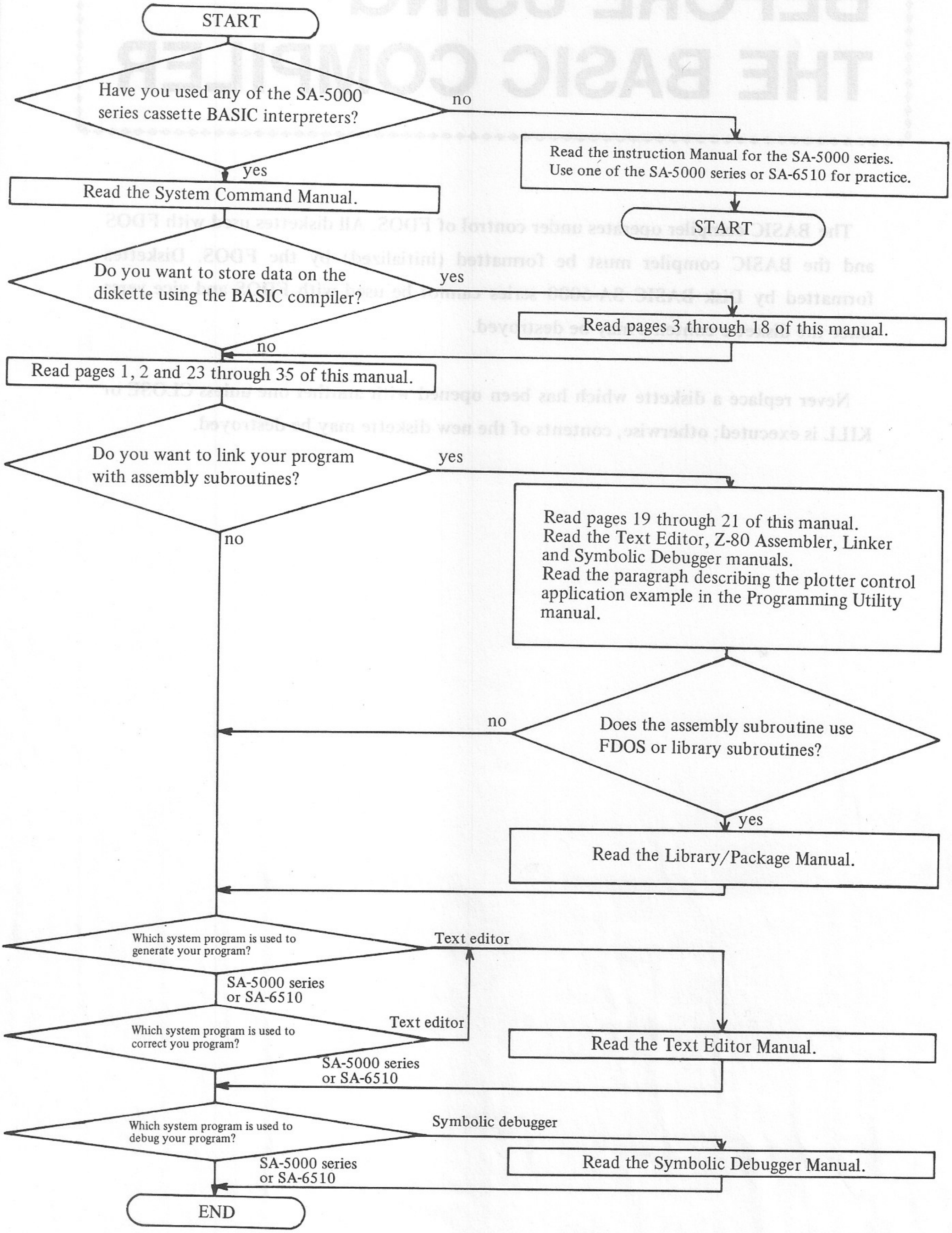
BEFORE USING THE BASIC COMPILER

The BASIC compiler operates under control of FDOS. All diskettes used with FDOS and the BASIC compiler must be formatted (initialized) by the FDOS. Diskettes formatted by Disk BASIC SA-6000 series cannot be used with FDOS and vice versa since the diskette contents may be destroyed.

Never replace a diskette which has been opened with another one unless CLOSE or KILL is executed; otherwise, contents of the new diskette may be destroyed.

—Guide for Reading this Manual—



— CONTENTS —

OUTLINE

The BASIC compiler translates a BASIC program into Z80 machine language. Use the following procedures to run a BASIC program with the BASIC compiler.

1. Generate a Source Program with Either of the Following Methods

a. Using the text editor (Refer to the Text Editor Manual.)

b. Using the BASIC interpreter and the CONVERT command (Refer to the explanation of CONVERT in the System Command Manual.)

2. Compile the Source Program

Example 1) 2 > BASIC PROG1 ↵ (↵ indicates a carriage return.)

(PROG1 = Filename)

Compiles source program PROG1 . ASC to generate relocatable file PROG1 . RB.

Example 2) 2 > BASIC PROG1. PROG1A/O ↵

Compiles source program PROG1 . ASC to generate relocatable file PROG1A . RB.

Example 3) 2 > BASIC/C PROG1 ↵

Compiles PROG1 . ASC to generate PROG1 . RB with the list displayed on $CRT.

Example 4) 2 > BASIC/P PROG1 ↵

Compiles PROG1 . ASC to generate PROG1 . RB with the list output to $LPT.

Example 5) 2 > BASIC/P/N PROG1 ↵

Outputs the list to $LPT but does not generate a relocatable file.

3. Link

Link the relocatable file generated in step 2 with RELO . LIB (the relocatable library which includes subroutines for arithmetic operations, SIN and COS functions, etc.). RELO . LIB is linked automatically.

Example 6) 2 > LINK PROG1 ↵

Links relocatable file PROG1 . RB with RELO, LIB in $FD1 to generate object file PROG1 . OBJ.

Example 7) 2 > LINK/P PROG1 ↵

The same as example 6 except that the link information is printed on $LPT.

4. Execute the Object Program

Example 8) 2 > RUN PROG1 ↵

Loads PROG1 . OBJ and executes it.

As described above, many steps are required for the compiler to generate, correct and execute a program, but the process has the following advantages.

1. Fast execution time
2. FDOS instructions can be executed.
3. Programs generated by the assembler can be easily linked.
4. Statements can be extended.

Note: Sometimes source files exceeding 15K bytes cannot be compiled.

5. Compilation and Execution of Long Program

It may occur that compilation or linking cannot be performed because of insufficient memory capacity when a long program is to be compiled and executed.

a. Requirement for compilation

The maximum length of a source program which the BASIC compiler can handle is about 10K−15K bytes. If the source program is too long, its length must be reduced or it must be divided into several short sections.

b. Requirement for linking

The standard linker can handle a BASIC source program the length of which is up to about 10K bytes and the resultant object program of which is about 23K bytes long. If the length exceeds the above values, special linker MLINK is used. This linker can handle a BASIC source program the length of which is up to about 15K bytes and the resultatnt object program of which is about 33K bytes long.

6. Program Debug

A BASIC source file is converted into a relocatable file with the BASIC compiler then into an object file with the linker. The source program does not exist in memory when the object program is executed. With the BASIC interpreter, the value of variables can be changed or the program can be corrected during execution by interrupting and restarting it. This is impossible with the BASIC compiler. The source program must be corrected, then it must be compiled, linked and executed.

There are three methods for debugging the source program.

a. Generating and debugging a program with the BASIC interpreter, then compiling the debugged program. (This method is the easiest but it does not allow use of statements which are unique to the BASIC compiler.)

b. Inserting PRINT statements in appropriate positions in a program to display data required for debugging.

c. Using the FDOS symbolic debugger.
(This method is effective when debugging programs which are linked with assembly subroutines.)

SEQUENTIAL FILE

A file is a set of related records which are treated as a unit. The FDOS has a file directory which controls access to files; a file cannot be accessed unless its name is stored in the file directory.

Cassette tape files are necessarily sequential files. To read the 100th record in a cassette tape file, the preceding 99 records must be skipped.

On the other hand, floppy disk files are usually random files, although sequential files can also be stored on a floppy disk. Assume an address list in which names are written at random. To find specified name, a search must be made from the beginning of the address list. Such a file is a sequential file.

The advantages and disadvantages of sequential files are as follows.

Advantages:
- A diskette can be used effectively because there exist no empty records in a file.
- They are effective when the entire contents of a file must be processed; that is, when no search operation is necessary.

Disadvantages:
- It takes a time to find a single specific record. When a record is inserted or deleted, all records must be rewritten.

Sequential file processing statements for both the BASIC compiler and the cassette based BASIC interpreter are listed in the following tables for comparison.

