Personal Computer

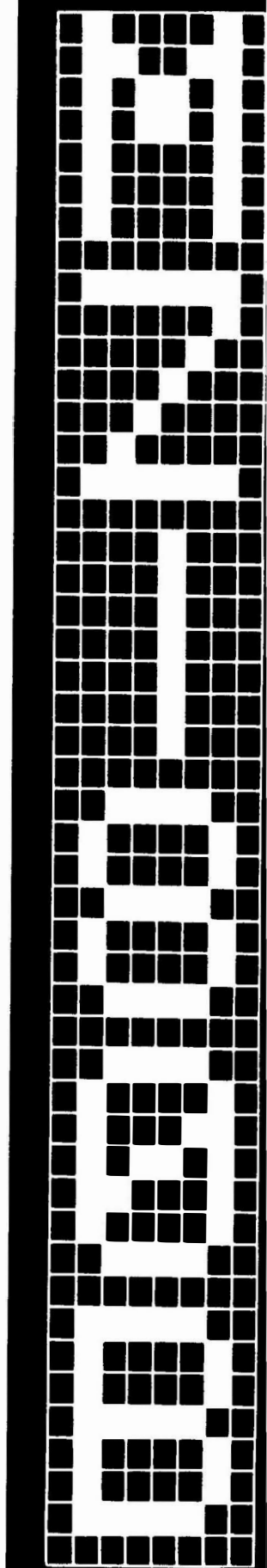
MZ-80B

Double Precision
DISK BASIC MANUAL

SHARP

Personal Computer
MZ-80B

Double Precision DISK BASIC MANUAL



Notice

This manual is applicable to the SB-6610 double precision DISK 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 or your nearest dealer for clarification.

Introduction

The greatest care must be taken in handling disk drives diskettes. Carefully read the notes in "Handling Diskettes" on page 69.

The master diskette and blank diskette will not be exchanged for new ones after purchase.

It is recommended that the master diskette be copied using the disk copy utility (refer to page 36) to generate a submaster diskette, and that the submaster diskette be used generally. Be sure to keep the master diskette in a safe place.

.Master Diskette Protection

A write protect seal is affixed to the write protect notch of the master diskette to prevent its contents from being accidentally erased through erroneous operation or accidents such as power failure.

Never remove the write protect seal; if it is damaged, replace it with a new one.

Contact your dealer for assistance if you should find any ambiguities in this manual.

Notes on converting BASIC text with BASIC Text Converter

When BASIC text for the double precision BASIC SP-6115 is converted with the Text Converter MZ-80T10C, "USING" of the statement "PRINT USING" is changed to "ERL" in the converted BASIC text program.

Therefore, you should change the "ERL" to "USING".

Contents

Chapter 1
Outline of double precision DISK BASIC SB-6610

This chapter outlines programming procedures and use of the double precision DISK BASIC interpreter SB-6610.

The chapter begins with a description of the procedure for activating the BASIC SB-6610, followed by general file control concepts.

For details of file control statements and use of the utility programs, see Chapter 2.

For other commands, statements, functions, operators and symbols, see Chapter 3.

1.1 Activating the DISK BASIC interpreter SB-6610

DISK BASIC SB-6610 is stored (along with MONITOR SB-1510) on a diskette file and must undergo initial program loading whenever it is to be used. Loading is easily performed. Ready the disk drive unit, place the master diskette (or submaster diskette†, if available) in disk drive 1 and simply turn on the power of the MZ-80B.

The MZ-80B's built-in IPL (Initial Program Loader) automatically starts loading both the DISK BASIC interpreter SB-6610 and the MONITOR SB-1510.

The SB-6610 automatically loads and executes the program assigned the file name "AUTO RUN" which is stored on the master (submaster) diskette. This program defines the functions assigned to the 10 special function keys. By assigning "AUTO RUN" to another program, the program can be automatically loaded and executed after IPL.

In this stage, system variables and default values are initialized as follows:

- Keyboard
  1) Operation mode : normal
  2) Lower case letters are entered with [SHIFT] key pressed.
  3) The function of each special function key is defined by program "AUTO RUN"

     F1: RUN ↵ F2: LIST↵ F3: CONSOLE↵ F4: CONT↵ F5: AUTO↵
     F6: CHR$ (F7: DIR FD1↵ F8: DIR FD2↵ F9: DIR↵ F10: LOAD ↵

  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 : ♩)

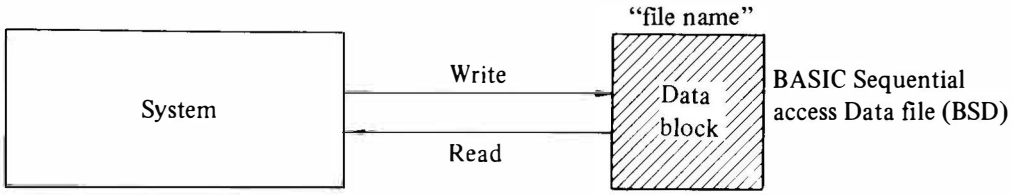
† Procedures for making a submaster diskette are explained on page 38.

1.2 Introduction to file control

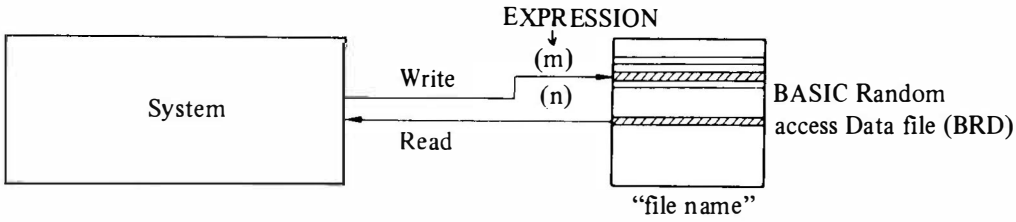
The DISK BASIC interpreter SB-6610 is a system software which has a superb file control function. It fully utilizes the large capacity and high speed accessing feature of the floppy disk file system so that files can be used not only for data storage but also as a random access data area connected to the system program. Further, with this interpreter disk files can be used as program segments which may be called for execution in job units by the program in memory with the CHAIN or SWAP statements.

Data files are classified into two groups according to the file access method: sequential access files and random access files.

A sequential access file is a block of file data which can be accessed sequentially. Data are accessed sequentially from the beginning by specifying the file name.



A random access file is a set of file data which can be accessed at random. Each data item is written in the file as an array element and is assigned with an expression with which the system controls it.



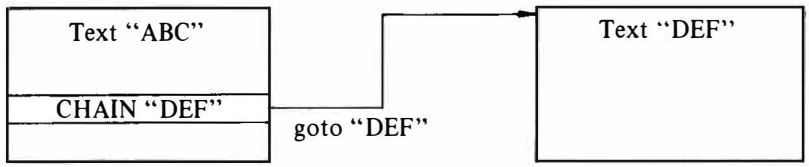
In general, when data can be treated in segments (e.g., decimal data used when coding a program by POKE statement) or it is arranged according to a certain rule (e.g., elements of a table), it is effective to write it as a sequential access file. When particular data items need to be accessed (e.g., in the case of information retrieval), it is effective to write it as a random access file.

To access data, first specify the file (a set of data assigned a file name) with a logical file number of 1 to 127. A logical file number is assigned to a file with a logical open statement as an alternative to the file name.

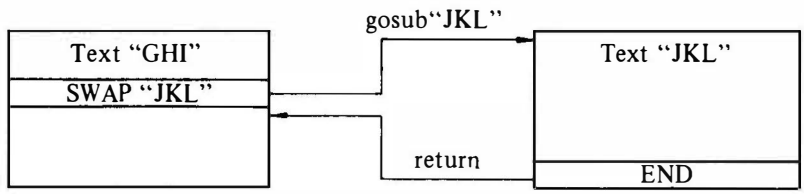
The file to which the specified logical file number has been assigned is accessed by the write or read command issued by a PRINT # or INPUT # statement or by a file close statement.

CHAIN and SWAP are statements which overlay a program upon another program in the memory and transfer control to the overlying program.

The CHAIN statement is used as a GOTO "file name" function.

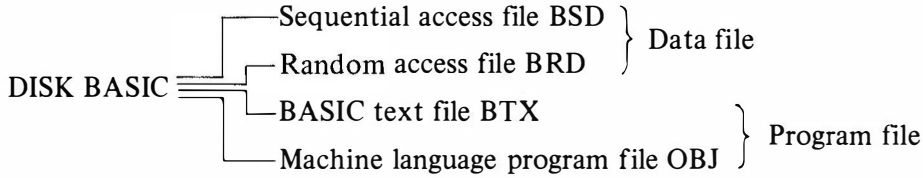


The SWAP statement is used as a GOSUB "file name" function. Control will be returned to the first program after the overlaid program has been completed. (In this case, overlay is performed again.)



Now we will discuss handling and control of various data and program files that make use of the large storage capacity and high-speed access function of our floppy disk unit.

We have already noted that DISK BASIC is capable of handling three kinds of files: two data files – the sequential access file (BSD) and random access file (BRD) – plus one program file – the BASIC text (BTX). One more file, the machine language program file (OBJ), has been constructed using a system program or MONITOR SB-1510 and recorded on the master diskette. This file is intended to be run alone or used linked with the program in the BASIC text area; hence, DISK BASIC can utilize it, but cannot write it or change its contents.



In discussing individual file control instructions, we will first explain procedures for constructing and using the two kinds of data files; second, we will explain use of the CHAIN and SWAP file statements.

1.3 Control of sequential access files

A sequential access file is a data file whose data is recorded or read with sequential access procedures, which accesses data sequentially starting with the first data item.

You already know how to handle data files on cassette tape using BASIC SB-5510. Sequential access with DISK BASIC is the same, except that the medium is not a cassette tape but a diskette. The method is, of course, far more practical and provides high speed access, enabling more versatile file control through seveal new file control statements.

First, let's compare the DISK BASIC and cassette-based BASIC sequential access statements.

Recording files (writing data)

	DISK BASIC	Cassette-based BASIC
File open statement	WOPEN #n, "File name"	WOPEN "file name"
Data write statement	PRINT #n, data	PRINT/T data
File close statement	CLOSE #n	CLOSE
Cancel statement	KILL #n	

Recalling files (reading data)

	DISK BASIC	Cassette-based BASIC
File open statement	ROPEN #n, "file name"	ROPEN "file name"
Data read statement	INPUT #n, variable	INPUT/T variable
File close statement	CLOSE #n	CLOSE
File end detection	IF EOF (#n) THEN	

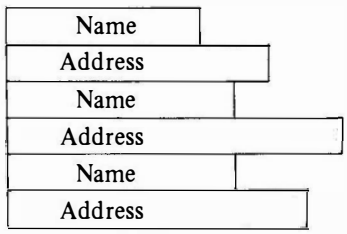
Note: The general form of a file open statement includes specification of the drive number and diskette volume number. These are not shown in the tables above.

As you can see, the statements of these two BASICs closely correspond to each other in composition. Did you notice that each DISK BASIC statement always contains the symbol "#n"? This is called a logical number (**any number from 1 through 127**), and it must be specified whenever a DISK-BASIC-based file is to be accessed.

Cassette-based BASIC permits either writing to or reading from only one file, whereas DISK BASIC statement utilizes the parallel arrangement of multiple files in the disk system to enable arbitrary access and control of multiple files (**maximum of 10 files**) simultaneously. In addition, it is deviced so that files opened can be defined with arbitrarily chosen logical numbers, making it unnecessary to write their file names every time you want to specify them thereafter.

(This difference is a result of differences in hardware cassette and diskette; that is, cassette files are essentially sequential in nature, while disk files are random.)

As a simple example of sequential access file control, let's discuss the recording of names and addresses of persons' homes in a sequential access file. Our file, an address list in this example, must be made in the following form.



file name = "ADDRESS LIST"

The reason the above rectangles are not the same length is that data recorded with the sequential access method is not fixed in length. In a random access file, as we will later mention, all data is given a fixed length of 32 bytes. When all data is handled in the blocks as in this example, or when most of the data (addresses in this example) is too long to be recorded in 32 bytes or is not fixed in length, the sequential access file may be more suitable.

Shown below is a program which causes the system to behave as follows: substitute string variables alternately with names and addresses with the INPUT statement, record a combination of names and addresses one by one to make "ADDRESS LIST" with 50 combinations in all, then read stored data out of the file (list) and display it on the CRT screen in groups of 10 items.

(Writing)

```
100 WOPEN #3, "ADDRESS LIST" .....
110 FOR P = 1 TO 50
120 INPUT "NAME="; NA$
130 INPUT "ADDRESS="; AD$
140 PRINT #3, NA$, AD$ .....
150 NEXT P
160 CLOSE #3 .....
                        :
                        :
                        :
                        :
                        :
                        .....
```

Difines the name of a sequential access file and opens it with logical number assigned.
In the program, the WOPEN statement defines the name of the sequential access file as "ADDRESS LIST" and assigns logical number 3 to it.

This statement, when it follows the WOPEN statement, generates a series of sequential access records. When the PRINT # statement is executed, the specified data record is added to a series of sequential access records. The sequential access file is not generated at this stage of program.

Writes a series of records generated by PRINT # statements in a file. A sequential access file (BSD) is generated when this statement is generated.

