EXTERNAL KEYBOARD
FOR BAR CODE PRINTER

Model:

KB-80

BASIC SPECIFICATION

First Edition: February 4, 1999
Second Edition: July 16, 1999

TOSHIBA TEC CORPORATION

TABLE OF CONTENTS

CONSTANTS AND VARIABLES

CONSTANTS

Character constants:

A string which is 255 digits or less including alphanumeric, Kana, symbol, and Kanji, enclosed in double quotation marks (”). If a double quotation mark (”) is used in the character string, the CHR$ function should be used.

Numeric constants:

Integer type:	Decimal format	:	-32768 to +32767
	Hexadecimal format	:	&H0000 to &HFFFF
Real type:	Single-precision fixed-point format		(Entry digits: 7, Effective digits: 6)
	Single-precision floating-point format		(Entry digits: 7, Effective digits: 6)

VARIABLES

A variable name should be a max. of 10 characters, including alphanumerics and periods, and must start with a letter. The max. number of variables is 256.

Integer type: A “%” should be attached to the end of the variable name.

Single-precision real type: A “!” should be attached to the end of the variable name.
The assignment value is in the same range as a numeric constant.

Character type: A ‘$’ should be attached to the end of the variable name.
The assignment value is in the same range as a character constant.

The max. 16 dimensions and 32767 subscripts for an array variable are allowable. However, a subscript starts from 0, and the practical number of elements is the number of subscripts + 1. 32 bytes are used per one array. An array exceeding 32 bytes cannot be used.

OPERATORS

Arithmetic operators:

Operator	Operation
^	Exponential operation
-	Sign
*, /	Multiplication, Real division
¥	Integer division (The quotient is output.)
MOD	Integer division (The remainder is output.)
+, -	Addition, Subtraction

The operations enclosed in parentheses are processed first.

Relational operators: Compares between two values. The result is true or false.

Operator	Operation
=	Equal to
<>, ><	Not equal to
<	Less than
>	Greater than
<=, =<	Less than or equal to
>=, =>	Greater than or equal to

Logic operators: Checks more conditions, performs bit handling.

Operator	Operation
NOT	Negation
AND	Logic product
OR	Logic OR
XOR	Exclusive-OR

LISTS OF COMMANDS, STATEMENTS, AND FUNCTIONS

Declarations and definitions

Statement		
CLEAR	Format	CLEAR
	Function	Initializes a variable.
DIM	Format	DIM <variable name> (<subscript> [, <subscript>] …) [, <variable name> (<subscript> [, <subscript>] …)] …
	Function	Specifies the max. number of array elements, and allocates the memory area in the memory.
ERASE	Format	ERASE <array name> [, <array name>] …
	Function	Erases the specified array from the program.
REM ’ (Apostrophe)	Format	REM [<comment>]
	Function	Enters a comment into the program.
STOP	Format	STOP
	Function	Terminates the execution of the program, and returns to the command level state.
END	Format	END
	Function	Terminates the execution of the program, closes all opened files, and returns to the command level state.

General instructions

Statement		
DATA	Format	DATA <constant> [, <constant>] …
	Function	Sets a numeric value or a character constant read by the READ statement.
READ	Format	READ <constant> [, <constant>] …
	Function	Reads a value defined by a DATA statement and assigns it to a variable.
RESTORE	Format	RESTORE
	Function	Starts reading from the first one of the DATA statements to be read by the READ statement.
GOTO	Format	GOTO <line number>
	Function	Moves the execution to the specified line without any conditions.
GOSUB	Format	GOSUB <line number>
	Function	Calls the subroutine program.
ON GOTO /GOSUB	Format	ON <expression> GOTO <line number> [, <line number>] … ON <expression> GOSUB <line number> [, <line number>] …
	Function	Branches the execution to the specified line number according to the value for <expression>.
RETURN	Format	RETURN
	Function	Declares the end of a subroutine, and returns the execution to the location where the subroutine was called up.

IF ~ THEN ~ ELSE	Format	IF <expression> THEN | <statement> | <line number> | [ELSE | <statement> | <line number> |] IF <expression> GOTO <line number> [ELSE | <statement> | <line number> |]
	Function	Judges the condition, and changes the flow of the program.
FOR ~ NEXT	Format	FOR <variable> = <initial value> TO <terminal value> [STEP <increment>] NEXT [<variable> [, <variable>] …]
	Function	Executes the statements included from the FOR statement to the NEXT statement repeatedly while the given conditions are satisfied.
WHILE ~ WEND	Format	WHILE <expression> WEND
	Function	Repeats the statements included between the WHILE statement and the WEND statement for as long as the given conditions are satisfied.
CHAIN	Format	CHAIN <file designation> [, ALL]
	Function	Executes another program.
COMMON	Format	COMMON <variable name> [, <variable name>] …
	Function	Declares the variable is to be passed from the original program to the program called by the CHAIN statement.

Numeric process

Statement		
ABS	Format	ABS (<numeric expression>)
	Function	Provides the absolute value.
INT	Format	INT (<numeric expression>)
	Function	Provides the max. integer value not exceeding the specified <numeric expression>.
CINT	Format	CINT (<numeric expression>)
	Function	Provides the integer value to which the real value is converted.
CSNG	Format	CSNG (<numeric expression>)
	Function	Provides the value which is converted to the single-precision real value.
SGN	Format	SGN (<numeric expression>)
	Function	Provides a sign for <numeric expression>.

Character string process

Statement		
ASC	Format	ASC (<character string>)
	Function	Provides the character code (ASCII) for the first character of the character string.
CHR$	Format	CHR$ (<numeric expression> [, <numeric expression>] ···)
	Function	Converts the character code (ASCII and internal sequential code) to a character.
RIGHT$	Format	RIGHT$ (<character string>, <numeric expression>)
	Function	Provides a character string of the length specified on the right side of the character string.
LEFT$	Format	LEFT$ (<character string>, <numeric expression>)
	Function	Provides a character string of the length specified on the left side of the character string.
MID$	Format	MID$ (<character string>, <numeric expression 1> [, <numeric expression 2>])
	Function	Provides the specified character in a given character string.
LSET RSET	Format	LSET <character variable> = <character expression> RSET <character variable> = <character expression>
	Function	Transfers data to the buffer for the random file. (In preparation for the PUT statement)
HEX$	Format	HEX$ (<numeric expression>)
	Function	Converts a decimal number to a hexadecimal number, and then provides the character string.
STR$	Format	STR$ (<numeric expression>)
	Function	Provides the character string indicating <numeric expression>.
VAL	Format	VAL (<character string>)
	Function	Converts a character string to a numeric value.
SPACE$	Format	SPACE$ (<numeric expression>)
	Function	Provides character strings of spaces in the specified length.
LEN	Format	LEN (<character string>)
	Function	Provides the length of the character string (the number of bytes). Kanji is counted as 2 bytes.
INSTR	Format	INSTR ([<numeric expression> ,] <character string 1>, <character string 2>)
	Function	Searches for the specified characters from the character string, and provides its first character position of the characters.
MKI$ MKS$	Format	MKI$ (<integer value>) MKS$ (<single-precision real value>)
	Function	Converts a numeric value to the numeric data of the character string type.
CVI CVS	Format	CVI (<2-byte character string>) CVS (<4-byte character string>)
	Function	Provides a value for the numeric value data which is converted to the numeric data.