Writing data

	BASIC compiler	Cassette based BASIC interpreter
File open statement	WOPEN #n, "filename"	WOPEN/T "filename"
Data write statement	PRINT #n, data	PRINT/T data
File close statement	CLOSE #n	CLOSE/T
Cancel statement	KILL #n	———

Reading data

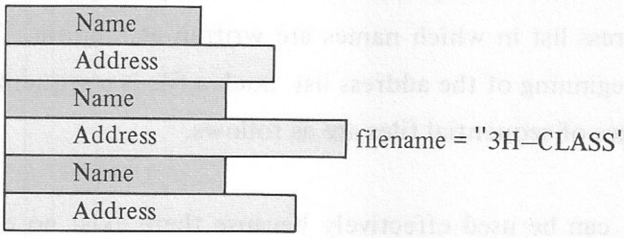
	BASIC compiler	Cassette based BASIC interpreter
File open statement	ROPEN #n, "filename"	ROPEN/T "filename"
Data read statement	INPUT #n, variable	INPUT/T variable
File close statement	CLOSE #n	CLOSE/T
File end detection	IF EOF (#n) THEN	———

Note: With the BASIC compiler, a device name ($KB, $PTR, etc.) can be written instead of a filename.

#n appearing in all the BASIC compiler file processing statements is called a **logical (file) number**; this must always be specified.

Only one file can be accessed at a time with the cassette based BASIC interpreter. With the BASIC compiler, however, multiple files can be dealt with simultaneously. To achieve this, a logical number is assigned to each file and all files are specified with logical numbers. (To prepare a file, assign a logical number to it; this is referred to as "opening a file").

Let's consider an address list as a simple example.



As shown above, **the length of records stored in a sequential file is not fixed.** That is, a sequential file is suitable for storing a set of records which have variable lengths.

The following program stores 50 names and 50 addresses in file 3H-CLASS. The following program reads records from file 3H-CLASS and displays names and their addresses 10 at a time on the displays names and their addresses 10 at a time on the display screen.

(Write)

```
100 WOPEN #3, "$FD2 ; 3H–CLASS"
110 FOR P = 1 TO 50
120 INPUT "NAME=" ; NA$
130 INPUT "ADDRESS=" ; AD$
140 PRINT #3, NA$, AD$
150 NEXT P
160 CLOSE #3
```

(Read)

```
200 ROPEN #4, "$FD2 ; 3H–CLASS"
210 FOR P = 1 TO 5 : FOR Q = 1 TO 10
220 INPUT #4, NA$, 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"
390 CLOSE #4
```

In this example, a file is stored on the diskette in floppy disk drive 2. The statements used are explained below.

WOPEN #n, "filename"

This statement defines the name of the sequential file to be generated as "filename" and assigns the logical number #n (1 ~ 126) to it to write the file; that is, it declares that the file is hereafter specified with the logical number #n.

In the example, the statement on line 100 defines the file name as 3H–CLASS, and assigns the logical number #3 to it and declares that the file is to be stored on the diskette in floppy disk drive 2.

PRINT #n, data

This statement appends a record whose value is given by data to the file opened with logical number #n assigned. The file directory, however, is not cataloged when this statement is executed. If is cataloged when a CLOSE # statement is executed.

Multiple records can be appended with a single PRINT # statement as follows.

PRINT #n, data, data, data, ...

OUT #n, data

Writes data byte-by-byte in the sequential file which is opened for writing with logical number #n assigned. When the data is a numeric value, it must be from 0 – 225. Its binary value is written in the file. When the data is a string, characters are written in the file. Data separators (e.g., CR) are not written. In other respects, this is the same as the PRINT # statement.

CLOSE #n (Corresponding to WOPEN)

This statement stores the names of files generated with PRINT # statements in the file directory. The logical number definition is cleared when this statement is executed.

KILL #n

This statement is not used in the example. It cancels a WOPEN # statement. If this statement is executed instead of the CLOSE # statement, the file directory is not cataloged. The logical number difinition is cleared by execution of this statement.

Notes:

- A CLOSE or KILL statement without a logical number #n closes or cancels all open files and stops motors of floppy disk drives.
- Any volume number which can be specified with D-BASIC SB-6510 cannot be specified with the BASIC compiler.

ROPEN #n,"filename"

This statement assigns the logical number (1 ~ 126) specified by #n to the file specified by "filename" for reading.

In the example, the statement on line 200 specifies sequential file 3H–CLASS and assigns logical number #4 to it. It also declares that the file is to be read from the diskette in floppy disk drive 2.

INPUT #n, variable

This statement assigns a record value read sequentially from the open sequential file assigned logical number #n into the variable.

In the example, the statement on line 220 sequentially reads two records from file 3H–CLASS and assigns them to variables NA$ and AD$. As is shown in the example, multiple records can be read with a single INPUT # statement by separating the variables from each other with a comma.

INPUT #n, variable, variable, variable, ...

INP #n, variable

• Reads data byte-by-byte from the sequential file which is opened for reading with logical number #n assigned and assigns it to the specified variable. When the data read is numeric data, its decimal value is assigned to the variable. When the data read is string data, it is assigned to the variable as a string whose length is one byte.

CLOSE #n (Corresponding to ROPEN)

This statement closes the file assigned logical number #n and clears the logical number definition. A KILL # statement issued subsequent to an ROPEN statement acts in the same manner as a CLOSE # statement.

WAIT X

Suspends program execution for the time specified by X. X must be numeric data from 0 – 32767; it indicates the time in milliseconds.

File Name Format

A file name must consist of a maximum of 16 characters. Characters permitted are alphabetic characters, numerals, and the symbols ! # % & ' ( ) + − < = > @ [\ and] . Small letters, graphic characters and/or spaces cannot be used.

A file name may be preceded by the name of the device from which it is accessed.

Correct format:	"$FD1 ;PROG1"	"PROG1" on the diskette in drive 1.
	"$CMT ; SAMPLE"	"SAMPLE" in the cassette file.
	"TEST2"	"TEST2" on the diskette in the default drive.
	"$PTR"	Paper tape reader
	"$USR2"	User I/O
	"$PTP/PE"	Paper tape punch with even parity
Incorrect format	"$FD1"	A file name is required.
	"$LPT ; PROG2"	$LPT cannot be assigned a file name.

The file mode is generally omitted and the .ASC mode is assumed.

File Mode

(1) WOPEN #, ROPEN #

For the WOPEN # and ROPEN # statements, the default file mode is .ASC.

(2) PRINT #, INPUT #

The PRINT # and INPUT # statements can be used only for files with file mode .ASC. For other file modes, use the OUT # and INP # statements.

(3) XOPEN #, WOPEN/T, ROPEN/T

The file mode must be .ASC for the XOPEN #, WOPEN/T and ROPEN/T statements. The file mode specification may be omitted.

Locating the File End

No error occurs when the INPUT # statement is executed after the last record of a file has been read. The variable(s) is loaded with zero or null. However, this serves no purpose. **A special function is provided for locating the end of a file; this is EOF(#n), which gives TRUE when the end of a file is reached.**

Executing

IF EOF (#n) THEN ....

after the INPUT # statement causes the statement following THEN to be executed when the file end is reached.

(Exercise) Rewrite the sample program on page 5 so that names and addresses are read in groups of 10 until the file end is reached, assuming that the number of names stored in the file is unknown.

(Example of solution)

```
300  ROPEN #5,"$FD1 ; 3H-CLASS"
310  FOR 1 = 1 TO 10
320  INPUT #5, NA$, AD$
330  IF EOF (#5) THEN 400
340  PRINT NA$ : PRINT AD$
350  NEXT I
360  PRINT "STRIKE ANYKEY"
370  GET X$ : IF X$ = "  " THEN 370
380  GOTO 310
400  CLOSE #5
410  PRINT "FILE END" : END
```

(Exercise) Make a program which reads sequential file 3H–CLASS and generates two sequential files, one for names and the other for addresses.

(Example of solution)

```
500  ROPEN #6,"$FD2 ; 3H-CLASS"
510  WOPEN #7,"$FD2 ; NAME"
520  WOPEN #8,"$FD2 ; ADDRESS"
530  INPUT #6, NA$, AD$
540  IF EOF (#6) THEN 600
550  PRINT #7, NA$
560  PRINT #8, AD$
570  GOTO 530
600  CLOSE
610  END
```

(Exercise) Make a program which writes string data input from the key board to a sequential file, and which closes or kills the opened file when CLOSE or KILL is entered from the keyboard.

(Example of solution)

```
100  WOPEN #30,"SEQ-DATAS"
110  INPUT "DATA=" ; A$
120  IF A$="CLOSE" THEN CLOSE #30 : END
130  IF A$="KILL" THEN KILL #30 : END
140  PRINT #30, A$ : GOTO 110
```

RANDOM FILE

A random file allows records to be written in or read from arbitrary file locations. It is useful when many related records are to be stored and read at random. The advantages and disadvantages of random access files are as follows.

Advantages:
- Any record can be easily located.
- It is easy to add or delete records.