(Reading)

```
200 ROPEN #4, "ADDRESS LIST" .....
210 FOR P = 1 TO 5
215 FOR Q = 1 TO 10
```

Specifies the name of the sequential access file to be read and assigns a logical number to it.
In the program, "ADDRESS LIST" is specified and the logical number #4 is assigned to it.

```
220 INPUT #4, NA$, AD$ .....
230 PRINT NA$: PRINT AD$
240 NEXT Q
250 PRINT "STRIKE ANYKEY"
260 GET X$: IF X$ = " " THEN 260
270 NEXT P
280 PRINT "END"
```

This statement, when it follows the ROPEN statement, reads records in sequential access file "ADDRESS LIST" from the beginning one by one into the specified variable.

```
290 CLOSE #4 .....
```

The CLOSE statement, when it follows the ROPEN statement, ends execution of ROPEN and resets the logical file number assignment.

1.4 Control of random access files

A random access file is a data file which permits data to be recorded or recalled using the "random access" method. The term "**random access**" refers to the process of recording or recalling each data item by specifying it as an array element. Unlike sequential access files, random access files permit addressing any data elements included in a collection of data.

The PRINT # and INPUT # statements used in random access statements contain an "expression" which specifies the array elements following the logical number, as shown below. This is because random access files require designation of the arrays of data of which they are composed.

PRINT #n (expression), data

INPUT #n (expression), variable

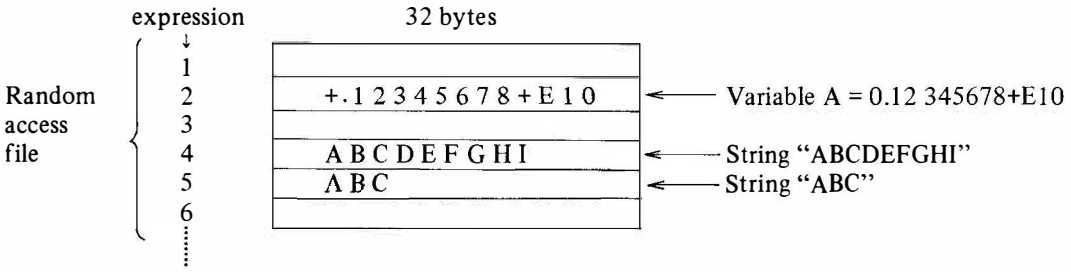
Array element designation

The "**expression**" must be given as a numerical value or variable. The statement

INPUT #7 (21), A$

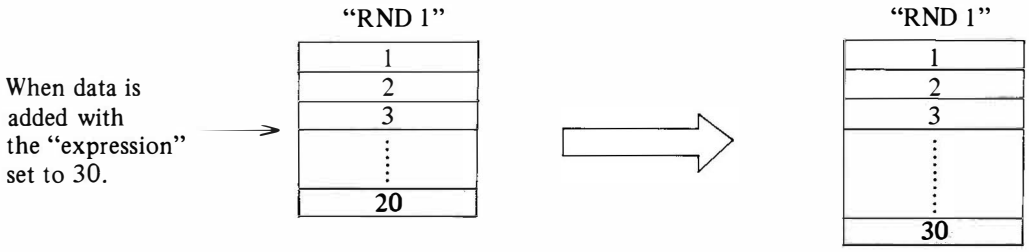
for example, commands the system to read the 21st data element of a group in a random access file opened with logical number #7 into variable A$.

Note that random data access requires that every data item be recorded in a fixed length. In other words, random access files require recording numeric and string variables in 32 bytes or less.



Numeric variables, including those expressed in exponential notation, do not usually exceed 32 bytes, whereas string variables may extend up to 255 bytes. String variables exceeding 32 bytes cannot be recorded in one data element of a random access file.

Another difference between random and sequential access files is that a random file can be expanded after it has been initially created. Given random access file "RND 1" recorded using an "expression" of 20, for instance, the file may be expanded to accommodate 30 "boxes" when data is newly entered with the "expression" set to 30.



Now, let's try to device a program for making a simple inventory list using a random access file. It is assumed that individual articles are given fixed item numbers from 1 to 50 and that the inventory list includes five fields of information: item name, unit price, number of units in stock, value (unit price X number items in) and comments.

When recording inventory data for each article, its item number must be entered first.

Recording inventory data

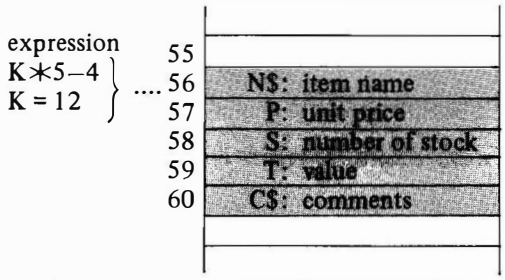
```
100 XOPEN #5, "STORE LIST" .....
110 INPUT "ITEM NO.="; K
120 IF K = 0 THEN 300
130 INPUT "ITEM NAME="; N$
140 INPUT "UNIT PRICE="; P
150 INPUT "NO. OF UNITS="; S
160 INPUT "COMMENTS="; C$
170 T = P * S
180 PRINT #5 (K * 5-4),
      N$, P, S, T, C$ .............
190 GOTO 110
300 CLOSE #5 .............
310 END                  .........
```

Specifies a file name and assigns a logical file number to the specified random access file for access. When a random access file is logically opened, no distinction is made between write and read by the open statement.
Logical number: 5 File name: "STORE LIST"

Writes data in the random access file assigned logical file number. The number of the first field in the record in which data is stored is indicated by the expression.

Closes the BRD opened by XOPEN and resets the logical file number definition.

A random access file made with the above program is as follows. If the item number assigned is K = 12, the five kinds of data entered are stored in elements indicated by the expressions corresponding to 56 through 60.



In this way, data can be arbitrarily arrayed in the file. Hence the file, unlike a sequential access file which is filled with data in succession, may include empty locations, providing for simple data rewriting. Next, let's devise a program to recall the random access file "STORE LIST" made as shown above and display inventory data for a certain article.

Recalling inventory data

```
500 XOPEN #17, "STORE LIST" ....
510 INPUT "ITEM NO.="; J
515 IF J = 0 THEN 700
520 INPUT #17 (J * 5-4),
      N$, P, S, T, C$ ..............
530 PRINT "NO."; J
535 PRINT "ITEM NAME:"; N$
540 PRINT "UNIT PRICE:"; P
550 PRINT "NO. OF UNITS:"; S
560 PRINT "VALUE:"; T
570 PRINT "COMMENTS:"; C$
580 GOTO 510
700 CLOSE #17 ..................
710 END
```

Specifies a file name to be read as "STORE LIST" and assigns logical file number 17 to it.

Reads the record indicated by the expression from the random access file assigned logical file number #17 into the specified variable.

Closes the BRD opened by XOPEN and resets the logical file number definition.

In this way, random access files enable the inventory data on specific articles to be called at once by inputting their article numbers, no matter how many articles are inventoried.

1.5 File access cancellation and how to detect file end

1.5.1 KILL #n

This statement, when it follows the WOPEN statement, cancels the WOPEN command.

The execution of KILL statement cancels the WOPEN and prevents the data array, even if it is under construction, from being recorded in the sequential access file. The statement is practical if the need for cancellation occurs during the construction of a sequential data array.

The KILL statement for the other use has same functions as the CLOSE statement.

1.5.1 How to detect file end

What is the result when the number of data reads exceeds the number of recorded items? In such cases, no error occurs and the variables are set with 0 or "null", then a special function, EOF (#n), detects file end. **EOF (#n), becomes "true" if it comes to the end of a file while data is being read with the INPUT # statement.** Hence, if the statement

```
IF EOF (#n) THEN
```

is placed after an INPUT # statement, instructions following THEN are executed when EOF (#n) becomes true (when the file end is detected).

The statement can be used in a random access file or a sequential access file to be read.

The following program reads string data from sequential access file "ABC" and displays it on the CRT screen until the file end is reached.

```
300 ROPEN #33, "ABC"
310 INPUT #33, A$
320 IF EOF (#33) THEN 350
330 PRINT A$
340 GOTO 310
350 CLOSE #33
```

1.6 Making a chain of programs

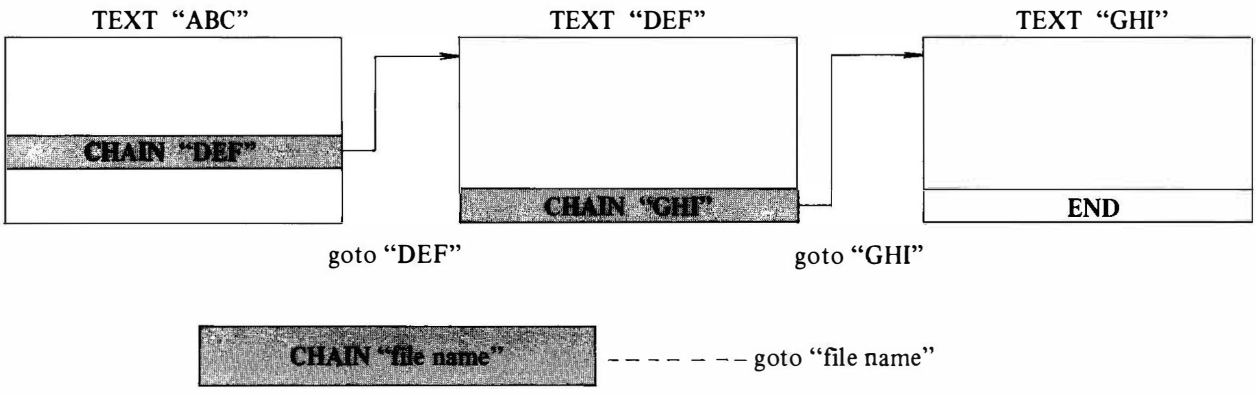
The topic of this section is two program file control statements. These are the CHAIN are SWAP statements. When some programs are recorded on a diskette, the use of these statements enables you to call another program while running the recorded programs and moves the control to it. In detail, the CHAIN statement enables you to connect any program to the ones recorded on a diskette, and the SWAP statement enables you to call any program in the form of subroutine. First is described the CHAIN statement to connect or join programs.

The form of the CHAIN statement is as follows.

CHAIN FD1@50, "TEXT 2"

This statement commands the system to clear a program then present in the text area (it, however, keeps the values of variables), overlays that area with the text named "TEXT 2" that is recorded on the diskette of volume number 50 present in drive 1 and moves control to the head of that text. The execution of this text frees the system from the control of the then running BASIC text and comples it to read the text "TEXT 2" anew, moving control to its head. **When two programs are connected, the values of variables and the function defined by the DEF FN** in the original program are kept.

The function of the CHAIN statement can be grasped as one of **"GOTO"** statement.



The use of the CHAIN statement enables you to process such a huge program as to overflow the BASIC text area by dividing it into pieces and then uniting them again as illustrated above. That is, the CHAIN statement joins component programs every time they are processed. Therefore, the statement and the SWAP statement we will next refer to can be said to be an indispensable aid in coping with complicated, versatile data processing in small businesses.

Apart from such a sophisticated application, it is quite exciting and interesting to join various texts on a diskette. The DISK BASIC, as seen from this, has an original world – which cannot be created by the conventional BASIC – in that enables programs to extend themselves.

1.7 Swapping programs

The SWAP statement reads a program from a diskette file, overlays another program with it or links them, and leaves control to that program text, resuming control by the original program the instant the execution of the text has been completed. Such behaviour is just the same as referring to a subroutine in a text; a fetched program returns to the location next to the one that has been subjected to the SWAP statement, Hence, the SWAP statement can be grasped as a subroutine call. To achieve the above-mentioned action correctly a program text that has the SWAP statement must be temporarily stored in a diskette before the execution of swapping. The program control process cannot then return to the stored original program text before the text area is renewed and the subprogram is called and completely executed. The SWAP statement is generally available in the following form.

SWAP FDd@v, "file name"

This form orders the system to swap a subprogram specified by "file name" that is stored on the diskette with volume number v present in drive d (d = 1 to 4). Storing of a program text prior to execution of a subprogram occurs onto the diskette present in the drive that has last executed the DIR FDd command. This means that the drive must be loaded with a diskette that allows temporary writing of a program text. The swapping level must be less than 1.

Let's follow the program file behaviour by taking a simple example in order to understand the SWAP statement. How does the file when the DIR FD1 command is executed?

[Program present in the text area]

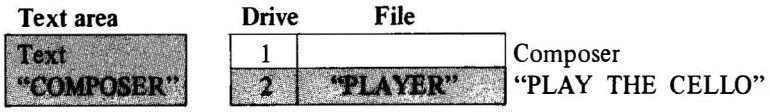
```
10 REM COMPOSER
20 M1$ = "A7B6 + C3A7A3"
30 M2$ = "B + C + D + E6A3"
40 M3$ = "+ F6A3 + E7"
50 PRINT "PLAY THE CELLO"
60 SWAP FD2@7, "PLAYER"
70 PRINT "VERY GOOD"
80 END
```

[Program file "PLAYER"]

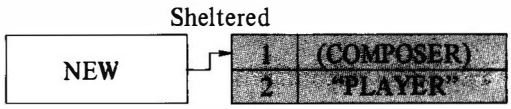
```
10 REM CELLO PLAYER
20 MUSIC M1$, M2$, M3$
30 PRINT "OK?"
40 END
```

This file is present on slave diskette No. 7 inserted in drive No. 2.

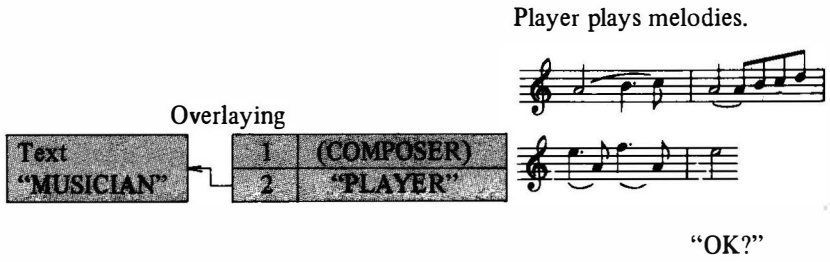
Initially, the text "COMPOSER", present in the text area, is executed.