Error process

Statement		
ON ERROR	Format	ON ERROR GOTO <line number>
	Function	Declares that the interrupt for the error process is enabled, and declares the line number of the subroutine to be executed when an error occurs.
RESUME	Format	RESUME [0] | NEXT | <line number>
	Function	Terminates an error process, and resumes execution of the program.
ERL/ERR	Format	ERR/ERL
	Function	Provides the line number in which an error occurs (ERL) and the error code (ERR).

Display input/output

Statement		
CLS	Format	CLS
	Function	Clears the screen.
PRINT	Format	PRINT [<expression list>] [;] ? [<expression list>] [;]
	Function	Displays the character string and contents of the variable on the screen.
LOCATE	Format	LOCATE [<line>] [, [<column>] [, < <switch>]]]
	Function	Specifies the position of the cursor on the screen.

Sound output

Statement		
BEEP	Format	BEEP
	Function	Sounds the internal buzzer.

Keyboard entry

Statement		
INPUT	Format	INPUT [;] [" <prompt statement> " ;] <variable> [, <variable>] …
	Function	Reads a numeric value or a character from the keyboard, and assigns it to a variable.
		Reads the whole line input from the keyboard (max. 255 characters), and assigns it to a variable.
INKEY$	Format	INKEY$
	Function	Provides the leading character if any key is pressed on the keyboard, or provides a null string if no key is pressed on the keyboard.
INPUT$	Format	INPUT$ (<no. of characters> [, [#] <file number>])
	Function	Reads the character string with the specified length from the keyboard or the file.

Communication input/output

Statement		
OPEN COM	Format	OPEN ”COM <line number> : [communication condition] ” AS # <file number>
	Function	Opens the communication file.
ON COM (n)	Format	ON COM (<line number>) GOSUB <line number>
	Function	Declares an interrupt which occurs when the data is input into the communication buffer, and declares the start line number for the subroutine to be executed.
COM (n)	Format	COM (<line number>) ON/STOP
	Function	Enables/Stops an interrupt from the communication file.
SENDCMD (n)	Format	SENDCMD <expression list>
	Function	Creates a command packet for the printer, and sends it to COM1.

Clock input/output

Statement		
ON TIMER	Format	ON TIMER (<n>) GOSUB <line number>
	Function	Declares an interrupt which occurs at specified intervals, and the line number from which the execution of the subroutine is started by the interrupt.
TIMER ON	Format	TIMER ON/STOP
	Function	Enables/Stops a timer interrupt.
TIMER	Format	TIMER
	Function	Returns the elapsed time after the system is reset in a single-precision floating point format.

File input/output

Statement		
OPEN	Format	OPEN <file designation> [FOR <file mode>] AS [#] <file number> [LEN= <record length>]
	Function	Opens a file.
CLOSE	Format	CLOSE [[#] <file number> [, [#] <file number>] …]
	Function	Closes a file.
INPUT#	Format	INPUT# <file number>, <variable> [, <variable>] …
	Function	Reads data from the sequential file, and assigns it to a variable.
LINE INPUT#	Format	LINE INPUT# <file number>, <character variable>
	Function	Reads one whole line (record) (max. 255 characters) from the sequential file.
INPUT$	Format	INPUT$ (<no. of characters> [, [#] <file number>])
	Function	Reads the character string with the specified length from the keyboard or the file.
PRINT#	Format	PRINT# <file number>, [USING <format control character string> ;], <expression list>
	Function	Outputs data (numeric value or character string) to the sequential file.
WRITE#	Format	WRITE# <file number>, <expression list>
	Function	Outputs data to a sequential file.
FIELD#	Format	FIELD [#] <file number>, <field width> AS <character variable>] [,<field width> AS <character variable>] …
	Function	Allocates the variable area to the random file buffer. (Record definition)
GET#	Format	GET [#] <file number> [, <numeric value>]
	Function	Inputs the data in a file into the buffer.
PUT#	Format	PUT [#] <file number>) [, <numeric value>]
	Function	Outputs the data to the file.
EOF	Format	EOF (<file number>)
	Function	Checks the end of the sequential file, or whether or not the communication buffer becomes empty.
LOC	Format	LOC (<file number>)
	Function	Provides the current theoretical location in the file.
LOF	Format	LOF (<file number>)
	Function	Provides the size of the file.

ABS

Function	Provides the absolute value.
Format	ABS (<numeric expression>)
Explanation	The absolute value for <numeric expression> (0 or a positive value) is provided as a function value.

ASC

Function	Provides the character code (ASCII) for the first character of the character string.
Format	ASC (<character string>)
Explanation	The character code for the first character (left side) of <character string> is provided. If <character string> is a null string, an error occurs.

BEEP

Function	Sounds the internal buzzer.
Format	BEEP
Explanation	The internal buzzer is sounded for 100 ms.

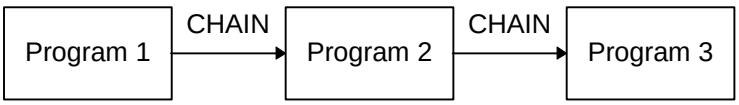
CHAIN

Function Loads another program, and executes it.

Format CHAIN <file designation> [, ALL]

Term <file designation>: The program file to be executed should be specified. (The file name should be included.)

Explanation The program designated in <file designation> is executed.



If the ALL option is designated, all variables are passed. If it is necessary to pass variables when the ALL option is omitted, the COMMON statement should be used.

```
CHAIN "PROG2", ALL
```

The CHAIN statement holds any file which has already been opened as is.

CHR$

Function Converts the character code (ASCII and internal sequence code) to a character.

Format CHR$ (<numeric expression> [, <numeric expression>] ⋯)

Term <numeric expression>: The numeric expression within the range between 0 and 255 (&H0 to &HFF) indicating ASCII code

Explanation The CHR$ function regards a value for <numeric expression> as ASCII code, and converts it to a character. The CHR$ is the opposite of the ASC function.

CINT

Function	Provides the integer value to which a real value is converted.
Format	CINT (<numeric expression>)
Explanation	The value for <numeric expression> is rounded off to the nearest whole number, making it into an integer.

CLEAR

Function Initializes a variable.

Format CLEAR

Explanation The memory used for storing data is freed without deleting any programs in the memory. After the CLEAR statement is executed, all numeric variables and character variables become 0 and null strings (“”), respectively. The array declaration becomes invalid. All files are closed.

