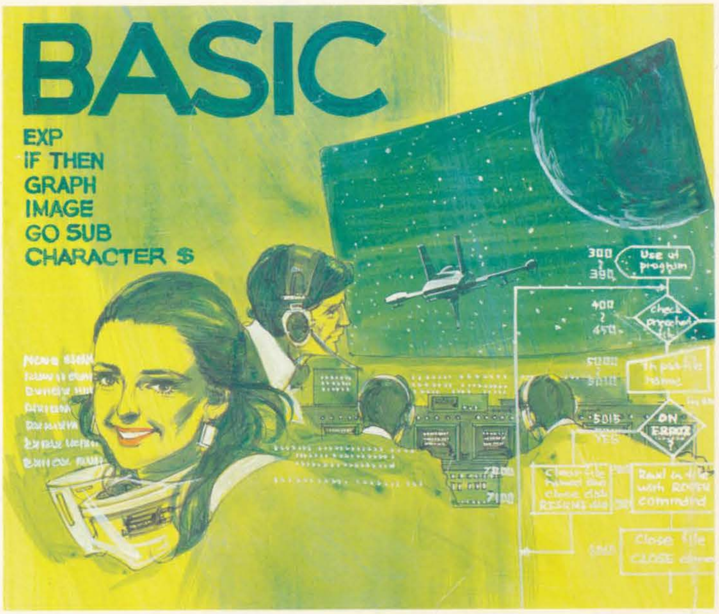
Personal Computer

MZ-80B

BASIC LANGUAGE MANUAL



SHARP

SHARP

Personal Computer

MZ-80B

BASIC Language Manual

January 1981

080221-150281

Printed in Japan

NOTICE

This manual is applicable to the SB-5510 BASIC interpreter used with the SHARP MZ-80B Personal Computer. The MZ-80B general-purpose personal computer is supported by system software which is filed in software packs (cassette tapes or diskettes).

All system software is subject to revision without prior notice, therefore, you are requested to pay special attention to file version numbers.

This manual has been carefully prepared and checked for completeness, accuracy and clarity. However, in the event that you should notice any errors or ambiguities, please feel free to contact your local Sharp representative for clarification.

Introduction

This Book describes the language specifications and grammar of the standard system software BASIC interpreter SB-5510. BASIC (an abbreviation for "Beginner's All-purpose Symbolic Instruction Code") was developed as an all purpose language to provide beginners with a means of easily programming computers to solve a diverse range of problems. Its simplicity and versatility make it well suited to personal programming applications.

BASIC SB-5510 is an extended BASIC interpreter which enables the MZ-80B computer to be used to its fullest capacity. Its sophisticated algorithms and output display processes, together with its high speed processing capability, make it well suited not only for beginners, but also for a variety of high level professional applications.

Language specifications for disk-based BASICs, that is, DISK BASIC, double-precision DISK BASIC, and BASIC compiler, are contained in their respective software manuals.

Contents

Chapter 1
Outline BASIC SB-5510

This chapter outlines programming procedures and use of the BASIC interpreter SB-5510.

The chapter begins with a description of the procedure for activating BASIC SB-5510, then discusses operating modes, the program architecture and constituents and the display screen in turn.

Detailed explanations of all commands, statements and functions provided with BASIC SB-5510 are contained in chapters 2, 3 and 4.

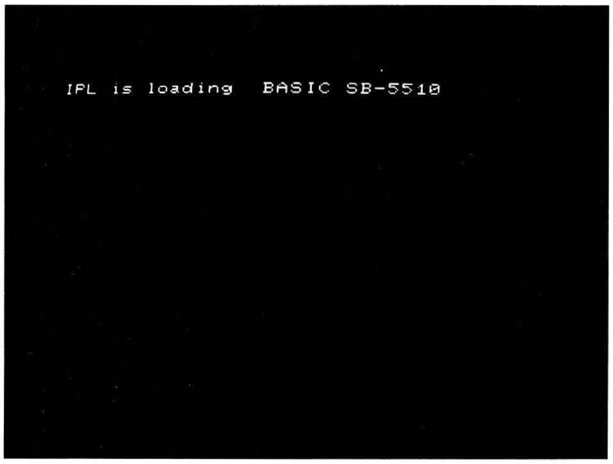
1.1 Activating the BASIC interpreter SB-5510

BASIC SB-5510 is stored (along with MONITOR SB-1510) on a cassette tape file, and must undergo initial program loading whenever it is to be used. Loading is easily achieved. Simply place the BASIC cassette file in the cassette tape deck and turn on the power.

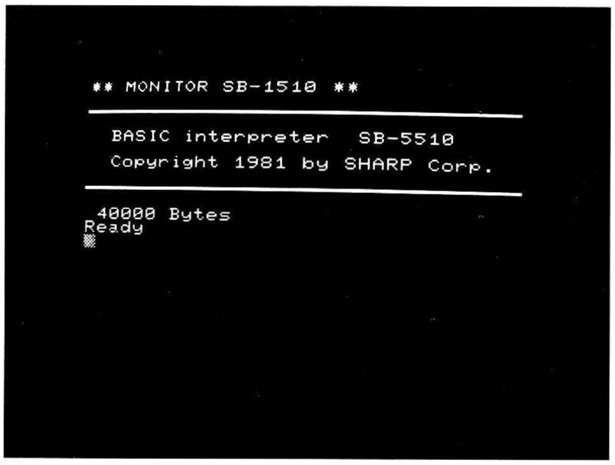
The MZ-80B's built-in IPL (Initial Program Loader) automatically starts (photo at left of FIGURE 1.1) loading both the BASIC interpreter SB-5510 and the MONITOR SB-1510. Upon completion of loading, the MZ-80B displays the message illustrated in the photo at right and the BASIC interpreter begins to operate.

The message "Ready" indicates that system control is at the BASIC command level and that the system is ready to accept any command.

(Please refer to the chapter on initial system program loading in the Owner's Manual for further information.)



```
IPL is loading  BASIC SB-5510
```



```
** MONITOR SB-1510 **
 BASIC interpreter  SB-5510
 Copyright 1981 by SHARP Corp.
 40000 Bytes
Ready
▒
```

FIGURE 1.1

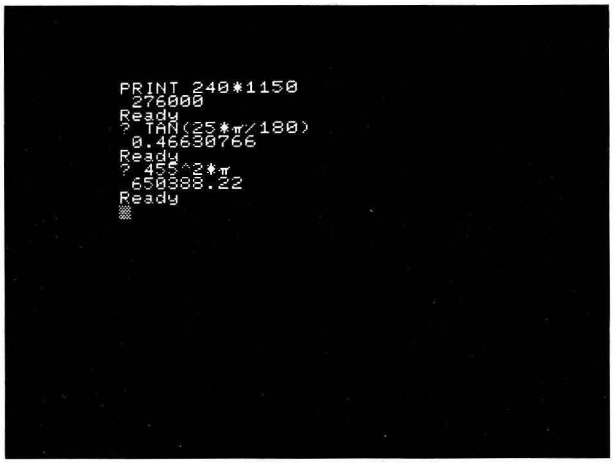
1.2 Operating modes

The system software, BASIC interpreter operates at either the command level or the statement level. At the command level, the BASIC interpreter executes BASIC commands and edits program texts. At the statement level, the BASIC interpreter interprets and executes BASIC statements of the program.

1.2.1 Operation at Command Level

When system control is at the BASIC command level, the MZ-80B can operate in either the direct or indirect mode. The direct mode allows you to manipulate the system just as you would operate an electronic calculator, using direct BASIC execution commands, such as RUN, LIST, and SAVE, or statements which serve the purpose of direct execution commands.

For example, the system may be used as illustrated at FIGURE 1.2 to obtain an immediate answer to an arithmetic expression using computation/display commands together with the PRINT statement.



```
PRINT 240*1150
 276000
Ready
? TAN(25*π/180)
 0.46630766
Ready
? 455^2*π
 650388.22
Ready
▒
```

FIGURE 1.2

This mode also enables you to interrupt program execution with the STOP statement or break operation to check the value of variables used and substitute other values, thus providing a means of tracing programming errors.

The indirect mode is used when creating and editing BASIC programs. A BASIC program, as discussed in the following paragraphs, is a group of numbered lines of BASIC statements and functions which describe the algorithm used in processing to obtain a specific result. The cursor which appears on the display screen in the indirect mode allows you to edit BASIC programs right on the screen. (See Section 1.5.)

1.2.2 Operation at Statement Level

The operating level of the BASIC interpreter is set to the statement level with a RUN command. The BASIC interpreter interprets each statement in the program text and performs data processing according to the statement. Communication between the operator and the system is performed as follows.

For example, when an INPUT statement is executed, the BASIC interpreter waits for the operator to input required data from the keyboard. When a PRINT statement is executed, the BASIC interpreter outputs the internal data to the console (or display screen).

After a program is completed, the operating level automatically returns to the command level. To forcibly return the operating level from the statement level to the command level, press the $\boxed{\text{BREAK}}$ key to stop program execution.

Machine language programs can be executed with the MZ-80B. In this case, the BASIC interpreter does not control the system and the system operates at the CPU level. To stop CPU level execution, press the RESET switch (on the rear panel of the main unit) to return the system operating level to the MONITOR command level.

1.3 Architecture of a BASIC program

A BASIC program is composed of a number of sequential lines of statements which control various operations, such as program execution and procedures.

With the exception of the various loop, transfer and branch operations, BASIC programs are executed in the order in which the lines of statements and commands are arranged in the program. Further, BASIC programs do not require any initial format specification or declarative statements, and program execution may be begun on any program line.

1.3.1 Line

Each program line consists of a line number and a sentence, separated from the following line by a carriage return code.

The entry of one line can be accomplished in the indirect mode by pressing the $\boxed{\text{CR}}$ (or $\boxed{\text{ENT}}$) key.

Line number

A line number is placed at the head of each program line to identify it. This number is also referred to as a definition line number—the number defining a line—to distinguish it from a reference line number, which is used for referencing from other lines.

The definition line number for each line must be an integer from 1 through 65535. Line numbers do not need to be assigned consecutively; in fact, it is advisable to assign line numbers in increments of ten to allow for insertion of additional lines during program editing.

The AUTO command serves to produce definition line numbers at fixed intervals.

If the same definition line number is input more than once, only the last entry will be valid.

Sentence

A sentence is a sequence of one or more statements, preceded by a definition line number and concluded with a carriage return code. Each sentence (including the definition line number and the carriage return code) is composed of up to 80 characters.

When a sentence is to include two or more statements, each must be separated by a semicolon (;).

1.3.2 Statements

BASIC statements are divided into two general classes: executable statements and nonexecutable statements.

An executable statement describes a program operation, such as computation, substitution, comparison or branching. A nonexecutable statement establishes information necessary for programming or controls the program pointer: examples include array declarator statements, data statements, definitive statements and comment statements.

An executable statement, when used in the direct mode, is sometimes referred to as a direct mode execution statement.

1.4 Constituent element of a sentence

A BASIC sentence is composed of reserved words—also called key words—which include statements, built-in functions and special signs (and also commands), and other elements, such as constants, variables, arrays and expressions.

1.4.1 Reserved Word

All reserved words have special meanings which are defined by the rules of BASIC, and cannot be altered. No reserved word specified in BASIC may be defined as variable names by programmer. Table 1.1 shows all reserved words of the BASIC interpreter SB-5510.

$\boxed{A}$	ABS		INP		PRINT/T
	ASC		INPUT	$\boxed{R}$	READ
	ATN		INPUT/T		REM
	AUTO		INT		RESET
$\boxed{B}$	BLINE	$\boxed{K}$	KLIST		RESTORE
	BOOT	$\boxed{L}$	LEFT$		RETURN
$\boxed{C}$	CHANGE		LEN		REW
	CHARACTER$		LET		RIGHT$
	CHR$		LIMIT		RND
	CLOSE/T		LINE		ROPEN/T
	CLR		LIST		RUN
	CONSOLE		LIST/P	$\boxed{S}$	SAVE
	CONT		LN		SET
	COPY/P		LOAD		SGN
	COS		LOG		SIN
	CSRH	$\boxed{M}$	MID$		SIZE
	CSRV		MON		SPACE$
	CURSOR		MUSIC		SQR
$\boxed{D}$	DATA	$\boxed{N}$	NEW		STEP
	DEF FN		NEXT		STOP
	DEF KEY	$\boxed{O}$	ON		STR$
	DIM		OUT		STRING$
$\boxed{E}$	END	$\boxed{P}$	PAGE/P	$\boxed{T}$	TAB
	EXP		PATTERN		TAN
$\boxed{F}$	FAST		PEEK		TEMPO
	FOR		POINT		THEN
$\boxed{G}$	GET		POKE		TI$
	GOSUB		POSH		TO
	GOTO		POSITION	$\boxed{U}$	USR
	GRAPH		POSV	$\boxed{V}$	VAL
$\boxed{I}$	IF		PRINT		VERIFY
	IMAGE/P		PRINT/P	$\boxed{W}$	WOPEN/T

TABLE 1.1 All reserved words of the BASIC interpreter SB-5510

1.4.2 Other constituent elements

Other constituent elements of a sentence are subdivided as follows.

Constant	:	numeric constant, string constant, system constant
Variable	:	numeric variable, string variable, system variable
Array	:	one-dimensional numeric array, two-dimensional numeric array, one-dimensional string array, two-dimensional string array
Expression	:	arithmetic expression, string connective expression, relational expression, logical expression
Separator	:	“,”

Constants, variables, array elements, arithmetic expressions and string connective expressions are program data elements, and are divided (depending upon type) into numeric data and string data.

1.4.3 Constant

The term “constant” refers to a data value that remains unchanged during program execution.

Numeric constant

A numeric constant is a decimal number represented by a combination of a sign (+ or –), numerals (0 through 9), and/or a decimal point (.); or, in scientific notation, by a combination of a sign (+ or –), mantissa, and exponent (indicated by “E”). Within the MZ-80B, such numbers are expressed with the floating decimal point system.

BASIC SB-5510 can represent numeric data of up to eight significant digits and numbers in scientific notation in the range from 10^{-19} through 10^{19}.

For positive numbers, the “+” sign may be omitted.

With the LIMIT, POKE, PEEK and USR statements, memory addresses may be specified directly with hexadecimal numbers. Such addresses are indicated by a four-digit hexadecimal number preceded by a dollar mark ($). Example: LIMIT $B000

Correct representation:	5215E–8 = 0.00005215
Incorrect representation:	15,300 – commas may not be used.
	1234567890 – the input number limit of eight digits is exceeded.
	300E+91 – the exponent limit is exceeded.

String constant

A string constant is a set of characters enclosed in quotation mark (" ") which is input from the keyboard. Quotation marks are not required with DATA statements.

The maximum number of characters in a string constant depends on the effective line length, but the total maximum number of characters of string data which are permitted per BASIC statement is 255.

A string constant may represent characters in the PRINT statement, data for musical notes in the MUSIC statement or bit data for graphic patterns in the PATTERN or IMAGE/P statements. Thus, the type of data represented by the constant depends on the statement with which is used.

System constant

A system constant is a value which is built into the BASIC interpreter; for example, the ratio of a circle's circumference to its diameter (indicated by the character "π") has a value equal to 3.1415927.

Example of use: The circumference of a circle with a radius of 10 can be computed by the expression:

```
2 * π * 10
```

1.4.4 Variable

The term "variable" refers to an element whose value may be arbitrarily changed during program execution without any change in data type. Each variable is identified by a name, and its initial value is 0 (zero) or null.

Numeric variable

Only numeric data can be assigned to numeric variables.

The name of each variable may be composed of any number of characters, but only the first two characters serve to identify the variable. The first character must be a capital letter (A thru Z), but the second may be any letter, numeral or symbol. No reserved word employed in BASIC may be used, however. The term "reserved word" covers all BASIC commands, statements, functions and operators, as well as special signs such as @ or #.

The numeric variable remains zero until it is loaded with numeric data.

Correct name : ABC and ABD are handled as the same variable. (First two characters are the same.)

Incorrect names : DATA 3 – Reserved word DATA may not be used.
C@ – Special sign "@" may not be used.

String variable

A string variable may be loaded only with string data, and its name is formed in the same manner as the name of a numeric variable, except that it is followed by a dollar sign ($).

Each string variable may contain a maximum of 255 characters of string data; it includes only null characters until loaded with string data.

Correct names : NAME$1 and NAME$2 are regarded as the same string variable (first two characters are the same).

Incorrect names : music$ – The name does not begin with capital letter.

System variable

System variables serve to indicate values that change during BASIC operation, and are classified in two types: numeric variable (e.g. SIZE to indicate the remained free memory size) and string variable (e.g. TI$ to indicate the reading of the built-in 24-hour based clock).

1.4.5 Array

An array is an arrangement of variables of the same type, and is called a one-dimensional array (or list) when given one subscript, and a two-dimensional array (or table) when given two subscripts.

Details on definition of numeric and string arrays will be found in Paragraph 3.5.1.

1.4.6 Expression

Arithmetic expression

An arithmetic expression, a means of expressing an arithmetic operation, is composed of operators and arithmetic element(s).

The table below shows the arithmetic operators arranged in order of operational priority.

Arithmetic operator		Operation	Example
^	:	Exponential calculation	X^Y
–	:	Minus sign	–X
*, /	:	Multiplication, division	X * Y, X/Y
+, –	:	Addition, subtraction	X+Y, X–Y

Exponential calculation takes priority over other operations, but any group of operations enclosed in parentheses has first priority. There is no limit on the number of levels of parentheses which may be used.

When an arithmetic expression includes operations of identical priority (multiplication and division or addition and subtraction), they are performed in sequence from left to right.

For successive exponential calculations, however, exponents on the right take priority over ones on the left.

