, after it is read from disk.

EXAMPLE:

```
:
60 Y=CVI(AB$)
:
```

3.14 CVS

SYNTAX: CVS (<4-byte-string>)

DESCRIPTION: This function converts a 4 character string to a single precision number after it is read from disk.

EXAMPLE:

```
:
KY=CVS(BYE$)
:
```

3.15 EOF

SYNTAX: EOF (<file-number>)

DESCRIPTION: This function is used to test for end-of-file during read. INPUT-PAST-END error during sequential input can be avoided by using EOF function.

REMARK: EOF (file-number) returns -1(true) if the end of a sequential file has been reached. A zero (false) is returned for "EOF record has not yet been read".

EXAMPLE:

```
:
40 IF EOF (4) THEN 200
:
```

3.16 EXP

SYNTAX: EXP(X)

DESCRIPTION: This function returns the "natural exponential" of X. This is the inverse of LOG function. In other words, it returns a number equal to the constant "e" (approximately 2.71828183) raised to the power of N.

EXAMPLE:

```
:
70 AMT=EXP(5.1)
:
```

then AMT equal to the value 164.002

3.17 FIX

SYNTAX: FIX (X)

DESCRIPTION: This function returns a truncated representation of the argument with all digits on the right of the decimal point being truncated or chopped off.

EXAMPLE:

```
:
40 PAYM=FIX(SYS)
```

If SYS is equal to 3.465 then PAYM will be equal to 3.

3.18 INP

SYNTAX: INP (<port-number>)

DESCRIPTION: This statement inputs a 8-bit value from the specified port. The Video Genie System is capable of handling 256 ports, numbered from Ø to 255. Usually this function is used only when the expression box is installed.

EXAMPLE:

```
:
6Ø A=INP (124)
:
```

3.19 INSTR

SYNTAX: INSTR (<n>,<expression-1>,<expression-2>)

DESCRIPTION: The INSTR function is used to search through a string to see if it includes another string. <n> is the position in <expression-1> where the search is to start. If <n> is not given then it is assumed to be 1. <expression-1> is the string to be search. <expression-2> is the substring required to search for. This function returns the starting position of the substring in the object string.
If zero is returned, it means that the string does not contain another string.

EXAMPLE:

For a string XY$ = "DEFGH"
The statement

```
   2Ø INSTR (XY$,"EFG")
```

returs 2

While the statement

```
  5Ø INSTR (XY$,"12")
```

returns Ø

Another example:
The statement

```
  7Ø INSTR (2, "2345234","23")
```

RETRUNS 5.

3.20 INT

SYNTAX: INT(X)

DESCRIPTION: This function returns an integer representation of the argument, using the largest integer that is not greater than the argument.
The argument is not limited to the range -32768 to +32768.

EXAMPLE: If Y=34.6 then the statement

```
50 A=INT(Y)
```

will set A=35

3.21 LEFT$

SYNTAX: LEFT$(<string>,<n>)

DESCRIPTION: This function returns the first n characters of the specified string. The arguments must be enclosed in parentheses. String may be a constant or an expression, and n may be a numeric expression.

EXAMPLE: For a string A$="ABCDEFG"
The statement

```
50 B$=LEFT$(A$,4)
```

will set B$="ABCD"

3.22 LEN

SYNTAX: LEN (<string>)

DESCRIPTION: This function returns the length of the specified string. The string may be a variable, expression or constant and must be enclosed in parentheses.

EXAMPLE: For the string A$="ABCDEFG"
The statement

```
Y=LEN(A$)
```

will set Y=7

3.23 LOC

SYNTAX:
(1) LOC (<file-number>)
(2) LOC (<file-number>)$
(3) LOC (<file-number>)%
(4) LOC (<file-number>)!
(5) LOC (<file-number>)#

where <file-number> is the file's buffer number, range from 1 to 15. The file must have been opened with OPEN statement before using this function.

DESCRIPTION:

Format (1): LOC (<file-number>)
This LOC function returns the number of the previous record GET or PUT for that file area. The return number is an integer, range from 1 to 32767.

Example:

```
:
30 PUT 2,46
40 X=LOC(2)
:
```

this example will set X equal to 46

Format (2): LOC (<file-number>)$

(I) For record segment files-

(a) return -1 (IF statement is true)

(i) if the start of the next record is greater than or equal to EOF;

or (ii) the current file position is greater than or equal to EOF.

(b) return Ø (IF statement false)

(i) if the start of the next record is less than EOF;

or (ii) the current file position is less than EOF.

(II) For non-record segement file and print/input files-

(a) returns -1 (IF statement true) if the current file positioning is greater than or equal to EOF,

(b) return Ø (IF statement false) if less than EOF.

Example:

```
:
40 IF LOC(1)$ THEN 340
:
```

the program will branch to line 340 if the next record is located at or beyond the file's EOF.

Format (3): LOC(<file-number>)%

When using Format (3), this function returns an Relative Byte Address equal to the file's EOF.

Example:
let the file contains 4200 bytes
then

```
:
200 Y=LOC(1)%
:
```

will set Y equal to the value 4200

Format (4): LOC(<file-number>)!

(a) Record segment files-

(i) If Remembered Record Address is valid, this funtion returns a Relative Byte Address value equal to the location of the file's next record.

(ii) If Remembered Record Address is invalid, this function returns a Relative Byte Address value equal to the current file position.

(b) Non-record segment files and print/input files- this function returns an Relative Byte Address equal to the current file position.

Example:
If the latest processed record for file's buffer 4 at relative file position 1234 and the next record starts at relative file position 2000, then

```
:
40 AAY=LOC(4)!
:
```

will set AAY equal to 2000.