The CLEAR statement executes the following processes.

1. All files are closed.
2. All COMMON statements and user variables are initialized.
3. The stack area and character area are initialized.

The ERASE statement is similar to the CLEAR statement. For the CLEAR statement, all variables are initialized, however, for the ERASE statement, only the specified array variable is initialized.

CLOSE

Function Close a file.

Format CLOSE [[#] <file number> [, [#] <file number>] ⋯]

Term <file number>: The number assigned to the file or the device by the OPEN statement

Explanation The file corresponding to <file number> is closed. One or more <file number>s can be specified once in the CLOSE statement. If <file number> is omitted, all opened files are closed.

Once a file has been closed, its number can be used for opening another file. The closed file can be opened by specifying the same file number as the previous one or a different file number.

For a file that has been opened for an output, the data remaining in the buffer is output by the CLOSE statement.

The END statement closes all opened files automatically.

CLS

Function	Clears the screen.
Format	CLS

COMMON

Function

Declares the variable is to be passed from the original program to a program called by the CHAIN statement.

Format

COMMON <variable name> [, <variable name>] ⋯

Term

<variable name>: The variable name to be passed should be specified.

Explanation

The COMMON statement is used together with the CHAIN statement, and declares that a variable is to be passed from the original program to a called program.

The same variable name cannot be used more than once in the COMMON statement. When the array variable is passed, two parentheses “( )” should be inserted after the variable name to indicate it is an array.

When all the variables are to be passed, not the COMMON statement but the ALL option in the CHAIN statement should be used.

A COMMON statement should be placed before the CHAIN statement.

When an array is to be passed to a called program, the array declaration should not be performed by the DIM statement. However, it is necessary to previously perform the array declaration by the DIM statement in the program to be called.

COM ON/STOP

Function

Enables/Stops an interrupt from the RS-232C communication file.

Format

COM (<line number>) ON
COM (<line number>) STOP

Term

<line number>: The RS-232C interface number (1 or 2)

Explanation

The interrupt which occurs when the communication flows to the RS-232C from an external source is enabled by the COM (<line number>) ON statement, or stopped by the COM (<line number>) STOP statement.

When the COM (<line number>) ON statement is used, an interrupt should be enabled previously by the ON COM statement. Then, an interrupt occurs every time data is input through the RS-232C interface, and the process specified by the ON COM statement is performed.

When the COM (<line number>) STOP statement is executed, an interrupt does not occur if data is input, however, the fact that data has been input is stored. After that, when the interrupt is enabled by the COM (<line number>) ON statement, the interrupt occurs immediately, and then the process is performed.

The COM (<line number>) STOP statement is used to prevent the process from being suspended by the occurrence of an interrupt.

When a communication interrupt is detected according to <line number> n, the interrupt routine executes the COM (<line number>) STOP statement automatically, and disables the interrupt which is being processed. The execution is automatically returned from the interrupt routine to the previous setting state.

CSNG

Function	Provides a value which is converted to a single-precision real value.
Format	CSNG (<numeric expression>)
Explanation	The value for <numeric expression> is converted to a 7-digit single-precision real value.

CVI/CVS

Function Provides a value for numeric value data which is converted to the numeric data.

Format

CVI (<2-byte character string>)
CVS (<4-byte character string>)

Explanation

All numeric values in the random file on the disk are converted to the character string type. This function returns the numeric data converted to the character string type to a numeric value.

CVI converts the first 2 bytes (2 characters) to an integer. CVS converts the first 4 bytes (4 characters) to a single-precision real value.

DATA

Function Sets a numeric value or a character constant read by the READ statement.

Format DATA <constant> [, <constant>] ⋯

Term <constant>: A numeric constant or character constant

Explanation The DATA statement is a non-executing statement, and should be placed before a READ statement. The data which can be included in one line (max. 255 characters) can be written in one DATA statement. There is no limit to the number of DATA statements that can be used in one program.

The contents written in the DATA statement are interpreted as the data list to be input. The data to be input is read by the READ statement in the ascending order of the line number.

A numeric constant or character constant can be written in <constant>. Any of the integer fixed-point, floating-point, octal notation, and hexadecimal notation can be used for a numeric constant. Double quotation marks (”) enclosing a character constant in the DATA statement can be omitted. However, they cannot be omitted when a comma (,), colon (:), or semicolon (;) are included in the character string, and when a space is entered in the beginning or end of the character string.

Note that the variable type should match the data type when the data in the DATA statement is read by the READ statement. If there are any data types which do not match, a “Syntax error” occurs.

Comments can not be attached to the line of the DATA statement.

DIM

Function

Specifies the max. number of array elements, and allocates the memory area in the memory.

Format

DIM <variable name> (<subscript>)] ⋯) [, <variable name> (<subscript> [, <subscript>] ⋯)] ⋯

Term

<variable name>: Array variable name
(It should be named according to the usual variable naming conventions.)

<subscript>: A numeric value or numeric expression indicating the max. value for the subscript of the array

Explanation

A specified area in which the array variable is stored is reserved, and the variable is initialized at the same time. 0 is assigned to all numeric array variables as an initial value by executing the DIM statement. Null strings are assigned to a character array variable. Therefore, the length of the character string is 0.

The max. of 16 dimensions and 32767 elements is available for an array, however, they are limited according to the memory capacity. In actuality, the maximum will not be used, since the number of characters which can be included in one line is also limited. The min. value for subscript is 0.

If an array variable is used without an array declaration by the DIM statement, the value for subscript is automatically set to 10. In this case, a value higher than 11 cannot be set. If the value for subscript exceeds the specified max. value, an error occurs.

The array variable declared by the DIM statement can be deleted by the ERASE statement, and the used memory area is cleared and ready to be used again.

END

Function

Terminates the execution of the program, closes all opened files, and returns to the command level state.

Format

END

Explanation

The END statement can be placed in any position in which the program is to be terminated. One or more END statements can also exist in the program. The END statement placed at the end of the program can be omitted.

EOF

Function

Checks the end of the sequential file, or whether or not the communication buffer becomes empty.

Format

EOF (<file number>)

Term

<file number>: The number assigned to the file by the OPEN statement

Explanation

The EOF function returns -1 (true) when the specified file reaches EOF (End of File).

If the specified file does not reach EOF, the EOF function returns 0 (false).

A sequential file and RS-232C communication file are available.

When the EOF function results in -1 in the RS-232C communication file, it means the buffer is empty.

ERASE

Function Erases the specified array from the program.

Format ERASE <array name> [, <array name>] …

Term <array name>: Array name to be erased

Explanation When there is not enough memory area while executing the program, arrays which become unnecessary should be erased by using the ERASE statement. The areas of erased arrays can be used for other arrays.