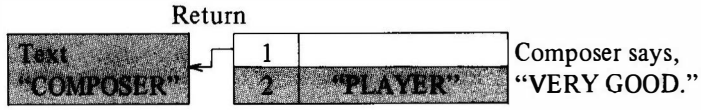
First the SWAP statement, line No. 60, shelters the text on the diskette present in the drive FD1 that has executed the DIR command, and renews the text area.



Second the text area is overlayed with BTX "PLAYER". and the program is executed to play melodies.



On the completion of playing, the sheltered COMPOSER returns, saying "VERY GOOD."



1.8 USR function in a logical open statement

The USR function generally calls a subroutine coded in machine language. When it is used in a logical open statement (WOPEN or ROPEN), however, logical open is performed with the assumption that the USR function is a logical file which is executed when a subsequent PRINT # or INPUT # statement is executed. Data input and output can be controlled in the same manner as for file access. After the USR function has been logically opened, the PRINT # and INPUT # statements are executed as shown in the examples below.

```
(Write)
100 WOPEN #10, USR (n)
200 PRINT #10, A$
300 CLOSE #10
```

100 : Assigns logical file number 10 to USR (n)

110 : Outputs the contents of string variable A$ to the write data buffer and sets the start address of the write data buffer in the DE register and the data length in the BC register.
For example, when A$ = "ABCD", ASCII codes corresponding to "ABCD" are stored in the buffer indicated by the DE register and the number of ASCII codes (excluding [CR] codes) is stored in the BC register.

DE ⟶

"A"	(41H)
"B"	(42H)
"C"	(43H)
"D"	(44H)
CR	(0DH)

BC ← 0004 (hex.)

Then, USR (n) is executed.
Program operation after control is returned from the machine language routine determined by the form of the PRINT # statement as follows.

(1) When a semicolon follows the data, the next statement is executed.

(2) When no semicolon follows the data, a [CR] code (0DH) is set in the location indicated by the DE register and 0001H is set in the BC register, then USR (n) is executed again.

Therefore, when the machine language routine is, for instance, one for controlling the line printer, a new line operation can be obtained by placing a semicolon in the PRINT # statement.

120 : Closes logical file # 10.

(Read)

```
200 ROPEN #11, USR (n)
210 INPUT #11, B$
220 CLOSE #11
```

200 : Assigns logical file number #11 to USR (n) and logically opens it.

210 : Executes USR (n). The machine language routine called must load string data in the read buffer starting at the address indicated by the DE register and load the length of the data string read in the BC register. Then, control is returned to the INPUT statement and the data read is stored in B$.

220 : Closes logical file #11.

[Note]

An error occurring during execution of the machine language routine can be linked with the BASIC error routine in the following manner. When USR (n) executed, the IYand IX registers contain special values. Therefore, when it is necessary to process an error occurring during execution of the machine language routine and when †ON ERROR is declared, system control can be transferred to the error routine by coding the machine language program to set an appropriate error code in the IY area indicated by the IY register and to jump to the address indicated by the IX register.

† Refer to "Error processing control" on page 28.

Chapter 2
Instructions Unique to SB-6610

This chapter describes SB-6610 direct commands, statements, updated commands and utilities which are not supported by the ordinary cassette BASIC interpreter SB-5510.

Command and statement format

Commands and statements must be coded according to the following conventions.

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

2.1 Direct commands

2.1.1 DIR

Format : DIR ⟨FD*d*⟩

d . . . . . drive number : 1 through 4

Function : Displays the file directory of the diskette specified.

Description : When FD*d* is omitted, the value defaults to the number of the drive against which the last DIR FD*d* command was executed.

The contents of the directory are as follows:

- Volume number
  For the master diskette, "MASTER" is displayed.
- The number of unused sectors remaining.
- Mode, lock condition and file name of each file on the diskette.
  The four file modes are indicated with the following codes:
  BTX : BASIC text file
  BSD : BASIC sequential access file
  BRD : BASIC random access file
  OBJ : Object file
  To indicate the lock condition, an asterisk is attached to the file mode.
  Locked files cannot be overwritten or deleted, nor can their names be changed.
  The file name specified during file creation must be always used to call the file.

When many files are contained on a diskette, the directory cannot be displayed in a single frame. The display is fixed once a frame is filled, and the cursor appears. The frame containing the remainder of the directory can then be brought to the screen by pressing the [CR] key. When the display is fixed, another command can be executed.

2.1.2 DIR/P

Format : DIR ⟨FD*d*⟩ /P

d . . . . . drive number : 1 through 4

Function : Prints the directory of the diskette in drive *d* on the line printer.

2.1.3 SAVE

Format : SAVE ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . .†diskette volume number

Function : Assigns the specified file name to the BASIC text contained in the text area and stores it on the diskette in the specified drive.

Description : The diskette on which the BASIC text is to be saved is specified with the FD*d*@*v* operand.

When this operand is omitted, the text will be stored on the diskette in the default drive.

"*file name*" consists of a string of up to 16 characters enclosed with quotation marks.

Example : SAVE "D" . . . Assigns the file name "D" to the BASIC text in the text area and stores it on the active diskette. The text is stored in the BTX file mode.

2.1.4 LOAD

Format : LOAD ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

Function : Loads the specified BASIC text file into memory from the specified diskette.

Description : The diskette is specified with the FD*d*@*v* operand.

When it is omitted, text is stored on the diskette in the default drive.

Example : LOAD FD2, "A" ... Loads the BASIC text assigned the file name "A" from the diskette in drive 2 into the text area.

LOAD "TEXT 1" . . . Loads BASIC text "TEXT 1" from the diskette in the active drive into the text area.

†The diskette volume number is assigned to a slave diskette when the diskette is made by using the utility program "Utility". See page 37.

2.1.5 RUN

Format : RUN ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

"*file name*" . . . . . BTX file or OBJ file

Function : Loads the BASIC text (BTX) assigned the file name "*file name*" from the diskette, and then executes it from its beginning.

Therefore,

RUN "*file name*" = LOAD "*file name*" + RUN

Loads the machine language program (OBJ) assigned the file name "*file name*" from the diskette, and then executes the program at the start address. In such cases, system control is transferred from the BASIC interpreter to the machine language program.

DIR, SAVE and RAN commands cannot be used as statement in programs.

2.2 File control statements

2.2.1 LOCK

Format : LOCK ⟨FD*d*@*v*,⟩ "*file name*"
d . . . . . drive number : 1 through 4
v . . . . . diskette volume number

Function : This statement locks a specified file.

Description : When a file is locked, requests to modify it will be denied. For example, the command prohibits DELETE or RENAME operations or writing of data in the case of random access files. It is good practice to lock files of a permanent or semi-permanent nature. The file mode symbols in the file directory display are followed by an asterisk to indicate protected files.
(The write protect seal serves as a hardware lock for an entire diskette.)

2.2.2 UNLOCK

Format : UNLOCK ⟨FD*d*@*v*,⟩ "*file name*"
d . . . . . drive number : 1 through 4
v . . . . . diskette volume number

Function : This statement unlocks a specified file.

2.2.3 RENAME

Format : RENAME ⟨FD*d*@*v*,⟩ "*file name* 1", "*file name* 2"
d . . . . . drive number : 1 through 4
v . . . . . diskette volume number

Function : This statement renames a specified file.

Description : To rename a file, its current name and its new name must be specified in this order. If a renamed file is identical in name and mode to any file currently stored on the same diskette, an error occurs.
The RENAME statement is prohibited for any locked file.

2.2.4 DELETE

Format : DELETE ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

Function : This statement deletes a specified file from the diskette.

Discription : This statement is prohibited for any locked file. If you want to delete locked files, it is necessary to execute the UNLOCK statement first, then the DELETE statement.

2.2.5 CHAIN

Format : CHAIN ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

Function : This statement chains the program execution to BASIC text on the diskette.

Description : CHAIN FD2@7, "TEXT B" . . . Chains the program in the BASIC text area to BASIC program "TEXT B" on the diskette volume 7 in drive 2. That is, program "TEXT B" is loaded in the BASIC text area and program execution is started at its beginning. Before the text is loaded, the BASIC text area is cleared but all variable values and contents of user functions are given to the program. The CHAIN statement has the same function as GOTO "*file name*".

CHAIN "PROGRAM 3" . . . Chains the program in the BASIC text area to program "PROGRAM 3" on the diskette in the active drive.

Statements LOCK, UNLOCK, RENAME and DELETE can be also used as direct commands.

2.2.6 SWAP

Format : SWAP ⟨FD*d*@*v*,⟩ "*file name*"

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

Function : This statement swaps the program execution to BASIC text on the diskette.

Description : SWAP FD2@7, "TEXT S-R" . . . Swaps the current program for BASIC program "TEXT S-R" on diskette volume 7 in drive 2. The current program text is saved on the diskette in the drive specified in the last DIR FDd command, then program "TEXT S-R" is loaded into the text area and is executed from its beginning. When the swapped program is finished, the saved program is loaded again and program execution is started at the statement following the SWAP statement. The values of variables and the contents of user functions are transferred between the two program. No SWAP statement can be used in a swapped program. The SWAP statement has the same function as GOSUB "*file name*".

■ BSD (BASIC Sequantial access Data file) control

2.2.7 WOPEN#

Format : WOPEN #*l*, ⟨FD*d*@*v*,⟩ "*file name*"

l . . . . . logical number

d . . . . . drive number : 1 through 4

v . . . . . diskette volume number

Function : This statement opens a diskette file to allow a sequential access file to be written on the diskette.

Description : WOPEN #3, FD2@7, "SEQ DATA 1" . . . Defines the file name of a BSD (BASIC sequential access data file) to be created as "SEQ DATA 1" and opens it with logical number 3 assigned on diskette volume 7 in drive 2.

2.2.8 PRINT

Format : PRINT #l, d_1 , ⟨, d_2 , . . . , d_n ⟩

l . . . . . logical number

d_i . . . . write data

Fonction : This statement writes the data d_1 , d_2 . . . d_v (numeric data or string data) in order in the BSD assigned logical number l which was opened by a WOPEN# statement.

Description : PRINT #3, A, A$ . . . Writes the contents of variable A and string variable A$ in order in the BSD assigned logical number 3 which was opened by a WOPEN# statement.

2.2.9 CLOSE

Format : CLOSE ⟨# l⟩

l . . . . . logical number

Function : This statement closes a BSD assigned logical number l.

Description : CLOSE #3 . . . Closes the BSD assigned logical number 3 which was opened by the WOPEN #3 statement.

By closing the BSD, the BSD which has the file name defined in the WOPEN# statement is created on the specified diskette, and the logical number assigned is made undefined.

2.2.10 KILL

Format : KILL ⟨# l⟩

l . . . . . logical number

Function : This statement kills a BSD assigned logical number l .

Description : KILL #3 . . . kills the BSD assigned logical number 3 by the WOPEN# statement. Logical number 3 is made undefined.

2.2.11 ROPEN

Format : ROPEN #*l*, ⟨FD*d*@*v*,⟩ "*file name*"

l . . . . . logical number

d . . . . . drive number : *l* through 4

v . . . . . diskette volume number

Function : This statement opens a diskette file to allow a sequential access file to be read from the diskette.

Description : ROPEN #4, FD2@7, "SEQ DATA 1" . . . Opens BSD "SEQ DATA 1" on diskette volume 7 in drive 2 with logical number 3 assigned to read data in BSD.

2.2.12 INPUT

Format : INPUT #*l*, v_1 ⟨, v_2, . . . , v_n⟩

l . . . . . logical number

v_i . . . . read data

Function : This statement reads data stored in the specified BSD in order and assignes to variables v_1, v_2 . . . v_n (or array elements).

Description : INPUT #4, A(1), B$ . . . Reads data sequentially from the beginning of the BSD assigned logical number 4 which was opened by the ROPEN# statement and substitutes numerical data into array element A(1) and string data into string variable B$.

CLOSE #4 statement closes the BSD assigned logical number 4 and the logical number undefined.

■ **BRD (BASIC Random access Data file) control**

2.2.13 XOPEN

Format : XOPEN #*l*, ⟨FD*d*@*v*,⟩ "*file name*"

l ..... logical number

d ..... drive number : 1 through 4

v ..... diskette volume number

Function : Generally, XOPEN # opens a BRD for writing and reading data (CROSS open).

Description : XOPEN #5, FD3@18, "DATA R1" . . . This statement cross-opens BRD "DATA R1" on diskette volume 18 in drive 3 with logical number 5 assigned or, if the file does not exist on the diskette, cross-opens a BRD by defining its file name as "DATA R1" to create it on the diskette with logical number 5 assigned.

2.2.14 PRINT # ( )

Format : PRINT #*l* (*n*), d_1 ⟨, d_2, . . . , d_n⟩

l ..... logical number

n ..... item expression

d_i .... write data

Function : This statement writes numeric or string data on elements *n*, *n* + 1, . . . , *n* + *n* of the BRD assigned logical number *l* which was opened by the XOPEN# statement.

Description : PRINT #5 (11), R(11) . . . Writes the contents of 1-dimensional array element R(11) on element 11 of the BRD assigned logical number 5 which was opened by the XOPEN# statement.