Format (5): LOC(<file-number>)#

If the Remembered Record Address is valid, then this function returns an Relative Byte Address value equal to Remembered Record Address. If Remembered Record Address is invalid, an error will occur.

Example: If the latest processed record for files' buffer 4 at relative file position 1234 and the next record starts at relative file posiition 2000, then

```
:
400 A=LOC(4)#
:
```

will set A equal to 1234.

3.24 LOG

SYNTAX: LOG (X)

DESCRIPTION: This function returns the natural logarithm of the argument.

NOTE: X must be greater than Ø.

EXAMPLE:

```
:
Y=LOG(4.5)
:
```

then Y is equal to the value 1.5Ø4Ø8.

3.25 LOF

SYNTAX: LOF(<file-number>)

DESCRIPTION: This function returns the number of the last record in a file.

EXAMPLE:

```
:
3Ø IF NUM>LOF(1) THEN PRINT "INCOMPLETE"
```

3.26 MEM

SYNTAX: MEM <expression>

DESCRIPTION: This function returns the number of unused and unprotected bytes in memory.

EXAMPLE:

```
:
60 IF MEM<456 THEN 600
:
```

NOTE: When used as a command, it must be accompanied with the PRINT command. That is PRINT MEM, to find out the amount of memory not being used to store program, variables, strings, arrays, etc.

3.27 MID$

SYNTAX: MID$(<string>,<P>,<n>)

DESCRIPTION: This function returns a substring of string starting at position <p>, with length <n>. The string, position and length must be enclosed in parentheses. String may be a constant or an expression, <p> and <n> may be numeric expressions or constants.

EXAMPLE: Let the string A$="ABCDEFG"
Then the statement

```
60 B$ = MID$(A$,3,4)
```

will set B$="CDEF"

3.28 MKD$

SYNTAX: MKD$(<double-precision-expression>)

DESCRIPTION: This function replaces a double precision value with an 8-byte string values.

EXAMPLE:

```
:
20 LSET M$=MKD$(AMT)
```

3.29 MKI$

SYNTAX: MKI$(<integer-expression>)

DESCRIPTION: This function converts integer value to a 2 byte string. The integer ranges from -32768 to +32768.

EXAMPLE:

```
:
30 D$=MKI$(PER)
:
```

3.30 MKS$

SYNTAX: MKS$(<single-precision-expression>)

DESCRIPTION: This function replaces single precision value with a 4-byte string value.

EXAMPLE:

```
:
20 LSET ABC$=MKS$(BUY)
:
```

3.31 OUT

SYNTAX: OUT<port-number>,<value>

DESCRIPTION: This function outputs an 8-bit value to the specified port. This function requires two arguments: port-number and the value. The Video Genie System is capable of handling 256 ports, numbered from 0 to 255.

EXAMPLE:

```
:
30 OUT 14, 240
:
```

3.32 PEEK

SYNTAX: PEEK <address>

DESCRIPTION: This function returns the 8-bit value stored at the specified decimal address in the computer's memory, and displays the value in decimal form. The value will be between Ø-255.

EXAMPLE:

```
:
2Ø B=PEEK(3ØØØ)
:
```

3.33 POINT

SYNTAX: POINT(X,Y)

DESCRIPTION: This function examines the specified graphics block to see whether it is ON or OFF. X and Y are the two coordinates.
If the block is ON (has been SET), then POINT returns a binary True (-1). If the block is OFF, POINT returns a binary False (Ø).

NOTE: If the system diskette is 64 mode:
The display is divided up into a 128 (horizontal) by 48 (vertical) grid. The X-coordinates are ranged from Ø to 127, organized from left to right. The Y-coordinates are ranged from Ø to 47, organized from top to bottom.
If the system diskette is 8Ø-mode:
The X-coordinates ranging from Ø to 159 is counted from left to right. The Y-coordinates ranging from Ø to 7Ø is counted from top to bottom.

EXAMPLE:

```
:
4Ø A=POINT(6,7)
```

For this statement, if point (6,7) has been set, then A has the value of -1. Otherwise A has the value of Ø.

3.34 POKE

SYNTAX: POKE <address>,<value>

DESCRIPTION: This statement sends a 8-bit value to the specified (decimal) memory address location. It requires two argument: address and value. The value must be between 0-255.

EXAMPLE:

```
:
40 POKE 400, A
:
```

this statement sends the value of A to the address 400.

3.35 POS

SYNTAX: POS(<dummy-argument>)

DESCRIPTION: This function takes a dummy numeric argument and returns a number from 0 to 63 indicating the current cursor position on the display. Usually, 0 is used for the dummy argument.

EXAMPLE:

```
:
400 B=POS(0)
:
```

3.36 RANDOM

SYNTAX: RANDOM

DESCRIPTION: This function causes the computer to generate a new set of random numbers every time when the computer is turned on and runs a program which has RND functions.
No argument is needed in this function.

3.37 RESET

SYNTAX: RESET (X,Y)

DESCRIPTION: This function turns off a graphics block on the display at the location specified by the coordinates X and Y. This function has the same limits and parameters as SET(X,Y).

NOTE: If the system diskette is 64 mode:
The display is divided up into a 128 (horizontal) by 48 (vertical) grid. The X-coordinates are ranged from Ø to 127, organized from left to right. The Y-coordinates are ranged from Ø to 47, organized from top to bottom.
If the system diskette is 8Ø-mode:
The X-coordinates ranging from Ø to 159 is counted from left to right. The Y-coordinates ranging from Ø to 7Ø is counted from top to bottom.