Disadvantages:
- Empty records are generated, reducing the efficiency of memory utilization.

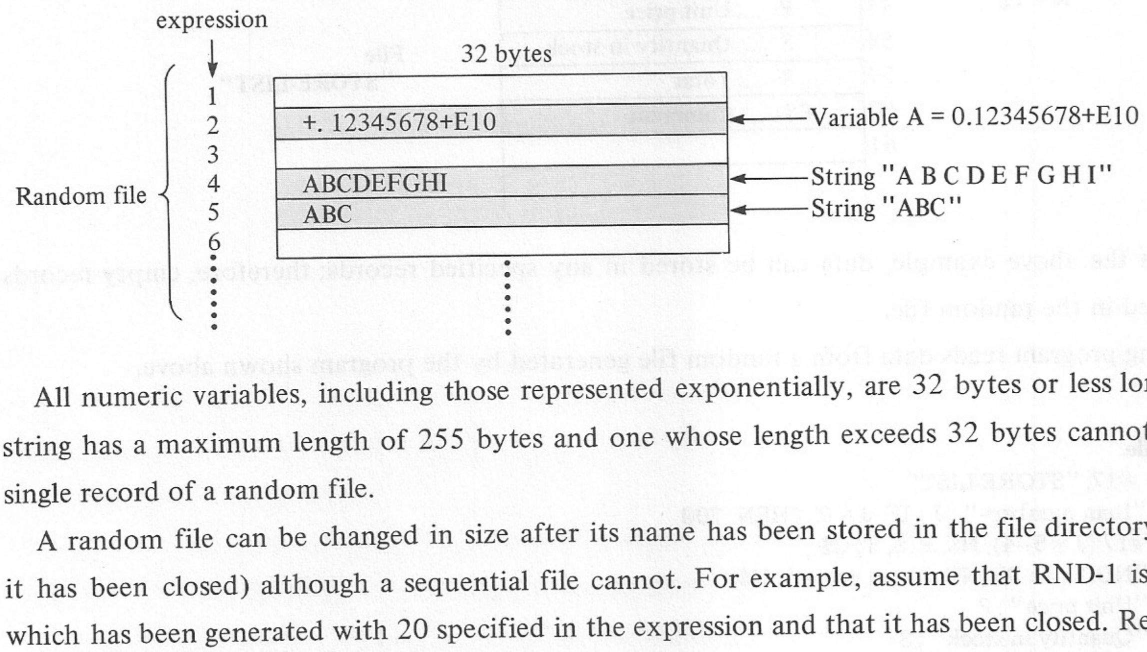
An expression is specified following the logical file number in the PRINT # and INPUT # statements to designate a record in the file as shown below.

PRINT #n (expression), data

INPUT #n (expression), variable (where expression is a numeric value, variable or expression.)

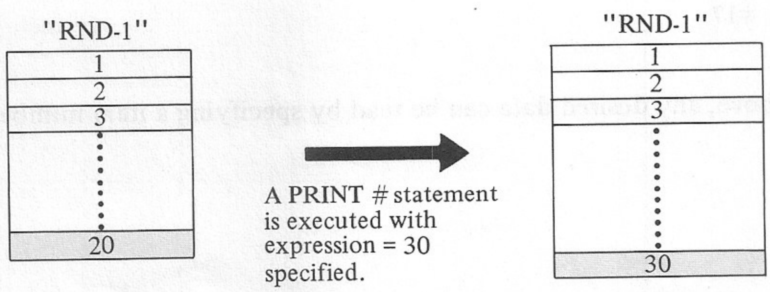
For example, INPUT #7(21), A$ reads record 21 of the random file assigned logical number 7 and assigns it to string variable A$.

A limitation is placed on random files to enable random data access; that is, **the record length is fixed to 32 bytes.**



All numeric variables, including those represented exponentially, are 32 bytes or less long. However, a string has a maximum length of 255 bytes and one whose length exceeds 32 bytes cannot be stored in a single record of a random file.

A random file can be changed in size after its name has been stored in the file directory (that is, after it has been closed) although a sequential file cannot. For example, assume that RND-1 is a random file which has been generated with 20 specified in the expression and that it has been closed. Reopening it and executing a PRINT # statement with 30 specified in the expression automatically increases the file size. See the figure below.



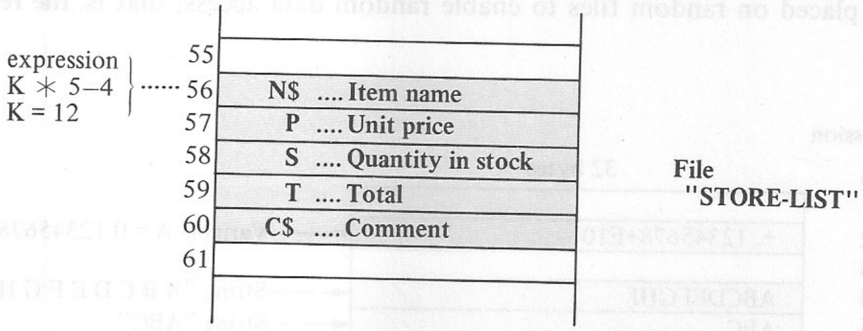
Let's make an inventory list using a random file. The inventory includes 50 lines and each line includes entries for item name, unit price, quantity in stock, total (unit price multiplied by quantity in stock) and comments.

An item number is input first followed by the other entries when storing the record for an inventory line in the file.

Storing inventory data in a file.

```
100 XOPEN #5, "STORE-LIST"
110 INPUT "Item number=" ; K
120 IF K = 0 THEN 300
130 INPUT "Item name=" ; N$
140 INPUT "Unit price=" ; P
150 INPUT "Quantity in stock=" ; S
160 INPUT "Comment=" ; C$
170 T = P * S
180 PRINT #5 (K*5-4), N$, P, S, T, C$
190 GOTO 110
300 CLOSE #5
310 END
```

The random file generated has the structure shown below.



As shown in the above example, data can be stored in any specified records; therefore, empty records can be generated in the random file.

The following program reads data from a random file generated by the program shown above.

Desire data in a file

```
500 XOPEN #17, "STORE-LIST"
510 INPUT, "Item number=" ; J : IF J = 0 THEN 700
520 INPUT #17 (J*5-4), N$, P, S, T, C$
530 PRINT "NO." ; J : PRINT "Item name" ; N$
540 PRINT "Unit price" ; P
550 PRINT "Quantity in stock" ; S
560 PRINT "Total" ; T
570 PRINT "Comment" ; C$
580 GOTO 510
700 CLOSE #17
710 END
```

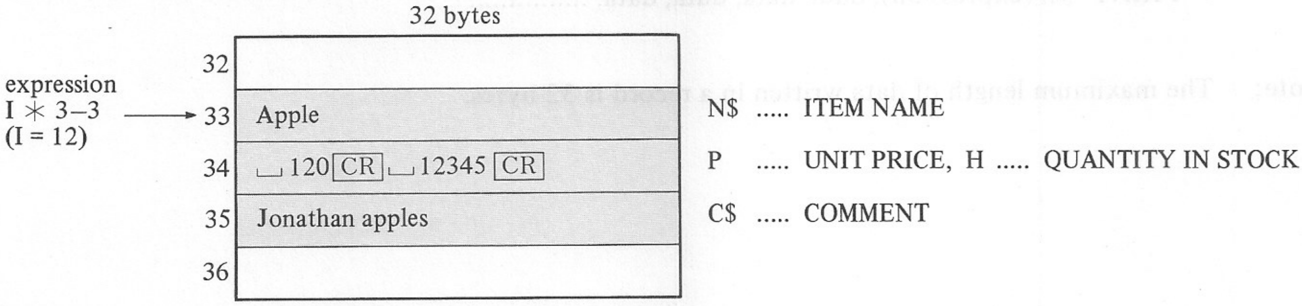
As shown above, any desired data can be read by specifying a item number.

The record length of random files is fixed to 32 bytes as explained previously. Therefore, useless memory space increases if the data length stored in each record is too short. To prevent this, two or more valuse can be stored in a record by the procedures as shown below. If the length of a string variable exceeds 32 bytes, only the first 32 bytes will be registered; whereas if it is less than 32 bytes, the remaining portion will be filled with spaces.

(Write)

```
100 XOPEN #5,"STORE-LIST"
110 INPUT "ITEM NUMBER=";I : IF I = 0 THEN 300
120 INPUT "ITEM NAME=" ; N$
130 INPUT "UNIT PRICE=" ; P
140 INPUT "QUANTITY IN STOCK=" ; H
150 INPUT "COMMENT=" ; C$
160 PRINT #5 (1*3-3), N$ , P ; H, C$
170 GOTO 110
300 CLOSE #5
310 END
```

A part of the random file generated by the above program is as shown below when I is set to 12. The values of P and H which are separated with a semicolon in the PRINT # statement are stored in the same record. In such a case, if the total length of values of P and H (including the carriage return code which is a data separator) exceeds 32 bytes, the "end of record" error results and –94 is set to ERN. The former record contents remain.