The ERASE statement can also be used for reinitializing the array which has been used in the program for another object.

If the array declaration for the same array name is performed by the DIM statement before the ERASE statement is used, a “Duplicate definition” error occurs.

If there is not a specified array name, an “Illegal function call” error occurs.

The ERASE statement erases the specified array only, however, the CLEAR statement initializes all variables.

ERR/ERL

Function Provides the line number in which an error occurs (ERL) and the error code (ERR).

Format ERR

ERL

Explanation The system variables, ERR and ERL, are used in the IF ~ THEN statement, and used for branching to error processes. (For details, refer to “ON ERROR”.)

Immediately after BASIC is started up, ERL is 0. A numeric value for the variable cannot be assigned to ERR and ERL.

FIELD

Function

Allocates the variable area to the random file buffer. (Record definition)

Format

FIELD [#] <file number>, <field width> AS <character variable> [, <field width> AS <character variable>] ⋯

Term

<file number>: The number assigned to the file by the OPEN statement

<field width>: The number of characters to be assigned to character variables

<character variable>: The variable name to be used for the input/output of the random file

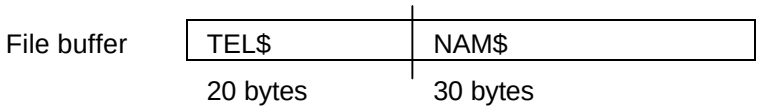
Explanation

The FIELD statement specifies the contents of the random file buffer to which read/write is performed by the PUT statement and the GET statement. Therefore, the FIELD statement should be executed before the PUT statement and the GET statement are executed.

Ex.)

```
FIELD #1,20 AS TEL$,30 AS NAM$
```

The above means the first 20 characters (20 bytes) from the furthest left in the random file buffer #1 (No. 1) are assigned to the character variable, TEL$, and the next 30 characters are assigned to NAM$. These are called “fields”.



The buffer in the random file can be efficiently allocated to more character variables by the FIELD statement.

The FIELD statement does not actually transfer data to the random file buffer. Data transmission to the buffer is performed by the LSET statement and RSET statement.

The total number of bytes to be allocated by the FIELD statement is not allowed to exceed the record length specified in the OPEN statement. The number of bytes set in the FIELD statement is the record length.

The FIELD statement defines the buffer from the first character for every execution. Therefore, one or more areas can be defined for the same data.

FOR ~ NEXT

Function

Executes the statements included from the FOR statement to the NEXT statement repeatedly while the given conditions are satisfied.

Format

FOR <variable> = <initial value> TO <terminal value> [STEP <increment>]
NEXT [<variable> [, <variable>] ⋯]

Term

<initial value>: A numeric value to be set at the beginning of the repetition
An integer or a single-precision real number

<variable>: A variable used as the counter which counts the number of executions
An integer or a single-precision real number

<increment>: Both positive and negative are available.

<terminal value>: A value for which the repetition is terminated
An integer or a single-precision real number

Explanation

An input or assignment to <variable> cannot be performed in the loop, since <variable> is used as the loop counter. However, <variable> can be used as a value having the current loop value.

<initial value> is set to <variable> first. When the execution reaches to the NEXT statement, <increment> specified in STEP is added to/subtracted from the counter value (<variable>). If STEP is not specified, <increment> is set to +1. A negative number can also be specified in <increment>. Then, whether or not the counter value exceeds <terminal value> is checked. If it does not exceed it, the execution is returned to the next statement of the FOR statement. If it exceeds it, the execution proceeds to the next statement of the NEXT statement.

For the FOR ~ NEXT statement, the evaluation is made before the execution. In the following cases, the FOR ~ NEXT statement is not executed once, and the execution proceeds to the next statement of the NEXT statement.

1) <increment> is positive, and <initial value> is larger than <terminal value>.

Ex.) **`FOR I=10 to 5 STEP2`**

2) <increment> is negative, and <initial value> is smaller than <terminal value>.

Ex.) **`FOR I=1 to 7 STEP-3`**

When <increment> is 0, the loop is performed endlessly, unless <variable> larger than <terminal value> is set in the loop.

Other FOR ~ NEXT statements can be used (nested) within the FOR ~ NEXT statement. In this case, each <variable> to be set should be different. One FOR ~ NEXT statement should be completely included in the other FOR ~ NEXT statement.

Ex.)

<Proper>

```
FOR I=1 TO 5
   FOR J=1 TO 5
      :
   NEXT J
      :
NEXT I
```

<Wrong>

```
FOR I=1 TO 5
   FOR J=1 TO 5
      :
NEXT I
      :
   NEXT J
```

If FOR ~ NEXT statements are terminated at the same point, their NEXT statements can be described in one statement as shown below.

Ex.)

```
FOR I=1 TO 10
   FOR J=1 TO 10
      FOR K=10 TO 1 STEP-1
         :
NEXT K,J,I
```

When <variable> is omitted in the NEXT statement, the NEXT statement pairs off with the nearest FOR statement. If the NEXT statement is executed before the paired FOR statement, a “NEXT Without FOR” error occurs.

<initial value> and <terminal value> are determined by the first FOR statement.

Ex.)

```
I=10
   FOR I=I TO I+5
      :
   NEXT
```

A loop is performed within the range of “I = 10 to 15”.

GET

Function

Inputs the data in a file into the buffer.

Format

GET [#] <file number> [, <numeric value>]

Term

<file number>: The number assigned to the file by the OPEN statement

<numeric value>: The record number between 1 and 65535, or the number of bytes of the data read from the RS-232C communication file

Explanation

The GET statement reads the data in the file specified in <file number>, and inputs the read data into the corresponding buffer. Note that the operation varies according to the specified file.

When the file specified in <file number> is a disk file, the record is read from the random file, and is input into the file buffer (the file buffer should be assigned by the FIELD statement). <numeric value> should be the record number in the random file. If <numeric value> is omitted, it is automatically set to the next number of the record number read by the last GET statement.

When the file specified in <file number> is the RS-232C communication file, <numeric value> indicates the number of bytes of the data to be read from the RS-232C communication buffer. If <numeric value> is omitted, it is the record length specified by the OPEN statement.

GOTO

Function Moves the execution to the specified line without any conditions.

Format GOTO <line number>

Term <line number>: The line number to which the execution is moved

Explanation The GOTO statement moves the execution to the specified line. When the specified line is a non-executable statement (such as a REM statement or a DATA statement), the execution starts from the next executable statement.

To move the execution to any line according to a calculation result and a numeric value, the ON ~ GOTO statement should be used.

HEX$

Function

Converts a decimal number to a hexadecimal number, and then provides the character string.

Format

HEX$ (<numeric expression>)

Explanation

<numeric expression> is converted to an integer value which is rounded off to a whole number before <numeric expression> is used for the HEX$ function.