EXAMPLE:
```
:
5Ø RESET (3Ø,6Ø)
:
```

3.38 RIGHT$

SYNTAX: RIGHT$ (<string>,<n>)

DESCRIPTION: This function returns a string made-up of the right most characters of the string <string>. If <n> is not an integer, only the integer part is used.

REMARK: If the length of the string is less than or equal to <n>, the entire string is returned.

EXAMPLE:

```
:
100 BBC$=RIGHT$("ABCDE",3)
:
```

This example sets BBC$ to the string "CDE".

3.39 RND

SYNTAX: RND(<number>)

DESCRIPTION: This function returns a pseudo-random using the current pseudo-random number (generated internally and has not access to the user).

REMARK: RND(0) returns a single-precision value between 0 and 1.
RND (X) returns an integer between 1 and X inclusive. X must be positive and less than 32768.

EXAMPLE:

```
:
50 B=RND(39)
:
```

3.40 SET

SYNTAX: SET(X,Y)

DESCRIPTION: This function turns on the graphics block on the display at the location specified by the coordinates X and Y. The argument X and Y may be numeric constants, variables or expressions. Since the SET (X,Y) function uses only the integer portion of X and Y, neither argument need be an integer.

NOTE: If the system diskette is 64 mode:
The display is divided up into a 128 (horizontal) by 48 (vertical) grid. The X-coordinates are ranged from 0 to 127, organized from left to right. The Y-coordinates are ranged from 0 to 47, organized from top to bottom.
If the system diskette is 80-mode:
The X-coordinates ranging from 0 to 159 is counted from left to right. The Y-coordinates ranging from 0 to 70 is counted from top to bottom.

EXAMPLE:

```
:
50 SET (23,45)
:
```

This statement will display a point at (23,45).

3.41 SGN

SYNTAX: SGN(X)

DESCRIPTION: This is a "sign" function, that is to return -1 if X is negative, 0 if X is zero, and +1 if X is positive.

EXAMPLE:

```
:
200 A=SGN(-480)
:
```

then A is set to be -1

3.42 SIN

SYNTAX: SIN (X)

DESCRIPTION: This function returns a number equal to the sine of X, where X is expressed in radians.

EXAMPLE:

```
:
30 A=49
40 SA=SIN(A)
:
```

then SA is set to be -0.953753

3.43 SQR

SYNTAX: SQR (X)

DESCRIPTION: This function returns the square root of the argument X. X is number or expression.

REMARK: The argument X must not be a negative number.

EXAMPLE:

```
:
ASQR=SQR(A)
:
```

3.44 STR$

SYNTAX: STR$(<expression>)

DESCRIPTION: This function converts a constant or numeric expression into a string of characters.

EXAMPLE:
```
Let A=1234
     :
     40 A$=STR$(A)
     :
```
will set A$ to be the string "1234"

3.45 STRING$

SYNTAX: STRING$(<n>,<character-or-number>)

DESCRIPTION: This function returns a string which composed of <n> number of the specified character(s).

REMARK: If it is number from 0-255, then it will be treated as as SACII, control or graphics code.

EXAMPLE:
```
     :
     40 PRINT STRING$(3,"$")
```
three "$" signs will display on the screen.

```
     100 PRINT STRING$(10, 63)
```
ten "?" signs will display on the screen.

3.46 TAN

SYNTAX: TAN (X)

DESCRIPTION: This function returns the tangent function of the argument (in radians).

EXAMPLE:

```
:
40 ATAN=TAN(A)
:
```

3.47 TIME$

SYNTAX: TIME$

DESCRIPTION: This function comes into no argument and is used to provide the value of the date and time currently stored in the Real-time clock memory area.

REMARK: In order to display the value, a PRINT must put before the TIME$.

EXAMPLE:

```
:
40 PRINT TIME$
:
```

the date and time will display at the screen.

3.48 USR

SYNTAX: USR<n> (<argument>)

DESCRIPTION: This function calls a machine language subroutine. <n> indicates for which USR routine is being called, and corresponds with the previous defined DEFUSR statement for that routine. <n> ranges from Ø to 9. If <n> is excluded, zero is assumed. <argument> ranges from -32768 to +32768 and is passed as an integer argument to the routine.

REMARK: The subroutine entry address should be POKEd into location 16526 - 16527. The least significant byte should be in location 16526.
To pass the argument to the subroutine, the subroutine should immediately execute a CALL 0A7FH (call 2687 dec.). The argument will then be placed in registers HL.
To return to your BASIC program without passing any value back, a RET instruction should be executed.
To return a value, load the value into the HL register pair as a two-byte signed integer and execute a JP ØA9AH instruction. (ØA9HA = 2714 Decimal).
USR routine reserves 8 stack levels for the users' subroutine.

NOTE: Users who are not familar with machine language programming are not recommended to use this function.

EXAMPLE:

```
:
A=USR(A%)
:
```

3.49 VAL

SYNTAX: VAL (<string>)

DESCRIPTION: This function performs the inverse of the STR$ function; that is to return the numeric value of the characters in a string argument.

EXAMPLE:
```
Let A$="1234"
     :
     50 C=VAL (A$)
     :
```
this example converts the string A$ to the numeric value 1234, and assigns it to C.

3.50 VARPTR

SYNTAX: VARPTR (<variable-name>)

DESCRIPTION: This function returns an address of the specified <variable-name>.