PRINT #5(20),AR$, AS$ . . . Writes the contents of string variables AR$ and AS$ on element 20 and element 21 of the BRD assigned logical number 5, respectively. All BRD elements have a fixed length of 32 bytes and, if the length of string variable exceeds 32 bytes, the excess part is discarded.

2.2.15 INPUT# ( )

Format : INPUT #$l(n), v_1 \langle, v_2, \ldots, v_n \rangle$

l ..... logical number

n ..... item expression

v_i .... read data

Function : This statement reads data stored in the specified elements of the specified BRD.

Description : INPUT #5(21), R$ . . . Reads the content of element 21 of the BRD assigned logical number 5 which was opened by the XOPEN# statement into string variable R$.

INPUT #5(11), A(11), A$(12) . . . Reads the contents of element 11 and element 12 of the BRD assigned logical number 5 into linear numeric array element A(11) and linear string array element A$(12), respectively.

CLOSE #5 statement closes the BRD assigned logical number 5 which was opened by the corresponding XOPEN # statement.

KILL #5 statement kills the BRD assigned logical number 5 and the logical number undefined.

CLOSE .... Closes all open files.

KILL ..... Kills all open files.

2.2.16 IF EOF(#) THEN

Format : IF EOF(#l) THEN lr (or *statement*)

l ..... logical number

lr ..... reference line number

Function : Transfers program control to the routine starting to specified line number lr if an EOF (End of file) is detected when as INPUT# statement is executed against a BSD or a BRD.

Example : IF EOF(#5) THEN 1200

2.3 Error processing control

2.3.1 ON ERROR GOTO

Format : ON ERROR GOTO *lr*

lr .... *reference line number* : error processing routine

Function : This statement declares the number of the line to which program execution is to be moved in order to correct errors.

Description : Declaring an error processing routine with the ON ERROR GOTO statement allows errors to be corrected during program execution without the system returning to the BASIC command level. When the ON ERROR GOTO statement is executed, program execution will be moved to ⟨error processing routine⟩ if any error has occurred. This enables the †error number (ERN) and the number of the line on which the error occurred (ERL) to be ascertained, and allows subsequent processing to be performed in accordance with the IF ERN or ERL statements. The RESUME statement serves to move program execution back to the point at which the error occurred.

Execution of a new ON ERROR GOTO statement invalidates any preceding one.

2.3.2 IF ERN

Format : IF ERN *expression* THEN *lr*

IF ERN *expression* THEN *statement*

IF ERN *expression* GOTO *lr*

lr .... *reference line number*

Function : This statement ascertains the identification numbers of errors, and causes branching when those numbers are ones specified.

Description : When an error occurs, the corresponding error number is placed in system variable ERN. This enables an IF ERN statement in an error correcting routine declared by the ON ERROR GOTO statement to determine what type of error has occurred. The IF ERN statement may be used in either of the following forms; either of the following forms:

(1) IF ⟨relational expression of ERN⟩ GOTO, or

(2) IF ⟨relational expression of ERN⟩ THEN statement or *lr*.

(See the descriptions of the IF ~ THEN and IF ~ GOTO statements.)

†For the error number, refer to the Error Message Table on page 66.

Example : The statement shown below causes program execution to jump to line 1200 when Error 5 (String Overflow) occurs, indicating that the string length exceeded 255 characters.

```
800 IF ERN = 5 THEN 1200
```

2.3.3 IF ERL

Format : IF ERL *expression* THEN *lr*
IF ERL *expression* THEN *statement*
IF ERL *expression* GOTO *lr*
lr .... *reference line number*

Function : This statement determines the number of the line on which an error has occurred and causes branching to a specified line.

Description : Since system variable ERL is loaded with the number of the line on which an error occurred, the IF ERL statement in the routine declared by the ON ERROR GOTO statement is able to ascertain this line number from system variable ERL. The IF ERL statement, like the IF ERN statement, may be used in two forms: IF ~ THEN or IF ~ GOTO.

Example : The statement shown below causes program execution to jump to line 1300 when an error occurs on line 250.

```
810 IF ERL = 250 THEN 1300
```

2.3.4 RESUME

Format : RESUME 〈NEXT〉

RESUME *lr*

lr .... *reference line number* or 0

Function : This statement returns program execution to the main program after correction of an error.

Description : The system holds the number of the line on which the error occurred in memory and returns program execution to that line or to another specified line after the error is corrected.

The RESUME statement may be used in any of the following four forms:

RESUME: This returns program execution to the statement in which the error occurred.

RESUME NEXT: This returns program execution to the statement just after the one in which the error occurred.

RESUME 〈line number〉: This returns program execution to the line specified by 〈line number〉.

RESUME 0 : This returns program execution to the beginning of the program, or to the line with the smallest line number.

If the system ecounters any RESUME statement when there is no error condition, Error 21 (RESUME - no ERROR) will occur.

2.4 Updated commands

2.4.1 PRINT USING

Format : PRINT USING *format* ; *variable name list*

format . . . . . string variable consisting of special characters

variable name list . . . numeric variables and/or numeric expression

Function : This statement displays the contents of the numeric variable indicated by† the variable name list operand in the specified format.

Description : Special characters used in the format operand are explained below.

■ # (used for specifying the number of digits)

The number of #'s is the number of digits to be displayed. Signs (+ and –) are not counted as digits.

When the number of digits to be displayed is less than that specified, displayed data is right-justified and vacant spaces are filled with blanks.

```
PRINT USING "##,###" ; 12345
 12,345
PRINT USING "##,###" ; 1234,567
␣1,234 ␣␣␣␣␣␣␣ 567 . . . . . Tabs are set every 10 digits.
PRINT USING "##.##" ; 12.345        " ␣ " represents a space.
 12.34 . . . . . Excess decimal places are omitted.
```

■ + and – signs

+### : When data is positive, it is perfixed with a + sign and when it is negative, it is prefixed with a – sign.

–### : When data is positive, a space precedes it and when it is negative, a – sign precedes it.

One character space is reserved for the sign.

```
PRINT USING "+###,###" ; 123456
+123,456
A$ = "–#,###"
PRINT USING A$ ; –1234
–1,234
```

†For a variable name list, numeric variables (expressions) are separated with cammas.

- Decimal point (.)

  The decimal point separates the decimal part from the integral part. Only one decimal point can be used in the format operand. When the number of specified decimal places is more than the number of decimal places in the given data, 0s are displayed for the excess places. One character space is reserved for the decimal point.

  ```
  PRINT USING "+#.##" ; 1.23
  +1.23
  PRINT USING "#.###" ; 1.23
  1.230
  ```

- Comma (,)

  Inserts a comma in the specified position. One character space is reserved for each comma.

  ```
  PRINT USING "##,###.#" ; 12345
  12,345.0
  PRINT USING "##,###" ; 12
  ␣␣␣12
  ```

- When ∗ s are used instead of #s, ∗ s are displayed instead of spaces.

  ```
  PRINT USING " ∗∗, ∗∗∗ " ; 12
  ∗∗∗∗12
  PRING USING "+ ∗∗∗∗ " ; 12
  ∗∗ +12
  ```

- £, @ and $

  £, @ or $ can be attached to the beginning of numeric data.

  ```
  PRING USING "£##,###.##" ; 1234.56
  ␣£1,234.56
  PRINT USING "@##.##" ; 12.3
  @12.30
  ```

- X

  When Xs are specified, spaces are displayed where the Xs are specified.

  ```
  PRING USING "##XXX##" ; 1234
  12 ␣␣␣ 34
  PRINT USING "#,###.XXX##" ; 123.4
  ␣␣ 123.␣␣␣40
  ```

- Format Over Display (%)

  When the data length is longer than that specified, % preceeds the data displayed.

  ```
  PRINT USING "##,###" ; 123456
  %123,456
  ```

- Other Characters except Format

  When any other characters are specified at the beginning or end of the format operand, they are displayed as they are.

  ```
  PRINT USING "UNIT PRICE @##.## YEN" ; 12.34
  UNIT PRICE @12.34 YEN
  A$ = "COMPUTER ### SYSTEMS"
  PRINT USING A$ ; 12
  COMPUTER ␣ 12 SYSTEMS
  PRINT USING "£#,###" ; 123,45678
  ␣␣ £123 ␣␣␣␣%£45,678
  ```

- Exchange of 0 (Zero) and O

  Either letter "O" or the numeric character "0" may be used for displaying or printing zeros with the following POKE statements.

  POKE $002D,1 The letter O is used to display zeros.

  POKE $002D,0 The numeric character 0 is used to zeros.

2.4.2 DELETE

Format : DELETE *lr*
DELETE *–lr*
DELETE *lr–*
DELETE *lr–lr*
lr .... reference line number

Function : Deletes all statements on lines specified.

Description : Refer to the samples shown below.

DELETE 10 : Deletes the statement on line 10.

DELETE 10–: Deletes all statements after line 10.

DELETE –10: Deletes all statements from the beginning of the program to line 10.

DELETE 10–50: Deletes all statements between line 10 and line 50.

2.4.3 DIM

Format : DIM a_1 (i_1)〈, a_2 (i_2), ... , a_n (i_n)〉
DIM b_1 (i_1, j_1)〈, b_2 (i_2, j_2), . . . , b_n (i_n, j_n)〉
ai . . . . one-dimensional array
bi . . . . two-dimensional array
i_n, j_n . . . . 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.
However, the number of array elements of an one-dimensional array is only limited by the amount of unused memory area.
In the case, the subscripts can be expressed with any number over 255.

Example DIM A (1000), AB$ (300)
DIM B (80, 80), BC$ (100, 100)

2.4.4 Function

Description : SB-6610 does not support the following functions.

SIN, COS, TAN, EXP, LOG, LN and ^ (power)

Subroutines for these functions are filed on the master diskette under the file name "FUNCTION"
These subroutines are referenced for calculating function. Use of the Disk BASIC SB-6510 is recommended when you need to calculate function.

2.5 Use of utility programs

Utility programs "Filing CMT" (OBJ) and "Utility" (OBJ) are stored on the master diskette together with DISK BASIC interpreter SB-6610, MONITOR SB-1510 and some application programs.

In the following paragraphs, use of utility programs are explained.

2.5.1 Use of utility program "Filing CMT"

This utility program transfers machine language program from cassette file to the diskette as it is. To call this utility program, enter

RUN "Filing CMT"

The display screen is as shown in Figure 2.1.

```
* TRANSFER FROM CMT (OBJECT TAPE) TO FD *
  SET TAPE! OK?
    (B KEY : BOOT START)
DRIVE NO. ▩
```

FIGURE 2.1

Set ready the cassette tape file which has to be transferred into the diskette, and specify the drive number.

The following example transfers BASIC interpreter SB-5510 from the cassette file to the diskette in drive 2.

```
* TRANSFER FROM CMT (OBJECT TAPE) TO FD *
  SET TAPE! OK?
    (B KEY : BOOT START)
DRIVE NO. 2
LOADING BASIC SB-5510
  (R) KEY : RESTART
  OTHER KEY : BOOT START
```

FIGURE 2.2

You obtain the object file (OBJ) "BASIC SB-5510" on the diskette in drive 2. Therefore, to call BASIC interpreter SB-5510 from the diskette file, simply enter

```
RUN "BASIC SB-5510"
```

2.5.2 Use of utility program "Utility"

This utility program has two functions, that is, initializing diskettes and copying diskettes. To call this utility program, enter

```
RUN "Utility"
```

The display screen is as shown below.

```
** UTILITY **
[COMMAND TABLE]
DISKETTE INIT     :   I
SLAVE-DISK INIT   :   S
DISKETTE COPY     :   C
BOOT START        :   B
? ■
```

FIGURE 2.3

When a I command is entered, the display is as shown in Figure 2.4.

```
[COMMAND TABLE]
DISKETTE INIT     :   I
SLAVE-DISK INIT   :   S
DISKETTE COPY     :   C
BOOT START        :   B
? I
DRIVE NO. ■
```

FIGURE 2.4

The utility program requests the operator to specify drive number.

When a new diskette is used, it must first be initialized (that is, formats the diskette so that data are able to be written or read). During initialization, the diskette is formatted.

S command assigns a volume number to a initialized diskette for making a slave diskette. Figure 2.5 shows an example where the slave diskette in drive 1 is made with volume number 2.5 assigned.

```
[COMMAND TABLE]
DISKETTE INIT     :   I
SLAVE-DISK INIT   :   S
DISKETTE COPY     :   C
BOOT START        :   B
? S
VOLUME NO. 25
```

FIGURE 2.5

Any number from 1 through 127 can be specified for the volume number. Different numbers must be assigned to each diskette so that a diskette can be specified with its volume number in a logical open statement.

C command copies a diskette. The following example copies the diskette in drive 1 on diskette in drive 2.

```
[COMMAND TABLE]
DISKETTE INIT     :   I
SLAVE-DISK INIT   :   S
DISKETTE COPY     :   C
BOOT START        :   B
? C
FROM
DRIVE NO. 1
TO
DRIVE NO. 2
```

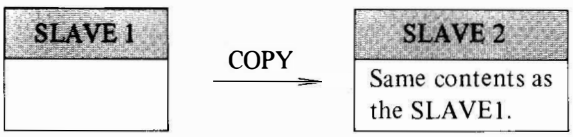


FIGURE 2.6 Slave diskette copying

Any number of submaster diskettes can be made by copying the master diskette using this diskette copy command explained above. However, submaster diskettes cannot be made by copying another submaster diskette.