A sample program which reads the random file generated in the above manner is shown below. Variables to which the values stored in the same record are assigned are separated with a semicolon in the INPUT # statement. If the value for P is stored but no value for H, 0 is assigned to H. That is, if no corresponding data is sotred, numeric variables are set to 0.

(Read)

```
500 XOPEN #17, "STORE-LIST"
510 INPUT "ITEM NUMBER=" ; J : IF J = 0 THEN 700
520 INPUT #17 (I*3-3) ; N$, P; H, C$
530 PRINT J, N$
540 PRINT "UNIT PRICE=" ; P
550 PRINT "QUANTITY IN STOCK=" ; H
560 PRINT "COMMENT=" ; C$
570 GOTO 510
700 CLOSE #17
710 END
```

Random file control statements are explained below.

XOPEN #n,"filename"

This statement writes the specified data in the record of (opened) logical file #n which is designated number (1 ~ 126) to it. This operation is referred to as cross opening a file.

In the example programs on the previous page, the statements on line 100 and 500 cross-open the random file "STORE-LIST".

XOPEN (Cross open)	Opens a random file for writing data. Opens a random file for reading data.

PRINT #n (expression), data

This statement writes the specified data in the record of (opened) logical file #n which is designated by the expression. Data items must be separated with commas when many data items are specified.

In the sample program, the statement on line 180 writes the values of variables N$, P, S, T and C$ in the 5 records starting with record (K∗5−4). As shown in this example, multiple records can be written in sequence.

```
PRINT #n (expression), data, data, data, data, ..............
```

Note: The maximum length of data written in a record is 32 bytes.

INPUT #n (expression), variable

This statement reads the specified record from the random file which is cross opened with logical number #n assigned and assigns it to the specified variable. The record is specified by the expression.

In the sample program, the statement on line 520 reads the 5 records starting at record (J∗5–4) and assigns their values to variables N$, P, S, T and C$. Multiple records can be read with a single statement as shown in this example.

INPUT #n (expression), variable, variable, variable, ...

Note:

Zero is assigned to a numeric variable and 32 spaces are assigned to a string variable when an empty record is read by an INPUT # statement.

CLOSE #n

Closes the random file assigned logical number #n and clears the logical number definition. The file directory is cataloged when this statement is executed if it was not cataloged previously.

Note:

The KILL # statement issued for a random file has the same function as the CLOSE # statement. However, for physical reasons it is not certain that all records will be stored on the diskette with the KILL # statement; therefore, the CLOSE statement should be used.

Using EOF (#n)

EOF (#n) can be used to detect file end (or out-of-file) for random files.

The following sample program displays all data stored in random file "STORE-LIST" from the beginning of the file to its end.

```
700 XOPEN #20, "STORE-LIST"
710 K=1
720 INPUT #20 (K∗5–4), N$, P, S, T, C$
730 IF EOF (#20) THEN 900
740 PRINT "Item number" ; K
750 PRINT "Item name" ; N$
760 PRINT "Unit price" ; P
770 PRINT "Quantity in stock" ; S
780 PRINT "Total" ; T
790 PRINT "Comment" ; C$
800 K = K + 1 : GOTO 720
900 CLOSE #20
910 PRINT "FILE END HERE" : END
```

EXCEPTION PROCESSING CONTROL

The BASIC program stops execution of a program and outputs and error message when an error occurs during program execution. However, it is not necessary for a program to stop when an error occurs if the cause of the error is known and an appropriate exception processing routine is included in the program to provide a countermeasure.

The exception processing statements are used for this purpose.

ON ERROR GOTO linenumber

This statement declares that control is transferred to the routine indicated by linenumber when an error occurs.

ERN, ERL

ERN and ERL are the special variables to which the error number and the error line number are assigned when an error occurs. See page 34 for the error numbers. When an error occurs during execution of a statement which has no line number, ERL is loaded with the first line number preceding the statement.

RESUME

This statement returns control to the main program after error processing has been completed.

RESUME linenumber	Returns control to the location specified by linenumber.
RESUME 0	Returns control to the beginning of the main program.

OFF ERROR

This statement cancels a preceding ON ERROR statement. That is, when an error occurs after this statement, a standard error message is displayed and program execution is stopped. Control is returned to FDOS.

ON BRKEY GOTO linenumber

This statement declares that control is transferred to the location indicated by linenumber when the [BREAK] key is pressed.

OFF BRKEY

This statement cancels a preceding ON BRKEY statement. That is, control is returned to FDOS when the [BREAK] key is pressed.

ON KEY GOTO linenumber

Declares that control is transferred to the routine starting at the specified line number when a key is pressed. The line number which is executed when the key is pressed is stored in variable ERL and the code for the key pressed is stored in variable KY$.

ON KEY GOSUB linenumber

Declares that control is transferred to the subroutine starting at the specified line number when a key is pressed. The line number which is executed when the key is pressed is stored in variable ERL and the code for the key pressed is stored in variable KY$. After the subroutine is finished, control is returned to the line number following the one stored in ERL.

OFF KEY

This statement cancels a preceding ON KEY statement.

Note:

An error occurring during execution of a CLI or RUN command included in a BASIC program is sometimes not processed by an ON ERROR statement.

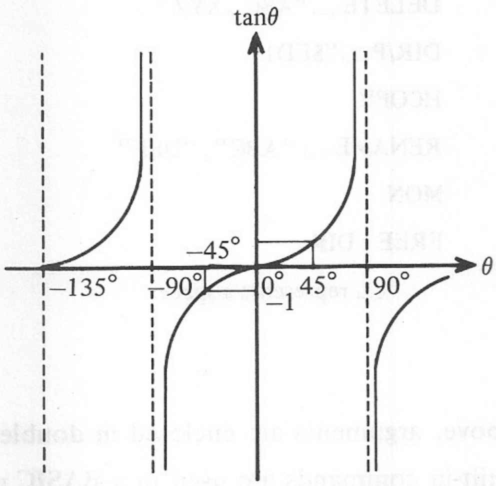
The following sampel program shows use of exception processing statements.

Calculation of tangent

```
100  ON ERROR GOTO 1000
110  FOR TH = 0 TO 180 STEP 30
120  PRINT TH, TAN (π * TH/180)
130  NEXT : END
1000 PRINT "OVERFLOW"
1010 RESUME 130
```

(Result)

```
RUN
0          0
30         0.57735027
60         1.7320508
90         OVERFLOW
120        -1.7320508
150        -0.57735027
180        0
```



In this sample program, "OVERFLOW" is displayed when overflow occurs upon calculation of the tangent on line number 120. Actually, when TH=90, TAN ($\pi/2$) $\rightarrow \infty$ and overflow results. Control is then transferred to the statement on line number 1000, "OVERFLOW" is displayed and the statement on line number 1010 returns control to the main program.

FDOS COMMANDS

FDOS commands can be included in a BASIC program.

—Built-in Commands—

FDOS built-in commands are commands whose processing routines are resident in the memory. Each FDOS built-in command used in a BASIC program is written with a mandatory space between the command and arguments and with the argumetns enclosed in double quotation marks. (Arguments may be string variables.)

Examples:		
	DELETE␣"SAMPLE" (↖ A space is mandatory.)	Deletes the file "SAMPLE"
	A$ = "SAMPLE" : DELETE␣A$	Same as the above.
	DELETE␣"ABC" , "XYZ"	Deletes files "ABC" and "DEF".
	DELETE␣"ABC , XYZ"	Same as the above.
	DIR/P␣"$FD1"	Outputs the directory of drive 1 to the printer.
	HCOPY	Outputs the ontents of the CRT screen to the printer.
	RENAME␣"ABC" , "DEF"	Renames the file.
	MON	Returns control to the monitor.
	FREE : DIR	Executes both FREE and DIR command.

␣ represents a space.

As shown above, arguments are enclosed in double quotation marks or specified with string variables when FDOS built-in commands are used in a BASIC program. Refer to the System Command Manual for details on each FDOS command.

—Transient Commands—

FDOS transient commands are commands whose processing routines are not resident in memory but are loaded from the master or submaster diskette when they are required for execution. Therefore, the master or a submaster diskette must be loaded in the floppy disk drive when FDOS transient commands are used in a BASIC command.

The CLI (Command Line Interpreter) statement is used to load and execute FDOS transient commands in a BASIC program. An operand of the CLI statement consists either of FDOS transient commands and arguments enclosed in double quotation marks or a string variable to which the FDOS transient commands and arguments are assigned in advance. The first operand must be preceded by a space. The operands of the CLI statement may be written in the multistatement form.