REMARK: If K is the returned address, the variables will be stored in the following structure:

(i) 2-byte integer
- K- LSB
- K+1 -MSB

(ii) single precision variable
- K - LSB
- K + 1 - Next MSB
- K + 2 MSB
- K + 3 - Exponent value

(iii) double precision value
- K - LSB
- K + 1 - Next MSB
- :
- K + 6 - MSB
- K + 7 - Exponent value

(iv) string variable
- K - length of string.
- K + 1 - LSB of string starting address.
- K + 3 - MSB of string starting address.

CHAPTER FOUR

BASIC EDITOR IN GENIE III SYSTEM

There are two BASIC editors in GENIE III system. One comes from the Microsoft BASIC interpreter; the other is the newly introduced SCREEN EDITOR, which has the 64-mode and the 80-mode corresponding to the NEWDOS 64-mode and NEWDOS 80-mode. 64-mode means 64 x 16 display format (GENIE I/II compatible) while 80-mode means 80 x 24 display format.

4.1 LEVEL II BASIC EDITOR

This line editor can be used as the System stays in the level II BASIC. Recall that we may enter the Level II BASIC by depressing the keys, RESET - BREAK simultaneously. The system is able to perform all the editing fun ctions presented below:

4.1.1 GETTING INTO EDITING MODE

EDIT <line-number>

The EDIT command shifts the Active Command Level to the Editing command level. The <line-number> specifies the line number to edit.
Example:

```
>EDIT 100
```

then the display will be

```
>100 -
```

and it is in editing mode.

4.1.2 CURSOR MOVEMENT

Move right:
One space to right: Press the SPACE BAR key.
n spaces to right: Type in a number n and press the SPACE BAR key

Move left:
One space to left: Press the left arrow key (<-).
n spaces to left: Type in a number n and press the left arrow key (<-).

4.1.3 SEARCH

n S c

Type a number (n), press the letter "S" key, and type a character (c).

The command searches for the n-th occurrance of the character c on that line and moves the cursor to that position.
If n is omitted, it searches the first occurance of the character specified and stop the cursor there.
If c is not found, the cursor will move to the end of the line.

Example:

Consider the follwoing example:

```
100 IF A=B THEN 230 : A = A+1
```

Then after entering the Edit mode:

```
100 -
```

Type in 2S=, then the display should be

```
100 IF A=B THEN 230 : A -
```

Now, the user may enter one of the subcommands at the current cursor position. For instance:
Type in H (hack and insert) followed by "= A + 2" (new data).
Then the line will become:

```
100 IF A=B THEN 230 : A = A+2
```

4.1.4 LIST A LINE

L

Press L letter key.
While the computer is in the Editing mode, and is not currently executing one of the editing subscommands, the L command will list the remaining part of the line onto the display.

Example:

```
>EDIT 100
100 -
```

Hit the letter L (without hitting NEW LINE key), the display should be:

```
100 If A = B THEN 150 : A = A + 1
100 -
```

The second line allows the user to do editing, while referencing the first line.

4.1.5 DELETE

n D

One character: Position the cursor over the character and hit "D" key. Then press the NEW LINE key.

n Characters: Position the cursor over the first character. Type in the value, n, the number of characters to be deleted. Then press the NEW LINE key.

The deleted character(s) will be enclosed in exclamation marks "!" to show you which character(s) are being affected.

Example:
If the statement is
```
100 IF A=B THEN 100 : A=A+1
```
and we want to delete A=A+1.
First enter the editing level and position the cursor on the ":". It displays:
```
100 IF A=B THEN 100
```
Now type in 7D then the display:
```
100 IF A=B THEN 100 !: A=A+1!
```
Press the NEW LINE key.
The statement will become:
```
100 IF A=B THEN 100
```

Delete all characters up to the n-th occurance of the character C:

n K c

The command will delete all characters up to the n-th occurance of character C, and move the cursor to that position.

Example:
If the statement is:
```
100 IF A = B THEN 100 : A=A+2
```
We want to delete "IF A = B THEN 100 :" . Enter into the editing mode:
```
100 -
```
First type 1k:. Then the display:
```
100 !IF A = B THEN 100 !
```
In order to delete ":" also, type in D. Display :
```
100 !IF A = B THEN 100 :!!:!
```
Then press NEW LINE, the statement will set as
```
100 A=A+2
```

4.1.6 INSERT

(i) Insert inside the statement:

I key

Position the cursor at the insert point, then Press the I letter key. Type in the addition characters.

(ii) Insert at the end of the line:

X key

After enter into the editing level, press the X letter key. The whole line will display and the user can type in the characters.

(iii) Hack and Insert:

H key

Position the cursor at the "Hack and Insert" point and press the H letter key. The remainder of the line will be deleted and the user can start type in the insert character(s).

4.1.7 CHANGE

n C key

Position the cursor at the change position and type the number (n) of character(s) want to exchange and press the C letter key.
If the number n is not specified, the computer assumes the user only wants to change a single character.

4.1.8 ESCAPE FROM INSERT SUBCOMMAND:

SHIFT ESC key

By pressing the SHIFT and ESC keys simultaneously, the computer will escape from any of the following Insert subscommands: H,I,X.
After escaping from an Insert subscommand, the user remains in the editing level, while the current cursor position is unchanged.
Another way to escape from these Insert subcommands, is by pressing the NEW LINE key, which will shift the computer back to the Active Command Level.

4.1.9 CANCEL AND RESTART

A key

In the editing level, when the letter A is pressed, the cursor will move back to the beginning of the line and all the editing changes previously made on that line will be cancelled. The former content of the line will be restored.

4.1.10 BACK TO ACTIVE MODE

(i) Record all change and back:

NEW LINE

Once the user presses the NEW LINE key while in the Edit mode, the computer will record all the changes made in that line, and returns back to the Active Command Level.