Example: BASIC representation of arithmetic expressions

Arithmetic expression	BASIC equivalent
$\frac{c}{a+b} - \frac{e}{d}$	C/(A+B) – E/D
$\sin^2 x + 1$	SIN(X) ^ 2 + 1

String connective expression

String connective expressions are used to combine two or more sets of string data into a single set.

Example: "ABC" + "DEF" — makes "ABCDEF"
"DEF" + "ABC" — makes "DEFABC"

Relational expression

The term "relational expression" refers to the combination of two sets of numeric or string data by a relational operator.

This expression is used in the IF statement. For details, see Paragraph 3.4.3.

Logical expression

A logical expression expresses the Boolean sum or product of true or false values – 1 or 0 (zero) – given by a relational expression, and is used in the IF statement.

1.4.7 Separator

A comma serves as a separator to subdivide a statement into its individual elements.

Example:
```
DATA 3.5, 4.66, DAT 1
ON A GOTO 100, 200, 300
```

1.5 Screen editor

The screen editor of BASIC works while you are writing or editing programs in the indirect mode, allowing fast and easy editing of program lines appearing on the screen by moving the cursor.

BASIC programs are stored in the BASIC text area by numbered line; as mentioned previously, lines are not stored until the CR key or the ENT key is pressed. Accordingly, the CR or ENT key must also be pressed when lines of the program are changed through cursor operation.

Cursor control

When entering a new program, each line number and sentence are entered in sequence and the cursor advances one space for each character entered.

When programs are being modified, the cursor is brought to the line which is to be revised before making the modification. It is not necessary to reinput the entire line. The cursor can be moved up, down, left or right by manipulating the four yellow keys at the top of the keyboard. The CLR HOME key may be used to return the cursor to the upper left-hand corner of the display screen.

Continuous movement of the cursor may be obtained by pressing one of the four yellow keys together with the SHIFT key.

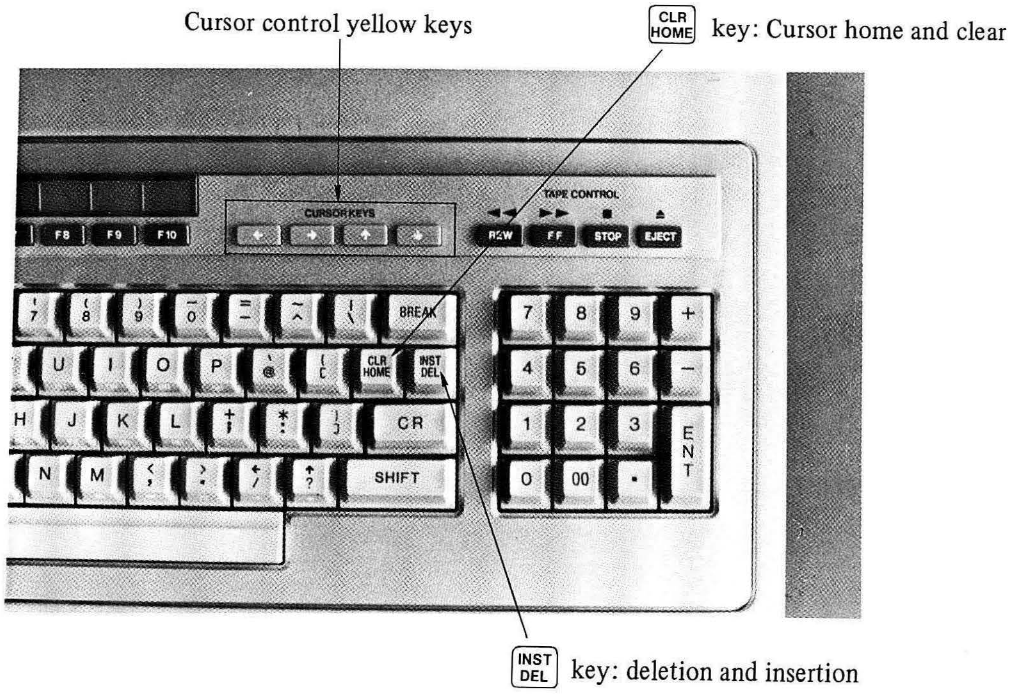

Cursor control yellow keys
CLR HOME key: Cursor home and clear
INST DEL key: deletion and insertion
CURSOR KEYS
TAPE CONTROL

FIGURE 1.3

Modification of lines may involve not only replacing certain characters with others, but insertion or deletion of characters as well.

To insert characters, first press and hold the SHIFT key, then press the INST DEL key once for each character to be inserted. This will open space ahead of the position where the cursor is located.

To delete characters, press the INST DEL key once for each character to be deleted. This will delete characters located ahead of the cursor.

Deletion of an entire line may be accomplished by entering only the line number and pressing the CR or ENT key. BASIC programs are not destroyed during program execution, nor by error detection or BREAK operation occurring during execution. Thus, debugging may be accomplished by repeatedly executing the program to locate errors, which may then be corrected through the screen editor. This conversational debugging/editing capability is one of the outstanding features of the BASIC interpreter SB-5510.

Setting tabs

The TAB key to the left of the space bar moves the cursor according to tabulation setting data stored in the area from $1141 to $114F. This function is convenient for inputting data entries in a formatted table displayed on the CRT screen.

Immediately after the BASIC interpreter SB-5510 has been activated, tabs are automatically set in the 10th, 20th and 30th character positions. For example, when the cursor is in the 13th character position and the TAB key is pressed, the cursor is moved to the 20th character position (i.e., the to first tab to the right).

Set tabs with the following procedures.

Set the first position in $1141. For example, when the first tab position is to be set in the 15th character position, execute the following statement.

```
POKE $1141, 15
```

The second tab position is to be set in $1142. For example, when the second tab position is to be set in the 23rd character position, execute.

```
POKE $1142, 23
```

To finish setting tabs, store the end data, 255, in the location just after that where the last tab position is stored. For example, execute the following statement.

```
POKE $1143, 255
```

Tab setting can be performed using the M command supported by the MONITOR SB-1510.

1.6 Initialization

When the BASIC interpreter SB-5510 is activated by the IPL, system variables and default values are initialized as follows:

- Keyboard
  1) Operation mode : normal
  2) Lower case letters are entered with the SHIFT key pressed.
  3) All special function keys are undefined.
- Display
  1) Character display mode : normal
  2) Character size : 40 characters/line
  3) Character display scrolling area : maximum (line 0 through line 24)
  4) Graphic display input mode : graphic area 1 (graphic area 1 cleared)
     Graphic display output mode : both graphic areas off
     Position pointer : POSH = 0, POSV = 0
- Array
  1) No arrays are declared.
- Clock
  1) The built-in clock is started with TI$ set to "000000".
- Music function
  1) Tempo : 4 (medium tempo : moderato)
  2) Duration : 5 (quarter note : ♩)

Chapter 2

BASIC SB-5510 Commands

This chapter describes all BASIC SB-5510 commands. All commands can be used only in the direct mode.

Command format.

Commands must be coded according to the following rules.

- *Small letters and reverse characters cannot be used for any commands.*
- *Operands which must be specified by the programmer are indicated in italics.*
- *Items in brackets "〈 〉" may be omitted or repeated any number of times.*
- *Separators (commas, semicolons, etc.) must be correctly placed in the specified positions.*

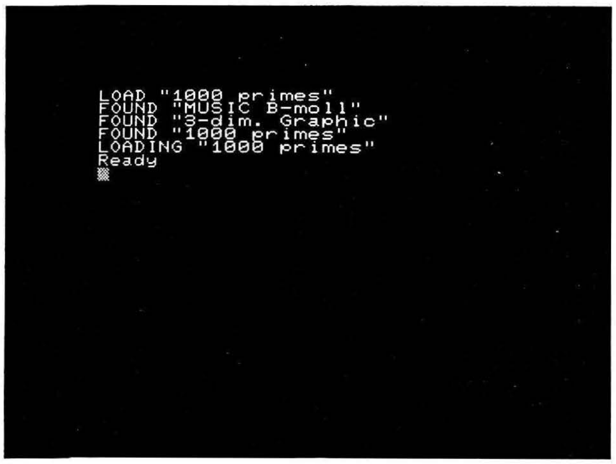
2.1 Input/output commands

2.1.1 LOAD

Format : LOAD 〈 *file name* 〉

Function : This command automatically loads the computer with a BASIC text or machine language program stored in a cassette tape file.

Description : If *file name* of the file to be loaded is specified, the MZ-80B system locates the file; if *file name* is not entered, the machine will be loaded with the first BASIC text file or machine-language program which the system encounters.
When the computer is loaded, it clears the current BASIC text area before accepting the BASIC text; when a machine language program is loaded, the BASIC text area is not cleared since the LIMIT statement is used to establish a machine language program area behind the BASIC text area.



```
LOAD "1000 primes"
FOUND "MUSIC B-moll"
FOUND "3-dim. Graphic"
FOUND "1000 primes"
LOADING "1000 primes"
Ready
▩
```

FIGURE 2.1

2.1.2 SAVE

Format : SAVE ⟨*file name*⟩

Function : This command automatically saves the program stored in the BASIC text area on cassette tape.

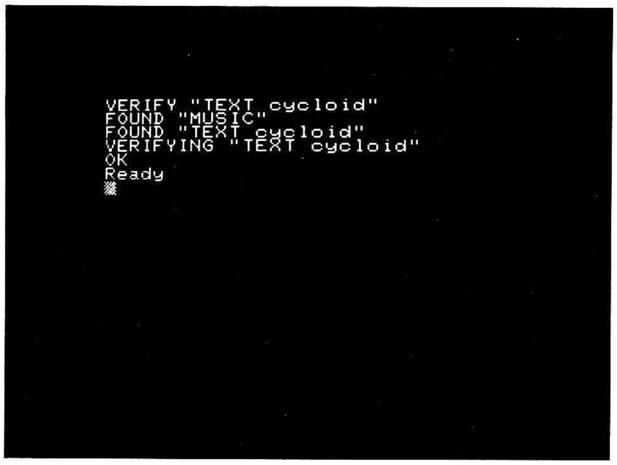
Description : *file name* is used to assign a name to the BASIC text file saved, so a suitable name must be given to each file. Each file name is composed of a string of up to 16 characters. If no *file name* is specified, the BASIC program file will have no name and later identification will be difficult.

2.1.3 VERIFY

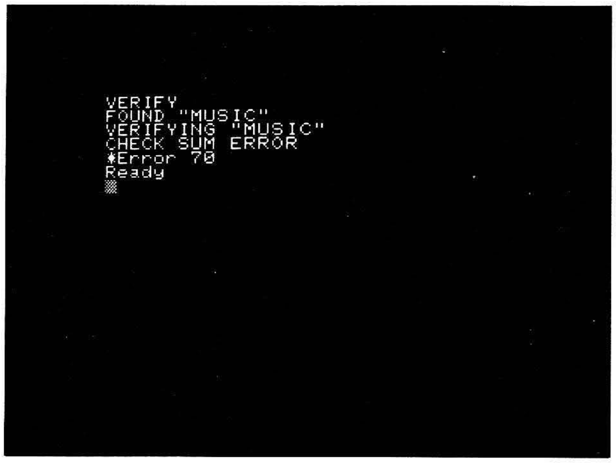
Format : VERIFY ⟨ *file name*⟩

Function : This command automatically compares the program contained in the BASIC text area with its equivalent text (file name: *"file name"*) in the cassette tape file.

Description : If the program and tape file coincide, "OK" will appear on the MZ-80B display screen; otherwise, "ERROR" is displayed as follows.



```
VERIFY "TEXT cycloid"
FOUND "MUSIC"
FOUND "TEXT cycloid"
VERIFYING "TEXT cycloid"
OK
Ready
▒
```



```
VERIFY
FOUND "MUSIC"
VERIFYING "MUSIC"
CHECK SUM ERROR
♦Error 70
Ready
▒
```

FIGURE 2.2

When ⟨ file name⟩ of a target file is specified, the MZ-80B system searches for the file and compares it with the program to be verified if it is located; if ⟨*file name*⟩ is not specified, the system compares the program with the first BASIC text file which it encounters.

2.2 Text edition commands

2.2.1 AUTO

Format	:	AUTO ⟨*ls*, *n*⟩
Function	:	This command automatically generates definition line numbers when a program is being entered in the indirect mode.
Description	:	Supplying the system with this command produces definition line numbers in specified increments each time the CR or ENT key is pressed, so the programmer needs only to input sentences of program text. ⟨*ls*⟩ indicates the beginning line number, and ⟨*n*⟩ indicates the increment; the default value of both is 10. The AUTO command is terminated when the BREAK key is pressed.

Example	:	AUTO 200, 20	This generates definition line numbers	200, 220, 240 ....
		AUTO	This generates definition line numbers	10, 20, 30 ....

2.2.2 LIST

Format	:	LIST ⟨*ls* - *le*⟩
Function	:	This command causes all or some of the program lines contained in the BASIC text area to be listed (displayed) on the computer screen.
Description	:	To list the entire program, enter only the LIST command. To list part of it, specify the starting line number ⟨*ls*⟩, the ending line number ⟨*le*⟩ or both. To stop program listing, press the space bar.

Example	:	LIST	This lists the entire program.
		LIST 100	This lists line 100 only.
		LIST 200-	This lists line 200 and succeeding lines.
		LIST -500	This lists lines 1 through 500.
		LIST 200-500	This list lines 200 through 500.

2.2.3 LIST/P

Format	:	LIST/P 〈 *ls* - *le* 〉
Function	:	This command causes all or some of the program lines contained in the BASIC text area to be listed on the printer.
Description	:	The MZ-80P5 80-digit line printer must be connected in order to use this command. The list range for this command is specified in the same manner as with the LIST command.

Errors	:	Error 65	Either the printer is OFF state or disconnected
		Error 66	Mechanical trouble with the printer
		Error 67	Paper has run out of the printer

2.2.4 NEW

Format	:	NEW
Function	:	This command clears the program contained in the BASIC text area and resets all variables.
Description	:	This command puts the system in the state it was in after initial program loading, making the system ready to accept a new program.

2.3 Execution control commands

2.3.1 RUN

Format :	RUN 〈 *ls* 〉
Function :	This command executes the program stored in BASIC text area.
Description :	Program execution may be started at any desired line by specifying the line number 〈 *ls* 〉 in the command. In such cases, the values of variables assigned and the contents of array elements are maintained just as when a GOTO statement is executed to jump to another line during program execution. If no line number is specified, program execution starts at the line with the smallest line number. In this case, all variables are reset and the array declaration is cancelled before execution begins.

2.3.2 CONT

Format :	CONT
Function :	This command serves to resume execution of program which has been interrupted by pressing the $\boxed{\text{BREAK}}$ key, or by a STOP statement.
Description :	This command is usually used during program debugging to continue execution interrupted (by a STOP statement or the $\boxed{\text{BREAK}}$ key), to check an intermediate result or to change variable data in the direct mode. To restart program execution at a line other than that at which execution was interrupted, use RUN 〈 *line number* 〉, or GOTO 〈 *line number* 〉.
Note :	When any program editing is carried out in the indirect mode during a pause in program execution, this command is not valid.

2.3.3 MON

Format : MON

Function : This command causes the system to leave the BASIC command level and await input of a command at the MONITOR SB-1510 level.

Description : As is indicated by the memory map (page 88), the system operates at either the BASIC or MONITOR system level. This command causes execution to jump from the BASIC to the MONITOR level.

The Monitor commands are M, D, J, S, V and L. For details regarding use of each Monitor command and return operation from the Monitor level to the BASIC interpreter level, refer to the MONITOR SB-1510 Reference Manual.

2.3.4 BOOT

Format : BOOT

Function : This command activates the MZ-80B System IPL (Initial Program Loader).

Description : IPL loads the system program from cassette tape or diskette into the memory. The BASIC interpreter, the monitor program and/or the user program which are currently stored in the memory area in which the system program is to be loaded are erased.

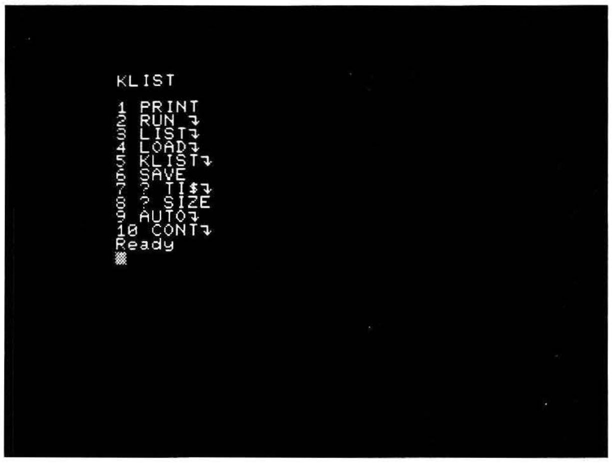
Executing a BOOT command results in the same operation as results from turning the power switch of the MZ-80B ON or pressing the IPL reset button.

2.4 Special function keys list command

2.4.1 KLIST

Format : KLIST

Function : This command displays a complete list of string definitions for special function keys, thereby enabling you to determine how individual special function keys are defined.



```
KLIST

1 PRINT
2 RUN ↴
3 LIST↴
4 LOAD↴
5 KLIST↴
6 SAVE
7 ? TI$↴
8 ? SIZE
9 AUTO↴
10 CONT↴
Ready
▒
```

FIGURE 2.3

Chapter 3
BASIC SB-5510 Statements

This chapter describes all BASIC SB-5510 statements and system variables.

Statement format.

Statements must be coded according to the following rules.

- *Small letters and reverse characters cannot be used for any statements.*
- *Operands which must be specified by thé programmer are indicated in italics.*
- *Items in brackets "⟨ ⟩" may be omitted or repeated any number of times.*
- *Separators (commas, semicolons, etc.) must be correctly placed in the specified positions.*

3.1 Assignment statement

3.1.1 LET

Format : ⟨LET⟩ $v = e$

v . . . . . . A name of numeric variable, numeric array, string variable or string array

e . . . . . . Arithmetic expression, string connective expression, variable or constant

Function : Assignment of data expressed by e to a variable or an array element

Description : The data represented by e and the corresponding variable or array must be identical in data type.