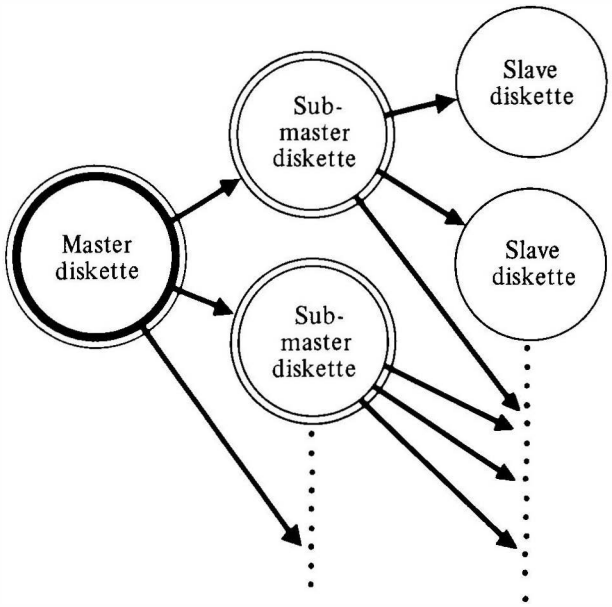


FIGURE 2.7

NOTE: It is recommended that a write protect seal be placed on the original diskette to prevent it from being accidentally erased.

ERROR INDICATIONS : DISK ERROR = 50 . . . . . The disk drive is not ready.
DISK ERROR = 41 . . . . . Disk drive hardware error.

Chapter 3
Programming Instructions

This chapter summarizes all commands, statements, operators, symbols and specifications of the double precision DISK BASIC interpreter SB-6610.

3.1 List of DISK BASIC interpreter SB-6610 commands, statements and functions

3.1.1 Commands

DIR	DIR FDd	Displays the file directory of the diskette in drive d (d=1~4). The contents of the directory are as follows: 1) Volume number 2) Number of unused sectors 3) Mode, lock condition and file name of each file on the diskette Note: When a directory is listed on the CRT, the display is fixed and the cursor appears when the frame is filled. To display the next frame of the directory, press the [CR] key. Other command may be executed once the display is fixed.
	DIR FD3	Displays the files directory of the diskette in drive 3. When a DIR FDd command is executed, the system stores the drive number so that it may be omitted (if the same drive is specified) for the direct execution instructions and file access instructions explained below.
	DIR	Displays the file directory of the diskette in the active drive which is last specified in a DIR FDd command.
LOAD	LOAD "A"	Loads BASIC text (BTX) assigned the file name "A" from the diskette in the active drive into the text area.
	LOAD FD2@10 "A"	Loads the BASIC text assigned the file name "A" from volume 10 in drive 2 into the text area.
	LIMIT $D000: LOAD "B"	To load a machine language program file (OBJ) to be linked with a BASIC text, the BASIC area of memory must be partitioned from the machine language area by the LIMIT statement.
SAVE	SAVE "D"	Assigns the file name "D" to the BASIC text in the text area and stores it on the diskette in the active drive. The text is stored in the BTX file mode.
RUN	RUN	Executes the BASIC text in the text area from the top. Note: The RUN command clears all variables (fills them with 0 or null string) before running text.
	RUN 1000	Executes the BASIC text starting at line number 1000.
	RUN "F" (BTX)	Loads the BASIC text assigned the file name "F" from the diskette in the active drive and executes it from its beginning.

	RUN FD3@7 "G" ↑ (OBJ)	Loads machine language program assigned the file name "F" from the diskette of volume 7 in drive 3, and then executes the program starting at the start address. In such cases, system control is transferred from the BASIC interpreter to the machine language program.
AUTO	AUTO	Automatically generates and assigns line numbers 10, 20, 30 . . . . . during creation.
	AUTO 200, 20	Automatically generates line numbers at intervals 20 starting at line 200. 200, 220, 240 . . . . . . An AUTO command is terminated by pressing the [BREAK] key.
LIST	LIST	Displays all lines of BASIC text currently contained in the text area.
	LIST –500	Displays all lines of BASIC text up through line 500.
NEW	NEW	Clears the text area and variable area. Further, disestablishes the machine language program area set by a LIMIT statement by removing the partition.
CONT	CONT	Continues program execution which was halted by a STOP statement or the [BREAK] key, starting at the statement following the STOP statement or the statement halted by the [BREAK] key.
MON	MON	Transfers system control from the BASIC interpreter to the MONITOR. (To transfer system control from the MONITOR to the BASIC interpreter, execute monitor command J.)
BOOT	BOOT	Activates the MZ-80B system initial program loader.
KLIST	KLIST	Displays a complete list of string definitions for special function keys, thereby enabling you to determine how individual special function keys are defined.
DELETE	DELETE 10	Deletes the statement on line 10.
	DELETE 10–	Deletes all statements after line 10.
	DELETE –100	Deletes all statements up to line 100.
	DELETE 10–100	Deletes all statements from line 10 to line 100.

3.1.2 File control statements

LOCK	LOCK "ABC"	Locks file "ABC" on the diskette in the active drive.
	LOCK FD4@7 "ABC"	Locks file "ABC" on the diskette (whose volume number is 7), in drive 4. The locked file cannot be updated or deleted. When the file directory is listed, the locked file name is indicated with an asterisk.
UNLOCK	UNLOCK "ABC"	Unlocks file "ABC" on the diskette in the active drive.
	100 UNLOCK FD1 "A"	Unlocks file "A" on the diskette in drive 1. (This is an example of a statement used in a program.)
RENAME	RENAME "A", "B"	Changes the name of file "A" on the diskette in the active drive to "B".
DELETE	DELETE "A"	Deletes file "A" from the diskette in the active drive.
CHAIN	CHAIN FD2@7 "TEXT B"	Chains the program in the BASIC text area to BASIC program "TEXT B" on the diskette volume 7 in drive 2. That is, program "TEXT B" is loaded in the BASIC text area and program execution is started at its beginning. Before the text is loaded, the BASIC text area is cleared but all variable values and contents of user functions are given to program "TEXT B". The CHAIN statement has the same function as GOTO "file name".
	CHAIN "TEXT B"	Chains the program in the BASIC text area to program "TEXT B" on the diskette in the active drive.
SWAP	SWAP FD2@7 "TEXT S-R"	Swaps the current program for BASIC program "TEXT S-R" on diskette volume 7 in drive 2. The current program text is saved on the diskette in the drive specified in the last DIR FDd command, then program "TEXT S-R" is loaded into the BASIC text area and is executed from its beginning. When the swapped program is finished, the saved program is loaded again and program execution is started at the statement following the SWAP statement. The values of variables and the contents of user functions are transferred between the two programs. No SWAP statement can be used in a swapped program. The SWAP statement has the same function as GOSUB "file name".

3.1.3 BSD (BASIC Sequential access Data file) control statements

WOPEN #	WOPEN #3, FD2@7, "SEQ DATA 1"	Defines the file name of a BSD (BASIC sequential access data file) to be created as "SEQ DATA 1" and opens it with logical number 3 assigned on diskette volume 7 in drive 2. For WOPEN # statements including a USR function operand, see page 55.
PRINT #	PRINT #3, A, A$	Writes the contents of variable A and string variable A$ in order in the BSD assigned logical number 3 which was opened by a WOPEN # statement. (In writing data, 256 bytes are treated as a unit.)
CLOSE #	CLOSE #3 (corresponding to WOPEN #)	Closes the BSD assigned logical number 3 which was opened by the WOPEN #3 statement. By closing the BSD, the BSD which has the file name defined in the WOPEN # statement is created on the specified diskette, and the logical number assigned is made undefined.
KILL #	KILL # 3	Kills the BSD assigned logical number 3 by the WOPEN # statement. Logical number 3 is made undefined.
ROPEN #	ROPEN #4, FD2@7 "SEQ DATA 1"	Opens BSD "SEQ DATA 1" on diskette volume 7 in drive 2 with logical number 3 assigned to read data in BSD. For ROPEN # statements including a USR function, see page 55.
INPUT #	INPUT #4, A (1), B$	Reads data sequentially from the beginning of the BSD assigned logical number 4 which was opened by the ROPEN # statement and substitutes numerical data into array element A(1) and string data into string variable B$.
CLOSE #	CLOSE #4 (corresponding to ROPEN #)	Close the BSD assigned logical number 4 and makes the file number undefined.

3.1.4 BRD (BASIC Random access Data file) control statements

XOPEN #	XOPEN #5, FD3@18, "DATA R1"	Generally, XOPEN # statement opens a BRD for writing and reading data (Cross open). This statement cross-opens BRD "DATA R1" on diskette volume 18 in drive 3 with logical number 5 assigned or, if the file does not exist on the diskette, cross-opens a BRD by defining its file name as "DATA R1" to create it on the diskette with logical number 5 assigned.
PRINT #( )	PRINT #5(11), R (3)	Writes the content of linear array element R (3) on field 11 of the BRD assigned logical number 5 which was opened by the XOPEN # statement.

PRINT #( )	PRINT #5(20), AR$, AS$	Writes the contents of string variables AR$ and AS$ on field 20 and field 21 of the BRD assigned logical number 5, respectively. All BRD fields have a fixed length of 32 bytes and, if the length of string variable exceeds 32 bytes, the excess part is discarded.
INPUT #( )	INPUT #5(21), R$	Reads the content of field 21 of the BRD assigned logical number 5 which was opened by the XOPEN # statement into string variable R$.
	INPUT #5(11), A(11), A$(12)	Reads the contents of field 11 and field 12 of the BRD assigned logical number 5 into linear numeric array element A(11) and linear string array element AR(12), respectively.
CLOSE #	CLOSE #5	Close the BRD assigned logical number 5 which was opened by the corresponding XOPEN # statement.
	CLOSE	Closes all open files.
KILL #	KILL	Kills all open files.
IF EOF (#)	10 IF EOF (#5) THEN 700	Transfers program control to the routine starting to line number 700 if an EOF (End of File) is detected when an INPUT # statement is executed against a BSD or a BRD.

3.1.5 Error processing statements

ON ERROR GOTO	ON ERROR GOTO 1000	Declares that the number of the line to which program execution is to be moved, if an error occurs is 1000.
IF ERN	IF ERN=44 THEN 1050	Jumps to the statement on line number 1050 if the error number is 44.
IF ERL	IF ERN=350 THEN 1090	Jumps to the statement on line number 1090 if the error line number is 350.
	IF (ERN=53)*(ERL=700) THEN END	Terminates the program if the error number 53 and the error line number is 700. With DISK-BASIC, the error number and error line number are set in special variables ERN and ERL, respectively, if an error occurs during program execution.
RESUME		Returns program execution to the main program after correction of an error.
	650 RESUME	Returns program execution to the statement in which the error occurred.

	700 RESUME NEXT	Returns program execution to the statement just after the one in which the error occurred.
	750 RESUME 400	Returns program execution to line number 400.
	800 RESUME 0	Returns program execution to the beginning of the program.

3.1.6 Cassette file input/output statements

LOAD/T	LOAD/T "C"	Loads the BASIC text assigned the file name "C" from the cassette tape into the text area. Note: When a LOAD command or a LOAD/T command is executed for a BASIC text file, the text area is cleared of any programs previously stored.
SAVE/T	SAVE/T "E"	Assigns the file name "E" to the BASIC text in the text area and automatically stores it on the cassette tape.
VERIFY	VERIFY "H"	This command automatically compares the program contained in the BASIC text area with its equivalent text assigned the file name "H" in the cassette tape file.
WOPEN/T	10 WOPEN/T "DATA-1"	Defines the file name of a cassette data file to be created as "DATA-1" and opens.
PRINT/T	20 PRINT/T, A$	Writes the contents of variable A and string variable A$ in order in the cassette data file which was opened by a WOPEN/T statement.
CLOSE/T	30 CLOSE/T	Closes the cassette data file which was opened by a WOPEN/T statement.
ROPEN/T	110 ROPEN/T "DATA-2"	Opens the cassette data file specified with file name "DATA-2".
INPUT/T	120 INPUT/T, B, B$	Reads data sequentially from the beginning of the cassette data file which was opened by the ROPEN/T statement and substitutes numerical data into variable B and string data into string variable B$ respectively.
CLOSE/T	130 CLOSE/T	Closes the cassette data file which was opened by a ROPEN/T statement.

3.1.7 Assignment statement

LET	⟨LET⟩ A=X+3	Substitutes X + 3 into numeric variable A. LET may be omitted.

3.1.8 Input/output statements

PRINT	10 PRINT A	Displays the numeric value of A on the CRT screen.
	? A$	Displays the character string of variable A$ on the CRT screen.
	100 PRINT A; A$, B; B$	Combinations of numeric variables and string variables can be specified in a PRINT statement. When a semicolon is used as the separator, no space is displayed between the data strings. When a colon is used, variable data to the right of the colon is displayed from the next tab set position. (A tab is set every 10 character positions.)
	110 PRINT "COST="; CS	Displays the string between double quotation marks as is, and CS.
	120 PRINT	Performs a new line operation (i.e., advances the cursor one line).
PRINT USING **Specifies format in which numeric data is to be output on the CRT screen.**	PRINT USING "####"	Displays data in the format specified with #'s. When the length of the data to be displayed is shorter than that specified, the data is right-justified and empty spaces are filled with blanks.
	10 PRINT USING "####" ; 123	Displays ␣123.
	20 PRINT USING "####" ; 98	Displays ␣␣ 98.
	PRINT USING "##,###.##"	Displays the decimal point and commas in the specified positions.
	10 PRINT USING "##,###.##" ; 5321.65	Displays ␣ 5,321.65.
	PRINT USING "✳✳✳✳,✳✳✳"	When ✳'s are used instead of #'s, ✳'s are displayed for spaces.
	10 PRINT USING "✳✳.✳✳✳" ; 1234	Displays ✳ 1,234.
	PRINT USING "£✳✳,✳✳✳" "@✳✳,✳✳✳" "$ ✳✳,✳✳✳"	Prefixes numeric data with £, @ or $.
	10 PRINT USING "£✳✳✳.✳✳✳" ;	Displays ✳ £82,546.