Examples:	CLI␣"EDIT"	Calls the FDOS text editor.
	CLI␣"ASM␣ABC"	Assembles source file ABC.ASC to generate relocatable file ABC.RB.
	CLI␣"ASM", "ABC"	Same as the above.
	A$ = "ASM␣ABC" : CLI␣A$	Same as the above.
	CLI␣"2>LINK␣XYZ"	Specifies $FD2 as the default drive and links XYZ.RB to generate object file XYZ.OBJ.
	CLI␣"XFER␣$PTR, XYZ : TYPE␣XYZ"	Any built-in commands and multistatement form can be included in double quotation marks.

An FDOS commands cannot be executed when the usable memory area is too small for it.

Note:

FDOS commands LIMIT, DEBUG and EXEC cannot be executed in a BASIC program.
(CLI␣"LIMIT␣$C000" not allowed)

—Changing the Default Drive—

The default drive is automatically selected when no drive number is specified in a file control statement. The default drive number is displayed to the left of ">" while in the FDOS command wait state. The default drive can be changed in the following manner.

CLI␣"1>"	Changes the default drive to $FD1.
CLI␣"3>"	Changes the default drive to $FD3.
CLI␣STR$(N) + ">"	Changes the default drive number to N.

—Run Statement—

The RUN statement is similar to the SWAP statement in D-BASIC SA-6510. A RUN statement used in program (A) generated with the BASIC compiler loads and runs another specified program (B) which was also generated with the BASIC compiler. In this situation, program control is returned to program (A) when a STOP or END statement is executed in program (B).

The RUN statement is different from the SWAP statement in the following.

1. Variables used in program (A) and program (B) are treated as different variables even if they have identical names.
2. Program A is stored in the memory while program (B) is being executed and program (B) is cleared after it has been completed.

It is convenient to use a pseudo device ($MEM) for transfer of data between program (A) and program (B). $MEM allows a memory area to be treated as an I/O device so that data can be read and written in the memory area.

(Sample program)

Program (A)

```
 :
 :
100  WOPEN  #9, "$MEM"  .............. Opens $MEM for writing data.
110  PRINT  #9, A, B, A$ ............. Writes A, B, A$ in $MEM.
120  CLOSE  #9........................ Closes $MEM.
130  RUN "PROGRAM(B)"................. Loads and executes program (B).
 :
 :
```

Program (B)

```
 10  ROPEN  #9, "$MEM"................ Opens $MEM for reading data.
 20  INPUT  #9, X, Y, N$ ............. Reads data.
 30  CLOSE  #9 ....................... Closes $MEM.
 40  DELETE/N  "$MEM"  ............... Clears $MEM to conserve memory space.
 :
 :
900  STOP ............................ Returns control to program (A).
```

The END statement in program (B) kills all files opened before returning control to program (A). The STOP statement in program (B) returns control program (A) with all files open.

EXTERNAL STATEMENT

The EXTERNAL statement allows a BASIC program to execute external commands and functions whose processing routines are coded with the assembler. (A sample program is shown in the Programming Utility Manual).

In the description below, **the subroutines in bold face type are stored in RELO.LIB on the master diskette**. For details, refer to the Library/Package.

—External Function Definition—

Ex 1) EXTERNAL FNA, FNSUB2

Defines external functions FNA and FNSUB2. A function name must be started with FN and must be no longer than 6 characters.

—Calling External Functions—

Ex 2) A = FNA (X)

The number of parameters of each external function is 1. Character strings cannot be used as parameters.

—External Command Definition—

Ex 3) EXTERNAL PLOT, SEND, RCV

Defines external commands PLOT, SEND and RCV. No command may be longer than 6 characters.

—Calling External Command—

Ex 4) PLOT X, Y: POINT 5, 8

Any numeric constant, numeric variable, string variable, array element, string array element or expression starting with + or − sign. (+A+B is acceptable. but A+B is not.)

—Coding External Functions—

An external function is coded with the assembler. The function name must be declared with the ENT instruction. The parameter is converted into signed 16-bit binary format and loaded into the HL register pair. (If it cannot be converted into 16-bit binary, 32767 or −32768 is loaded into the HL register pair and the carry flag is set.) The RET instruction is used to return control to the BASIC program. The HL content upon return is used as the value of the function. Routine **BEERR** is called when an error occurs.

Ex 5)

```
FNSUB2 : ENT
         LD  (PARAM), HL        ; The HL register pair contains a parameter value.
           :
         JR  C, ERR
           :
         LD  HL, (ANS)          ; Loads the return value into HL.
         RET                    ; Returns control to the BASIC program.
ERR:     CALL BEERR             ; Calls the error routine.
         DEFB 101               ; Error number
         DEFM 'IL PARA'         ; Error message
         DEFB 0DH
         END
```

—Coding External Commands—

Ex 6)

```
PLOT:  ENT                     ; Command name
       DEFB  2                 ; The number of parameters is 2.
       DEFB  0                 ; The first parameter is a real number.
X :    DEFS  2                 ; The area in which the first parameter address is stored.
       DEFB  0                 ; The second parameter is a real number.
Y :    DEFS  2                 ; The area in which the second parameter address is stored.
;                              ; The beginning of the program
       LD    HL, (X)           ; HL ← First parameter address
       CALL  . . INT0          ; HL ← 16-bit binary
       JP    C, ER3            ; Indicates error 3 when the parameter value cannot be converted
                               into 16-bit binary.
       :
       RET                     ; Returns control to the BASIC program.
       END
```

Ex 7) The above program is modified as follows when data is to be transferred from the assembly program to the BASIC program.

```
       LD    HL, .........     ; Loads data to be transferred to the BASIC program into HL.
       LD    DE, (X)           ; Loads the first parameter address into DE.
       CALL  . . FLT0          ; Calls the routine which converts data into real numbers.
```

Ex 8) String transfer (from the BASIC program to the assembly program)

```
SENT :  ENT                    ; Command name
        DEFB  1                ; The number of parameter is 1.
        DEFB  80H              ; The parameter is a string.
DATA:   DEFS  2                ; The area in which the string address is stored.
;                              ; The beginning of the program
        LD    HL, (DATA)       ; HL ← string address (type 1)
        CALL  . MOVE'          ; Calls the routine which converts strings from type 1 to type 2.
                               After execution of the routine, DE contains the first address of
                               the type 2 string.
        :
        RET
```

```
Type 1 string                          Type 2 string
  DEFB length     CALL  . MOVE'        DEFM ' ........ '
  DEFM ' ........ '  ────────→         DEFB 0DH
```

Ex 9) String transfer (from the assembly program to the BASIC program)

```
RCV:    ENT                       ; Command name
        DEFB    1                 ; The number of parameters is 1.
        DEFB    80H               ; The parameter is a string.
DATA:   DEFS    2                 ; The area in which the string address is stored.
;                                 ; The beginning of the program



        LD      HL, ADRS          ; HL ← the starting address of the string (type 1) to be transferred.
        LD      DE, (DATA)        ; DE ← the starting address of the area in which the result is stored.
        CALL    . LOAD$           ; Calls routine .LOAD$. This subroutine transfers string data from
        RET                       ; an assembly subroutine to a BASIC program. The contents of
                                  ; registers BC', DE', HL', IX and IY are saved.
```

—Linking External Functions and Commands with BASIC Programs—

A BASIC program generated by the BASIC compiler and external routines generated by the assembler are relocatable files. These are linked with RELO.LIB to generate an object file.

Ex 10)

```
2 > LINK PROG1, PLOT
         ↑      └ Generated by the assembler
Generated by the BASIC compiler

2 > RUN PROG1
```

SAMPLE PROGRAM

The sample program "8-QUEEN" is stored in the master diskette. Compile, link and execute the program according to following procedures.