The result of the HEX$ function is a hexadecimal number (0 to FF). However, a character variable should be used for the assignment since the result is handled as a character.

The range for <numeric expression> is as described below. Any value exceeding the range results in an “Overflow” error.

Decimal number: -32768 ~ 0 ~ 65535
Hexadecimal number: &H8000 ~ 0 ~ &HFFFF

HEX$ (-n) is equal to HEX$ (65536-n).

IF ~ THEN ~ ELSE

Function

Judges the condition, and changes the flow of the program.

Format

```
IF <expression> THEN | <statement>   | [ELSE | <statement>   | ]
                     | <line number> |      | <line number> |
IF <expression> GOTO   <line number>   [ELSE | <statement>   | ]
                                             | <line number> |
```

Term

<expression>: A theoretical expression, an arithmetic expression, or a variable

<statement>: BASIC statement list

<line number>: The line number for the next execution

Explanation

The flow of the program is changed according to the evaluation of <expression>. When <expression> is true (other than 0), <statement> which follows THEN is executed, or the execution is moved to <line number> which follows GOTO.

A line number or one or more statements can be described after THEN. However, only the line number is specified after GOTO.

When <expression> is false (0), the THEN statement or GOTO statement are ignored. And if there is an ELSE statement after them, it is executed.

The IF ~ THEN ~ ELSE statement can be multiplexed (nested) by writing another IF ~ THEN ~ ELSE statement in the <statement> which follows THEN and ELSE.

In this case, if the numbers of THEN statements and ELSE statements are not the same, each ELSE statement pairs off with the nearest THEN statement.

The IF ~ THEN ~ ELSE statement is one block. Therefore, the ELSE cannot be put on a separate line. The ELSE statement should be included in one line (max. 255 characters with the If ~ THEN).

INKEY$

Function

Provides the leading character if any key is pressed on the keyboard, or provides a null string if no key is pressed on the keyboard.

Format

INKEY$

Explanation

The number of characters provided for the INKEY$ variable is 0 (null string) or 1. It is determined according to the key entry.

“0” indicates that key entry is not performed.

The INKEY$ function does not display the key entry on the screen.

INPUT

Function

Reads a numeric value or a character from the keyboard, and assigns it to a variable.

Format

INPUT [;] [" <prompt statement> " ;] <variable> [, <variable>] ⋯

Term

<prompt statement>: A character string which is output on the screen when the input is performed
(It is used to make it easy to understand the numeric value or character to be input.)

<variable>: The numeric or character variable to which the input data is assigned

Explanation

When the INPUT statement is executed, the prompt statement is displayed on the screen, and the program waits for the input from the keyboard. If a semicolon (;) follows <prompt statement>, a question mark (?) is displayed after <prompt statement>. If a comma (,) is used instead of the semicolon (;), the question mark (?) is not displayed.

The input from the keyboard is entered by pressing the [ENTER] key, and the execution proceeds to the next statement. Until the [ENTER] key is pressed, the numeric value or character string which is input and displayed can be modified.

If only the [ENTER] key is pressed when there is one <variable> in the INPUT statement, it is interpreted as a no entry or a null string, and then it is assigned to <variable>.

For the key entries, refer to "KEY ENTRY CODE LIST".

INPUT#

Function Reads data from the sequential file, and assigns it to a variable.

Format INPUT# <file number>, <variable> [, <variable>] ⋯

Term <file number>: The number assigned to the file by the OPEN statement

<variable>: The variable name to which data is assigned.
A numeric variable or character variable

Explanation The data should be read from the sequential file on the disk or the RS-232C communication device. The type of <variable> should match the type of data.

How to fetch the input data is shown below.

Type	Code ignored the leading character of the character string	Delimiter	Remark
Numeric	Space Carriage return (CR) Line feed (LF)	Comma (,) Space Carriage return Line feed	
Character	Same as above	Comma (,) Carriage return Line feed	Max. 255 characters a variable
“Character”	Same as above	Double quotation mark (”)	

When the end of file (EOF) is confirmed when reading the data of numeric type or character type, the data item is delimited at that time. The end of file can be confirmed by the EOF function. In reading the data of character type, a space immediately before a delimiter is ignored.

INPUT$

Function Reads the character string with the specified length from the keyboard or the file.

Format INPUT$ (<no. of characters> [, [#] <file number>])

Term

<no. of characters>: The number of characters to be read from the keyboard or file (1 to 255.)

<file number>: The number assigned to the file by the OPEN statement

Explanation

The character string with the length specified in <no. of characters> is read from the file specified in <file number>.

If <file number> is omitted, the input from the keyboard is available. However, characters input from the keyboard are not displayed on the screen, unlike the INPUT statement. When the keyboard entry reaches the specified length, the keyboard entry is disabled automatically, and the program proceeds. Therefore, it is unnecessary to press the [ENTER] key.

Since the INPUT$ statement is not limited by the input data, it can be used for reading the ASCII code which cannot be input by the INPUT statement and LINE INPUT statement.

INSTR

Function

Searches for the specified characters from the character string, and provides the first character position of the characters.

Format

INSTR ([<numeric expression> ,] <character string 1>, <character string 2>)

Term

<numeric expression>: Indicates the position in <character string 1> where the searching of <character string 2> is started.
(Unit: bytes, Range: 1 to 255)

<character string 1>, <character string 2>: A character variable, character expression, or character constant

Explanation

The INSTR statement searches for <character string 2> from the specified position (the <numeric expression>th bytes from the left) in <character string 1>, and then provides the first position (byte) of <character string 2>. If <numeric expression> is omitted, the searching starts from the first byte in <character string 1>.

In the following cases, “0” is provided.

- <character string 2> is not found.
- The value for <numeric expression> is larger than the value for <character string 1>.
- <character string 1> is a null string.

When a null string is specified in <character string 2>, “1” is provided.
When <numeric expression> is specified in <character string 2>, the value is provided.

If <numeric expression> exceeds the range of 1 to 255, an “Illegal function call” error occurs.

INT

Function	Provides the max. integer value not exceeding the specified <numeric expression>.
Format	INT (<numeric expression>)
Explanation	The max. integer value not exceeding <numeric expression> is provided.

LEFT$

Function

Provides a character string of the length specified on the left side of the character string.

Format

LEFT$ (<character string>, <numeric expression>)

Term

<character string>: Any character string

<numeric expression>: The length is specified on the left side of <character string>. (Unit: bytes, Range: 0 to 255)

Explanation

<numeric expression> is converted to an integer value rounded off to a whole number before it is evaluated.

When the value for <numeric expression> is larger than the number of strings of <character string>, all character strings are provided. When the value is 0, a null string is provided.

LEN

Function Provides the length of the character string (the number of bytes).
Kanji is counted as 2 bytes.

Format LEN (<character string>)

Explanation The length should be between 0 and 255 bytes. A space and a code which is not displayed as the control code are also counted and included in the length.