20 PRINT USING "£###,###.##" ; 7658.35	Displays ␣␣£7,658.35.
30 PRINT USING "@###,###" ; 2935	Displays ␣␣ @2,935.
40 PRINT USING "$###,###" ; 81965	Displays ␣ $81,965.
PRINTUSING "#XX#. ##"	Displays spaces for X's.
10 PRINT USING "#XX#.##" ; 98.76	Displays 9␣␣8.76.
PRINT USING "+###.#"	When data is positive, a + sign precedes it and when it is negative, a – sign precedes it. (One character space is reserved for the sign.)
PRINT USING "–###.#"	When data is positive, a space precedes it and when it is negative, a – sign precedes it. (One character space is reserved for the sign.)
10 PRINT USING "+###.#" ; 12.3	Displays ␣ +12.3.
20 PRINT USING "+###.#" ; –5.6	Displays ␣␣ –5.6.
30 PRINT USING "–###.#" ; 58.3	Displays ␣␣ 58.3.
40 PRINT USING "–###.#" ; –58.3	Displays ␣ –58.3.
	When the length of the data to be displayed is longer than that specified, % precedes the data displayed.
10 PRINT USING "##.##" ; 135.68	Displays %135.68.
	When the number of decimal places specified is less than that of the given data, the excess digits are omitted.
10 PRINT USING "#.#" ; 1.23	Displays 1.2.

	10 A$= "###.##"	The format operand can be specified with a string variable as shown at the left.
	20 PRINT USING A$; 1.58	Displays ␣␣ 1.58.
	30 PRINT USING A$; 28.3	Displays ␣ 28.30.
INPUT	10 INPUT A	Obtains numeric data for variable A from the keyboard.
	20 INPUT A$	Obtains string data for string variable A$ from the keyboard.
	30 INPUT "VALUE?" ; D	Displays "VALUE?" on the screen before obtaining data from the keyboard. A semicolon separates the string from the variable.
	40 INPUT X, X$, Y, Y$	Numeric variables and string variables can be used in combination by separating them from each other with a comma. The types of data entered from the keyboard must be the same as those of the corresponding variables.
GET	10 GET N	Obtains a numeral for variable N from the keyboard. When no key is pressed, zero is substituted into N.
	20 GET K$	Obtains a character for variable K$ from the keyboard. When no key is pressed, a null is substituted into K$.
READ~DATA		Substitutes constants specified inthe DATA statement into the corresponding variables specified in the READ statement. The corresponding constant and variable must be of the same data type.
	10 READ A, B, C 1010 DATA 25, –0.5, 500	In READ and DATA statements at left, values of 25, –0.5 and 500 are substitutes for variables A, B and C, respectively.
	10 READ H$, H, S$, S 30 DATA HEART, 3 35 DATA SPACE, 11	In the example at left, the first string constant of the DATA statement on line number 10 is substituted into the first variable of the READ statement; that is; "HEART" is substituted into H$. Then, numeric constant 3 is substituted into numeric variable H, and so on.
RESTORE		With a RESTORE statement, data in the following DATA statement which has already been read by preceding READ statements can be re-read from the beginning by the following READ statements.
	10 READ A, B, C 20 RESTORE 30 READ D, E 100 DATA 3, 6, 9, 12, 15	The READ statement on line number 10 substitutes 3, 6 and 9 into variables A, B and C, respectively. Because of the RESTORE statement, the READ statement on line number 30 substitutes not 12 and 15, but 3 and 6 again into D and E, respectively.

3.1.9 Loop statement

FOR ~ TO NEXT	10 FOR A=1 TO 10 20 PRINT A 30 NEXT A	The statement on line number 10 specifies that the value of variable A is varied from 1 to 10 in increments of one. The initial value of A is 1. The statement on line number 20 displays the value of A. The statement on line number 30 increments the value of A by one and returns program execution to the statement on line number 10. Thus, the loop is repeated until the value of A becomes 10. (After the specified number of loops has been completed, the value of A is 11.)
	10 FOR B=2 TO 8 STEP 3 20 PRINT B^2 30 NEXT	The statement on line number 10 specifies that the value of variable B is varied from 2 to 8 in increments of 3. The value of STEP may be made negative to decrement the value of B.
	10 FOR A=1 TO 3 20 FOR B=10 TO 30 30 PRINT A, B 40 NEXT B 50 NEXT A	The FOR-NEXT loop for variable A includes the FOR-NEXT loop for variable B. As is shown in this example, FOR-NEXT loops can be enclosed in other FOR-NEXT loops at different levels. Lower level loops must be completed within higher level loops. The maximum number of levels of FOR-NEXT loops is 16.
	60 NEXT B, A 70 NEXT A, B	In substitution for NEXT statement at line numbers 40 and 50, a statement at line number 60 shown at left can be used. However, statement at line number 70 cannot be used, causing an error to occur.

3.1.10 Branch statements

GOTO	100 GOTO 200	Jumps to the statement on line number 200.
GOSUB ~ RETURN	100 GOSUB 700 ..... 800 RETURN	Calls the subroutine starting on line number 700. At the end of subroutine, program execution returns to the statement following the corresponding GOSUB statement.
IF ~ THEN	10 IF A>20 THEN 200	Jumps to the statement on line number 200 when the value of variable A is more than 20; otherwise the next line is executed.
	50 IF B<3 THEN B=B+3	Substitues B+3 into variable B when the value of B is less than 3; otherwise the next line is executed.
IF ~ GOTO	100 IF A>=B GOTO 10	Jumps to the statement on line number 10 when the value of variable A is equal to or greater than the value of B; otherwise the next line is executed.

IF ~ GOSUB	30 IF A=B*2 GOSUB 90	Jumps to the subroutine starting on line number 700 when the value of variable A is twice the value of B; otherwise the next statement is executed. (When other statements follow a conditional statement on the same line and the conditions are not satisfied, those following an ON statement are executed sequentially, but those following an IF statement are ignored and the statement on the next line is executed.)
ON ~ GOTO	50 ON A GOTO 70, 80, 90	Jumps to the statement on line number 70 when the value of variable A is 1, to the statement on line number 80 when it is 2 and to the statement on line number 90 when it is 3. When the value of A is 0 or more than 3, the next statement is executed. This statement has the same function as the INT function, so that when the value of A is 2.7, program execution jumps to the statement on line number 80.
ON ~ GOSUB	90 ON A GOSUB 700, 800	Jumps to the subroutine on line number 700 when the value of variable A is 1 and jumps to the subroutine on line number 800 when it is 2.

3.1.11 Definition statements

DIM		When an array is used, the number of array elements must be declared with a DIM statement. For an one-dimensional array, the number of array elements is only limited by the amount of the unused memory area. For a two-dimensional array, however, it is limited by the maximum value of each subscript which is 255.
	10 DIM A(300)	Declares that 301 array elements, A (0) through A (300), are used for one-dimensional numeric array A (n).
	20 DIM B (79, 79)	Declares that 6400 array elements, B (0, 0) through B (79, 79), are used for two-dimensional numeric array B (m, n).
	30 DIM C1$ (10)	Declares that 11 array elements, C1$ (0) through C1$ (10), are used for one-dimensional string array C1$ (n).
	40 DIM K$ (7, 5)	Declares that 48 array elements, K$ (0, 0) through K$ (7, 5), are used for two-dimensional string array K$ (m, n).
DEF FN	100 DEF FNA(X)=X^2−X 110 DEF FNB(X)=LOG(X) +1 120 DEF FNZ(Y)=LN(Y)	A DEF FN statement defines a function. The statement on line number 100 defines FNA(X) as X^2-X. The statement on line number 110 defines FNB(X) as $\log_{10}X+1$ and the statement on line number 120 defines FNZ(Y) as $\log_e Y$. The number of variables included in the function must be 1.

DEF KEY	15 DEF KEY(1)=LIST 25 DEF KEY(2)=LOAD! RUN	A DEF KEY statement defines a function for any of the ten special function keys. The statement on line number 15 defines special function key 1 as LIST. The statement on line number 25 defines special function key 2 as the multi-command LOAD: RUN.

3.1.12 Comment and control statements

REM	200 REM JOB-1	Comment statement (not executed).
STOP	850 STOP	Stops program execution and awaits a command entry. When a CONT command is entered, program execution is continued.
END	1999 END	Declares the end of a program. Although the program is stopped, the following program is executed if a CONT command is entered.
CLR	300 CLR	Clears all variables and arrays, that is, fills all numeric variables and arrays with zeros and all string variables and arrays with nulls.
CURSOR	50 CURSOR 25, 15 60 PRINT "ABC"	The CURSOR command moves the cursor to any position on the screen. The first operand represents the horizontal location of the destination, and must be between 0 and 39 in 40-character mode, and must be between 0 and 79 in 80-character mode. The second operand represents the vertical location of the destination and must be between 0 and 24. The left example displays "ABC" starting at location (25, 15) (the 26th position from the left side and the 16th position from the top).
CSRH		System variable indicating the X-coordinate (horizontal location) of the cursor.
CSRV		System variable indicating the Y-coordinate (vertical location) of the cursor.
CONSOLE	10 CONSOLE S10, 20	Sets the scrolling area to lines 10 through 20.
	20 CONSOLE C80	Sets the display in the 80 characters/line mode.
	30 CONSOLE C40	Sets the display in the 40 characters/line mode.
	40 CONSOLE R	Sets the display in the reverse mode.
	50 CONSOLE N	Sets the display in the normal mode.
CHANGE	10 CHANGE	Reverses the function of the SHIFT key concerned with alphabetic keys.

REW	710 REW	Rewinds the cassette tape.
FAST	720 FAST	Fast-forwards the cassette tape.
SIZE	? SIZE	Displays the amount of unused memory area in bytes.
TI$	100 TI$ = "102030"	Sets the built-in clock to 10:20:30 AM. Data between the double quotation marks must be numerals.

3.1.13 Music control statements

MUSIC **TEMPO**		The MUSIC statement generates a melody from the speaker according to the melody string data enclosed in quotation marks or string variables at the tempo specified by the TEMPO statement.
	300 TEMPO 7 310 MUSIC "DE#FGA"	The TEMPO statement on line number 300 specifies tempo 7. The MUSIC statement on line number 310 generates a melody consisting of D, E, F sharp, G and A. Each note is a quarter note. When the TEMPO statement is omitted, default tempo is set.
	300 M1$ = "C3DG + C" 310 M2$ = "BGD – G" 320 M3$ = "C8R5" 330 MUSIC M1$, M2$, M3$	In this example, the melody is divided into 3 parts and substituted in 3 string variables. The following melody is generated from the speaker at tempo 4. 

3.1.14 Graphic control statements

GRAPH	10 GRAPH I1	Places graphic area 1 in the input mode. (That is, data are to be transferred to graphic area 1.)
	20 GRAPH O1	Places graphic area 1 in the output mode.
	30 GRAPH O2	Places graphic area 2 in the output mode.
	40 GRAPH O12	Places graphic areas 1 and 2 in the output mode.
	50 GRAPH O0	Resets the graphic output.
	60 GRAPH C	Clears graphic area that is in the input mode.
	70 GRAPH F	Fills graphic area that is in the input mode.
	80 GRAPH I1, C, O1	Places graphic area 1 in the input mode, then clears it and places it in the output mode.

SET

Sets a dot in the specified position in a graphic area operating in the input mode.
The first operand specifies the X-coordinates (0-319) and the second operand specifies the Y-coordinates (0-199).

```
300 SET 160, 100
```

Displays a dot in the center of the screen.

RESET

Resets a dot in the specified position in a graphic area operating in the input mode.

```
310 RESET 160, 100
```

Resets a dot from the center of the screen.

LINE

Draws lines connecting positions specified by operands.

```
400 LINE 110, 50, 210, 50,
    210, 150, 110, 150, 110,
    50
```

Draws a square the length of whose side is 100 in the center of the display screen.

BLINE

Draws black lines connecting positions specified by operands.

POSITION

Sets the location of the position pointer in a graphic area. The PATTERN statement (see below) is executed starting at the location indicated by the position pointer.

```
20 GRAPH I2, C, O2
30 POSITION 0, 50
40 PATTERN 8, A$
```

Places graphic area 2 in the input mode, sets the position pointer to the position corresponding to the position on the display screen which is at (0, 50), then transfers data from variable A$ to graphic area 2 so that the pattern corresponding to the contents of A$ is drawn on the screen starting at (0, 50).

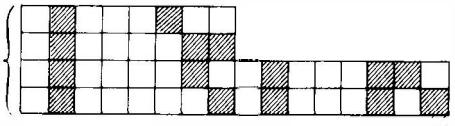
PATTERN

Draws the dot pattern specified by operands in a graphic area which is in the input mode. Each dot pattern unit consists of 8 dots arranged horizontally and corresponds to 8 bits representing a character. Elements are stacked in the number of layers specified by the value of the first operand and the direction in which layers are stacked is specified by the sign of the first operand.

```
10 C$ = "ABCDEF"
20 PATTERN 4, C$
```

Draws the dot pattern shown as follows.

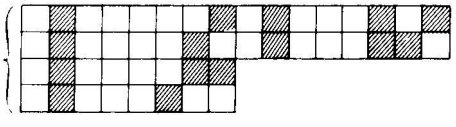
4 layers



```
30 PATTERN -4, C$
```

Draws the following dot pattern.