(ii) Saves all changes and back:

E key

This command shifts the computer from Editing level, and saves all the changes previously made. Make sure that the computer is not executing any subcommand before entering E.

(iii) Cancel all changes and back:

Q key

This command shifts the computer from Editing level back to the Active Command level, but cancel all the changes made in the current edit mode. Just type in Q to cancel the changes made and return to the Active Command level.

4.2 SCREEN EDITOR

This screen Editor allows us to edit at the same time all BASIC program lines displayed on the screen. This Editor is applicable in NEWDOS 64-mode BASIC and NEWDOS 80-mode BASIC.

4.2.1 INTRODUCTION:

In NEWDOS 80-mode BASIC, this Screen Editor is already resident in main momery. This means that we can use the Screen Editor functions as the system has entered the NEWDOS BASIC.

For NEWDOS 64-mode BASIC (Genie I/II compatible), initially we are supposed to use the Level II BASIC line editor. If we want to use the Screen Editor for 64-mode, we have to load the Editor from the disk to replaced the line editor in main memory.

There are two cases to load the Screen Editor (64-mode):-

(1) As the System is in DOS level, type in

EDIT64/CMD.

Then, we may type in the command BASIC followed by NEW LINE key to enter the NEWDOS 64-mode BASIC.

(2) As the system is already in NEWDOS BASIC, type in

CMD"EDIT64/CMD"

Then, we can use the editing functions and subcommands of the Screen Editor as illustrated in the following section.

4.2.2 EDITING FUNCTION AND SUBCOMMANDS OF SCREEN EDITOR

This Screen Editor allows us to delete, change and insert characters to our BASIC programs displayed on the screen.

4.2.3 ENTER THE EDIT MODE

While the system stays in NEWDOS BASIC, type in the EDIT command followed by NEW LINE. There are 4 version of EDIT command.

(a) EDIT X-Y
Where X is the first line number of our BASIC program to be displayed and edited.
Y is the last line number of our BASIC program to be displayed and edited.

(b) EDIT X-
Which edits program lines from X to the last line of a program.

(c) EDIT -Y
which edits program lines from the first line to the line Y of a program.

(d) EDIT or "," (comma)
which edits all program lines provided that they are displayed on the screen.

As the EDIT command is entered, the program lines specified from X to Y will be displayed on the screen. If the number of lines specified is beyond the screen can display, the last 24 lines (in 80 x 24 format) or 16 lines (in 64 x 16 format), and the first line on the screen must have a line header. Every line to be edited has a line header, indicating the mode had been entered.

4.2.4 CURSOR MOVEMENT

The cursor position determines where editing is to be performed. The cursor can be moved anywhere on the screen by four position keys.

' UP ARROW KEY '	-	Upward
' DOWN ARROW KEY '	-	Downward
' LEFT ARROW KEY '	-	Left
' RIGHT ARROW KEY'	-	Right

Everytime a position key is hit, the cursor will move one space in the direction of the position key.

4.2.5 EXECUTE THE EDITOR COMMAND AND UPDATE THE PROGRAM MEMORY

This is accomplished by hitting the NEW LINE key. As all the displayed lines have been edited on the screen, NEW LINE will bring the updated information shown on the screen into the Program Statement memory. This will also cause exit from the Edit Mode, and 'READY' will appear.

NOTE : The editor will edit only those program lines displayed on the screen just before hitting NEW LINE key.

PRECAUTION : DO NOT press NEW LINE unless you are sure the desired characters/ program lines are to be deleted, cleared or changed.

4.2.6 TERMINATED THE EDIT MODE WITHOUT CHANGING THE OLD PROGRAM

Hit the BREAK key to terminate the Edit Mode. 'READY' will appear. The old program in this case will remain in the porgram memory without any changes although the program has been edited on the screen.

4.2.7 CHANGE A CHARACTER ON THE SCREEN

This is done by positioning the cursor at the character to be changed, and type in the new character to replace the old character. Even the line number of a program can be changed on the screen. See also section 4.2.11.

* NOTE that the program is not yet changed in the program memory before the NEW LINE key in pressed.

4.2.8 DELETE A CHARACTER ON THE SCREEN

The cursor is positioned at where delete operation is to be performed. Then, press 'SHIFT' and LEFT ARROW key simultaneously to delete the character or even a blank on the screen. The characters on the right of the cursor will shift left by one character position. This is illustrated in the following example.

Example 1 : Delete a character

```
■10 REM THE ITEMMS ARE SORTED AS BELOW
```

(1) Move the cursor-

```
■10 REM THE ITEMMS ARE SORTED AS BELOW
                  ↑
                  _
```

move the cursor to here, under the character to be deleted.

(2) Press 'SHIFT' key and LEFT ARROW key simultaneously

```
■10 REM THE ITEMS ARE SORTED AS BELOW
                  ↑
                  _
```

cursor will under "S" and "M" is deleted.

(3) Move the cursor to other position for editing.

The delete operation is inhibited under three conditions:

(1) When the cursor is positioned at the line header, or
(2) When the cursor is positioned after the last character of a line, or
(3) When the cursor is positioned at the bottom right most corner of the screen.

4.2.9 INSERT OPERATION

The insert operation includes:
(1) inserting a blank
(2) inserting a character
(3) inserting a blank line

(1) Insert a Blank

The cursor is positioned at where a blank is to be inserted. Then press 'SHIFT' and RIGHT ARROW key simultaneously. A blank will be obtained at the cursor position, and the characters on the right of the cursor will shift right by one character position.