LINE INPUT#

Function

Reads one whole line (record) (max. 255 characters) from the sequential file.

Format

LINE INPUT# <file number>, <character variable>

Term

<file number>: The number assigned to the file by the OPEN statement

<character variable>: A character variable to which one whole line is assigned

Explanation

The LINE INPUT# statement reads one whole line (record) from the sequential file. The record of the carriage return (CR) is used as a delimiter which delimits the line.

When the carriage return and line feed is placed in this order, the characters up to that are read. The next LINE INPUT# statement reads the line up to the next carriage return.

Both the line feed code and carriage return code are each regarded as one character, and are stored as part of the character string.

The LINE INPUT# statement is useful when there are many lines in the file.

LOC

Function Provides the current theoretical location in the file.

Format LOC (<file number>)

Term <file number>: The number assigned to the file by the OPEN statement

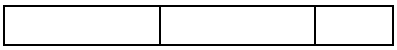
Explanation The file specified in <file number> is:

1. Random file

   The LOC function returns the record number for which the read/write (GET/PUT) was performed last in a random file. This function is useful for performing read/write in the order of the record number; it is possible to obtain the record number accessed last.

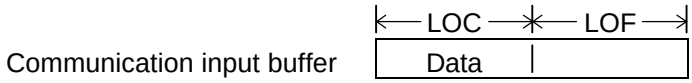
2. Sequential file

   The LOC function returns the position of the current file pointer in bytes.



3. RS-232C communication file

   The LOC function returns the number of characters (bytes) which are accepted by the interrupt from the RS-232C and stored in the input buffer. If read (INPUT# or INPUT$) is not performed, the buffer becomes full and an error occurs.



LOCATE

Function

Specifies the position of the cursor on the screen and determines the display of the cursor.

Format

LOCATE [<line>] [, [<column>] [, < <switch>]]]

Term

<line>: The vertical position on the screen (1 to 2)

<column>: The horizontal position on the screen (1 to 16)

<switch>: Determines whether or not the cursor is displayed when the program is executed.

0: Not displayed

1: Displayed

Explanation

The cursor is moved to the position specified in <line> and <column>. The output of characters by the PRINT statement is performed from the position of the cursor.

The display of the cursor can be selected in <switch>. The cursor is not usually displayed in the program execution state, except when waiting for input from the INPUT statement. When 1 is set in <switch>, the cursor is also displayed during execution of the program.

LOF

Function Provides the size of the file.

Format LOF (<file number>)

Term <file number>: The number assigned to the file by the OPEN statement

Explanation The file specified in <file number> is:

1. Disk file

   The file size is provided in units of bytes.

2. RS-232C communication file

   The remaining number of bytes of the input buffer is returned. It is the result of the LOC function value subtracted from the input buffer size

LSET/RSET

Function Transfers data to the buffer for a random file. (In preparation for the PUT statement)

Format LSET <character variable> = <character expression>

RSET <character variable> = <character expression>

Term <character variable>: The character variable used for the assignment in the FIELD statement.

<character expression>: The character string to be transferred

Explanation When the character string to be transferred is shorter than the number of characters assigned in the FIELD statement, the character string to be transferred is left-justified and right-justified in the LSET statement and RSET statement, respectively. For the FIELD statement, it is right-justified. Blanks are filled with spaces in all of these statements.

When the character string to be transferred is longer than the number of characters assigned in the FIELD statement, it is left-justified in both LSET statement and the RSET statement. Note that excess characters are lost from the right side of the character string at this time.

The LSET statement and the RSET statement can also be used for a variable other than the variable assigned in the FIELD statement. In this case, the required number of characters should be previously assigned to a variable as dummies.

Ex.)

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

The above indicates that the character string N$ is right-justified in the variable area consisting of 20 characters.

These statements are useful for arranging the print format.

Numeric data cannot be used for the LSET statement and RSET statement if it is not converted into character data. To convert numeric data into character data, the MKI$ and MKS$ functions should be used.

MID$

Function Provides the specified character in a given character string.

Format MID$ (<character string>, <numeric expression 1> [, <numeric expression 2>])

Term

<character string>: Any character string

<numeric expression 1 >: The position in <character string>
(Unit: bytes, Range: 1 to 255)

<numeric expression 2>: The length to be specified in <character string>
(Unit: bytes, Range: 0 to 255)

Explanation The MID$ function provides the character string from the character of the <numeric expression 1>th byte from the left in <character string> to the character of the <numeric expression 2>th byte. <numeric expression 1> and <numeric expression 2> should be integer values which are rounded off to whole numbers before they are evaluated.

In the following cases, all character strings at the right side of the <numeric expression 1>th byte are provided.

- <numeric expression 2> is omitted.
- The number of characters from the <numeric expression 1>th byte to the end of the character string is smaller than <numeric expression 2>

In the following cases, a null string is provided.

- There is no character for the <numeric expression 1>th byte.
(<numeric expression 1> is longer than the character string length.)
- <numeric expression 2> is 0.

MKI$/MKS$

Function Converts a numeric value to the numeric data of the character string type.

Format
MKI$ (<integer value>)
MKS$ (<single-precision real value>)

Explanation When a random file is created by using the LSET and RSET statements, all values to be written in the buffer should be assigned as the numeric data of the character string type. MKI$ converts an integer value to a 2-byte character string. MKS$ converts a single-precision real value to a 4-byte character string.

ON COM

Function

Declares an interrupt which occurs when the data is input into the communication buffer, and declares the start line number for the subroutine to be executed.

Format

ON COM (<line number>) GOSUB <line number>

Term

<line number>: The line number to which an interrupt is processed (1 or 2)

<line number>: The line number from which the subroutine for the interrupt process is started should be specified.

Explanation

The ON COM statement declares the interrupt process for receiving data only, and does not execute the process.

When the ON COM statement is executed, BASIC checks whether or not data is received in the specified line while executing the program. When data reception is confirmed in the specified line, the interrupt process routine is executed if an interrupt is enabled by the COM (n) ON statement. When “0” is specified in <line number>, the interrupt process is disabled. To prevent another interrupt process from being received during the execution of the first interrupt process, the COM (n) STOP statement is automatically executed if the interrupt process is being executed. When the RETURN statement for terminating the subroutine is sent, the program is returned to the previous state.
An interrupt occurs only when the program is executed.

ON ERROR

Function

Declares that the interrupt for the error process is enabled, and declares the line number of the subroutine to be executed when an error occurs.

Format

ON ERROR GOTO <line number>

Term

<line number>: The first line number of the subroutine for the error process

Explanation

When an error interrupt is enabled by this statement, the subroutine specified in <line number> is executed for all errors including the command level state (in the direct mode). If <line number> is not specified, an “Undefined line number” error occurs.

To disable an error interrupt, “ON ERROR GOTO 0” should be executed.