LET may be omitted.

Note : *Assignment* and *equal* have different meanings, so that A = A + 1 is a correct assignment statement.

Correct :

```
10   R (10) = R (10) + 1
20   LET S1 = SIN (TH * π + C) * A
30   N3$ = "Give me file name"
```

I = 1 . . . . An assignment in direct mode

Incorrect :

100 D$ = A + B . . . . . . The left side is a string variable, while the right side is numeric data. These are different data types.

110 LOG (K) = LK

3.2 Input/output statements

Broadly speaking, input/output statements are used for general control of the keyboard, display screen, audio output, input/output terminals, printer and external data files. Here only the fundamental input/output statements (PRINT, INPUT, GET, READ-DATA-RESTORE) are discussed.

The other input/output statements are grouped under the headings of data file input/output, music output, graphic output, printer output and input/output port access.

3.2.1 PRINT

Format : PRINT ⟨ e_1 d_1 e_2 d_2 .... e_n d_n ⟩

ei .... Output data

di .... Separator or tabulation function

Function : This statement displays consecutive lines of the data (values of constants, variables, array elements and expressions) designated as output list, in order of listing.

Description : The cursor works as the data output pointer. The data sets are displayed in sequence with the first line appearing immediately to the right of the position in which the cursor is located before execution of the PRINT statement.

The data output format depends on the separator, TAB function and SPC function, as well as cursor control and reverse control codes included in the string data.

If the data sets are omitted, the cursor advances down one line on the screen without any data being displayed.

Numeric data output is displayed either in the standard format or in scientific notation. Numbers larger in magnitude than 0.00000001 and smaller than 99,999,999 are displayed in the following form.

```
Blank
         X – – – X.X – – – X
  –
```

Each place is filled with a digit (from 0 to 9), and the total number of digits (not including the units place when the number is less than one) is eight or fewer.

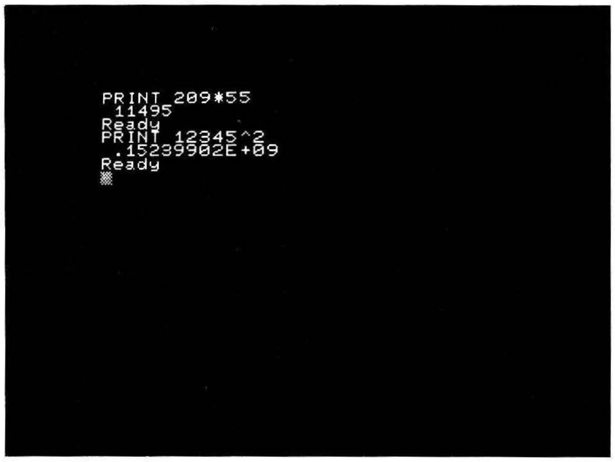
When the number includes a decimal point, it may be located between any two digit positions. Positive numbers are preceded by a blank, and negative ones by the minus sign "–".

The exponential display format is used for numbers which cannot be displayed in the standard format. The exponential display format is as indicated below.

$$\left\{\begin{matrix}\text{Blank}\\ -\end{matrix}\right\} .\text{X} \text{-----} \text{X E}\left\{\begin{matrix}+\\ -\end{matrix}\right\}\text{YY}$$

When the number is positive, it is preceded by blank; when negative, it is preceded by a minus sign 〈"–"〉.

The mantissa (.X ---- X) is composed of up to eight numerals (from 0 through 9), with zeroes suppressed in the first and last positions. The exponent is a two digit number YY (Y: 0 through 9) which is preceded by the exponential sign "E" and the plus "+" or minus sign "–".



```
PRINT 209*55
 11495
Ready
PRINT 12345^2
 .15239902E+09
Ready
▒
```

FIGURE 3.1

String data output occurs in succession, starting from the point at which the cursor is located.

If the string data is, for example, the PRINT statement control code responsible for cursor control, the control operation will be performed. Since ASCII codes $00 through $0F (0 through 15) may also be specified as data sets with the PRINT statement, a summary of their functions is set forth below.

PRINT CHR$ ($00)	No control.
PRINT CHR$ ($01)	Moves the cursor downward one line.
PRINT CHR$ ($02)	Moves the cursor upward one line.
PRINT CHR$ ($03)	Moves the cursor to the right one position.
PRINT CHR$ ($04)	Moves the cursor to the left one position.
PRINT CHR$ ($05)	Brings the cursor to the upper left hand corner of the display screen.

PRINT CHR$ ($06)	Clears the display and brings the cursor to the upper left hand corner of the display screen.
PRINT CHR$ ($07)	Deletes the character to the left of the cursor and moves the cursor to the left one position.
PRINT CHR$ ($08)	Opens one blank space to the right of the cursor.
PRINT CHR$ ($09)	Sets the key input mode of the keyboard to GRAPHIC.
PRINT CHR$ ($0A)	Sets the key input mode of the keyboard to SHIFT LOCK.
PRINT CHR$ ($0B)	No control
PRINT CHR$ ($0C)	Sets the key input mode of the keyboard to REVERSE.
PRINT CHR$ ($0D)	No control
PRINT CHR$ ($0E)	Cancels the GRAPHIC and SHIFT LOCK key input modes.
PRINT CHR$ ($0F)	Cancels the REVERSE key input mode.
PRINT CHR$ ($10 ~ $1E)	Prints one space.

Two or more output lists are punctuated by two types of separators which differ in function.

; (Semicolon)	Use of this separator results in display of output lists on successive lines.
, (Comma)	This separator causes output lists to be displayed in a tabulated format; i.e., the first character of the followed output list will appear in the position corresponding to the position of the forward output list plus 10 spaces. If the forward output list consists of more than 10 characters, successive tabulation is performed in 10 spaces increments as required to prevent overlapping.

The TAB function (see paragraph 4.3.1) and SPACE$ function (see paragraph 4.2.4) are used to control tabulation in the PRINT statement, allowing any desired tabulation.

3.2.2 INPUT

Format : INPUT ⟨ *String message* ;⟩ v ⟨ , v_2, ——–, v_n ⟩

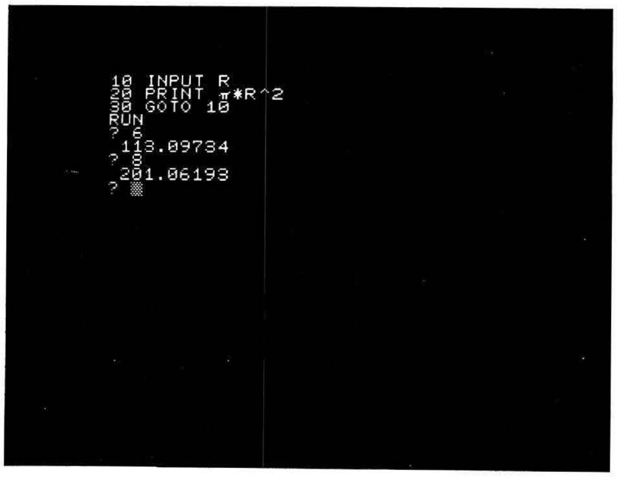
v_i . . . . A name of numeric variable, numeric array, string variable or string array

Function : This statement momentarily interrupts program execution to allow entry of data (numeric constant, string constant) from the keyboard, and assigns the data input in sequence to the variables or array elements (referred to as the input list) specified in the statement.

Description : Execution of this statement causes a question mark (?) to be displayed and the cursor to flicker to indicate that the system is awaiting data entry. If the statement includes a *string message*, the system displays that message instead of the question mark. A semicolon is used to separate the string message and the input list.

When the input list includes two or more variables or array elements, the input data is punctuated with commas (,). Data entry is concluded by pressing the $\boxed{\text{CR}}$ (or $\boxed{\text{ENT}}$) key.

When the number of data constants entered is smaller than the input list, the MZ-80B system displays "?" on the following line and awaits entry of subsequent data; when the number of data constants is greater than the input list, the excess constants are ignored. The data constants and the input list must be of the same data type. Spaces entered ahead of and behind string data are ignored. Normally, a string constant can be entered without being enclosed in quotation marks when it is preceded or followed by a space or when it includes a comma.



```
10 INPUT R
20 PRINT π*R^2
30 GOTO 10
RUN
? 6
 113.09734
? 8
 201.06193
? ▒
```

FIGURE 3.2

3.2.3 GET

Format : GET v

v . . . . . . A name of numeric variable, numeric array, string variable or string array

Function : During program execution, this statement will ascertain what key is being pressed and will assign the data to a corresponding variable or array element.

Description : This statement, when executed, assigns the individual data entered when a key is pressed to a specified variable or array element; when no key is pressed, "0" (zero) is assigned to a specified numeric variable or numeric array element and " " (null string) is assigned to a string variable or string array element.

The GET statement allows the entry of one character of key input data each time it is executed. When the input is numeric data, it is one of integers 0 through 9; when the input is string data, it consists of one character. When two or more keys are pressed, only the key with the highest priority is valid. Numeric variables or array elements will not accept any input from other than a numeric key.

3.2.4 READ ~ DATA

Format : READ $v_1 \langle, v_2, ---, v_m \rangle$

DATA $d_1 \langle, d_2, ---, d_n \rangle$

v_i . . . . A name of numeric variable, numeric array, string variable or string array

d_i . . . . A numeric constant or a string constant

Function : A READ statement reads a data table described in a DATA statement and assigns it to variables or array elements.

Description : The READ and DATA statements are used in pairs. Each DATA statement contains a data table composed of one or more statements, and the related READ statement reads the statements one after another and assigns them one by one to the individual variables or array elements of the input table. Accordingly, data present in the data table and the variable or array elements present in the corresponding input table must be of the same data type. If they do not agree, Error 4 (data mismatch) occurs. If the data table of a DATA statement is exceeded during execution of the related READ statement, Error 24 (Out of DATA) occurs.

If another READ statement is executed after the preceding one has read half of a data table, the remaining data is read without interruption.
The RESTORE statement can be used to fix the position of the data table that is to be read by the READ statement to be executed next.

3.2.5 RESTORE

Format : RESTORE ⟨ *line number* ⟩

Function : This statement positions the data read pointer at the beginning of the first data table to be read by a READ statement, or to a data table with a specified line number.

Description : Given no operand, this statement restores the data read pointer to the beginning of the first data table; that is, to the beginning of the DATA statement with the smallest line number. If ⟨ *line number* ⟩ is specified in the operand, the pointer is restored to the corresponding DATA statement, or to the beginning of the following DATA statement.

3.3 Loop Statement

3.3.1 FOR ~ NEXT

Format : FOR *cv* = *iv* TO *fv* ⟨STEP *sv*⟩

. . . . .

NEXT ⟨*cv*⟩

cv .... *control variable* : numeric variable or array element

iv .... *initial value* : numeric constant, variable, array element or expression

fv .... *final value*

sv .... *step value*

Function : These statements cause a specified routine to be repeated.

Description: The *control variable* for the repeat block (loop) consisting of the FOR ~ NEXT statements is first filled with an *initial value.* The NEXT statement is executed at the end of the routine. The increment specified by *step value* is then added to the value of *control variable*.

If the sum is below the *final value*, program execution returns to the executable statement directly behind the FOR statement to repeat the routine.

The *step value* is usually a positive increment, but negative ones may be used as well. When the *step value* set is a negative increment, the *final value* must be set below the *initial value.* The loop is executed until the value of the *control variable* is smaller than the *final value.* If the *step value* is not given, the increment is fixed at 1.

Multiple loop

FOR ~ NEXT loops may be overlapped in multiple layers. In such cases, however, these loops must be nested. Inner loops must be entirely contained within outer loops, and all loops must use different *control variables.*

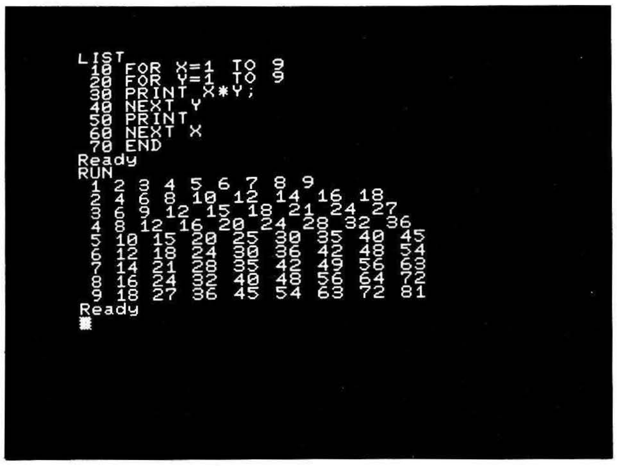
The nest may contain a maximum of 15 loops. If multiple layers end at the same location, then they can be collected in one NEXT statement. At such times, the operand of the NEXT statement must contain a string of *control variable* labels that are separated with commas, starting with the label of the innermost loop.

If the NEXT statement corresponds to the last FOR statement, its *control variable* label can be omitted.

Example : **Program**

```
10   FOR X = 1 TO 9
20   FOR Y = 1 TO 9
30   PRINT X*Y ;          inner loop     outer loop
40   NEXT Y
50   PRINT
60   NEXT X
70   END
```

Number of nested layers is 2. FIGURE 3.3 shows a result of execution of this multiple loop.



```
LIST
 10 FOR X=1 TO 9
 20 FOR Y=1 TO 9
 30 PRINT X*Y;
 40 NEXT Y
 50 PRINT
 60 NEXT X
 70 END
Ready
RUN
 1  2  3  4  5  6  7  8  9
 2  4  6  8  10  12  14  16  18
 3  6  9  12  15  18  21  24  27
 4  8  12  16  20  24  28  32  36
 5  10  15  20  25  30  35  40  45
 6  12  18  24  30  36  42  48  54
 7  14  21  28  35  42  49  56  63
 8  16  24  32  40  48  56  64  72
 9  18  27  36  45  54  63  72  81
Ready
```

FIGURE 3.3

Incorrect : **Program**

```
  ⋮
100   FV = 30
110   FOR N = 0 TO FV STEP 3
120   PRINT A$(N)
130   FOR I = 1 TO 5
140   PRINT A (N, I),            loop are not nested
150   NEXT N, I
  ⋮
```

If the system encounters a NEXT statement without a corresponding FOR statement, Error 13 occurs.

If more than 15 loops are nested, Error 11 occurs.

3.4 Branch statements

3.4.1 GOTO

Format : GOTO *lr*

lr . . . reference line number

Function : This statement causes program operation to jump unconditionally to the statement on line number *lr*.

Description: If the statement on line number *lr* is executable, it and subsequent statements will be executed; if it is non-executable, program execution jumps to the first subsequent executable statement.

3.4.2 GOSUB ~ RETURN

Format : GOSUB *lr*

. . . .

RETURN

lr . . . reference line number

Function : This statement unconditionally transfers program operation to the subroutine beginning on line number *lr* ; after execution of the subroutine, program operation is returned to the line immediately following the GOSUB statement by a RETURN statement at the end of the subroutine.

Description: A subroutine is a sequence of statements that is used to process a specific type of problem at one or more points in the program. Use of subroutines to handle repetitive occurrences of such problems allows the available BASIC text area to be used more efficiently by shortening the program text, contributing to systematic program construction.

Up to 15 levels of subroutines may be invoked either by the main program or by higher level subroutines.

The GOSUB statement achieves program branching, and the corresponding RETURN statement at the end of the subroutine causes program return. Each RETURN statement must, of course, correspond to one GOSUB statement.

3.4.3 IF ~ THEN

Format : IF *e* THEN *lr*

IF *e* THEN statement

e . . . . relational or logical expression

lr. . . . *reference line number*

Function : This statement evaluates the state described by a relational or logical expression, and causes conditional branching depending upon the result.

Description: If a relational expression is true, or if the specified conditions of a logical expression are fulfilled, the program process following THEN in the sentence is executed. When a *reference line number* follows THEN, program operation jumps to the indicated line. THEN can also be followed by other statements, which allows constructions such as IF THEN IF ~ .

If a relational expression is false, or if the specified conditions of a logical expression are not satisfied, program execution advances to the following line and the program process following the IF ~ THEN statement is ignored. The logical diagram below illustrates the function of the IF ~ THEN statement.