4 layers



POINT

```
100 ON POINT (X, Y)
GOTO 10, 20, 30
```

Ascertains the dot (X, Y) whether it is set or reset, and branches according to the result.

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.

POSH		System variable indicating the X-coordinate (horizontal location) of the position pointer.
POSV		System variable indicating the Y-coordinate (vertical location) of the position pointer.

3.1.15 Machine language control statements

LIMIT	100 LIMIT 49151	Limits the area in which BASIC programs can be loaded to the area up to address 49151 ($BFFF in hexadecimal).
	100 LIMIT A	Limits the area in which BASIC programs can be loaded to the area up to the address indicated by variable A.
	100 LIMIT $BFFF	Limits the area in which BASIC programs can be loaded to the area up to $BFFF (hexadecimal). Hexadecimal numbers are indicated by a dollar sign as shown at left.
	300 LIMIT MAX	Set the maximum address of the area in which BASIC programs can be loaded to the maximum address of the memory installed.
	200 LIMIT $BFFF 210 LOAD FD2 "S-R1"	Loads machine language program (object program) "S-R1" in the machine language link area from the diskette in drive 2 when the loading address of the program is $C000 or higher.
POKE	120 POKE 49450, 175	Stores 175 in address 49450.
	130 POKE AD, DA	Stores data (between 0 and 255) specified by variable DA into the address indicated by variable AD.
PEEK	150 A=PEEK (49450)	Substitutes data stored in address 49450 into variable A.
	160 B=PEEK (C)	Substitutes the contents of the address indicated by variable C into variable B.
USR	500 USR (49152)	Transfers program control to address 49152. This function is the same as that performed by the CALL instruction, which calls a machine language program. When a RET command is encountered in the machine language program, program control is returned to the BASIC program.
	550 USR (AD)	Calls the program starting at the address specified by variable AD.
	570 USR ($C000)	Calls the program starting at address $C000.

600 WOPEN #8, USR ($C000) 610 PRINT #8, A$ 620 CLOSE #8	The statement on line 600 opens a file which is to be written by the machine language program called by USR ($C000) with logical number 8 assigned. At this stage of program execution the USR function is not executed. The statement on line 610 loads the beginning address of the memory area set with variable A$ into the DE register of the CPU and its length (max. 255 bytes) into the BC register. This enables the program called by USR (SC000) to obtain data in A$. It then executes USR ($C000).
700 ROPEN #9, USR ($C100) 710 INPUT #9, B$ 720 CLOSE #9	The statement on line number 700 opens a file which is to be read by the machine language program called by USR ($C100) with logical number 9 assigned. The statement on line number 710 executes USR ($C100). The machine language program called loads string data in the memory area starting at the address indicated by the DE register and loads the length of the data string read in the BC register. It then returns program control to the BASIC program. The BASIC program refers to this memory area as B$.

3.1.16 Printer control statements

PRINT/P		Performs the nearly same operation as the PRINT statement on the optional printer.
	PRINT/P A$	Outputs to the printer just as it is the contents of string variable A$.
	PRINT/P CHR$ (N)	For an N of $32 \leqq N \leqq 255$, it considers this as an ASCII code, and outputs a matching character to the printer.
	PRINT/P CHR$ (5)	Feeds paper to top of the form position on the next page. It is called form feed. The function of the control button "TOP OF FORM" of the printer is controlled by software.
	PRINT/P CHR$ (6)	Returns the printing mode to its initial condition. Furthermore, the form feed is carried out. It is called initial mode set. Initial mode means 80 digit mode, line space mode.
	PRINT/P CHR$ (16)	Sets the printing mode for line spacing. It is called line space mode.
	PRINT/P CHR$ (17)	Sets the printing mode, completely closing up printing line space.
	PRINT/P CHR$ (18)	Sets the mode to double the present printing size of the characters. It is called double size mode. There is a 40 digit mode and a 68 digit mode.

	PRINT/P CHR$ (19)	Cancels the double size mode. Returns to the 80 digit mode or 136 digit mode.
	PRINT/P CHR$ (20)	Sets the printing mode as reduced characters of the normal size printing (80 digit mode). It is called reduced mode or 136 digit mode. With the bit image mode, it sets the 816 bit data in one line in the printing mode.
	PRINT/P CHR$ (21)	Cancels the reduced mode.
IMAGE/P	30 IMAGE/P CHR$ (255), "UU"	Draws a desired dot pattern (image) specified in the operand on the line printer according to the operating mode (image mode 1 or 2).
COPY/P	10 COPY/P 1	Causes the printer to copy the character display.
	20 COPY/P 2	Causes the printer to copy the dot pattern set in graphic area 1.
	30 COPY/P 3	Causes the printer to copy the dot pattern set in graphic area 2.
	40 COPY/P 4	Causes the printer to copy the dot pattern set in both graphic area 1 and graphic area 2.
PAGE/P	100 PAGE/P 20	Specifies 20 lines to be contained in one page of the MZ-80P5 line printer.
PRINT/P USING	The same as the PRINT USING statement.	Specifies the format in which numeric data is to be output on the printer. (Refer to PRINT USING on page 46.)
LIST/P	LIST/P	Prints out all lines contained in the BASIC txext area on the line printer.
DIR/P	DIR FDd/P	Prints the file directory of the diskette in drive d on the line printer.

3.1.17 I/O input/output statements

INP		Reads data on the specified I/O port.
	10 INP @12, A 20 PRINT A	The statement on line number 10 reads data on I/O port 12.
OUT		Outputs data to the specified I/O port.
	30 B = A^2+0.3 40 OUT @13, B	The statement on line 40 outputs the value of B to I/O port 13.

3.1.18 Arithmetic functions

ABS	100 A = ABS (X)	Substitutes the absolute value of variable X into variable A. X may be either a constant or an expression. Ex) ABS (–3) = 3 ABS (12) = 12
INT	100 A = INT (X)	Substitutes the greatest integer which is less than X into variable A. X may be either a numeric constant or an expression. Ex) INT (3.87) = 3 INT (0.6) = 0 INT (–3.87) = –4
SGN	100 A = SGN (X)	Substitutes one of the following values into variable A: –1 when X<0, 0 when X=0 and 1 when X>0. X may be either a constant or an expression. Ex) SGN (0.4) = 1 SGN (0) = 0 SGN (–400) = –1
SQR	100 A = SQR (X)	Substitutes the square root of variable X into variable A. X may either a numeric constant or an expression; however, it must be greater than or equal to 0.
RND		This function generates random numbers which take any value between 0.000000000000001 and 0.999999999999999, and works in three manners depending upon the value in parentheses.
	100 A = RND (0)	When the value in parentheses is 0, the random number generating routine is initialized and the function always gives the first number of the random number group generated. Therefore, statement on line 100 gives the same value to variables A and B.
	110 C = RND (–3)	When the value in parentheses is negative, the random number generating routine is given time information from the built-in clock and generates a random number between 0.000000000000001 and 0.999999999999999. The statement on line 110 generates a random number in this manner.
	200 A = RND (1) 210 B = RND (10)	When the value in parentheses is positive, as shown in the statements on lines 200 and 210, the function gives the random number following the one previously given in the random number group generated. The value obtained is independent of the value in parentheses.

3.1.19 String control functions

LEFT $	10 A$ = LEFT$ (X$, N)	Substitutes the first N characters of string variable X$ into string variable A$. N may be either a constant, a variable or an expression.

MID $	20 B$ = MID$ (X$, M, N)	Substitutes the N characters following the Mth character from the beginning of string variable X$ into string variable B$.
RIGHT $	30 C$ = RIGHT$ (X$, N)	Substitutes the last N characters of string variable X$ into string variable C$.
SPACE $	40 D$ = SPACE$ (N)	Substitutes the N spaces into string variable D$.
STRING $	50 E$ = STRING $ ("*", 10)	Substitutes the ten repetitions of "*" into string variable E$.
CHR $	60 F$ = CHR $ (A)	Substitutes the character corresponding to the ASCII code in numeric variable A into string variable F$. A may be either a constant, a variable or an expression.
ASC	70 A = ASC (X$)	Substitutes the ASCII code (in decimal) corresponding to the first character of string variable X$ into numeric variable A.
STR$	80 N$ = STR$ (I)	Converts the numeric value of numeric variable I into string of numerals and substitutes it into string variable N$.
VAL	90 I = VAL (N$)	Converts string of numerals contained in string variable N$ into the numeric data as is and substitutes it into numeric variable I.
LEN	100 LX = LEN (X$)	Substitutes the length (number of bytes) of string variable X$ into numeric variable LX.
	110 LS = LEN (X$ + Y$)	Substitutes the sum of the lengths (number of bytes) of string variable X$ and Y$ into numeric variable LS.

3.1.20 Tabulation function

TAB	10 PRINT TAB (X) ; A	Displays the value of variable A at the Xth position from the left side.

3.1.21 Arithmetic operators

The number to the left of each operator indicates its operational priority. Any group of operations enclosed in parentheses has first priority.

❶ ^	10 A = X^Y (power)	Substitutes X^Y into variable A. (If X is negative and Y is not an integer, an error results.)
❷ –	10 A = –B (negative sign)	Note that "–" in –B is the negative sign and "–" in 0-B represents subtraction.
❸ *	10 A = X*Y (multiplication)	Multiplies X by Y and substritutes the result into variable A.
❸ /	10 A = X/Y (division)	Divides X by Y and substitutes the result into variable A.
❹ +	10 A = X + Y (addition)	Adds X and Y and substitutes the result into variable A.
❹ –	10 A = X – Y (subtraction)	Subtracts X from Y and substitutes the result into variable A.

3.1.22 Logical operators

=	10 IF A = X THEN ...	If the value of variable A is equal to X, the statement following THEN is executed.
	20 IF A$ = "XYZ" THEN ...	If the content of variable A$ is "XYZ", the statement following THEN is executed.
<> or ><	10 IF A <> X THEN ...	If the value of variable A is not equal to X, the statement following THEN is executed.
>= or =>	10 IF A >= X THEN ...	If the value of variable A is greater than or equal to X, the statement following THEN is executed.
<= or =<	10 IF A <= X THEN ...	If the value of variable A is less than or equal to X, the statement following THEN is executed.
*	40 IF (A>X)*(B>Y) THEN ...	If the value of variable A is greater than X and the value of variable B is greater than Y, the statement following THEN is executed.
+	50 IF (A>X) + (B>Y) THEN ...	If the value of variable A is greater than X or the value of variable B is greater than the value of Y, the statement following to THEN is executed.

3.1.23 Other symbols

?	200 ? "A+B=" ; A+B 210 PRINT "A+B=" ; A+B	Can be used instead of PRINT. Therefore, the statement on line number 200 is identical in function to that on line number 210.
:	220 A=X : B=X ^ 2 : ? A, B	Separates two statements from each other. This separator is used when multiple statements are written on the same line. Three statements are written on line number 220.
;	230 PRINT "AB" ; "CD" ; "EF"	Displays characters to the right of separators following characters on the left. The statement on line 230 displays "ABCDEF" on the screen with no spaces between characters.
	240 INPUT "X=" ; X$	Displays "X=" on the screen and awaits entry of data for X$ from the keyboard.
,	250 PRINT "AB", "CD" , "E"	Displays character strings in a tabulated format; i.e. AB first appears, then CD appears in the position corresponding to the starting position of A plus 10 spaces and E appears in the position corresponding to the starting position of C plus 10 spaces.
	300 DIM A(20), B$(3, 6)	A comma is used to separate two variables.
" "	320 A$ = "SHARP BASIC" 330 B$ = "MZ-80B"	Indicates that characters between double quotation marks form a string constant.
$	340 C$="ABC"+"CHR$(3)	Indicates that the variable followed by a dollar sign is a string variable.
	500 LIMIT $BFFF	Indicates that numeric data following a dollar sign is represented in hexadecimal notation.
π	550 S = SIN (X*π/180)	π represents 3.141592653589793 (ratio of the circumference of a circle to its diameter).

3.2 Specifications of double precision BASIC SB-6610 interpreter

■ **Type**

Interpreter system

Program size: about 16.5K bytes

Start address: $1220 (hexadecimal)

■ **Numeric data**

BCD floating point system

Real number: +1E-48 ~ +9.999999999999999E + 78

Hexadecimal: Can be used only when a hexadecimal address is directly specified in LIMIT, POKE, PEEK or USR. Expressed in 4-digit hexadecimal notation following $.

Example: LIMIT $8FFF, USR ($A000)

■ **Numeric variables**

Numeric variable: Only the first two character variable name are significant. The first character must be alphabetic. Special character, BASIC key words or names including key words cannot be used.

Example: A, X F1 and AA are correct.
ABC and ABD are processed as the same name.
DATA, XDATA, A#, etc. cannot be used.

One-dimensional array variable: The size of a one-dimensional array is limited only by the amount of unused memory space. A one-dimensional array must be defined by a DIM statement. Conditions on characters which can be used are the same as these shown above.

Example: DIM Q (500) —— Q (0) through Q (500)

Two-dimensional array variable: A variable with two subscripts. The maximum value of each subscript is 255, but it is limited by the amount of unused memory space.

Example: DIM A3 (7, 7) —— A3 (0, 0) through A3 (7, 7)

■ **String data**

Maximum length: 255 characters

Internal data string: A train of ASCII character codes forming a data string followed by a carriage return code (0DH).

■ **String variable**

Kinds of string variables: String variable
One-dimensional string variable
Two-dimensional string variable

Format: A string variable is expressed by a name following $. Requirements for string variable name are the same as those for numeric variable.