Example:

```
■ 1Ø PRINT "ENTERTHE REQUIRED PARAMETERS"
```

(a) move the cursor

```
■ 1Ø PRINT "ENTERTHE REQUIRED PARAMETERS"
                  -
                  move the cursor under "T"
```

(b) press 'SHIFT' key and 'RIGHT ARROW' key simultaneously

```
■ 1Ø PRINT "ENTER THE REQUIRED PARAMETERS"
```

(2) Insert a Character

Inserting a character is similar to inserting a blank. The cursor is positioned at where the character is to be inserted. Then press 'SHIFT' and 'RIGHT ARROW' key simultaneously. The characters on the right of the cursor will shift right by one character position. Type in the character to be inserted.

Example: Insert a character

```
■ 6Ø REM RANDOM NMBER• GEN
```

(a) Move the cursor under the "M" in the word "NMBER"

```
■ 6Ø REM RANDOM NMBER GEN
                  -
                  cursor under "M"
```

(b) Press 'SHIFT' key and 'RIGHT ARROW' key simultaneously.

```
■ 6Ø REM RANDOM N MBER GEN
                    -
```

(c) Type in the character to be inserted

```
■ 6Ø REM RANDOM NUMBER GEN
```

(3) Insert a Blank Line

Move the cursor or the last character of a line to the 8Øth character position of a line (in 8Ø x 24 format) or the 64th character position of a line (in 64 x 16 format). Then press 'SHIFT' key and 'RIGHT ARROW' key simultaneously. We shall obtain a blank line following the line that the cursor is positioned.

New character can be typed in the blank line just obtained to extend the above program line.

Example: Insert a blank line

```
     2Ø A=A+1 : B=B+A
     3Ø REM INCREMENT THE COUNTER
```

(a) Move the cursor to the 8Øth character position.

```
■ 2Ø A=A+1 : B=B+A                    -
                                    cursor
■ 3Ø REM INCREMENT THE COUNTER
```

(b) Press 'SHIFT' key and 'RIGHT ARROW' simultaneously

```
■ 2Ø A=A+1 :B=B+A
       ( a blank line)
■ 3Ø REM INCREMENT THE COUNTER
```

(c) Move the cursor to where characters are to be inserted.

Type in new characters in the blank line.

```
■ 2Ø A=A+1 :B=B+A
  :C=C+A : REM CREATE ANOTHER NO. -
■ 3Ø REM INCREMENT THE COUNTER
```

REMARK: Limitations in Insert Operation

Insert operation is inhibited under the following conditions:

(a) When the cursor is positioned at a line header, or

(b) When the cursor is positioned at the bottom right most corner of the screen, or

(c) When the cursor is positioned after the 24Øth character position of a program line.

NOTE : a program line can have a maximum of 24Ø characters.

4.2.10 CLEAR

(1) We may press the CLEAR key to clear all characters on the right of the cursor in a line.

(2) Also, we can clear a whole program line by placing the cursor just after the line number and pressing the CLEAR key.

Example:

```
■ 10 REM INITIALIZATION
■ 15 OUT 253,0 : OUT 254,78
■ 20 A=0 : B=170 : C =255
```

(a) Move the cursor to just after the line number.

```
■ 10 REM INITIALIZATION
■15-OUT 253,0 : OUT 254,78
■ 20 A=0 : B=170 : C=255
```

(b) Press CLEAR key

```
10 REM INITIALIZATION
15-
20 A=0 : B=170 : C =255
```

(c) Hit NEW LINE key and type in LIST followed by NEW LINE key.

```
10 REM INITIALIZATION
20 A=0 : B=170 : C=255
```

REMARK: It is invalid to clear a porgram line with the cursor in front of the line number. That porgram line will not be cleared if we try to do so.

(3) By using the CLEAR key, we can add a new program line on the screen during editing. Position the cursor just after the line header of a line and hit the CLEAR key.
The old program line following that line header will disappear on the screen but it remains unaltered in the program memory.
Then, type in the new line number and statements in the blank line obtained above.

Example:

(a) Position the cursor just after header (line 60).

```
 50 REM INPUT A NEW LINE
 60 PRINT "PLEASE INPUT A NEW LINE"
 ↑
cursor position before "60"
 70 INPUT A$, B$, C$
```

(b) Hit CLEAR key

```
 50 REM INPUT A NEW LINE

 70 INPUT A$, B$, C$
```

(c) Type in the new line number and statement

```
 50 REM INPUT A NEW LINE
 45 PRINT "EDITING HAS COMPLETED"
 70 INPUT A$, B$, C$
```

(d) press NEW LINE and type in LIST followed by NEW LINE key.

```
 45 PRINT "EDITING HAS COMPLETED"
 50 REM INPUT A NEW LINE
 60 PRINT "PLEASE INPUT A NEW LINE"
 70 INPUT A$, B$, C$
```

4.2.11 DUPLICATE A PROGRAM LINE

It is easy to duplicate a program line by changing the line number of the program line displayed on screen to the desired line number. It is illustrated in the example below.

Example:

(i) Original program

```
■ 10 REM DISPLAY THE RESULT
■ 20 PRINT "THE PARTIAL SUM WILL BE"
  :
  :
■ 50 REM END OF SECTION X
```

(ii) Position the cursor at the line number of which the program line will be duplicated. Type in the new line number. (line 20 to line 90)

```
■ 10 REM DISPLAY THE RESULT
■ 90 PRINT "THE PARTIAL SUM WILL BE"
  -
  cursor position
  :
■ 50 REM END OF SECTION X
```

(iii) Press the NEW LINE key to execute the editing. LIST the program for inspection.

```
10 REM DISPLAY THE RESULT
20 PRINT "THE PARTIAL SUM WILL BE"
:
50 REM END OF SECTION X
90 PRINT "THE PARTIAL SUM WILL BE"
```

NOTE:

(1) The original program line whose line number is edited on the screen will remain in main memory without any changes.