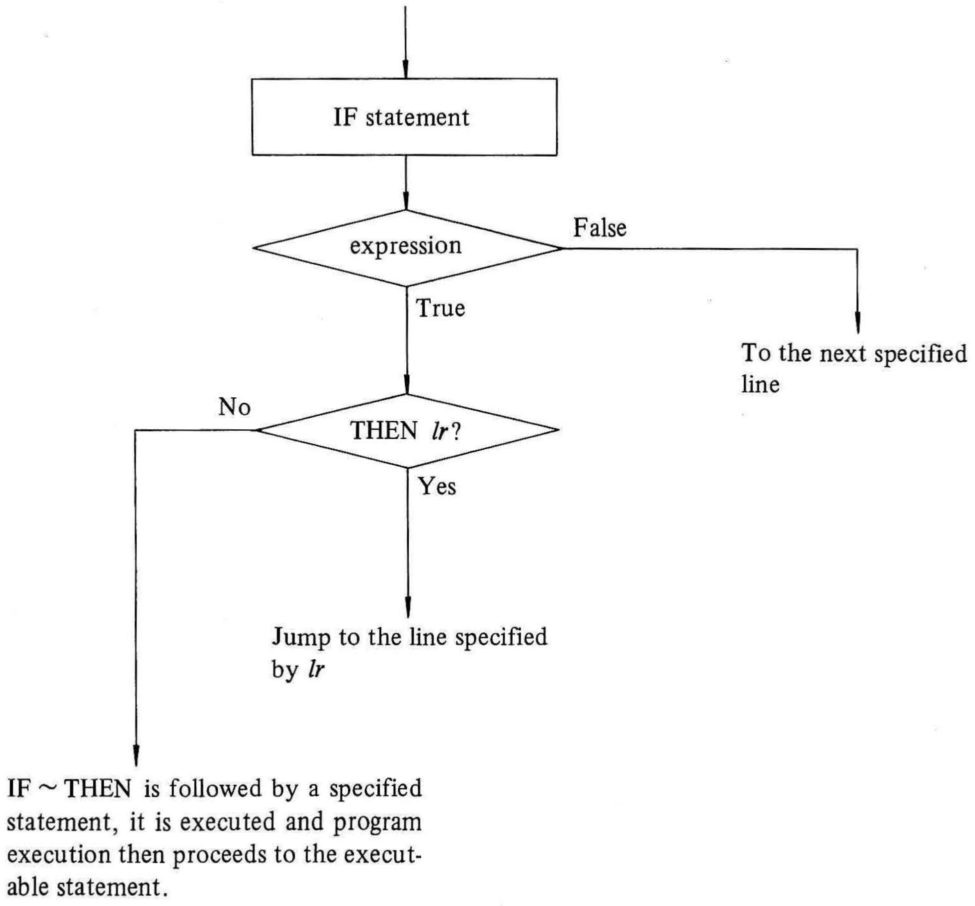

IF statement
expression
False
True
To the next specified line
No
THEN lr?
Yes
Jump to the line specified by lr
IF ~ THEN is followed by a specified statement, it is executed and program execution then proceeds to the executable statement.

FIGURE 3.4 Function of the IF ~ THEN statement

Cautionary notes on comparison of numeric data

Every numeric value handled by BASIC SB-5510 is internally represented in the binary floating decimal point system. Binary notation does not always provide for exact representation of numbers other than integers; it should therefore be noted that when a relational or a logical expression which includes the result of a mathematical operation is subjected to a comparison test, the expected program operation may or may not be performed depending on whether there is any disagreement between the result of the mathematical operation and the expected result. Discrepancies may also exist between the value expressed in internal notation and that output in external notation. For example, even when the result of an operation is mathematically expected to be an integer, the value expressed in internal notation may not be an integer if any data other than integers is processed.

Example : Internal representation of the numeric value of (ten times the value obtained by dividing 1 by 10) and that of the integer 1 differ from each other.

Program

```
10  A = 1 / 10 *10
20  IF  A = 1  THEN  PRINT "TRUE" : GOTO  40
30  PRINT "FALSE"
40  PRINT  "A =" ; A
50  END
```

Operation

```
RUN . . . . . . . . . . . . . . . . entered from the keyboard
FALSE  }
A =  1 } . . . . . . . . . display on screen
```

This result shows that the internal value of A differs from the mathematically expected value since "TRUE" was not printed, and it also shows the error is not in the first 8 decimal places since the number display is 1 (the expected value). However, if the IF statement is of the form;

IF ABS (A − 1) < . 1E−8 THEN . . .

the value of A will be compared with 1 to an accuracy of $1/10^8$.

Practice : 1) Varying the accuracy of comparison, determine what degree of error exists between the value of A and 1.

2) Determine whether or not any error occurs when the arithmetic expression 10 A = 1 *10 / 10 is evaluated.

3.4.4 IF ~ GOTO

Format : IF *e* GOTO *lr*

e ...... relational or logical expression

lr ...... *reference line number*

Function : This statement evaluates the state represented by a relational or logical expression, and causes branching according to the result.

Description : This statement induces conditional branching in the same manner as the IF ~ THEN statement; the statement causes program operation to jump to line number *lr* when specified conditions are fulfilled, and advances program execution to the following line when the conditions are not fulfilled. IF ~ GOTO *lr* and IF ~ THEN *lr* have identical functions; however, multi-statements following an IF ~ GOTO statement have no meaning in program execution.

3.4.5 IF ~ GOSUB

Format : IF *e* GOSUB *lr*

e ..... relational or logical expression

lr ..... *reference line number*

Function : This statement evaluates the state represented by a relational or logical expression, and causes program operation to jump to a subroutine according to the result.

Description : This statement, like the IF ~ THEN statement, activates conditional branching; it calls the subroutine beginning on line number *lr* when specified conditions are satisfied. Return from the subroutine will be made to the first executable statement following the IF ~ GOSUB statement (or to a statement preceded by the IF ~ GOSUB statement when it includes a multi-statement).

3.4.6 ON ~ GOTO

Format : ON *e* GOTO lr_1 ⟨, lr_2 , lr_3 , ... , lr_n ⟩

e . . . . numeric variable, array element or expression

lr_i. . . . *reference line number*

Function : This statement causes program operation to jump to one of several specified line numbers according to the value of the expression *e*.

Description: The target lines of a transfer caused by ON ~ GOTO statements are specified by placing the corresponding *reference line numbers* after GOTO – with each line number separated by a comma (,). Any number of line numbers may be specified provided they can be placed on one line. Transfer to the first line specified will be made if the integer part of the value of the expression is 1; transfer to the second line specified if the value is 2, and so forth. If the value of the expression is less than 1, or if its integer part exceeds the number of line specifications made, program execution advances to the first executable statement following the ON ~ GOTO statement.

If the internally represented value of the expression is not an integer, its integer part decides the target line for the transfer; e.g., when the internal representation of value A is 1.9999999, transfer is made to the first target line specified. Accordingly, it is necessary to consider the internal representation of each value of the expression being used. (See the description of the IF ~ THEN statement.)

3.4.7 ON ~ GOSUB

Format : ON *e* GOSUB lr_1 ⟨, lr_2 , lr_3 , ... , lr_n ⟩

e . . . . . numeric variable, array element or expression

lr_i. . . . . *reference line number*

Function : This statement calls one of several subroutines whose line numbers are specified.

Description: The ON ~ GOSUB statement is fundamentally the same as the ON ~ GOTO statement, but differs in that program operation returns to the first executable statement following the ON ~ GOSUB statement after execution of the subroutine.

3.5 Definitive statements

3.5.1 DIM

Format : DIM $a_1\ (i_1)\ \langle,\ a_2\ (i_2), \ldots\ldots, a_n\ (i_n)\rangle$
DIM $b_1\ (i_1, j_1)\ \langle,\ b_2\ (i_2, j_2), \ldots, b_n\ (i_n, j_n)\rangle$
ai . . . . one-dimensional array
bi . . . . two-dimensional array
in, *jn* . . . . dimensions

Function : This statement declares the dimensions of one-dimensional or two-dimensional arrays and secures necessary memory area.

Description: Use of either one-dimensional or two-dimensional arrays (numeric or string arrays) requires that the size of each array be declared by the DIM statement.
The subscripts which indicate the elements of an array can be expressed with any numbers from 0 to 255, but the range of usable numbers may be limited according to how the memory is used.
When two or more arrays are declared they must be punctuated with commas (,).

Array declaration for the one-dimensional array (list)

DIM A (20). . . . . This prepares 21 array elements – A (0) to A (20) – for one-dimensional numeric array A ( ).

DIM ST$ (99) . . . This prepares 100 array elements – ST$ (0) to ST$ (99) – for one-dimensional string array ST$ ( ).

Array declaration for the two-dimensional array (table)

DIM N1 (5, 11) . . . . This prepares 72 (6×12) array elements – N1 (0, 0) to N1 (5, 11) – for two-dimensional numeric array N1 (,).

DIM S$ (11, 30) . . . . This prepared 372 (12×31) array elements – S$ (0, 0) to S$ (11, 30) – for two-dimensional string array S$ (,).

On execution of the DIM statement, all elements of the declared array are loaded with 0 (zeroes), or "" (null string).
If any array is specified which is larger than the declared array size, Error 7 (Dimension Overflow) occurs.
Execution of the CLR statement disables all array declarations.

3.5.2 DEF FN

Format : DEF FN $f(x) = e$

f . . . . name of function: one of the alphabetical capital letters A~Z

x . . . . name of variable: one of the alphabetical capital letters A~Z

e . . . . numeric expression

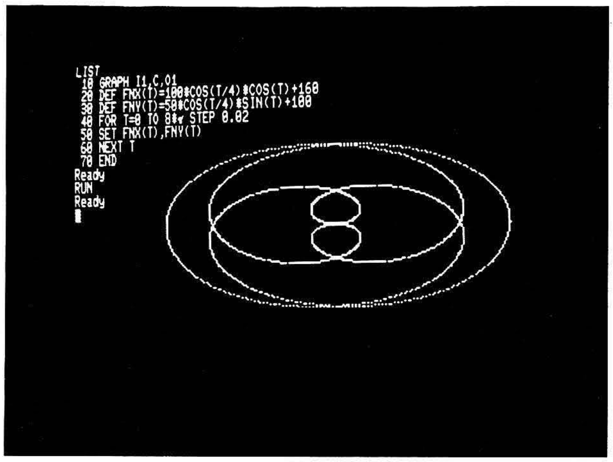
Function : This statement can be used to define any single-variable function.

Description: The name of each function to be defined can be specified by putting an appropriate alphabetical character directly after FN, and the name of the variable to which the desired function is to be assigned can be specified by placing the corresponding set of characters in parentheses following the name of the function. Accordingly, a maximum of 26 user functions may be concurrently defined. The statement may also be used to define nested functions by placing the definitive expression of a previously defined function in the function which is currently being defined. Nested functions may contain a maximum of 5 levels of definitions.

If functions are nested beyond this limit, Error 12 (Function Nesting) occurs. A single DEF statement can define only one user function.

Examples of definition of commonly used functions other than those built into the system (trigonometric, inverse trigonometric, and hyperbolic functions) are given in Table A.4 at the end of this manual.

Example : FIGURE 3.5 shows a graphic display using two defined functions.



```
LIST
 10 GRAPH I1,C,01
 20 DEF FNX(T)=100*COS(T/4)*COS(T)+160
 30 DEF FNY(T)=50*COS(T/4)*SIN(T)+100
 40 FOR T=0 TO 8*π STEP 0.02
 50 SET FNX(T),FNY(T)
 60 NEXT T
 70 END
Ready
RUN
Ready
█
```

FIGURE 3.5

Correct : 10 DEF FNA (X) = TAN (X – π / 6)

20 DEF FNB (X) = FNA (X) / C + X

. . . . FNA (X), a function previously defined, is used in the definition of FNB (X). Nested function.

Incorrect : 10 DEF FNK (X) = SIN (X / 3 + π / 4), FNL (X) = X ˆ 2

. . . . DEF and FN are not coupled.

3.5.3 DEF KEY

Format : DEF KEY (*k*) = *s*

k . . . . *key number* : 1 ~ 10

s . . . . character string

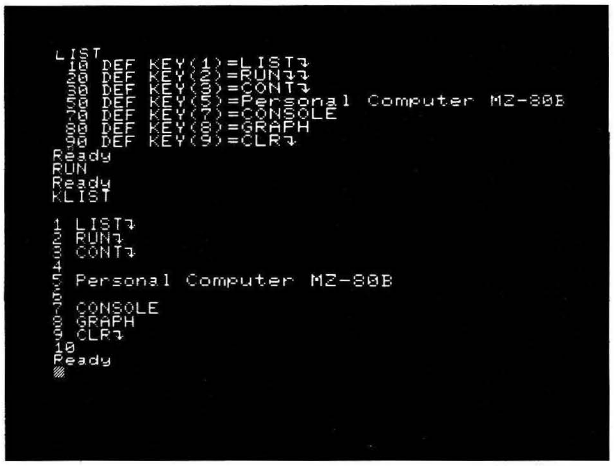
Function : This statement defines function for any of the ten function keys.

Description: A number from 1 through 10 is assigned to *key number*, and a string or command representing the function is indicated on the left of the definitive expression. The carriage return function may be included in each function assigned to a special function assigned to a special function key; if the SFT LOCK and GRPH keys are simultaneously pressed, the "⌐" symbol is input, so that the carriage return function is performed when the function key is pressed.

If two or more DEF KEY statements are executed against the same function key, only the last definition is valid.

When defining a multi-command or multi-statement to a special function key, use "!" as a separator. For example, when the multi-command, LOAD: RUN CR, is defined to special function key 1, execute DEF KEY (1) = LOAD! RUN ⌐

For the KLIST command, "!" is displayed as ":".



```
LIST
 10 DEF KEY(1)=LIST⌐
 20 DEF KEY(2)=RUN⌐⌐
 30 DEF KEY(3)=CONT⌐
 50 DEF KEY(5)=Personal Computer MZ-80B
 70 DEF KEY(7)=CONSOLE
 80 DEF KEY(8)=GRAPH
 90 DEF KEY(9)=CLR⌐
Ready
RUN
Ready
KLIST

1 LIST⌐
2 RUN⌐
3 CONT⌐
4
5 Personal Computer MZ-80B
6
7 CONSOLE
8 GRAPH
9 CLR⌐
10
Ready
▨
```

FIGURE 3.6

3.6 Remark statement and control statements

3.6.1 REM

Format :	REM *r*
	r . . . . a remark message
Function :	This statement refers to a comment statement contained in the program list.
Description:	The REM statement, a non-executable statement, serves as a comment statement to make it easy to review the program list. When this statement is encountered during program execution, execution jumps to the next executable statement.

3.6.2 STOP

Format : STOP

Function : This statement stops program execution and returns system operation to the command level.

Description: Execution of a STOP statement stops program execution and displays "Stop in *l*", where *l* is the number of the line at which program execution has been interrupted. This allows data content to be checked. The CONT command serves to restart program execution. (For details, see the CONT command.) The STOP statement, unlike the END statement, does not close any files (see below).

3.6.3 END

Format : END

Function : This statement terminated program execution and returns system operation to the command level.

Description: Execution of this statement terminates program execution, closes all files, displays the message "Ready", and returns system operation to the command level.

3.6.4 CLR

Format : CLR

Function : This statement clears all variables and arrays.

Description: This statement sets the value of all numeric variables to 0 (zero), and clears all string variables; it also cancels dimensional declarations for all arrays.
Hence, when an array is required after the CLR statement has been executed, it must be redeclared by executing the DIM statement.

3.6.5 CURSOR

Format : CURSOR *x*, *y*

x ...... X-coordinate : arithmetic expression

y ...... Y-coordinate : arithmetic expression

Function : This statement positions the cursor on the display.

Description: Messages issued by a PRINT or an INPUT statement appear beginning at the cursor position.

The CURSOR statement can move the cursor to any position on the display. Coordinate data carried by the CURSOR statement is represented by arithmetic expression. The data may be composed of integers within the following range for the 80-character and the 40-character modes:

- 80-character mode

  X-coordinate : 0 to 79

  Y-coordinate : 0 to 24

- 40-character mode

  X-coordinate : 0 to 39

  Y-coordinate : 0 to 24

If the value of arithmetic expression is not an integer, its decimal fraction is discarded. The Y-coordinate may extend beyond the scrolling area.

3.6.6 CSRH

Format : CSRH

Function : This is a system variable which indicates the current location of the cursor on the horizontal axis.

Description: The cursor position changes each time the CURSOR, PRINT, or INPUT statement is executed, and its X-coordinate is shown by this variable. The value this function takes stays within the following ranges for each character display mode:

80-character mode: $0 \leqq \text{CSRH} \leqq 79$

40-character mode: $0 \leqq \text{CSRH} \leqq 39$

3.6.7 CSRV

Format : CSRV

Function : This is a system variable which indicates the current location of the cursor on the vertical axis.

Description: The value CSRV takes stays within the following range for both character modes mentioned above:

$0 \leqq \text{CSRV} \leqq 24$

3.6.8 CONSOLE

Format : CONSOLE ⟨S*ls*, *le*⟩ ⟨, C *n*⟩ ⟨, R⟩ ⟨, N⟩

ls ...... start line of the scrolling area

le ...... end line of the scrolling area

n ...... number of the characters/line

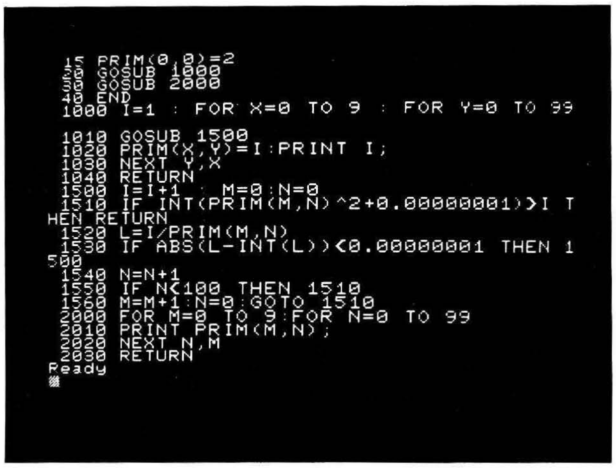
Function : This statement fixes the scrolling area of the display, changes the character display mode between 80 characters/line and 40 characters/line, or character and graphic display mode between reverse mode and normal mode.

Description: The operand of the CONSOLE statement determines which of three functions shown below are activated.

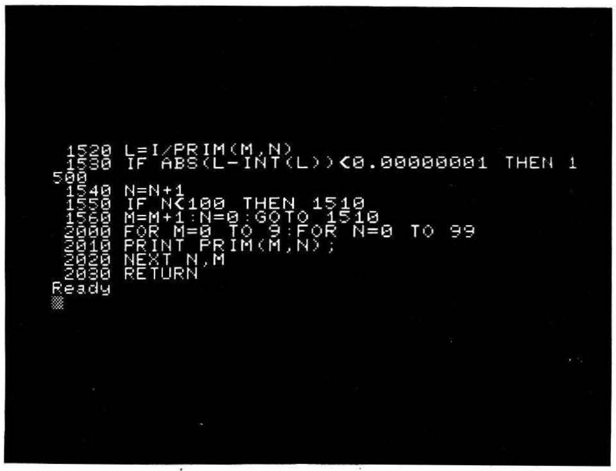
■ **Fixing the scrolling area**

CONSOLE S*ls*, *le* .... The top line refers to line 0 of the display and the bottom line to line 24. *ls* and *le* fix the scrolling area. Herce, $0 \leqq ls < le \leqq 24$.