Example: A$, ST$ and NI$ are normal string variables.
NAME1$ and NAME2$ are processed as the same string variable.
TI$, CHR$ ( ), etc. are special string variables.
DIM S$ (3, 3) defines a two-dimensional string array including 16 elements S$ (0, 0) through S$ (3, 3).

■ **Others**

Line numbers: 1 – 65535

File name: Significant digits in a file name specified in LOAD, VERIFY or SAVE commands are a maximum of 16 digits.

Cursor position in a CURSOR statement:

X = 0 – 79
Y = 0 – 24 } 80 characters/line mode

X = 0 – 39
Y = 0 – 24 } 40 characters/line mode

Point position in a SET or RESET statement:

X = 0 – 319
Y = 0 – 199

Clock string TI$: 6-digit decimal string
The 1st 2 characters = 00 – 23 (hour digits)
The 2nd 2 characters = 00 – 59 (minute digits)
The 3rd 2 characters = 00 – 59 (second digits)

Levels of FOR ... NEXT loops: Maximum of 15

Levels of GOSUB loops: Maximum of 15

Levels of function definition routines with DEF FN:
Maximum of 6

Port address and data in INP or OUT statement:
Port address = 0 – 255
Data = 8 bit data or decimal numbers (0 – 255)

Value of π: π = 3.141592653589793

APPENDIX

The Appendix includes the following;

- *ASCII Code Table . . . . . Table A.1*
- *DISK BASIC interpreter SB-6610 Error Message Table . . . . . Table A.2*
  This table list 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*
- *Handling diskettes*

A.1 ASCII Code Table

Code in parentheses represents a hexadecimal code.

CODE	(HEX)	CHARACTER	CODE	(HEX)	CHARACTER	CODE	(HEX)	CHARACTER	CODE	(HEX)	CHARACTER	CODE	(HEX)	CHARACTER
0	(00)	NULL	26	(1A)		52	(34)	4	78	(4E)	N	104	(68)	h
1	(01)	↓	27	(1B)		53	(35)	5	79	(4F)	O	105	(69)	i
2	(02)	↑	28	(1C)		54	(36)	6	80	(50)	P	106	(6A)	j
3	(03)	→	29	(1D)		55	(37)	7	81	(51)	Q	107	(6B)	k
4	(04)	←	30	(1E)		56	(38)	8	82	(52)	R	108	(6C)	l
5	(05)	HOME	31	(1F)	▒	57	(39)	9	83	(53)	S	109	(6D)	m
6	(06)	CLR	32	(20)		58	(3A)	:	84	(54)	T	110	(6E)	n
7	(07)	DEL	33	(21)	!	59	(3B)	;	85	(55)	U	111	(6F)	o
8	(08)	INST	34	(22)	"	60	(3C)	<	86	(56)	V	112	(70)	p
9	(09)	GRPH	35	(23)	#	61	(3D)	=	87	(57)	W	113	(71)	q
10	(0A)	SFT LOCK	36	(24)	$	62	(3E)	>	88	(58)	X	114	(72)	r
11	(0B)		37	(25)	%	63	(3F)	?	89	(59)	Y	115	(73)	s
12	(0C)	RVS	38	(26)	&	64	(40)	@	90	(5A)	Z	116	(74)	t
13	(0D)		39	(27)	'	65	(41)	A	91	(5B)	[	117	(75)	u
14	(0E)	SCRIPT	40	(28)	(	66	(42)	B	92	(5C)	\	118	(76)	v
15	(0F)	RVS CANCEL	41	(29)	)	67	(43)	C	93	(5D)	]	119	(77)	w
16	(10)		42	(2A)	*	68	(44)	D	94	(5E)	^	120	(78)	x
17	(11)		43	(2B)	+	69	(45)	E	95	(5F)	←	121	(79)	y
18	(12)		44	(2C)	,	70	(46)	F	96	(60)	`	122	(7A)	z
19	(13)		45	(2D)	—	71	(47)	G	97	(61)	a	123	(7B)	{
20	(14)		46	(2E)	.	72	(48)	H	98	(62)	b	124	(7C)	|
21	(15)		47	(2F)	/	73	(49)	I	99	(63)	c	125	(7D)	}
22	(16)		48	(30)	0	74	(4A)	J	100	(64)	d	126	(7E)	~
23	(17)		49	(31)	1	75	(4B)	K	101	(65)	e	127	(7F)	↴
24	(18)		50	(32)	2	76	(4C)	L	102	(66)	f			
25	(19)		51	(33)	3	77	(4D)	M	103	(67)	g			

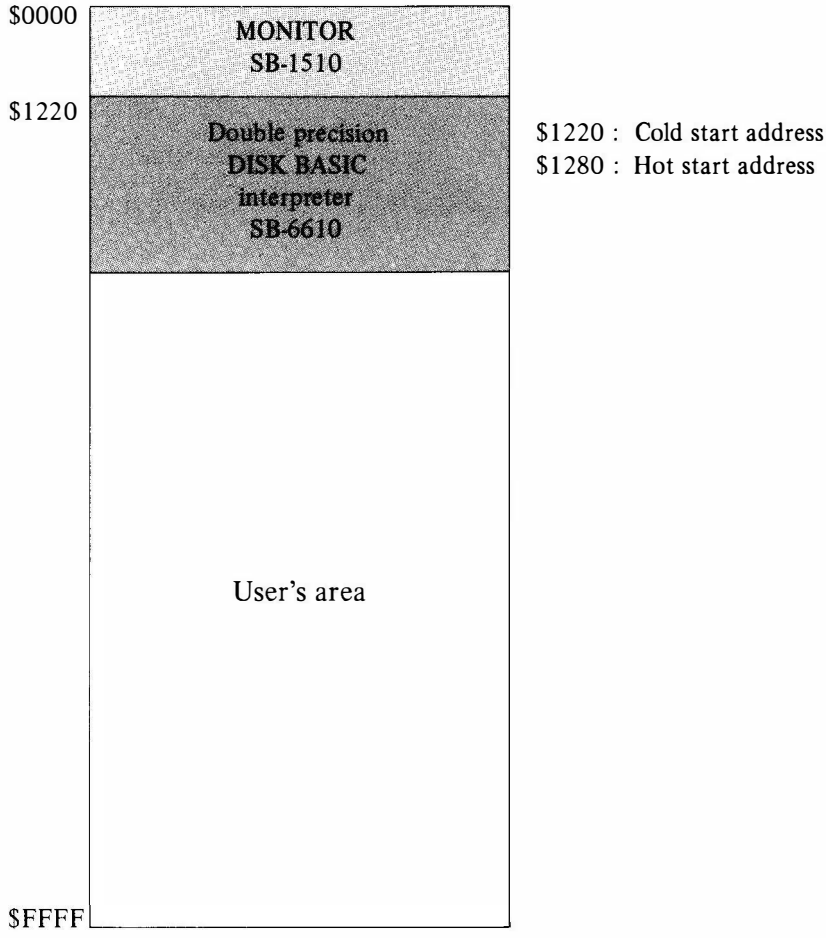
CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
128 (80)		154 (9A)		180 (B4)	4	206 (CE)	N	232 (E8)	h
129 (81)		155 (9B)		181 (B5)	5	207 (CF)	O	233 (E9)	i
130 (82)		156 (9C)		182 (B6)	6	208 (D0)	P	234 (EA)	j
131 (83)		157 (9D)		183 (B7)	7	209 (D1)	Q	235 (EB)	k
132 (84)		158 (9E)		184 (B8)	8	210 (D2)	R	236 (EC)	l
133 (85)		159 (9F)		185 (B9)	9	211 (D3)	S	237 (ED)	m
134 (86)		160 (A0)		186 (BA)	:	212 (D4)	T	238 (EE)	n
135 (87)		161 (A1)	!	187 (BB)	;	213 (D5)	U	239 (EF)	o
136 (88)		162 (A2)	"	188 (BC)	<	214 (D6)	V	240 (F0)	p
137 (89)		163 (A3)	#	189 (BD)	=	215 (D7)	W	241 (F1)	q
138 (8A)		164 (A4)	$	190 (BE)	>	216 (D8)	X	242 (F2)	r
139 (8B)		165 (A5)	%	191 (BF)	?	217 (D9)	Y	243 (F3)	s
140 (8C)		166 (A6)	&	192 (C0)	@	218 (DA)	Z	244 (F4)	t
141 (8D)		167 (A7)	'	193 (C1)	A	219 (DB)		245 (F5)	u
142 (8E)		168 (A8)	(	194 (C2)	B	220 (DC)		246 (F6)	v
143 (8F)		169 (A9)	)	195 (C3)	C	221 (DD)		247 (F7)	w
144 (90)		170 (AA)	*	196 (C4)	D	222 (DE)		248 (F8)	x
145 (91)	¥	171 (AB)	+	197 (C5)	E	223 (DF)		249 (F9)	y
146 (92)	£	172 (AC)	,	198 (C6)	F	224 (E0)		250 (FA)	z
147 (93)		173 (AD)	-	199 (C7)	G	225 (E1)	a	251 (FB)	{
148 (94)		174 (AE)	.	200 (C8)	H	226 (E2)	b	252 (FC)	|
149 (95)		175 (AF)	/	201 (C9)	I	227 (E3)	c	253 (FD)	}
150 (96)		176 (B0)	0	202 (CA)	J	228 (E4)	d	254 (FE)	~
151 (97)		177 (B1)	1	203 (CB)	K	229 (E5)	e	255 (FF)	π
152 (98)		178 (B2)	2	204 (CC)	L	230 (E6)	f		
153 (99)		179 (B3)	3	205 (CD)	M	231 (E7)	g		

A2. 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 15.
11	The number of levels of FOR-NEXT nests exceeded 15.
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 reference line number was specified in a statement.
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	RESUME statement cannot be executed.
21	A RESUME statement was used without a corresponding error process.
22	
23	
24	A READ statement was used without a corresponding DATA statement.
25	The number of SWAP levels exceeded 1.
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	

Error No.	Meaning
36	
37	
38	
39	
40	File was not found.
41	Disk drive hardware error.
42	A file name already used was defined again.
43	OPEN, DELETE, RENAME statements were issued to an open file.
44	An unopened file was reference or a CLOSE or KILL statement was issued to it.
45	A file was accessed in different mode from ROPEN, WOPEN or XOPEN.
46	A protected file was accessed for writing.
47	
48	
49	
50	The disk drive is not ready.
51	The total number of files on a volume exceeded 63.
52	Volume number error
53	File space on the diskette is insufficient.
54	A diskette which has not been initialized was loaded.
55	The number of data of a BSD file exceeded 64K bytes.
56	Data error occurred on an FDC routine call.
57	The diskette cannot be used.
58	
59	
60	Illegal file name was specified.
61	Illegal file mode was specified.
62	
63	Out of file
64	Illegal logical number was specified.
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 Handling diskettes

The master diskette must be handled especially carefully. Make a submaster diskette by means of the diskette-copy program in the OBJ file "Utility" on the master diskette. Be sure to keep the master diskette in a safe place.

All optional blank diskettes supplied by the Sharp Co. are not initialized. Be sure to initialize them before use.

Notes on handling of diskettes

- Fingerprints on a diskette may permanently render it unusable. Never touch the diskette surface through the head window.
- Insert the diskette straight into the drive until it stops, then close the front door gently. Rough handling may damage the diskette.
- Do not fold or bend the diskette, or it may be rendered unusable.
- Write the index label before it is affixed to the jacket. If it is written after it is affixed to the jacket, use a felt marker or other soft tip pen.
- Ashes and drinks are the most common contaminants to guard against.
- Ambient temperature: 4~53°C.

  Storage temperatures for the diskette are 4°C to 53°C. Do not leave the diskette exposed to direct sunlight, or locate it in a place subject to the temperatures exceeding 53°C. This may cause the jacket to be deformed and unusable.

  When using the diskette, ensure that the temperature range described on the protective envelope is observed. Environmental conditions may differ between the storage and operation places. This requires the diskette to be placed under the proper operating environment for a while before use.

Notes on storing diskettes

- Keep the diskettes away from magnets. Even a magnet ring or magnet necklace may damage data on the diskette. Electrical equipment such as the display unit of the computer, a cassette tape recorder, or a TV set generates magnetic flux, so keep diskettes away from such equipment.
- Keep the diskette in the envelope supplied. Make it a habit to put the diskette in the envelope immediately after it has been take out of the drive. This will prevents almost all problems which result from careless handling of diskettes. The master diskette must be handled especially carefully. The envelopes supplied are made of special materials and guard against static electricity and moisture.
- When storing diskettes for a long time, keep the envelopes in the storage case. Be sure the envelopes are stored vertically in the storage case. Do not incline or bend the envelope. The master diskette is not supplied with a storage case.
- Do not clip diskettes with paper clips or the like.
- Do not place any heavy objects on diskettes.

SHARP CORPORATION

TINSE0018PAZZ
810429-500-K
Printed in Japan

MZ-80DPB
E1

PRECAUTION FOR USE OF DISK BASIC SB-6510, AND DOUBLE-PRECISION DISK BASIC SB-6610

Do not replace the disket with another until a KILL instruction has been executed, when the disket is inserted into the floppy disk drive and the read/write operation is in action. Replacing the disket with the file open will destroy the contents of the newly inserted disket.

To save the program into different diskets with SAVE instructions, each SAVE instruction should be followed by execution of a KILL or DIR instruction. If the disket is replaced with another one immediately after the SAVE instruction and the SAVE instruction is executed again, the contents of the replaced new disket will be destroyed.

SHARP CORPORATION

Printed in Japan