To resume executing the program after the error message is displayed, the RESUME statement should be used.

ON GOTO/GOSUB

Function

Branches the execution to the specified line number according to a value for <expression>.

Format

ON <expression> GOTO <line number> [, <line number>] ⋯
ON <expression> GOSUB <line number> [, <line number>] ⋯

Term

<expression>: An expression to specify <line number> which is placed after GOTO/GOSUB.
It should be 255 or less.
(If a value is not an integer, a decimal is rounded off to the nearest integer).

<line number>: Line number to which the execution is branched by GOTO/GOSUB.

Explanation

The line number to which the execution is branched is determined according to the value of <expression>. See an example below.

```
ON DATA1 GOTO 10, 20, 30, 40
```

In the above statement, if a value for <DATA1> is 1, 2, and 3, the execution is branched to line number 10, 20, and 30, respectively.

In the ON ~ GOSUB statement, the program which starts in each line number must be the sub program which the RETURN statement is placed at the end of the program. When the value for <expression> is 0, or is larger than the number of <line number>s, ON ~ statement is ignored, and the execution is moved to the next statement (GOSUB statement).

ON TIMER

Function

Declares an interrupt which occurs at specified intervals, and the line number from which the execution of the subroutine is started by the interrupt.

Format

ON TIMER (<n>) GOSUB <line number>

Term

<n>: The time interval at which an interrupt occurs should be specified in units of 100 ms. (Range: 1 to 32767)

<line number>: The line number from which the subroutine for the interrupt process is started should be specified.

Explanation

The ON TIMER statement only declares the interrupt process; it does not execute the process.

When the ON TIMER statement is executed, it counts down the specified time by using the system timer. When it reaches the specified time, the interrupt process routine is executed if an interrupt is enabled by the TIMER ON statement. When “0” is specified in <line number>, the interrupt process is disabled. To prevent another interrupt process from being received during the execution of the first interrupt process, the TIMER STOP statement is automatically executed if the interrupt process is being executed. If there is no TIMER STOP statement during the execution of this process, the TIMER ON statement is automatically executed by the RETURN statement for terminating the subroutine, and an interrupt is received.
An interrupt occurs only when the program is executed.

OPEN

Function

Opens a file.

Format

OPEN <file designation> [FOR <file mode>] AS [#] <file number> [LEN= <record length>]

Term

<file designation>: The file name to be opened should be specified.

<file mode>: The type of file should be specified.

OUTPUT	Sequential access / New output mode
INPUT	Sequential access / Input mode
APPEND	Sequential access / Append mode
RANDOM	Random access / Input/output mode
Omitted	Random access / Input/output mode

<file number>: An integer between 1 and 9

<record length>: The record length in the random access mode should be specified in bytes. (Range: 1 to 32767, or 128 if the designation is omitted)

When the file specified in <file number> is a disk file, the record length is the number of bytes for the record specified in the FIELD statement.

OPEN COM

Function Opens the RS-232C communication file.

Format OPEN ”COM <line number> : [<communication baud rate>] [, [<parity>] [, [<data length>] [, [<stop bit>]]] ” AS [#] <file number>

Term

<line number>: The RS-232 C interface number
(COM 1:, COM2:)

<communication baud rate>: The baud rate for the line
COM 1: (9600, 19200, 38400)
COM 2: (4800, 9600, 19200)

<parity>: The parity type is specified. (O (Odd), E, (Even), N (Non))

<data length>: The bit length of one character is specified. (7, 8)

<stop bit>: The number of stop bits to be added (1, 2)

<file number>: An integer between 1 and 9

When COM 1 is connected to the B-452 printer, 38400, E, 8, 1 should be set in <communication baud rate>, <parity>, <data length>, <stop bit>, and <file number>, respectively.

PRINT

Function Displays the character string and contents of the variable on the screen.

Format PRINT [USING <format control character string>] [<expression list>] [;]

Term <format control character string>: A character string to specify the format

<expression list>: Numeric expressions, character expressions and character strings are delimited by a semicolon (;), listed.
(The character string should be enclosed with double quotation marks (").)

Explanation The character strings of the specified expression are displayed on the screen.

The displayed position varies according to how the elements of <expression list> are delimited. One line is delimited every 16 characters from the left in BASIC. When a semicolon (;) is used, the character string is displayed following the preceding display, regardless of areas. Also, when one or more spaces are inserted between <expression list>s, the same result as when a semicolon (;) is used is obtained.

The control of the cursor position should be performed by the LOCATE statement.

Format control for a numeric value can be performed by using a format control character string.

- # : Specifies the number of digits for the value to be output. One "#" represents one digit. When the number of digits for the numeric value is smaller than the specified number of digits, the data to be output is aligned to the right.
- . : Specifies the position of the decimal point.
- + : When it is placed at the beginning of <format control character string>, a sign for a numeric is output first.

PRINT#

Function Outputs data (numeric value or character string) to the sequential file.

Format PRINT# <file number>, [USING <format control character string> ;], <expression list>

Term

<file number>: The number assigned to the file by the OPEN statement.

<expression list>: Numeric expressions and characters are delimited by a comma (,) or semicolon (;), and listed.

<format control character string>: A character string formatted for outputting data to the file is specified in detail

Explanation

The PRINT# statement is similar to the PRINT statement, however, the data is not displayed on the screen, and the character string is output to a file. The format designation by the USING statement is also similar. For details, refer to the PRINT USING statement.

The PRINT# statement does not compress data in the file. The data is written to the file with the same format as displayed on the screen by the PRINT statement.

When <expression> is a numeric value, is should be delimited by a semicolon (;). If a comma (,) is used, every 14-byte data is output like the PRINT statement, resulting in more file areas being unnecessarily used.

```
PRINT #1, 12;-34.5;"ABC";"DEFG"
```

In the above case, the data written to the file is shown below.

␣12␣-34.5␣ABCDEFG [CR][LF] (␣ : space)

When <expression> is the character string, it should be delimited by a comma (,). If a semicolon (;) is used, character strings are connected to each other, and they are output as one character string to the file.

Ex.) A$="CAMERA", B$="93604-1"

```
PRINT #1,A$;B$
```

If the statement is described as above, A$ is connected to B$, and "CAMERA93604-1" is output as one string to the files.

To prevent the above, a comma (,) should be inserted as a delimiter between character strings as below.

```
PRINT #1,A$;",";B$
```

If the statement is described as above, data can be read properly by the INPUT# statement, and the data is written to the file properly as below.

```
CAMERA, 93604-1
```

When a comma (,), semicolon (;), or space is placed at the beginning of the string, carriage return and line feed are included in the character string, and the output can be performed by the PRINT statement. However, when the data is read by the INPUT# statement later, commas, semicolons, carriage returns and line feeds are regarded as delimiters, and a space placed at the beginning of the string is ignored.