This area, however, must cover at least three lines.



```
 15 PRIM(0,0)=2
 20 GOSUB 1000
 30 GOSUB 2000
 40 END
 1000 I=1 : FOR X=0 TO 9 : FOR Y=0 TO 99

 1010 GOSUB 1500
 1020 PRIM(X,Y)=I:PRINT I;
 1030 NEXT Y,X
 1040 RETURN
 1500 I=I+1 : M=0:N=0
 1510 IF INT(PRIM(M,N)^2+0.00000001)>I T
HEN RETURN
 1520 L=I/PRIM(M,N)
 1530 IF ABS(L-INT(L))<0.00000001 THEN 1
500
 1540 N=N+1
 1550 IF N<100 THEN 1510
 1560 M=M+1:N=0:GOTO 1510
 2000 FOR M=0 TO 9:FOR N=0 TO 99
 2010 PRINT PRIM(M,N);
 2020 NEXT N,M
 2030 RETURN
Ready
```



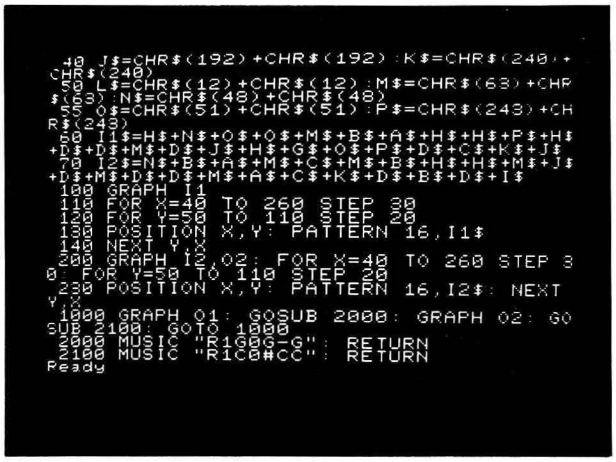
```
 1520 L=I/PRIM(M,N)
 1530 IF ABS(L-INT(L))<0.00000001 THEN 1
500
 1540 N=N+1
 1550 IF N<100 THEN 1510
 1560 M=M+1:N=0:GOTO 1510
 2000 FOR M=0 TO 9:FOR N=0 TO 99
 2010 PRINT PRIM(M,N);
 2020 NEXT N,M
 2030 RETURN
Ready
```

FIGURE 3.7

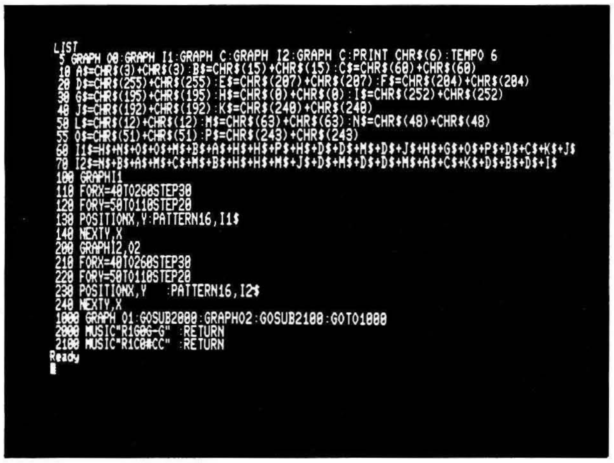
■ **Changing the character display mode**

CONSOLE C80 ...... This sets the character display mode to "80 characters/line".

CONSOLE C40 ...... This sets the character display mode to "40 characters/line".



```
 40 J$=CHR$(192)+CHR$(192):K$=CHR$(240)+
CHR$(240)
 50 L$=CHR$(12)+CHR$(12):M$=CHR$(63)+CHR
$(63):N$=CHR$(48)+CHR$(48)
 55 O$=CHR$(51)+CHR$(51):P$=CHR$(243)+CH
R$(243)
 60 I1$=H$+N$+O$+O$+M$+B$+A$+H$+H$+P$+H$
+D$+D$+M$+D$+J$+H$+G$+O$+P$+D$+C$+K$+J$
 70 I2$=N$+B$+A$+M$+C$+M$+B$+H$+H$+M$+J$
+D$+M$+D$+D$+M$+A$+C$+K$+D$+B$+D$+I$
 100 GRAPH I1
 110 FOR X=40 TO 260 STEP 30
 120 FOR Y=50 TO 110 STEP 20
 130 POSITION X,Y: PATTERN 16,I1$
 140 NEXT Y,X
 200 GRAPH I2,O2: FOR X=40 TO 260 STEP 3
0: FOR Y=50 TO 110 STEP 20
 230 POSITION X,Y: PATTERN 16,I2$: NEXT
Y,X
 1000 GRAPH O1: GOSUB 2000: GRAPH O2: GO
SUB 2100: GOTO 1000
 2000 MUSIC "R1G0G-G": RETURN
 2100 MUSIC "R1C0#CC": RETURN
Ready
```



```
LIST
 5 GRAPH O0:GRAPH I1:GRAPH C:GRAPH I2:GRAPH C:PRINT CHR$(6):TEMPO 6
 10 A$=CHR$(3)+CHR$(3):B$=CHR$(15)+CHR$(15):C$=CHR$(60)+CHR$(60)
 20 D$=CHR$(255)+CHR$(255):E$=CHR$(207)+CHR$(207):F$=CHR$(204)+CHR$(204)
 30 G$=CHR$(195)+CHR$(195):H$=CHR$(0)+CHR$(0):I$=CHR$(252)+CHR$(252)
 40 J$=CHR$(192)+CHR$(192):K$=CHR$(240)+CHR$(240)
 50 L$=CHR$(12)+CHR$(12):M$=CHR$(63)+CHR$(63):N$=CHR$(48)+CHR$(48)
 55 O$=CHR$(51)+CHR$(51):P$=CHR$(243)+CHR$(243)
 60 I1$=H$+N$+O$+O$+M$+B$+A$+H$+H$+P$+H$+D$+D$+M$+D$+J$+H$+G$+O$+P$+D$+C$+K$+J$
 70 I2$=N$+B$+A$+M$+C$+M$+B$+H$+H$+M$+J$+D$+M$+D$+D$+M$+A$+C$+K$+D$+B$+D$+I$
 100 GRAPHI1
 110 FORX=40TO260STEP30
 120 FORY=50TO110STEP20
 130 POSITIONX,Y:PATTERN16,I1$
 140 NEXTY,X
 200 GRAPHI2,O2
 210 FORX=40TO260STEP30
 220 FORY=50TO110STEP20
 230 POSITIONX,Y   :PATTERN16,I2$
 240 NEXTY,X
 1000 GRAPH O1:GOSUB2000:GRAPHO2:GOSUB2100:GOTO1000
 2000 MUSIC"R1G0G-G"  :RETURN
 2100 MUSIC"R1C0#CC"  :RETURN
Ready
```

FIGURE 3.8

■ **Changing the character and graphic display mode**

CONSOLE R .... This sets the character and graphic display mode to reverse mode.

CONSOLE N .... This sets the character and graphic display mode to normal mode.

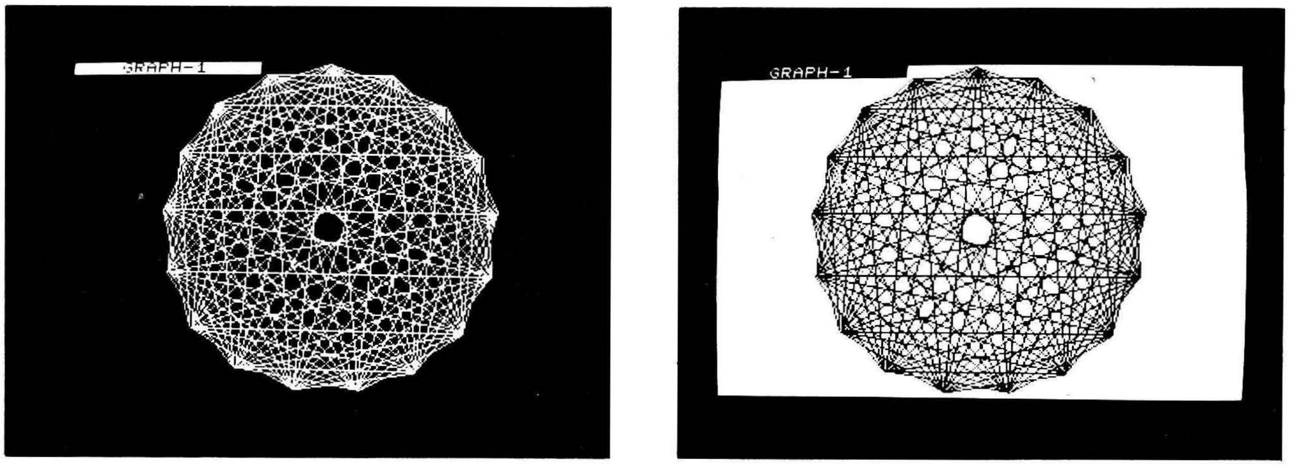


FIGURE 3.9

3.6.9 CHANGE

Format : CHANGE

Function : Reverses the function of the SHIFT key concerned with alphabetic keys.

Description : Small letters are input from the keyboard by pressing the SHIFT key after the BASIC interpreter has been activated. This is convenient for entry of BASIC commands and statements, which mainly consist of capital letters. However, it is natural to input messages with capitals entered by pressing the SHIFT key in the same manner as with an ordinary typewriter. This can be achieved by issuing the CHANGE statement. Issuing the CHANGE statement again returns the function of the SHIFT key to its original condition.

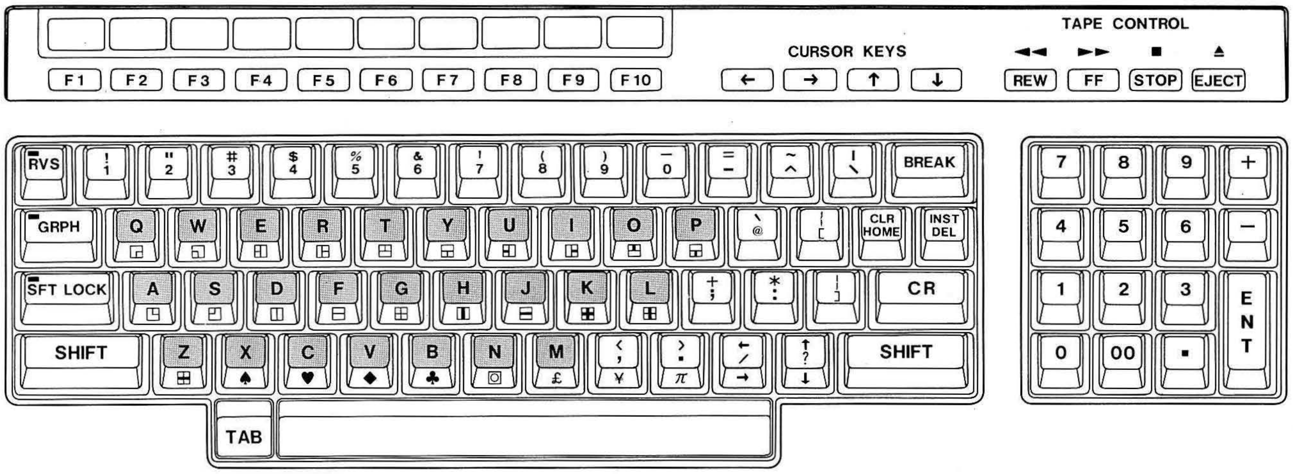


FIGURE 3.10 Locations of alphabetic keys

3.6.10 REW

Format : REW

Function : This statement rewinds the cassette tape.

Description : The REW statement operates in the same manner as the cassette deck REW key. The system executes the next statement immediately after the REW statement has been issued to the cassette tape deck. The cassette tape deck will automatically stop when rewind is completed. If no cassette tape is loaded, no operation is performed and the next statement is executed.

3.6.11 FAST

Format	:	FAST
Function	:	Fast-forward the cassette tape.
Description	:	The FAST statement operates in the same manner as the cassette deck FF key. The system executes the next statement immediately after the FAST statement has been issued to the cassette tape deck.

3.6.12 SIZE

Format	:	SIZE
Function	:	This variable shows the amount of unused memory area.
Description	:	In detail, this variable shows the number of bytes of unused BASIC program memory area. If "PRINT SIZE" is executed, the number of bytes of unused BASIC memory area is displayed on the screen in decimal notation.

3.6.13 TI$

Format	:	TI$
Function	:	This system variable is a 6-digit string which represents the current time told by the built-in clock.
Description	:	The clock indicates the current time with the six digit number of the TI$ variable as it ticks off seconds. This variable shows the hour with its first two digits, the minute with the next two digits, and the second with the last two digits. For example, when TI$ is "131753", it indicates that the clock time is 13 hours 17 minutes 53 seconds. TI$ begins with "000000" when the BASIC interpreter is activated. The clock may be adjusted to any time. To adjust the clock to 8:00 pm according to a time signal, enter the string TI$ = "200000" and input a carriage return at the moment the time signal occurs. Of course, indication is made within the following limits: 00 to 23 hours, 00 to 59 minutes, 00 to 59 seconds.

3.7 Music control statements

3.7.1 MUSIC

Format	:	MUSIC *x*$ *x*$ . . . . . string data
Function	:	This statement automatically plays music.
Description	:	Musical data is composed of string data which symbolize musical notes; the tempo of the musical performance is as set by the TEMPO statement.

The following indicates how the melody or sound effect is converted into string data.

Musical notes are assigned according to pitch (octave and scale) and duration;

a musical note . . . . . ⟨ *octave assignment* ⟩ ⟨ # *(sharp)* ⟩ *scale* ⟨ *duration* ⟩

Octave assignment

The sound range covers three octaves as shown in Figure 3.11.

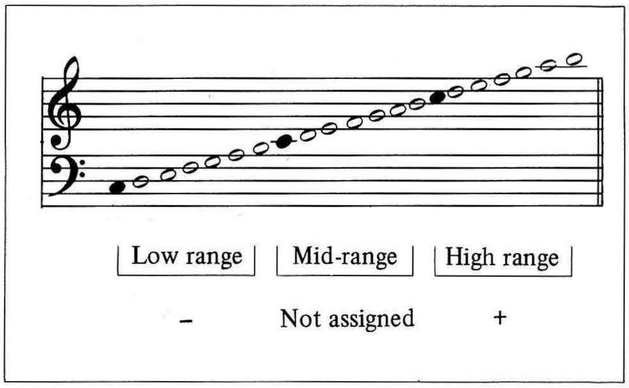


FIGURE 3.11

The black points indicate C notes, and the three C notes are separated by octave assignments as follows;

Low C . . . . –C
Middle C C
High C . . . . +C

Scale assignment

CDEFGAB and # are used for scale assignment.

Relationship between the scale and CDEFGAB is shown in Figure 3.12. The # symbol is used for semitone assignment.

Rests (no sound) are assigned with R.

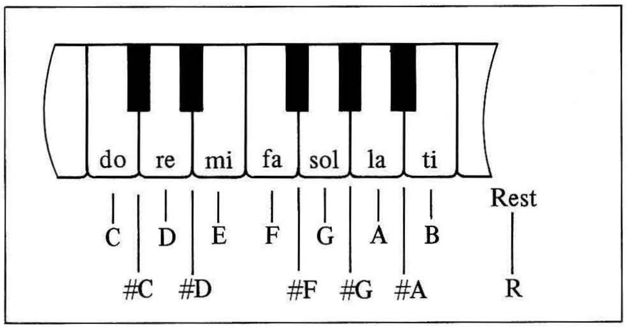
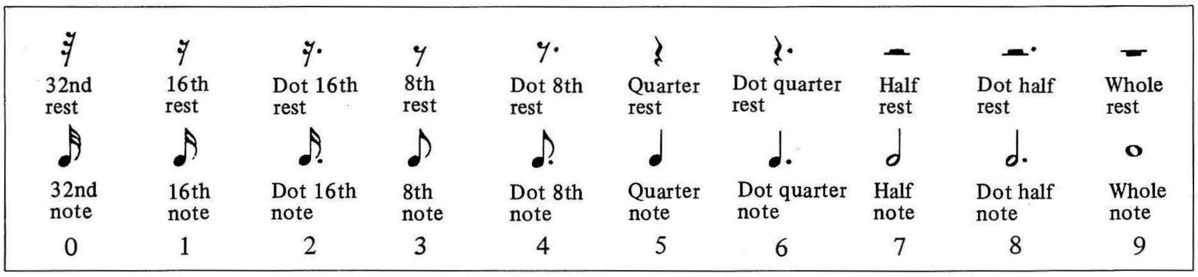


FIGURE 3.12

Duration assignment

This assignment determines the duration of a note whose pitch has already been assigned. Note durations from thirty-second to whole are assigned with numbers from 0 to 9 as shown in Figure 3.13. This assignment also applies to rests (R).



32nd rest	16th rest	Dot 16th rest	8th rest	Dot 8th rest	Quarter rest	Dot quarter rest	Half rest	Dot half rest	Whole rest
32nd note	16th note	Dot 16th note	8th note	Dot 8th note	Quarter note	Dot quarter note	Half note	Dot half note	Whole note
0	1	2	3	4	5	6	7	8	9

FIGURE 3.13

When notes of identical duration are repeated, duration assignment for the second note may be omitted.

If no duration assignment is made, program execution is carried out with quarter note durations (duration 5) regarded to be assigned.