(2) Under two conditions we really want to change the line number of a program:

(a) duplication the program line as in the above example to the desired line number.

(b) re-enter edit mode to clear the program line with old line number as mentioned in section 4.2.10 item (2).

APPENDIX A: ERROR MESSAGES

CODE	ERROR MESSAGE
1	NEXT without FOR
2	syntax error
3	return without GOSUB
4	out of data
5	illegal funciton call
6	overflow
7	out of memory
8	undefined line
9	subscript out of range
1Ø	redimensioned array
11	division by zero
12	illegal direct
13	type mismatch
14	out of string space
15	string too long
16	string formula too complex
17	can't continue
18	NO RESUME
19	RESUME without error
2Ø	unprintable error
21	missing operand
22	bad file data
51	field overflow
52	internal error
53	bad file #
54	file not found
55	bad file mode
56	file already open
58	DOS ERROR
59	file already exists
62	disk full
63	input past end
64	bad record #
65	bad file name
66	mode mismatch
67	direct statement in file
68	too many file
69	disk write protected

70	file access denied
71	seq. # overflow
72	record overflow
73	illegal to extend file
75	previously displayed error
76	can't process line 0
77	bad file type
78	IGEL SYNTAX ERROR
79	IGEL item syntax error
80	bad/illegal/missing IGEL item prefix
82	bad record length
83	stmt. uses 2 file names
84	bad file positioning PARAM

CODE	MESSAGE

1 NEXT WITHOUT FOR

A variable in NEXT statement does not match with the variable in the previously executed FOR statement or a NEXT statement with no FOR statement.

Example :

```
>
READY
>LIST
10 A=20
20 AMT=A-1
30 NEXT
40 END
READY
>RUN
NEXT WITHOUT FOR IN 30
READY
>
```

2 SYNTAX ERROR

incorrect sequence of characters in the program; such as misspelled command, unmatched parenthesis, and etc..

Example:

```
>
>LLLLIST
SYNTAX ERROR
```

3 RETURN WITHOUT GOSUB

A RETURN statement is found in a program without any GOSUB statement.

Example:

```
>LIST
10 FOR B=192 TO 0 STEP -16
20 WT=5*B
30 NEXT B
40 RETURN
50 END
READY
>RUN
RETURN WITHOUT GOSUB IN 40
READY
```

==========

4 OUT OF DATA

A READ statement with no DATA statement.

Example:

```
>LIST
10 READ A,B,C
20 IF A>B GO TO 10
30 C=A
40 PRINT C
50 END
READY
>RUN
OUT OF DATA IN 10
READY
>
```

==========

5 ILLEGAL FUNCTION CALL

Occurs when:

(a) an out-of-range parameter is passed to a math or string function.
(b) a negative or unreasonably large subscript.
(c) a negative or zero argument with LOG.
(d) a negative argument with SQR.
(e) a USR function call without starting address
(f) an improper argument to LEFT$, RIGHT$, MID$, and etc..

```
Example:
>LIST
10 A=INP(330)
20 PRINT A
30 END
READY
>RUN
ILLEGAL FUNCTION CALL
READY
```

==========

6 OVERFLOW

Too large for number format or array.

```
Example:
>LIST
10 DIM A(7777777),R(111111111)
20 FOR I=1 TO 60000000
30 A=DIM(I)
40 PRINT A
50 NEXT I
60 END
READY
>RUN
OVERFLOW IN 10
READY
>
```

==========

7 OUT OF MEMORY

A program is too large for the memory to store, or has too many FOR loops or too many GOSUB.

==========

8 UNDEFINED LINE

A non-existent line is specified by a GOTO, GOSUB, IF-THEN-ELSE, or DELETE command.

Example:

```
>LIST
10 READ A,B
20 IF A=B GOTO 100
30 40 DATA 1,3
40 END
READY
>RUN
UNDEFINED LINE # IN 20
READY
>
```

==========

9 SUBSCRIPT OUT OF RANGE

An array element is specified either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

Example:

```
>LIST
10 DIM A(10)
20 FOR I=1 TO 40
30 K=I
40 READ A(K)
50 NEXT I
60 END
70 DATA 1,2,3,4,5,6,7,8,9,1,1,1,1,2,4,5,6,6
READY
>RUN
SUBSCRIPT OUT OF RANGE IN 40
READY
>
```

==========

10 REDIMENSIONED ARRAY

Two or more DIM statements are given for the same array (same name for two array).

Example:

```
>LIST
10 DIM W(40) W(10)
20 FOR I=1 TO 40
30 INPUT W(I)
40 NEXT I
50 END
READY
>RUN
REDIMENSIONED ARRAY IN 10
READY
>
```

==========

11 DIVISION BY ZERO

An attempt was made to use a value of zero in the denominator.

Example:

```
>A=45/0
DIVISION BY ZERO
READY
>
```

==========

12 ILLEGAL DIRECT

The use of INPUT as a direct command or any illegal direct mode statement is encountered in direct mode.

Example:

```
>
>INPUT "AMOUNT";AMT
ILLEGAL DIRECT
READY
>
```

==========

13 TYPE MISMATCH

(a) An attempt was made to assign a non-string variable to a string or vice-versa.
(b) An attempt was made to assign a numeric argument to a non-numeric function or vice versa.