BASIC 8-Queen

LINK 8-Queen

RUN 8-Queen

```
     BASIC compiler SA-7701  <8-QUEEN>  page 1                12.28.81

000F        TI$="000000": PRINT "▣"
0040        DIM LOCATE(8),HFLAG(8),D1FLAG(15),D2FLAG(15)
007F        CURSOR 12,0: PRINT "▀▀▀▀▀▀▀▀▀▀▀▀▀▀"
00B4        CURSOR 12,1: PRINT "  8-QUEEN   "
00DD        CURSOR 12,2: PRINT "▄▄▄▄▄▄▄▄▄▄▄▄▄▄"
0106        CURSOR 9,4: PRINT "┌─┬─┬─┬─┬─┬─┬─┬─┐"
013C        FOR I=0 TO 6
0168          PRINT "          │ │ │ │ │ │ │ │ │"
0190          PRINT "          ├─┼─┼─┼─┼─┼─┼─┼─┤"
01B8        NEXT I
01C3        PRINT "          │ │ │ │ │ │ │ │ │"
01EB        PRINT "          └─┴─┴─┴─┴─┴─┴─┴─┘"
0213        PRINT "▣▣▣▣▣(     )"
022C        Y=1: LOCATE(1)=1: X=1
025C   100  HFLAG(X)=1: D1FLAG(Y+X-1)=1: D2FLAG(Y-X+8)=1: Y=Y+1: X=1
02FC   200  IF HFLAG(X)<>0 THEN 300
0339        IF D1FLAG(Y+X-1)<>0 THEN 300
0374        IF D2FLAG(Y-X+8)<>0 THEN 300
03AF        LOCATE(Y)=X: IF Y<8 THEN 100
03E0        GOSUB 400
03F2   300  IF X<8 THEN X=X+1: GOTO 200
041F        Y=Y-1: IF Y=0 THEN CURSOR 0,20: PRINT TI$: PRINT "FINISH": END
0485        X=LOCATE(Y): HFLAG(X)=0: D1FLAG(Y+X-1)=0: D2FLAG(Y-X+8)=0
0505        GOTO 300

050B    400 REM ↑1
0511        NUMBER=NUMBER+1: CURSOR 1,4: PRINT NUMBER;
053C        CURSOR 0,4
0545        FOR I=1 TO 8
0560          PRINT "▣          │ │ │ │ │ │ │ │ │"
0589        NEXT I
058F        FOR I=1 TO 8: CURSOR LOCATE(I)*2+8,I*2+3: PRINT "♦";
0609         CURSOR 26,CSRV: PRINT LOCATE(I)
0638        NEXT
0640        PRINT TI$
064F        RETURN
** Compiler found no errors.
```

BASIC COMPILER STATEMENT LIST

The format and function of every statement is subject to change when new versions of the BASIC compiler are issued. The following lists are based on BASIC Compiler SA-7701.

—FDOS commands (For details, refer to the System Command Manual.)—

BUILT-IN COMMAND	100 DIR 110 DIR/P 120 DIR "$FD3" 130 DELETE "SEQ-DATA-1" 140 RENAME "NAME1", A$ 150 RENAME "NAME2", "NAME3" 160 FREE 170 RUN A$+B$+"HEAD—ON"	Any FDOS built-in command can be included in a BASIC program as shown at left. In this case, arguments must be enclosed in double quotation marks or must be represented as string variables or expressions.
CLI	210 CLI "2 > XFER $PTR/PE, ABC" 220 CLI "EDIT" 230 CLI "ASM ABC" 240 CLI "ASM", A$	Either FDOS built-in and transient commands can be executed during execution of a BASIC program by using the command line interpreter (CLI). However, the LIMIT, DEBUG and EXEC commands cannot be executed.

—Sequential file control statements—

WOPEN #	WOPEN #3, "SFD2 ; SEQ-DATA 1"	Defines the name of a sequential file to be generated as "SEQ—DATA 1" and opens it with logical number 3 assigned on the diskette in drive 2.
PRINT #	PRINT #3, A, A$	Writes the contents of variable A and string variable A$ in succession in the sequential file assigned logical number #3. The file directory is not cataloged until a CLOSE # statement is executed.
OUT #	OUT #3, A	Writes the data in variable A byte-by-byte to the file opened with a WOPEN # statement with logical number #3 assigned.
CLOSE #	CLOSE #3 (corresponding to WOPEN #)	Closes the sequential file assigned logical number #3 which was previously opened with the WOPEN #3 statement. The directory of the file generated by the WOPEN # statement is cataloged and the logical file number assignment is cleared when the file is closed.
KILL #	KILL #3	Kills the sequential file assigned logical number #3 with the WOPEN # statement. The file directory is not cataloged. The logical file number assignement is cleared.

ROPEN #	ROPEN #4, "$FD2 ; SEQ-DATA 1"	Read-opens sequential file "SEQ-DATA 1" on the diskette in drive 2 with logical number #4 assigned.
INPUT #	INPUT #4, A(1), B$	Reads data from the beginning of the sequential file assigned logical number #4 and assigns data to array A(1) and string variable B$ in that order.
INP #	INP #4, A	Reads data byte-by-byte from the sequential file assigned logical number #4 and assigns data to variable A.
CLOSE #	CLOSE #4 (corresponding to ROPEN #)	Closes the sequential file assigned logical number #4 and clears the logical file number assignment.

—Random file control statements—

XOPEN #	XOPEN #5, "$FD3 ; DATA-R1"	This XOPEN statement opens random file "DATA-R1" on the diskette in drive 3 with logical number #5 assigned for reading or writing data; if the file does not exist, the file name "DATA R1" is defined with the logical number #5 assigned, then the file is opened.
PRINT #( )	PRINT #5(11), R(11)	Writes the contents of element R(11) of one-dimensional array R( ) in record 11 of the random file assigned logical number #5.
	PRINT #5(20), AR$, AS$	Writes the contents of string variables AR$ and AS$ in records 20 and 21, respectively, of the random file assigned logical number #5. If the length of the contents of variable exceeds 32 bytes, the excess is ignored.
INPUT #( )	INPUT #5(21), R$	Reads the contents of record 21 of the random file assigned logical number #5 into string variable R$.
	INPUT #5(11), A(11), A$(12)	Reads the contents of records 11 and 12 of the random file assigned logical number #5 into array element A(11) and string array element A$(12).
CLOSE #	CLOSE #5	Closes the random file assigned logical number #5 which was opened by the corresponding XOPEN # statement and clears the logical file number assignment.
CLOSE	CLOSE	Closes all open files and stops motors of floppy disk drives.
KILL #	KILL	Kills all open files and stops motors of floppy disk drives. This statement does not access any floppy disk drive.
IF EOF (#)	IF EOF (#5) THEN 700	Transfers program control to the routine starting to line number 700 if the file end is detected during execution of an INPUT # statement against a sequential or random file.

—Exception processing statements—

ON ERROR GOTO	ON ERROR GOTO 1000	Declares that control is transferred to line number 1000 when an error occurs.
ERN	IF ERN = 44 THEN 1050	Transfers control to line number 1050 when the error number is 44.
ERL	IF ERL = 350 THEN 1090	Transfers control to line number 1090 when an error occurs on line number 350.

ERM$	IF (ERN = 53) ∗ (ERL = 700) THEN PRINT ERM$: END	Displays the error message and terminates program execution when an error of error number 53 occurs on line number 700. The error number, the line number, and the error message are stored in variable ERN, ERL, and ERM$, respectively.
RESUME	RESUME 100	Returns control to the line number 100 when error processing is completed.
	RESUME 0	Transfers control to the beginning of the main program.
OFF ERROR		Cancels a preceding ON ERROR statement.
ON BRKEY GOTO	ON BRKEY GOTO 2000	Declares that control is transferred to line number 2000 when the [BREAK] key is pressed.
OFF BRKEY	OFF BRKEY	Cancels a preceding ON BRKEY statement.
ON KEY GOTO	ON KEY GOTO 3000	Declares that control is transferred to line number 3000 when key is pressed. The line number which is executed when the key is pressed is stored in variable ERL and the code for the key pressed is stored in variable KY$.
ON KEY GOSUB	ON KEY GOSUB 4000	Control is transferred to the subroutine starting at line number 4000 when a key is pressed. The line number which is executed when the key is pressed is stored in ERL and the key code is stored in KY$.
OFF KEY	OFF KEY	Cancels a preceding ON KEY statement.
KY$	IF KY$ = "5" THEN 4000	Transfers control to line number 4000 when the [5] key is pressed after an ON KEY trapping.

—Assignment statement—

LET	<LET> A = X + 3	Assigns the sum of the value of variable X and 3 to variable A. LET may be omitted.

—Input and Output statements—

PRINT	10 PRINT A	Displays the value of variable A on the CRT screen.
	20 ? A$	Displays the contents of variable A$ on the CRT screen.
	100 PRINT A ; A$, B ; B$	Numeric variables and string variables may be used mixedly in a PRINT statement. The value of a variable following a semicolon is displayed closed to the previous value displayed. The value of a variable following a comma is displayed at the next tab set position. (Tabs are set every 10 characters).
	110 PRINT "COST=" ; CS	Displays the string enclosed in double quotation marks.
	120 PRINT	Makes a line feed.
INPUT	10 INPUT A	Obtains numeric data for variable A from the keyboard.
	20 INPUT A$	Obtains string data for variable A$ from the keyboard.
	30 INPUT "VALUE?" ; D	Displays string "VALUE?" before obtaining data for D 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 A from the keyboard. When no key is pressed, zero is assigned to A.
	20 GET K$	Obtains a character for variable K$ from the keyboard. When no key is pressed, a null is assigned to K$.
READ~DATA		Assigns constants specified in the DATA statement into the corresponding variables specified in the READ statement. The corresponding constant and variable must be of the same data type.
	10 READ A, B, C 1010 DATA 25, –0.5, 500	Assigns 25, –0.5 and 500 to variables A, B and C, respectively.
	10 READ H$, H, S$, S 30 DATA HEART, 3 35 DATA SPADE, 11	Assigns string "HEART" to string varialbe H$ and assigns 3 to numeric variable H and so on.
RESTORE	500 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.
RESTORE	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 assigns 3, 6 and 9 into variable A, B and C, respectively. Because of the RESTORE statement, the READ statement on line number 30 assigns 3 and 6 again into D and E, respectively.