Example : Substitute the 2 octave G major scale into G$ using quarter and eighth notes and play G$.

```
 10 G$="–G5–A3–BCDE#FG5A3B+C+D+E+#FG8R5"
 20 MUSIC G$
```

3.7.2 TEMPO

Format	:	TEMPO *x*
		x . . . . . an integer from 1 through 7
Function	:	This statement sets a musical tempo.

Musical notes contained in MUSIC statements are played at a speed corresponding to the tempo set with this statement.

Tempo data represented by *x* must be an integer from 1 through 7. "1" activates the slowest tempo and "7" the fastest tempo:

TEMPO 1	Lento, Adagio (slowest speed)
TEMPO 4	Moderato (medium speed)
TEMPO 7	Molto Allegro, Presto (fastest speed)

3.8 Graphic control statements

3.8.1 GRAPH

Format : GRAPH ⟨I*a*⟩ ⟨, O*b*⟩ ⟨, C⟩ ⟨, F⟩

a . . . . . graphic area number : 1 or 2

b . . . . . graphic area number : 1, 2, 12 or 0

Function : This statement sets the graphic input or output mode and clears or fills the graphic memory area.

Description : This statement performs the following four functions; the particular function performed depends on the operand of the GRAPH statement.

■ **Assignment of the graphic input mode to graphic areas**

GRAPH I1 . . . . . Assigns the graphic input mode to graphic area 1.

GRAPH I2 . . . . . Assigns the graphic input mode to graphic area 2.

■ **Assignment of the graphic output mode to graphic areas**

GRAPH O1 . . . . Assigns the graphic output mode to graphic area 1.

GRAPH O2 . . . . Assigns the graphic output mode to graphic area 2.

GRAPH O12 . . . Assigns the graphic output mode to graphic areas 1 and 2. (or O21)

GRAPH O0 . . . . Resets the graphic output mode.

■ **Clearing graphic areas**

GRAPH C . . . . . Clears the graphic area that is in the graphic input mode.

■ **Graphic area filling**

GRAPH F . . . . . Fills the graphic area that is in the graphic input mode.

Graphic display information is stored in areas that are in the graphic input mode by the GRAPH, SET, RESET, LINE, BLINE, or PATTERN statements, whole data contained in areas in the graphic output mode is displayed on the screen. Other information appearing on the screen may be overlaid with information stored in graphic areas 1 and/or 2.

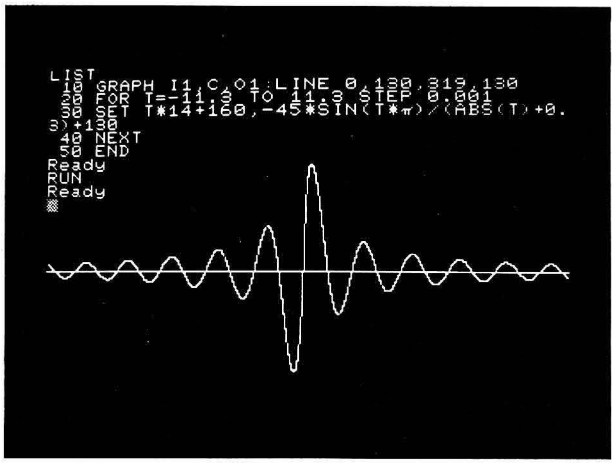
Operands of the GRAPH statement may be punctuated by commas (,).

Example : The GRAPH statement shown below clears graphic data from the display, puts graphic area 2 in the input mode, clears graphic area 2, puts graphic area 1 in the input mode, clears graphic area 1 and puts graphic area 1 in the output mode.

```
GRAPH   O0, I2, C, I1, C, O1
```

3.8.2 SET

Format	:	SET x, y x . . . . . numeric data : X-coordinates y . . . . . numeric data : Y-coordinates
Function	:	This statement sets a dot in any position in a graphic area operating in the input mode.
Description	:	The dot position is specified with X- and Y-coordinates. As shown in Figure 3.17, the X-coordinate of the graphic area can range from 0 to 319 – from left to right – and the Y-coordinate from 0 to 199 – from top to bottom. If specified coordinates lie outside of the graphic area they are ignored and no error occurs as long as the coordinates stay within the following ranges: $0 \leqq$ X-coordinate $\leqq 16383$ $0 \leqq$ Y-coordinate $\leqq 16383$ If the limits of the graphic area are exceeded during a sequence of operations, interruption of program execution does not occur as long as data stays within these ranges.
Example	:	Figure 3.14 shows a graphic display using the SET statement.



```
LIST
 10 GRAPH I1,C,O1:LINE 0,130,319,130
 20 FOR T=-11.3 TO 11.3 STEP 0.001
 30 SET T*14+160,-45*SIN(T*π)/(ABS(T)+0.
3)+130
 40 NEXT
 50 END
Ready
RUN
Ready
```

FIGURE 3.14

3.8.3 RESET

Format : RESET *x*, *y*

x . . . . . numeric data : X-coordinates

y . . . . . numeric data : Y-coordinates

Function : This statement resets any dot in a graphic area operating in the input mode.

Description : Specification of dot positions with X-Y-coordinates and the range of the out-of-area coordinates are the same as for the SET statement.

Example : The following program is intended to fill graphic area 1 and draw concentric circles with radii of 10, 20, 30 . . . , 150.

Coordinates outside the graphic area are ignored.

Figure 3.15.

Program

```
10 GRAPH I1, F,O1 : FOR R=0 TO 150 STEP 10
20 FOR T=0 TO 2 STEP 0.01
30 RESET R*COS (T*π) + 200, R*SIN (T*π) + 160
40 NEXT T, R
50 END
```

Operation

Figure 3.15 shows the result of execution this program.

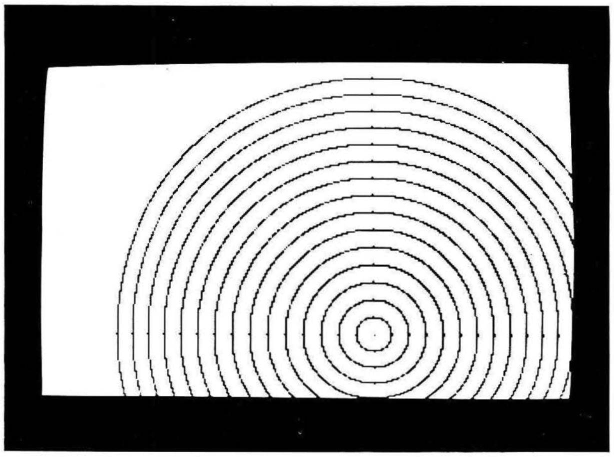


FIGURE 3.15

3.8.4 LINE

Format : LINE x_1, y_1, x_2, y_2 ⟨ $, x_3, y_3, \ldots, x_n, y_n$ ⟩

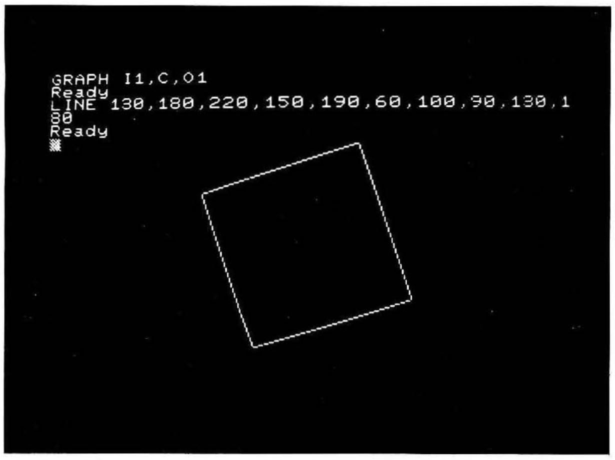
x_i .... numeric data : X-coordinates

y_i .... numeric data : Y-coordinates

Function : This statement draws a line in the graphic area that is in the input mode.

Description : This statement draws a line by setting dots from the first set of coordinates to the second set of coordinates. When the operand specifies three or more sets of coordinates, the system draws corresponding segments one after another.

Example : Drawing a square in the graphic area 1.



```
GRAPH I1,C,O1
Ready
LINE 130,180,220,150,190,60,100,90,130,1
80
Ready
```

FIGURE 3.16

3.8.5 BLINE

Format : BLINE x_1, y_1, x_2, y_2 ⟨ $, x_3, y_3, \ldots, x_n, y_n$ ⟩

x_i .... numeric data : X-coordinates

y_i .... numeric data : Y-coordinates

Function : This statement draws a black line in a graphic area.

Description : The procedure for drawing a black line and for describing the operand are the same as for the LINE statement.

3.8.6 POSITION

Format : POSITION x, y

x . . . . . numeric data : X-coordinates

y . . . . . numeric data : Y-coordinates

Function : This statement sets the location of the position pointer in the graphic area.

Description : The PATTERN statement is executed starting at position coordinates indicated by the position pointer. The position pointer moves each time any one of these statements is executed. The POSITION statement specifies the coordinates to which the pointer is to be moved.

The position pointer can be set to any point within the following maximum and minimum limits.

$0 \leqq$ X-coordinate $\leqq 319$

$0 \leqq$ Y-coordinate $\leqq 199$

Because, the graphic display area extends only from 0 to 319 on the X axis and from 0 to 199 on the Y axis. Figure 3.17.

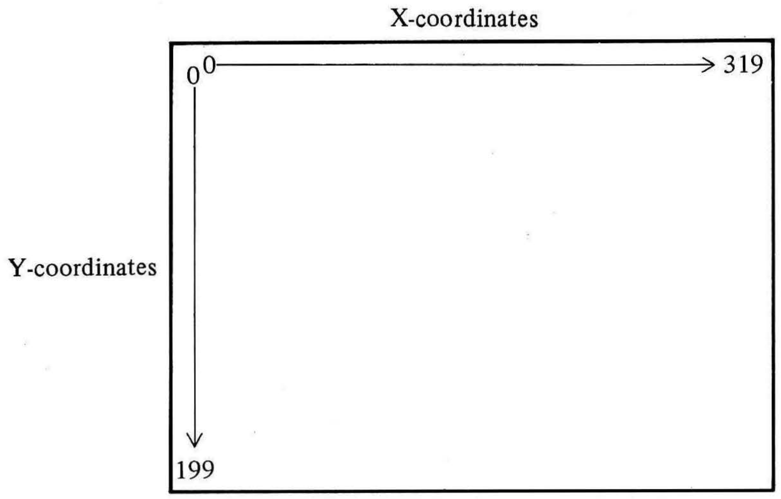


FIGURE 3.17

Position pointer

The position pointer indicates the dot position in the graphic area. The pointer is controlled by graphic control statement PATTERN.

Execution of a statement moves the position pointer to the position indicated.

Thus, the position pointer has a function similar to that of the cursor for character display.

3.8.7 PATTERN

Format : PATTERN ⟨ x_1 , ⟩ $x\$$ ⟨ , x_2 ⟩ ⟨ , $x_2\$$ ⟩ ..... ⟨ , x_n ⟩ ⟨ , $x_n\$$ ⟩

x_i ..... numeric data : number of layers

$x_i\$$ .... string data : dot pattern data in units of 8 bits

Function : This statement draws a desired dot pattern in the graphic area which is in the input mode.

Description : The statement includes a number of pairs of numeric values and string variables separated by commas.

The string variable specifies the arrangement of dots in a single line of the graphic pattern, and the numeric value indicates the number of layers of that line. The pattern is drawn starting at the location specified by the position pointer. The position pointer moves automatically as the statement is executed.

The direction in which lines are stacked depends upon the sign of the numeric value. If the value's sign is "–", stacking progresses from the top down; if the value is positive, stacking is performed in the opposite direction. Any number of layers of dots may be stacked, but any dots outside the graphic area are not displayed. When the operand does not include a numeric value to specify the number of layers, the data last specified serves as the default value.

The default value when BASIC is activated is "8". Dot patterns are given in units of 8 bits, and dot units of 8 dots each are set along the X axis from the location of the position pointer according to each 8 bit unit. Since each unit consists of 8 bits, there are 256 possible combinations of dots. Each combination is expressed as a binary number.

For example, the following dot-pattern is given by binary number "01001110", or decimal number 78. This dot pattern is represented by string data CHR$(78) or CHR$($4E).

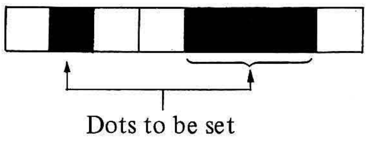


FIGURE 3.18

String constant "N" may be used in this case because CHR$(78) = CHR$($4E) = "N"

If A$ = "ABCDEFG", for example, the graphic pattern shown below is drawn when the following PATTERN statement is executed:

```
PATTERN –5, A$
```

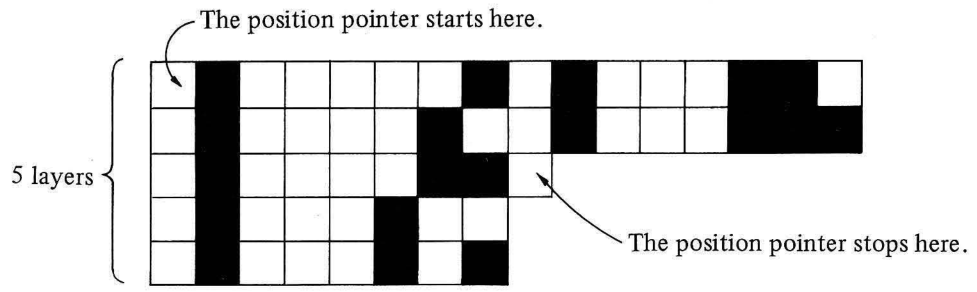

The position pointer starts here.
5 layers
The position pointer stops here.

FIGURE 3.19

The 8-dot pattern is drawn from the top downwards because the numerical value specified by the operand is negative –5, and "FG" moves to the right as it is drawn since a 5-layer stack is specified.

Example : Let's produce Gothic characters as a graphic pattern. Let capital letters be formed of 16 x 16 dot patterns and small letters of 16 x 8 dot patterns, keeping in mind that dots are set along the X-axis in units of 8 dots each. Dot pattern data is generated by writing each character on graph paper. For example, "A" is written on graph paper as shown in FIGURE 3.20. This 16 x 16 dot pattern data is stored in array CL$ (1). The layer is to be stacked from the top downward.

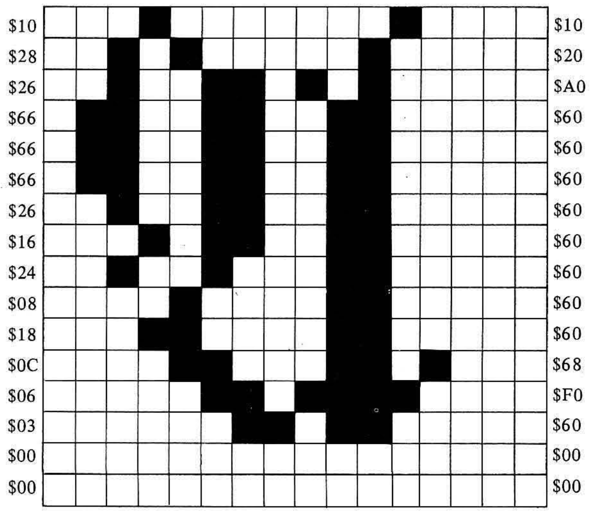
Then, the array is given by

CL$ (1) = CHR$ ($10) + CHR$ ($28) + ––––– + CHR$ ($00)

In a similar manner, CL$ (1) through CL$ (26) and SL$ (1) through SL$ (26) are given data. Then, issue

PATTERN –16, CL$ (1)

and Gothic type will be displayed on the CRT screen.



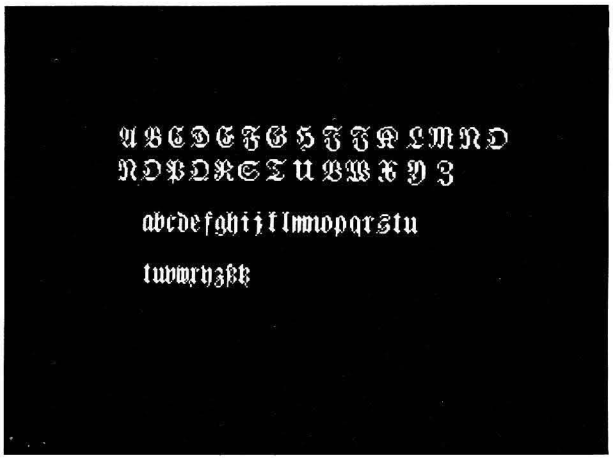


FIGURE 3.20 Die Fraktur

3.8.8 POINT

Format : POINT (x, y)

x . . . . . numeric data : X-coordinates

y . . . . . numeric data : Y-coordinates

Function : This function scans graphic areas to determine whether specified dots are set or reset.

The results are indicated by numerals 0 through 3.

Result of the POINT function	**Point information**
0	Points in both graphic areas 1 and 2 are reset.
1	Only point in graphic area 1 is set.
2	Only point in graphic area 2 is set.
3	Points in both graphic areas 1 and 2 are set.

When the system is provided only with graphic area 1, the result of the POINT function is 0 or 1.

With this statement, for example, the coordinates of the intersection of two curves can be obtained.

3.8.9 POSH

Format : POSH

Function : This is a system variable which indicates the current location on the horizontal axis of the position pointer in the graphic display area.

The value POSH takes stays within the following range:

$0 \leqq \text{POSH} \leqq 319$

3.8.10 POSV

Format : POSV

Function : This is a system variable which indicates the current location on the vertical axis of the position pointer in the graphic display area. The value POSV takes stays within the following range:

$0 \leqq \text{POSV} \leqq 199$

3.9 Data file input/output statements

3.9.1 WOPEN/T

Format : WOPEN ⟨ /T ⟩ ⟨ *file name* ⟩

Function : This statement opens a cassette file to allow a sequential data file to be written on cassette tape.