Example:

```
>LIST
10 CLEAR 500
20 A=20
30 C=STRING$(10,A)
40 PRINT C
50 END
READY
>RUN
TYPE MISMATCH IN 30
READY
>
```

==========

14 OUT OF STRING SPACE

The amount of string space allocated was exceeded.

Example:

```
>LIST
10 B$=STRING$(500,"A")
20 C$=STRING$(500,"A")
30 END
READY
>RUN
OUT OF STRING SPACE IN 10
READY
>
```

==========

15 STRING TOO LONG

A string variable was assigned a string value which exceeded 255 characters in length.

Example:

```
>LIST
10 CLEAR 5000
20 A$="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA"
30 B$="BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
40 C$="CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
50 D$=A$+B$+C$
60 PRINT D$
70 END
READY
>RUN
STRING TOO LONG IN 50
READY
>
```

==========

16 STRING FORMULA TOO COMPLEX

A string operation was too complex to handle. Break up the operation into shorter steps.

==========

17 CAN'T CONTINUE

A CONT was issued at a point where no continuable program exists.

Example:

```
>LIST
1Ø A=23
:
:
7Ø END
READY
>RUN
AMOUNT IS 12222

READY
>CONT
CAN'T CONTINUE
READY
```

==========

18 NO RESUME

End of program reached in error-trapping mode.

Example:

```
>LIST
1Ø ON ERROR GOTO 6Ø
2Ø READ A
3Ø PRINT A
4Ø GOTO 2Ø
5Ø END
6Ø DATA 4,2,2
READY
>RUN
 4
 2
 2
NO RESUME IN 6Ø
READY
>
```

==========

19 RESUME WITHOUT ERROR

A RESUME was encountered before ON ERROR GOTO was executed.

```
READY
>LIST
10 PRINT "A"
20 RESUME
30 IF I=4 THEN GO TO 10
40 END
READY
>RUN
A
RESUME WITHOUT ERROR IN 20
READY
>
```

==========

20 UNPRINTABLE ERROR

AN attempt was made to generate an error using an ERROR statement with an invalid code.

==========

21 MISSING OPERAND

An operation was attempted without providing one of the required operands.

Example:

```
READY
>LIST
10 A=10
20 B=A+
30 END
REDY
>RUN
MISSING OPERAND IN 20
READY
>
```

22 BAD FILE DATA

Data input from source was not correct or was in improper sequence, etc.

50 FIELD OVERFLOW

An attempt is made to allocate more than 255 bytes to a random-access buffer.

51 INTERNAL ERROR

Error in the DISK BASIC operating system.

52 BAD FILE #

(a) an improper file number.
(b) a number has not been assigned to a file with an OPEN statement.

Example:

```
>LIST
10 OPEN"O",57,"METER/TXT:1"
20 FOR I%=1 TO 5
:
:
100 END
READY
>RUN
BAD FILE # IN 10
READY
>
```

54 FILE NOT FOUND

An attempt is made to read a non-existing file from the disk.

Example:

```
READY
>LOAD"KKK/BAS:Ø"
FILE NOT IN DIRECTORY
FILE NOT FOUND
READY
>
```

==========

55 BAD FILE MODE

An attempt is made to carry out disk file input or output which conflicts with the mode in which the file was opened.

Example:

```
READY
>LIST
1Ø CLEAR 5ØØ
2Ø DIM FACT(5)
3Ø OPEN"A",1,"METER/TXT:1"
:
:
12Ø END
READY
>RUN
BAD FILE MODE IN 3Ø
READY
>
```

==========

56 FILE ALREADY OPEN

A sequential output mode OPEN is issued for a file that is already open.

==========

58 DOS ERROR

An error occurred during I/O operation between the Computer and a disk file.

==========

59 FILE ALREADY EXISTS

The filename specified in statement is the same as the filename that is using.

==========

62 DISK FULL

All available disk storage space on the diskette has been used.

==========

63 INPUT PAST END

During sequential input to a variable, the end of file was reached before any data characters were read.

==========

64 BAD RECORD #

In a PUT or GET statement, the record number is exceeded the range .

==========

65 BAD FILE NAME

An illegal form of file specification was provided.

```
READY
>LOAD"KKK.
BAD FILE NAME
READY
>
```

==========

66 MODE MISMATCH

An OPEN file mode is not match with the PUT, or GET or vice versa.

==========

67 DIRECT STATEMENT IN FILE

An attempt is made to LOAD, RUN, or MERGE a disk file which is not a BASIC program.

==========

68 TOO MANY FILE

An attempt is made to create a new file when all directory entries are full.

==========

69 DISK WRITE PROTECTED

An attempt is made to write to a disk with write-protect notch covered.

==========

70 FILE ACCESS DENIED

An attempt is made to access existing file with incorrect password.

---

71 SEQ. # OVERFLOW

---

72 RECORD OVERFLOW

---

73 ILLEGAL TO EXTEND FILE

---

75 PREVIOUSLY DISPLAYED ERROR

---

76 CAN'T PROCESS LINE Ø

---

77 BAD FILE TYPE

---

78 IGEL SYNTAX ERROR

Note: IGEL is Item Group Expression List. i.e. a list of expressions corresponding to a group of file items.

79	IGEL ITEM SYNTAX ERROR
80	BAD/ILLEGAL MISSING IGEL ITEM PREFIX
82	BAD RECORD LENGTH
83	STMT. USES 2 FILE NAMES
84	BAD FILE POSITIONING PARAM