—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 30 PRINT B2 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.
	60 NEXT B, A 70 NEXT A, B	The statements on lines 40 and 50 in the above example can be combined into one shown on line 60. Line 70 results in an error.

—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 a 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 statement is executed.
	50 IF B < 3 THEN B = B+3	Assigns B+3 to variable B when the value of B is less than 3; otherwise the next statement is executed.
IF ~ GOTO	100 IF A >=B THEN 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 statement is executed.
IF ~ GOSUB	30 IF A=B * 2 GOSUB 90	Jumps to the subroutine starting on line number 90 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. Handling of the multistatement after IF statement is the same as in MZ-80K/A/B BASIC interpreter, but different from that in MZ-80K compiler.)
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	Calls the subroutine starting on line number 700 when the value of variable A is 1 and calls the subroutine starting on line number 800 when it is 2. When the value of A is 0 or more than 3, the next statement is executed.

—Definition statements—

DIM		When an array is used, the maximum number of array elements must be declared with a DIM statement. The number of elements is limited by the available memory space.
	10 DIM A(20)	Declares that 21 array elements, A(0) through A(20), are used for one-dimensional numeric array A( ).
	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( ).
	30 DIM C1$(10)	Declares that 11 array elements, C1$(0) through C1$(10), are used for one-dimensional string array C1$( ).
	40 DIM K$(7, 5)	Declares that 48 elements, K$(0, 0) through K$(7, 5), are used for two-dimensional string array K$( ).

DEF FN	100 DEF FNA(X)= X↑2−X 110 DEF FNB(X)=LOG (X) +1 120 DEF FNZ(Y)=LN(Y) 130 DEF FNC(X, Y)=SQR (X↑2+Y↑2) 140 DEF FNA$(X$)=M1D$ (X$, 3, 3)	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 LN(Y). The statement on line number 130 defines FNC(Y, Y) as $\sqrt{X^2+Y^2}$ and the statement on line number 140 defines FNA$(X$) as MID$ (X$, 3, 3). The number of parameters is arbitrary.

—Comment statement and control statements—

REM	200 REM JOB-1	Comment statement (not executed)
	300 REM ↑3	Performs three line feeds on the list during compiling.
	400 REM ↑	Performs a form feed on the list during compiling.
STOP	850 STOP	The same as END.
END	1999 END	Stops program execution and kills all open files. (See page 18 for exceptions).
CURSOR	50 CURSOR 25, 15 60 PRINT "ABC"	The CURSOR statement positions the cursor to any position on the screen. The X coordinate ranges from 0 to 39 (from left to right) and the Y coordinate from 0 to 24 (from top to bottom). The statements shown at left display the string "ABC" at the location starting at the position 26th characters from the left and 16th characters from the top.
CSRH		System variable which contains the X coordinate of the current cursor position.
CSRV		System variable which contains the Y coordinate of the current cursor position.
TI$	100 TI$ = "102030"	Sets the built-in clock to 10 : 20 : 30. The time data is a 6-digit string enclosed with double quotation marks.
WAIT	10 WAIT 20	Suspends program execution for 20 ms.

—Music control statements—

MUSIC **TEMPO**		The MUSIC statement generates a melody from the speaker according to the melody string enclosed in quotation marks at the tempo specified by the TEMPO statement.
	300 TEMPO 7 310 MUSIC "DE # FGA"	The TEMPO statement on line number 300 specified tempo 7 (fastest speed). 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, tempo 4 is set.
	300 M1$= "C3EG+C" 310 M2$= "BGD−G" 320 M3$= "C8R5" 330 MUSIC M1$, M2$, M3$	In this example, the melody is divided into 3 parts and assigned to 3 string variables. The melody shown below is played through the speaker at tempo 4.

—Graphic control statements—

SET	100 SET X, Y 200 RESET X, Y	This statement sets or resets a dot in the CRT screen. The dot position is specified with X coordinates (0 ~ 79 from left to right) and Y coordinates (0 ~ 49 from top to bottom).
	300 SET 40, 25	Set a dot in the center of the screen.
RESET	310 RESET 0,0	Reset a dot in the left-uper of screen.

—Machine language program control statements—

POKE	120 POKE 49450, 175	Stores 175 (decimal) in decimal address 49450.
	130 POKE AD, DA	Stores the value of variable DA (0 ~ 255) in the memory location specified by variable AD.
	140 POKE $D000, 83, 68, 70, 65	Stores data in succession into the memory, starting at address D000H (hexadecimal).
	150 POKE $C000, A$	Stores the contents of A$ in the memory, starting at address C000H (hexadecimal).
PEEK	150 A=PEEK(49450)	Converts the content of decimal address 49450 into decimal representation and assigns it to variable A.
	160 B=PEEK(C)	Converts the content of the decimal address specified in variable C into decimal and assigns it to variable B.
USR	500 USR(49152)	Transfers control to decimal address 49152. This statement has the same function as the CALL command. Accordingly, when the RET command is encountered during execution of the machine language program, control is returned to the BASIC program.
	550 USR(AD)	Transfers control to the memory location specified by variable AD.
	570 USR($C000)	Transfers control to memory location C000H.
EXTERNAL	10 EXTERNAL PLOT, FNPX	Defines external command PLOT and external function FNPX. (The routines for PLOT and FNPX must be created with the assembler.)
	20 PLOT X, Y 30 A=FNPX(10)	Example of use of the external statement and function defined with the EXTERNAL statement.

—Printer control statements—

PRINT/P		Performs the same operation as the PRINT statement on the optional printer. If no printer is connected, execution of this statement results in an error.
	10 PRINT/P A, A$	Outputs the contents of numeric variable A, then the contents of string variable A$ to the printer.
	20 PRINT/P CHR$(5)	Performs a form feed on the printer. (CHR$(5) is a printer control code).
COPY/P	10 COPY/P 1	Makes a copy of the contents of the display screen on the printer.
PAGE/P	100 PAGE/P 20	Sets the number of lines per page on the printer form to 20.

—I/O statements—

INP @		Reads data from the specified I/O port.
	10 INP @12, A 20 PRINT A	The statement on line 10 reads data from I/O port 12 (decimal) into variable A.
OUT @		Outputs data to the specified I/O port.
	39 B=A∗2 40 OUT @13, B	The statement on line 40 outputs the contents of variable B to I/O port 13.

—Arithmetic functions—

ABS	100 A = ABS (X)	Assigns the absolute value of variable X to variable A. X may be either a constant or an expression. Ex) ABS (−3) = 3 ABS (12) = 12
INT	100 A = INT (X)	Assigns the greatest integer which is less than X to 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)	Assigns one of the following values to 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)	Assigns the square root of variable X to variable A. X may be either a numeric constant or an expression: however, it must be greater than or equal to 0.
SIN	100 A = SIN (X)	Assigns the sine of variable X in radians to variable A, X may be either a numeric constant or an expression. The relationship between degrees and radians is as follows. 1 degree = $\frac{\pi}{180}$ radians
	110 A = SIN (30∗π/180)	Therefore, when assigning the sine of 30° to A, the statement is written as shown on line number 110 at left.
COS	200 A = COS (X)	Assigns the cosine of variable X in radians to variable A. X may be either a numeric constant or an expression. The same relationship as shown in the explanation of the SIN function is used to convert degrees into radians. The statement shown on line number 210 assigns the cosine of 200° to variable A.
	210 A = COS (200∗π/180)	
TAN	300 A = TAN (X)	Assigns the tangent of variable X in radians to variable A. X may be either a numeric constant or an expression. The statement on line number 310 is used to assign the tangent of numeric variable Y in degrees to variable A.
	310 A = TAN (Y∗π/180)	
ATN	400 A = ATN (X)	Assigns the arctangent of variable X to variable A in radians. X may be either a numeric constant or an expression. only the result between −π/2 and π/2 is obtained. The statement on line number 410 is used to assign the arctangent in degrees.
	410 A = 180/π∗ATN (X)	

EXP	100 A = EXP (X)	Assigns the value of exponential function e^X to variable A. X may be either a numeric constant or an expression.
LOG	100 A = LOG (X)	Assigns the value of the common logarithm of variable X to variable A. X may be either a numeric constant or an expression; however, it must be positive.
LN	100 A = LN (X)	Assigns the natural logarithm of variable X to variable A. X may be either a numeric constant or an expression; however, it must be positive.
	100 A = LOG(X)/LOG(Y) 120 A = LN(X)/LN(Y)	To obtain the logarithm of X with the base Y, the statement on line number 110 or line number 120 is used.
RND		This function generates random numbers which take any value between 0.00000001 and 0.99999999, and works in two manners depending upon the value of the integer in parentheses.
	100 A = RND (1) 110 B = RND (10)	When the value of the integer in parentheses is positive, 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.
	200 A = RND (0) 210 B = RND (–3)	When the value of the integer in parentheses is less than or equal to 0, the function gives the initial value of the random number group generated. Therefore, statements on line numbers 200 and 210 both give the same value to variables A and B.