Description : The WOPEN/T statement declares "Write Open" for each sequential data file, and *file name* specifies the name of the sequential data file to be written.

If the WOPEN/T statement has been executed, numeric and string data are assembled in sequence in the memory when the PRINT/T statement is executed.

This data set is not stored on cassette tape as a sequential data file until the CLOSE/T statement is executed.

Nameless data files will result it no file names are specified. It is advisable to give to each data file a distinct name to indicate its contents. This will prevent confusing data files with each other.

3.9.2 PRINT/T

Format : PRINT/T d_1 ⟨ , d_2, d_3, ... , d_n ⟩

d_i . . . . . numeric or string data

Function : This statement writes data in succession to the cassette data file opened with the WOPEN/T statement.

Description : Execution of the PRINT/T statement writes the output list specified in its operand in succession on a cassette data file. Output lists can contain both numeric and string data. When assigning two or more sets of data to one PRINT/T statement, the data sets must be separated by commas (,).

Note : Do not attempt to replace or run a cassette tape while any file stored there is open; otherwise correct file control will become impossible.

Example : The following program creates the sequential data file "Name list" on the cassette tape. This file may have two hundred Name-strings.

```
10 WOPEN/T "Name list"
20 FOR N=1 TO 200:PRINT/T N$(N) : NEXT
30 CLOSE/T
```

3.9.3 CLOSE/T

Format : CLOSE ⟨ /T ⟩

Function : When the WOPEN/T statement has been executed, this statement closes WOPEN/T and creates a sequential data file on the cassette tape.
When the ROPEN/T statement has been executed, it closes ROPEN/T.

Description : When WOPEN/T is closed, the statement stores the sequential data file data list set up by the PRINT/T statement on cassette tape, under the file name declared by the WOPEN/T statement. Closing ROPEN/T allows ROPEN/T or WOPEN/T to be declared for other data files.

3.9.4 ROPEN/T

Format : ROPEN ⟨ /T ⟩ ⟨ file name ⟩

Function : This statement opens cassette files, enabling the system to read data from sequential data files on a cassette tape.

Description : The ROPEN/T statement declares "Read Open" for each sequential data file, and *file name* specifies the name of the sequential data file to be opened for reading. If the ROPEN/T statement has been executed, data stored on the cassette tape can be sequentially assigned to variables or array elements with the INPUT/T statement. After the data has been read, the CLOSE/T statement is executed to close the file.
If the ROPEN/T statement is not accompanied by a *file name*, it opens the BASIC sequential data file which is first found.

3.9.5 INPUT/T

Format : INPUT/T v_1 ⟨ , v_2 , v_3 , . . . , v_n ⟩
v_i . . . . numeric or string variable or array element

Function : This statement reads data in sequence from the cassette data file opened with the ROPEN/T statement.

Description : When the INPUT/T statement is executed, data read from the cassette file are assigned in succession to the variables or array elements of the input list specified by the INPUT/T statement's operand. Hence, the data list on the cassette·file and that of the INPUT/T statement must be identical in data type. Out of File occurs if the file data is exhausted before completion of the INPUT/T statement. If numeric data elements on a sequential data file read by the INPUT/T statement correspond to string variables (or array elements), they are assigned accordingly. When such numeric data is "5.17", for example, it is handled as a string data " 5.17".

Handling the file end

If "Out of File" occurs while data is being read from a sequential data file, an error is generated and the system returns to the BASIC command level. When the number of field data elements is known, the error is prevented by setting an equal number of data reads. When the number of filed data elements is unknown and program execution is not to be interrupted by "Out of File", it is advisable to attach the character string "END OF FILE" to the end of each sequential data file to indicate its end, thereby preventing the Error 63 (out of file).

3.10 Machine language control statements

3.10.1 LIMIT

Format : LIMIT *x*

x . . . . . *address* : numeric data or a four-digit hexadecimal number

Function : This statement limits the BASIC program memory area.

Description : When a machine language program is to be used in conjunction with a BASIC program, or when specific data is to be placed in memory, reservation of user memory area with the LIMIT statement is required to partition it from the BASIC program area. To divide the memory into two areas, the last address of the BASIC program area must be specified in the operand; the value of that address is specified with a numeric variable, or a hexadecimal number.

The LIMIT MAX statement is used to restore the original BASIC text area.

Refer to the memory map in the Appendix.

Example : LIMIT $DFFF . . . . . This statement limits the BASIC program memory area up to $DFFF (hexadecimal).

3.10.2 POKE

Format : POKE *x*, *d*

x . . . . . *address* : numeric data or a four-digit hexadecimal number

d . . . . . numeric data

Function : This statement stores data in arbitrary memory addresses.

Description : This statement writes one byte of data in the memory address specified in *address*; any address may be specified.

Each byte of data stored must have a binary value from 0 to 255. If it is numeric data, any integer from 0 to 255 may be stored; if it is string data, the ASCII code corresponding to its leading character is stored. The POKE statement may be executed for any memory address irrespective of the LIMIT statement, and thus may destroy the entire BASIC or MONITOR if not used with care.

The user area secured with the LIMIT statement is not used at all by the BASIC program. When entering machine language programs or data with the POKE statement, it is advisable to execute the LIMIT statement in advance.

3.10.3 PEEK

Format : PEEK (*x*)

x . . . . . *address* : numeric data or a four-digit hexadecimal number

Operation : This function gives the contents of the memory address indicated by the value of numeric data *x*. Since data in the memory consists of 8 binary digits (bits), a result from 0 to 255 will be obtained.

The value of *x* must be from 0 to 65535. Memory addresses may also be specified by four hexadecimal digits preceded by a $ sign. To store data in specified memory addresses, the POKE statement is used.

3.10.4 USR

Format : USR (*x*)

x . . . . . *address* : numeric data or a four-digit hexadecimal number

Operation : In this format, the USR function transfers program control to the memory address indicated by the value of numeric data *x*. This operation is the same as that resulting from CALL *x*, the machine-language command which causes branching to a subroutine. Accordingly, when the system encounters any return command – RET, RETcc – during execution of a machine-language program, program control is returned to the statement following the statement which executed the USR function.

The value of *x* must be in the range from 0 to 65535. Memory addresses may also be specified by four hexadecimal digits preceded by a sign ($).

Format : USR (*x*, *x*$)

x . . . . . . *address* : numeric data or a four-digit hexadecimal number

x$ . . . . . string data

Operation : When string data is given together with address data, this USR function places the first address of the memory area containing string data *x*$ in the CPU's DE register and the length of *x*$ in the BC register prior to execution of a CALL command contained in a machine-language program. This function can also serve to deliver string data used in a BASIC program to a machine-language program.

3.11 Printer control statements

Refer to the Printer (MZ-80P5) Manual for details of BASIC program operation and printer handling.

3.11.1 PRINT/P

Format	:	PRINT/P ⟨ e_1 d_1 e_2 d_2 . . . . . e_n d_n ⟩ e_i . . . . . Output data d_i . . . . . Separator or tabulation function
Function	:	This statement outputs print data (characters or control codes) to the printer.
Description	:	Print data output by PRINT/P statement are handled in nearly the same manner as the PRINT statement does with the display.

In detail, the PRINT/P statement causes the printer to print numeric data (value of numeric constants, numeric variables, numeric array elements or expressions) and string data (contents of string constants, string variables, string array elements, or string connective expressions), and uses separators ";", "," and the TAB function in the same manner as does the PRINT statement.

Only when transfer codes CHR$ (0) to CHR$ (30), or CHR$ ($00) to CHR$ ($1E), are output does the PRINT/P statement cause data processing different from that of the PRINT statement.

The character print mode has eight settings which determine the size of individual characters printed (or the number of characters per line) and whether to space lines or not as shown below. These eight settings are selected by CHR$ (16) to CHR$ (21) and maintained, once chosen, until another character print mode conversion code is received. Both character print mode conversion and control codes can be placed in any locations in the output data list.

CHR$ (5)	Home
CHR$ (6)	Cancels the enlarged character mode or reduced character mode, and sets the normal mode 1 with line spacing, and home.
CHR$ (16)	Sets the normal mode 1 with line spacing
CHR$ (17)	Sets the normal mode 2 without line spacing
CHR$ (18)	Enlarged character mode
CHR$ (19)	Cancels the enlarged character mode
CHR$ (20)	Reduced character mode
CHR$ (21)	Cancels the reduced character mode

3.11.2 IMAGE/P

Format : IMAGE/P *x*\$

x\$ . . . . . string data

Function : This statement causes the printer to draw a desired dot pattern according to the operating mode (image mode 1 or 2).

Description : The MZ-80P5 printer can operate in the normal mode, used for printing characters, and in two image modes.

Image mode 1

When the printer is operating in image mode 1, data output to the printer with the IMAGE/P statement are handled as following dot patterns.

Each dot pattern consists of 8 bits of data which determine the vertical arrangement of eight dots. Image mode 1 allows the printer to print 480 dot patterns on one line, so that the dots are arranged as a square lattice. CHR\$ functions 0 to 255 may be used to represent any possible arrangement of dots.

Image mode 2

When the printer is operating in image mode 2, output data are handled in the same manner as in image mode 1. Image mode 2 differs from image mode 1 in that it allows the printer to print 816 dot patterns on one line; horizontal spacing of dots is smaller than vertical spacing.

3.11.3 COPY/P

Format : COPY/P *n*

n..... display area : 1, 2, 3 or 4

Function : This statement causes the printer to copy an entire frame of data displayed on the computer screen.

Description : There are three types of data display: character display, graphic display of graphic area 1 and graphic display of graphic area 2. The type of data display to be copied is specified in the operand of the COPY/P statement. The statement may be used without regard for the mode in which the printer is operating.

Copying a character display

To copy a character display, the following COPY/P statement must be executed.

COPY/P 1

Copying a graphic display

Dot pattern data is output for printing in the same manner as in the image mode. Since the system has two graphic areas, the following three types of COPY/P statements are allowed:

COPY/P 2 ... This causes the printer to copy the dot patterns set in graphic area 1.

COPY/P 3 ... This causes the printer to copy the dot patterns set in graphic area 2.

COPY/P 4 ... This causes the printer to copy the dot patterns set in both graphic area 1 and graphic area 2.

The COPY/P statement can be used without regard for any GRAPH statements executed.

3.11.4 PAGE/P

Format : PAGE/P *x*

x ..... numeric data

Function : This statement sets the number of lines to be contained in one page of the printer.

Description : The printer normally prints 66 lines per page. When the number of lines is specified by the PAGE statement, the system forces the printer to print out information on the specified number of lines as one page. This holds true for both the character and image print mode.

3.12 I/O input/output statements

3.12.1 INP

Format : INP *@p*, *v*

p . . . . . *port number* : numeric data

v . . . . . numeric variable or array element, or string variable or array element

Function : The I/O *port number* is specified by *@p*. This number is identical to the I/O port number for the CPU which is represented by a decimal number ($0 \leqslant p \leqslant 255$).

Each I/O port is a parallel terminal of 8 bits, and is loaded with input data from 0 to 255. A number from 0 to 255 is assigned to each numeric variable, and a name from CHR$ (0) to CHR$ (255) is assigned to each string variable.

3.12.2 OUT

Format : OUT *@p*, *x*

p . . . . . *port number* : numeric data

x . . . . . numeric or string data

Function : The I/O *port number* is specified in the same manner as with the INP statement. Data to be output must range from 0 to 255. If the data is numeric data, a number from 0 to 255 is output; if it is string data, the ASCII code corresponding to its first character is output.

Use of the INP and OUT statements depends on how the I/O port is used. For details, consult the universal I/O card manual.

Chapter 4
BASIC SB-5510 Functions

This chapter lists all built-in functions of BASIC SB-5510 in the order of arithmetic functions, string control functions and tabulation functions. For functions relating to graphic display and machine language control, refer to the description of each group in the preceding chapter.

Any of the built-in functions can be called at any program location without prior definition.

4.1 Arithmetic functions

4.1.1 ABS

Format : ABS (x)

Function : This function gives the absolute value $|x|$ of numeric data x. That is, when $x \geqq 0$, ABS (x) = x, and when $x < 0$, ABS (x) = $-x$.

Example :

```
PRINT  ABS (3 − 8)
  5
Ready
```

4.1.2 INT

Format : INT (x)

Function : This function gives the largest integer smaller than x for numeric data x.

Example :

```
PRINT  INT (9.99), INT (π)
  9              3
Ready
PRINT  INT (−35.6)
−36
Ready
```

4.1.3 SGN

Format : SGN (x)

Function : This function ascertains whether the value of numeric data x is greater than, less than or equal to zero and indicates the result with 1, 0 or −1 as follows:

When $x > 0$, SGN (x) = 1

When $x = 0$, SGN (x) = 0

When $x < 0$, SGN (x) = −1

Example :

```
PRINT  SGN (SGN (A*(−A)) −0.5)
−1
Ready
```

4.1.4 SQR

Format : SQR (x)

Function : This function gives the square root $\sqrt{x}$ of numeric data x. x must be greater than or equal to zero.

Example : **Program**

```
10 FOR X = 1 TO 5
20 PRINT X, SQR (X)
30 NEXT
```

Operation

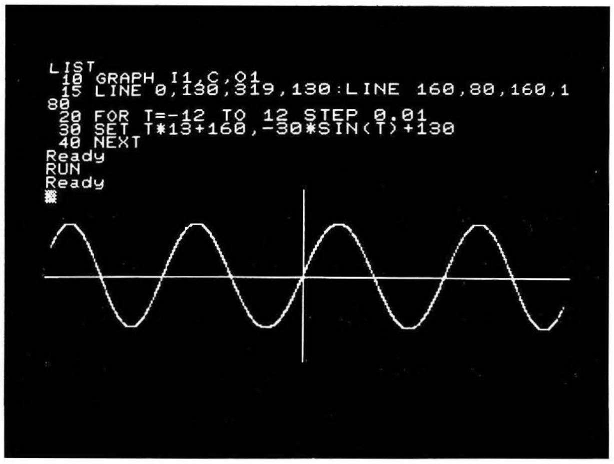
```
RUN
 1            1
 2            1.4142136
 3            1.7320508
 4            2
 5            2.236068

Ready
```

4.1.5 SIN

Format : SIN (x)

Function : This function gives the sine of numeric data x in radians.

Example : Figure 4.1 shows a sine curve.



```
LIST
 10 GRAPH I1,C,O1
 15 LINE 0,130,319,130:LINE 160,80,160,1
80
 20 FOR T=-12 TO 12 STEP 0.01
 30 SET T*13+160,-30*SIN(T)+130
 40 NEXT
Ready
RUN
Ready
```

FIGURE 4.1 A sine curve

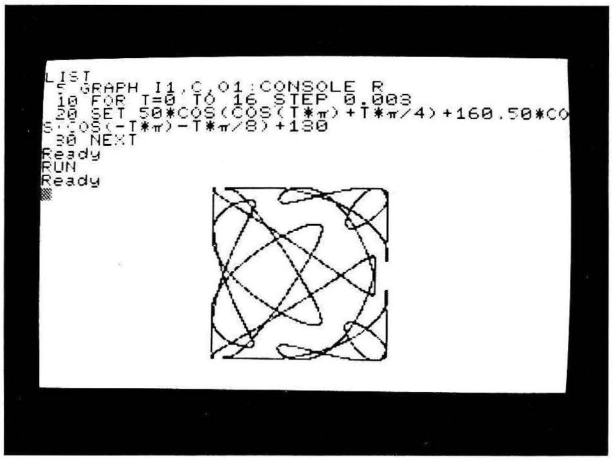
4.1.6 COS

Format : COS (x)

Function : This function gives the cosine of numeric data x in radians. When the value of numeric data D is given in degrees, its cosine is obtained by converting it to radians and applying the COS function is as follows:

COS (D*π/180)

Example : Figure 4.2 shows a curve.



```
LIST
 5 GRAPH I1,C,O1:CONSOLE R
 10 FOR T=0 TO 16 STEP 0.003
 20 SET 50*COS(COS(T*π)+T*π/4)+160,50*CO
S(COS(-T*π)-T*π/8)+130
 30 NEXT
Ready
RUN
Ready
```

FIGURE 4.2

4.1.7 TAN

Format : TAN (x)

Function : This function gives the tangent of numeric data x in radians. If the value of TAN (x) is too large, Error 2 (Operation result overflow) occurs.

4.1.8 ATN

Format : ATN (x)

Function : This function gives the arctangent of numeric data x (the angle whose tangent is x) in radians. Though arctan x has an infinite number of results, only the result between $-\pi/2$ and $\pi/2$ will be obtained (that is, the principal value). When the result is to be expressed in degrees, the expression ATN (x)*180/π is used.

4.1.9 EXP

Format : EXP (x)

Function : This function gives the value of exponential function e^x (the natural logarithmic base e raised to the x power) for numeric data x.

Example : PRINT EXP (1), EXP (2)

```
 2.7182818    7.3890561
Ready
```

4.1.10 LOG

Format : LOG (x)

Function : This function gives $\log_{10} x$ (the value of the common logarithm of x) for numeric data x.

x must be greater than 0.

When $A > 0$, $A \neq 1$ and numeric data X is given, $\log_A X$ (the value of the logarithm of X with the base A) can be obtained with either of the following expressions:

```
LOG (X) / LOG (A)
or
LN (X) / LN (A)
```

4.1.11 LN

Format : LN (x)

Function : This function gives ln x (the value of the natural logarithm of x) for numeric data x.

x must be greater than 0.

4.1.12 RND

Format	:	RND (x)
Function	:	This function generates pseudo-random numbers, which take any value between 0.00000001 and 0.99999999, and works in two manners depending on the value of numeric data x.

When the value of x is 0 or a negative number, the function gives the initial value of the pseudo-random number group it generates, and when the value of x is larger than 0, the function gives a pseudo-random number next to the one previously given in the pseudo-random number group.

Usually, a number larger than 0 is used as the value of x. However, initializing the pseudo-random number group by setting x at 0 or a negative number gives duplicability to the group.

4.2 String control functions

4.2.1 LEFT $

Format : LEFT \$ (*x*\$, *n*)

Function : This function gives string data comprised of the left *n* characters of string data *x*\$.

n must be a number between 0 and 255.

If $n >$ LEN (*x*\$), then LEFT \$ (*x*\$, *n*) = *x*\$; and if $n = 0$, then LEFT \$ (*x*\$, *n*) = " " (null string).

Example : **Program**

```
10 A$ = "Personal Computer MZ-80B"
20 B$ = LEFT $ (A$, 17)
30 PRINT B$
```

Operation

```
RUN
Personal Computer
Ready
```

4.2.2 MID $

Format : MID \$ (*x*\$, *m*, *n*)

Function : This function gives string data comprised of the *n* characters following the *m*th character from the beginning of string data *x*\$.

m must be a number between 1 and 255, and *n* a number between 0 and 255.

If $n >$ LEN (*x*\$) – *m*, then MID (*x*\$, *m*, *n*) = RIGHT \$ (*x*\$, LEN (*x*\$), – *m* +1); and if $n = 0$, then MID \$ (*x*\$, *m*, *n*) = " " (null string).

Example : **Program**

```
10 A$ = "Personal Computer MZ-80B"
20 C$ = MID $ (A$, 10, 8)
30 PRINT C$
```

Operation

```
RUN
Computer
Ready
```

4.2.3 RIGHT $

Format : RIGHT $ (x\$, n)

Function : This function gives string data comprised of the right n characters of string data x\$.

n must be a number between 0 and 255.

If n > LEN (x\$), then RIGHT \$ (x\$, n) = x\$; and if n = 0, then RIGHT \$ (x\$, n) = " " (null string).

Example : **Program**

```
  10 A$ = "Personal Computer MZ-80B"
  20 D$ = RIGHT $ (A$, 6)
  30 PRINT D$
```

Operation

```
RUN
MZ-80B
Ready
```

4.2.4 SPACE $

Format : SPACE $ (x)

Function : This function gives a string of successive spaces whose length is expressed by the value of numeric data x.

This function, when used with the PRINT statement, may serve as tabulation or to delete items displayed.

Example : **Program**

```
  10 PRINT SPACE $ (5); "Code"; SPACE $ (7); "Meaning"
```

Operation

```
RUN
     Code       Meaning
Ready
```

4.2.5 STRING $

Format : STRING $ (*x*$, *n*)

Function : This function gives a string of *n* repetitions of the first character of string data *x*$.

A string of consecutive spaces is given by the SPACE $ function.

Example : **Program**

```
10  R$ = STRING $ ("*", 10)
20  PRINT R$; " Table 1 "; R$
```

Operation

```
RUN
********** Table 1 **********
Ready
```

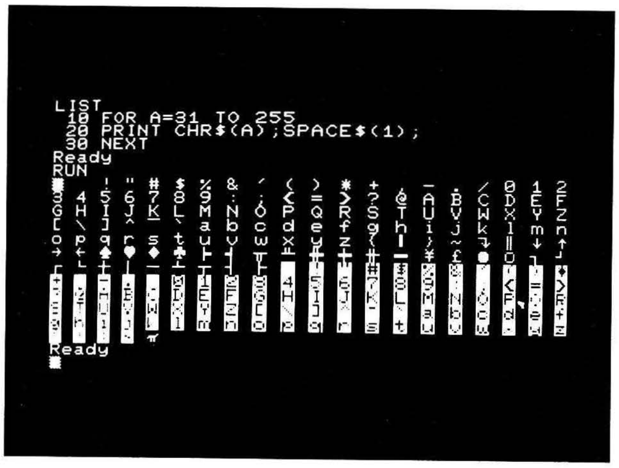
4.2.6 CHR$

Format : CHR$ (*x*)

Function : This function gives characters corresponding to ASCII codes expressed as numeric data *x*.

To convert characters into ASCII codes, the ASC function is used.

Example : Figure 4.3 shows all characters corresponding to ASCII codes 31 – 255. Operation of PRINT CHR$ (*x*) statement in case of ASCII codes 0 through 30 is summarized in pages 84 – 85.



```
LIST
 10 FOR A=31 TO 255
 20 PRINT CHR$(A);SPACE$(1);
 30 NEXT
Ready
RUN
■   ! " # $ % & ' ( ) * + , - . / 0 1 2
3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F
G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ ` a b c d e f g h i j k l m n
o p q r s t u v w x y z { | } ~
[illegible]
Ready
■
```

FIGURE 4.3

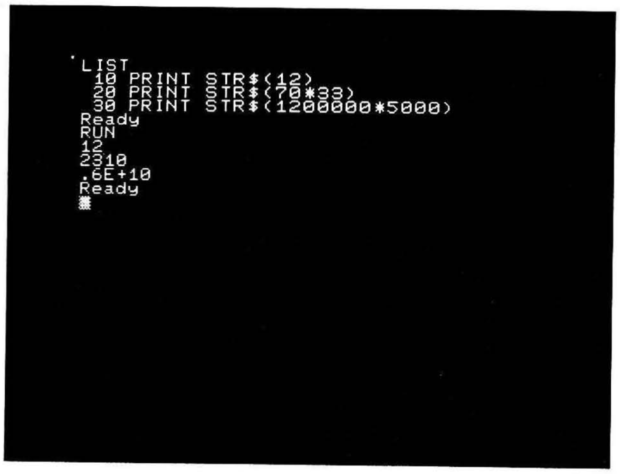
4.2.7 ASC

Format	:	ASC (*x*$)
Function	:	This function gives the ASCII code – a decimal number – of the first character of string data *x*$. For the relations between characters and ASCII codes, see the ASCII Code Table (Table A.2). To convert ASCII codes into characters, the CHR$ function is used.
Example	:	PRINT ASC ("ABC")

```
PRINT  ASC ("ABC")
  65
Ready
```

4.2.8 STR$

Format	:	STR$ (*x*)
Function	:	This function gives a string that expresses the value of numeric data *x*. This string will be expressed in scientific notation if that is the form in which the value of *x* is expressed. To convert a numeric string into a number, the VAL function is used.
Example	:	Figure 4.4 shows operation of the STR$ function.



```
LIST
 10 PRINT STR$(12)
 20 PRINT STR$(70*33)
 30 PRINT STR$(1200000*5000)
Ready
RUN
12
2310
.6E+10
Ready
▒
```

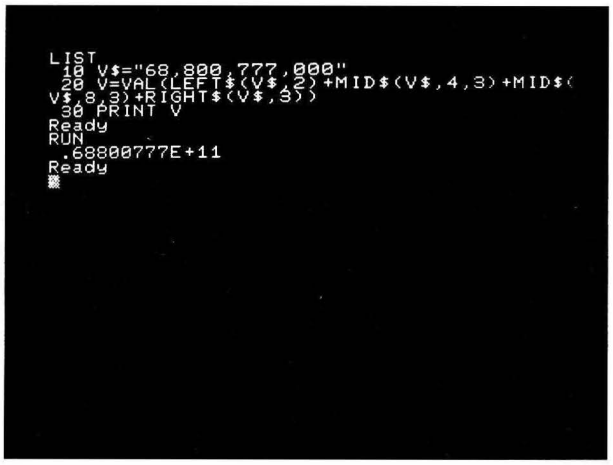
FIGURE 4.4

4.2.9 VAL

Format : VAL (*x*$)

Function : This function gives the numeric constant represented by string data *x*$. To convert numeric data into a string, the STR$ function is used.

Example : Figure 4.5 illustrates operation of the VAL function.



```
LIST
 10 V$="68,800,777,000"
 20 V=VAL(LEFT$(V$,2)+MID$(V$,4,3)+MID$(
V$,8,3)+RIGHT$(V$,3))
 30 PRINT V
Ready
RUN
 .68800777E+11
Ready
▒
```

FIGURE 4.5

4.2.10 LEN

Format : LEN (*x*$)

Function : This function gives the number of characters that make up string data *x*$, including spaces and characters which are not displayed by the PRINT statement.

Example : PRINT LEN ("Personal Computer MZ-80B")

```
 24
Ready
```

4.2.11 CHARACTER $

Format : CHARACTER $ (x, y)

Function : This function gives characters located on the screen in the positions specified by x and y as string data.

Coordinate data are given by arithmetic expressions, variables or constants.

Coordinates must lie within the ranges shown below.

- 80-character mode

  x-coordinate : 0 to 79

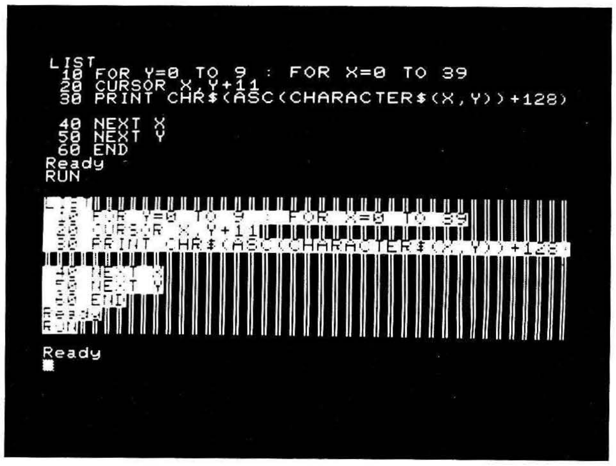
  y-coordinate : 0 to 24

- 40-character mode

  x-coordinate : 0 to 39

  y-coordinate : 0 to 24

Example : Figure 4.6 illustrates operation of the CHARACTER $ function.



```
LIST
 10 FOR Y=0 TO 9 : FOR X=0 TO 39
 20 CURSOR X,Y+11
 30 PRINT CHR$(ASC(CHARACTER$(X,Y))+128)
 40 NEXT X
 50 NEXT Y
 60 END
Ready
RUN
```

```
Ready
```

FIGURE 4.6

4.3 Tabulation function

4.3.1 TAB

Format : TAB (x)

Function : This function, used with the PRINT statement, causes the cursor to be moved forward to the line position indicated by numeric data x.

If the current cursor position exceeds the value of x, the tabulation function is inoperative (no action is performed).

The value of x must be between 0 and 255 (although 0 and 1 are considered identical).

APPENDIX

The Appendix includes the following;

- *ASCII Code Table . . . . . Table A.1*
- *BASIC interpreter SB-5510 Error Message Table . . . . . Table A.2*
  This table lists all the possible errors which may occur during program execution. The interpreter notifies the operator of occurrence of an error during program execution or operation in the direct mode with the corresponding error number.
- *Memory Map*
- *Trigonometric and hyperbolic functions*

A.1 ASCII Code Table

A table of hexadecimal ASCII codes is shown in FIGURE 2.22 of the Owner's Manual.

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0	NULL	26		52	4	78	N	104	h
1	↓	27		53	5	79	O	105	i
2	↑	28		54	6	80	P	106	j
3	→	29		55	7	81	Q	107	k
4	←	30		56	8	82	R	108	l
5	HOME	31	▒	57	9	83	S	109	m
6	CLR	32		58	:	84	T	110	n
7	DEL	33	!	59	;	85	U	111	o
8	INST	34	"	60	<	86	V	112	p
9	GRPH	35	#	61	=	87	W	113	q
10	SFT LOCK	36	$	62	>	88	X	114	r
11		37	%	63	?	89	Y	115	s
12	RVS	38	&	64	@	90	Z	116	t
13		39	'	65	A	91	[	117	u
14	L SCRIPT	40	(	66	B	92	\	118	v
15	RVS CANCEL	41	)	67	C	93	]	119	w
16		42	*	68	D	94	^	120	x
17		43	+	69	E	95	‾	121	y
18		44	,	70	F	96	`	122	z
19		45	−	71	G	97	a	123	{
20		46	.	72	H	98	b	124	|
21		47	/	73	I	99	c	125	}
22		48	0	74	J	100	d	126	~
23		49	1	75	K	101	e	127	↴
24		50	2	76	L	102	f		
25		51	3	77	M	103	g		

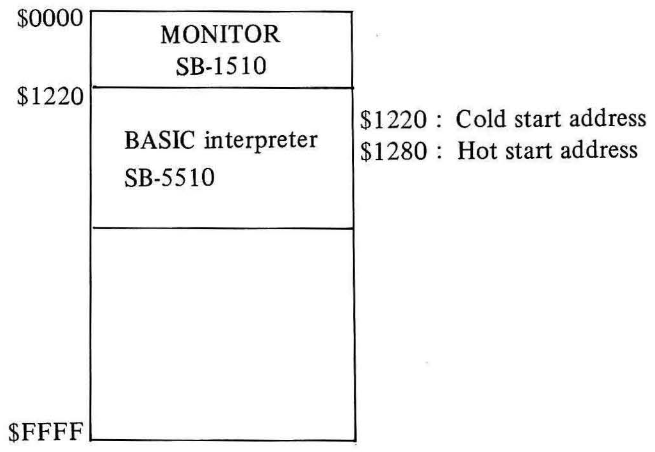
CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
128	║	154	│	180	4	206	N	232	h
129	↓	155	─	181	5	207	O	233	i
130	↑	156	┴	182	6	208	P	234	j
131	→	157	┬	183	7	209	Q	235	k
132	←	158	┤	184	8	210	R	236	l
133	♠	159	├	185	9	211	S	237	m
134	♥	160	█	186	:	212	T	238	n
135	♦	161	!	187	;	213	U	239	o
136	♣	162	"	188	<	214	V	240	p
137	╞	163	#	189	=	215	W	241	q
138	╡	164	$	190	>	216	X	242	r
139	╥	165	%	191	?	217	Y	243	s
140	╨	166	&	192	@	218	Z	244	t
141	╬	167	'	193	A	219	[	245	u
142	╪	168	(	194	B	220	\	246	v
143	╫	169	)	195	C	221	]	247	w
144	═	170	*	196	D	222	^	248	x
145	¥	171	+	197	E	223	‾	249	y
146	£	172	,	198	F	224	`	250	z
147	●	173	-	199	G	225	a	251	{
148	○	174	.	200	H	226	b	252	|
149	┐	175	/	201	I	227	c	253	}
150	┘	176	0	202	J	228	d	254	~
151	┌	177	1	203	K	229	e	255	π
152	└	178	2	204	L	230	f		
153	┼	179	3	205	M	231	g		

A.2 Error Message Table

Error No.	Meaning
1	Syntax error
2	Operation result overflow
3	Illegal data
4	Data type mismatch
5	String length exceeded 255 characters
6	Insufficient memory capacity
7	The size of an array defined was larger than that defined previously.
8	The length of a BASIC text line was too long.
9	
10	The number of levels of GOSUB nests exceeded 16.
11	The number of levels of FOR-NEXT loops exceeded 16.
12	The number of levels of functions exceeded 6.
13	NEXT was used without a corresponding FOR.
14	RETURN was used without a corresponding GOSUB.
15	Undefined function was used.
16	Unused line number was used.
17	CONT command cannot be executed.
18	A writing statement was issued to the BASIC control area.
19	Direct mode commands and statements are mixed together.
20	
21	
22	
23	
24	A READ statement was used without a corresponding DATA statement.
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	

Error No.	Meaning
36	
37	
38	
39	
40	
41	
42	
43	OPEN statement (ROPEN or WOPEN) was issued to a file which is already open.
44	
45	
46	
47	
48	
49	
50	
51	
52	
53	
54	
55	
56	
57	
58	
59	
60	
61	
62	
63	Out of file
64	
65	The printer is not ready.
66	Printer hardware error
67	Out of paper
68	
69	
70	Check sum error

A.3 Memory Map



A.4 Trigonometric and hyperbolic functions

Some functions which are not provided as built-in functions can be easily obtained by using built-in functions in combination as shown below:

secant	SEC(X) = 1/COS(X)
cosecant	CSC(X) = 1/SIN(X)
cotangent	COT(X) = 1/TAN(X)
arcsine	ARCSIN(X) = ATN(X/SQR(–X*X+1))
arccosine	ARCCOS(X) = –ATN(X/SQR(–X*X+1)) + 1.5708
arcsecant	ARCSEC(X) = ATN(SQR(X*X–1)) + (SGN(X)–1)*1.5708
arccosecant	ARCCSC(X) = ATN(1/SQR(X*X–1)) + (SGN(X)–1)*1.5708
arccotangent	ARCCOT(X) = –ATN(X) + 1.5708
hyperbolic sine	SINH(X) = (EXP(X) – EXP(–X))/2
hyperbolic cosine	COSH(X) = (EXP(X) + EXP(–X))/2
hyperbolic tangent	TANH(X) = –EXP(–X)/(EXP(X) + EXP(–X))*2+1
hyperbolic secant	SECH(X) = 2/(EXP(X) + EXP(–X))
hyperbolic cosecant	CSCH(X) = 2/(EXP(X) – EXP(–X))
hyperbolic cotangent	COTH(X) = EXP(–X)/(EXP(X) – EXP(–X))*2+1
hyperbolic arcsine	ARCSINH(X) = LOG(X + SQR(X*X+1))
hyperbolic arccosine	ARCCOSH(X) = LOG(X + SQR(X*X–1))
hyperbolic arctangent	ARCTANH(X) = LOG((1+X)/(1–X))/2
hyperbolic arcsecant	ARCSECH(X) = LOG((SQR(–X*X+1) + 1)/X)
hyperbolic arccosecant	ARCCSCH(X) = LOG((SGN(X)*SQR(X*X+1) + 1)/X)
hyperbolic arccotangent	ARCCOTH(X) = LOG((X+1)/(X–1))/2