—String control functions—

LEFT $	10 A$ = LEFT$ (X$, N)	Assigns the first N characters of string variable X$ to string variable A$. N may be either a constant, a variable or an expression.
MID $	20 B$ = MID$ (X$, M, N)	Assigns the N characters following the Mth character from the beginning of string variable X$ to string variable B$.
RIGHT $	30 C$ = RIGHT $ (X$, N)	Assigns the last N characters of string variable X$ to string variable C$.
SPACE $	40 D$ = SPACE $(N)	Assigns N spaces to string variable D$.
STRING $	50 E$ = STRING $ ("*", 10)	Assigns 10 continuous asterisks to string variable E$.
CHR $	60 F$ = CHR $(A)	Assigns the character corresponding to the ASCII code in numeric variable A to string variable F$. A may be either a constant, a variable or an expression.
ASC	70 A = ASC (X$)	Assigns the ASCII code (in decimal) corresponding to the first character of string variable X$ to numeric variable A.
STR$	80 N$ = STR$ (I)	Converts the numeric value of numeric variable I into a string of numerals and assigns it to string variable N$.

VAL	90 I = VAL (N$)	Converts string of numerals contained in string variable N$ in to the numeric data as is and assigns it to numeric variable I.
LEN	100 LX = LEN (X$)	Assigns the length (number of characters) of string variable X$ to numeric variable LX.
	110 LS = LEN (X$+Y$)	Assigns the sum of lengths of string variables X$ and Y$ to numeric variable LS.

—Tabulation functions—

TAB	10 PRINT TAB(X) ; A	Displays the value of variable A at the (X+1)th character position from the left.
	20 PRINT TAB(5) ; A$	Displays a character string in string variable A$ starting at the 6th character position from the left.

—Arithmetic operators—

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

=	10 A = X+3	Assigns X+3 to variable A.
	20 B = π	Assigns π (3.1415927) to variable B.
❶ ↑	10 A = X↑Y (power)	Assigns X^Y to variable A. (If X is negative and Y is not an integer, an error results.)
❷ *	10 A = X*Y (multiplication)	Multiplies X by Y and assigns the result to variable A.
❷ /	10 A = X/Y (division)	Divides X by Y and assigns the result to variable A.
❸ +	10 A = X+Y (addition)	Adds X and Y and assigns the result to variable A.
❸ –	10 A = X–Y (subtraction)	Subtracts Y from X and assigns the result to variable A.
❸ –	10 A = –B (negative sign)	Note that "–" in –B is the negative sign and "–" in 0–B represents subtraction.

—Logical operators—

❹ =	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.
❹ <> or ><	10 IF A<>X THEN	If the value of variable A is not equal to X, the statement following THEN is executed.
❹ >= 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.

❹<= 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.
❷*	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.
❸+	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 THEN is executed.

—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. The 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 correponding 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-80"	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 A=$BFFF	Indicates that data (2 or 4 digits) following a dollar sign is represented in hexadecimal notation.
π	550 S=SIN (X*π/180)	π represents 3.1415927 (ratio of the circumference of a circle to its diameter).

ERROR MESSAGES

—Error Messages Issued During Compiling—

Error number	Message	Meaning
1	syntax	Syntax error
2	too big number	There is a numeric value which is too large
3	il constant	Illegal constant value
4	different type	Data type mismatch
6	too many variables	There are too many variables.
8	too long statement	A BASIC text line is too long.
15	undefined function	An undefined function is used.
16	undefined line-number	An undefine line number is specified.
22	double defined function	A function is defined two or more times.
30	il expression	Illegal expression format
31	mismatch " ("and ") "	Opening and closing parentheses do not correspond.
32	reserved word	A reserved word is used for another purpose.
33	il line-number	Illegal line number
34	too many " ("	Too many levels of parentheses are used.
35	il function	Illegal function name
36	il array	Illegal array
99	table overflow	A program is too long and it cannot be compiled.

Note: il is an abbreviation for illegal.

—Error Messages Issued During Program Execution (BASIC Level)—

Error number	Message	Meaning
1	syntax	Syntax error
2	overflow	Operational result overflow
3	il data	Illegal data
4	data mismatch	Data type mismatch
5	string too long	String length exceeds 255 characters.
6	memory over	Insufficient memory
13	NEXT, no FOR	NEXT is used without a correspoinding FOR.
14	RETURN, no GOSUB	RETURN is used without a corresponding GOSUB.
21	RESUME, no ERROR	RESUME is used without a corresponding error processing statement.
24	READ, no DATA	READ is used without a corresponding DATA.
37	Break	BREAK was pressed.
38	out of index	Illegal value was assigned to an element of an array defined with a DIM statement.
39	undefined array	An undefined array was used.
64	il lu#	Illegal logical number

—Error Messages Issued During Program Execution (FDOS Level)—

Errors occurring during execution of an FDOS subroutine can be detected by using the ON ERROR statement. (Sometimes errors are not detected in the case of RUN and CLI.) When errors are detected, an FDOS error number is stored in ERN following a minus (–) sign. For example, –50 is stored in ERN when no file is found. For FDOS error numbers, refer to the System Error Messages in the System Command Manual.

COMPARISON WITH D-BASIC SA-6510

Item	D-BASIC SA-6510	BASIC compiler
RESUME	RESUME 0 RESUME linenumber RESUME RESUME NEXT	RESUME 0 RESUME linenumber
RESTORE	RESTORE RESTORE linenumber	RESTORE (The line number cannot be specified.)
Logical open of USR function	ROPEN #n, USR (. . . .) WOPEN #n, USR (. . . .)	Impossible (ASSIGN of FDOS can be used in its place.)
FDn@v	ROPEN #n, FDn@v, "filename" DELETE FDn@v, "filename"	ROPEN #n, "$FDn ; filename" DELETE "$FDn ; filename"
LOAD	LOAD "filename" LOAD T	CLI "LOAD filename" } Can be used for CLI "LOAD $CMT" } object files only.
LIMIT	LIMIT $B000 LIMIT MAX	Impossible The memory area must be limited with FDOS before running the BASIC program.
LOCK UNLOCK	LOCK "filename" UNLOCK "filename"	CHATR "sign, filename, W" CHATR "sign, filename, 0"
SWAP	SWAP "filename"	RUN "filename" can be used in its place. See page 16.
Commands	SAVE, LIST, CONT, VERIFY, AUTO, NEW, SIZE, CHAIN, CLR, SWAP, LOCK, UNLOCK, WOPEN/T, ROPEN/T, PRINT/T, INPUT/T, CLOSE/T	Commands at left cannot be used, but other commands are substituted for some of these commands.
FOR-NEXT	Nesting is possible to a maximum of 16 levels.	The number of levels is limited by usable memory capacity.
GOSUB	Nesting is possible to a maximum of 16 levels.	The number of levels is limited by usable memory capacity.
DEF FN	Nesting is possible to a maximum of 6 levels. The number of parameters is 1.	Nesting is possible to a maximum of 6 levels. The number of parameters is not limited. The DEF statement must be executed before functions are used.
DIM	The number of dimensions is1 or 2. The number of elements is 0 ~ 255. A (. .) and A (. . , . . .) can be identified.	The number of elements is limited only by the usable memory capacity. A (. .) and A (. . , . . .) cannot be identified, so different names must be used. When a name is declared twice or more by DIM, only the first declaration is valid.
POKE	POKE address, date	POKE address, data 1, data 2 ....
Expression	2 ↑ 3 is treated as 2 ↑ (–3). –2 ↑ 3 is treated as –(2 ↑ 3).	2 ↑ –3 results in an error. –2 ↑ 3 is treated as –(2 ↑ 3).
Variable name	1 or 2 alphanumerics, starting with letter.	The length of variables is not limited.
Line number	1 ~ 65535 Line numbers are mandatory.	1 ~ 65535 Line numbers are not always necessary.
Hexadecimal constant	Can be used for certain statements such as POKE and USR.	Can be used anywhere.
Logical number	Constant from 1 to 127.	Constant from 1 to 126 are variable.
File name	16 characters or less.	16 characters or less. Only alphanumerics and specific symbols can be used. For details, refer to paragraph 4.2.2 of the System Command Manual. Device names $KB, $PTP, etc. can be used instead of file names.
Error number	Somewhat different from BASIC compiler	Somewhat different from D-BASIC.
PAGE/P n	n may be either a constant, a variable or an expression.	n must be a constant.