To prevent the above, double quotation mark (") (CHR$ (34)) should be added before and after each character string.

Ex.) A$="CAMERA, AUTOMATIC", B$="93604-1"

```
PRINT #1,A$;B$
```

If the statement is described as above, the data is written to the file as follows.

```
CAMERA, AUTOMATIC 93604-1
```

And "CAMERA" and "AUTOMATIC 93604-1" are assigned to A$ and B$, respectively by the following statement.

```
INPUT #1,A$;B$
```

To prevent this problem, the WRITE# statement should be used, instead of the PRINT# statement. The example is shown below.

```
WRITE #1,A$,B$,12,-34.5
```

When the above statement is executed, the data is written to the file as shown below.

"CAMERA, AUTOMATIC", "93604-1", 12, -34.5

The PRINT# statement can control the file format with the USING option.

For example:

```
PRINT #1,USING "####.##";12.3;456;9876.5
```

When the above statement is executed, the following data is written to the file, and the comma (,) placed at the end of the format is used for delimiting the data.

␣␣12.30,␣456.00, 9876.50,

PUT

Function

Outputs the data to the file.

Format

PUT [#] <file number> [, <numeric value>]

Term

<file number>: The number assigned to the file by the OPEN statement.

<numeric value>: A record number between 1 and 4294967295, or the number of bytes of the data read from the RS-232C communication file.

Explanation

The PUT statement writes the data to the file specified in <file number>. The operation varies according to the specified file.

When the file specified in <file number> is a disk file, the record of the file buffer is written to a random file. (The file buffer should be allocated by the FIELD statement.)

<numeric value> is the record number in the random file.

When <numeric value> is omitted, the record number becomes the next number read by the last PUT statement.

To transfer data to the file buffer, the LSET statement and the RSET statement should be used.

When the file specified in <file number> is an RS-232C communication file, <numeric value> indicates the number of bytes of data to be written to the RS-232C communication buffer.

The number of bytes should be smaller than the record length defined by the OPEN statement.

READ

Function Reads a value defined by a DATA statement and assigns it to a variable.

Format READ <constant> [, <constant>] ⋯

Term <variable>: The variable to which the value for the DATA statement is assigned (Numeric or character variables)

Explanation The READ statement should always be used with the DATA statement. The READ statement reads the data of the DATA statement starting from the beginning, and assigns one data to one variable.

Both numeric and character variables are available for the variable in the READ statement. However, if the variable type in the READ statement does not match the data type in the DATA statement, a “Syntax error” occurs. When the value for the DATA statement is numeric, both numeric variables and character variables are allowed.

One READ statement can read the values for more than one DATA statement, and more than one READ statement can also be used to read the value for one DATA statement. If the number of <variable>s exceeds the number of data of the DATA statement, an “Out of data” error occurs. When the number of <variable>s is less than the number of data of the DATA statement, the next READ statement assigns the data to the variable in the order starting from the first data in the data which has not yet been read. If there are not more READ statements, the remaining data is ignored.

REM/APOSTROPHE (')

Function Enters a comment into the program.

Format REM [<comment>]

Term <comment>: Any character string

Explanation The REM statement is ignored when the program is executed. The REM statement is used for entering a comment into the program to make the program easy to understand.

The REM statement occupies memory and affects the execution speed a little. (It takes a little time to read the comment.)

The REM statement is useful for places that are jumped to when using the GOTO statement and the GOSUB statement. However, the execution starts from the statement following the REM statement.

The REM statement can be used after a colon (:) inserted at the end of the program. However, other statements cannot follow the REM statement, even if a colon (:) is put after it. Everything after a REM statement is regarded as a comment.

An apostrophe (') can be used instead of "REM" or ": REM". The apostrophe (') can also be used for entering a comment after a statement. After the DATA statement the DATA statement cannot be used again. The apostrophe (') in the DATA statement is regarded as data, therefore, ": REM" should be used for writing comments.

RESUME

Function

Terminates an error process, and resumes execution of the program.

Format

RESUME [0] | NEXT | <line number>

Term

<line number>: The line number from which the execution of the program is resumed should be specified.

Explanation

The subroutine defined by the ON ERROR statement processes an error which occurs while the program is being executed. The RESUME statement is used as the termination declaration of the subroutine.

Three options can be selected from for the RESUME statement, where the execution is returned when the process is terminated.

1) RESUME or RESUME0

   The execution resumes from the statement in which the error occurred.

2) RESUME NEXT

   The execution resumes from the statement following the statement in which the error occurred.

3) RESUME <line number>

   The execution resumes from the line number specified in <line number>.

   If the RESUME statement is used in other than the error process subroutine, “RESUME without error” is displayed. The RETURN statement cannot be used instead of the RESUME statement.

RETURN

Function Declares the end of a subroutine, and returns the execution to the location where the subroutine was called up.

Format RETURN

Explanation The RETURN statement should be placed at the end of the subroutine called by the GOSUB statement. When the RETURN statement is executed, the execution proceeds from the statement following the GOSUB statement which called the subroutine.

RIGHT$

Function Provides a character string of the length specified on the right side of the character string.

Format RIGHT$ (<character string>, <numeric expression>)

Term <file number>: Any character string

<numeric expression>: The length is specified on the right side of <character string>. (Unit: bytes, Range: 0 to 255)

Explanation <numeric expression> is converted to an integer value rounded off to a whole number before it is evaluated.

When the value is 0, a null string is provided. When the value for <numeric expression> is larger than the length of <character string>, the whole character string is provided.

SGN

Function	Provides a sign for <numeric expression>
Format	SGN (<numeric expression>)
Explanation	If <numeric expression> is positive, 0, and negative, 1, 0, -1 is provided, respectively.

SPACE$

Function	Provides character strings of spaces in the specified length
Format	SPACE$ (<numeric expression>)
Term	<numeric expression>: Indicates the number of spaces
Explanation	Character strings of spaces for the number specified in <numeric expression> are provided. A value in <numeric expression> is an integer in which a decimal is rounded off to the nearest integer.

STR$

Function Provides the character string indicating <numeric expression>.

Format STR$ (<numeric expression>)

Explanation The value for <numeric expression> is converted to a character string.
For example, the numeric value, 123, is converted to the character string “123”.

Both integer and real types are available for the value for <numeric expression>.
The STR$ function is the opposite of to the VAL function.

TIMER

Function	Returns the elapse time after the system is reset in a single-precision floating point format. (In units of 100 ms)
Format	X = TIMER
Explanation	A fraction is rounded off to the nearest value.

TIMER ON/STOP

Function

Enables/Stops a timer interrupt.

Format

TIMER ON

TIMER STOP

Explanation

The interrupt specified by the ON TIMER statement is enabled by the TIMER ON statement, or stopped by the TIMER STOP statement.

Before TIMER ON/STOP statements are executed, the interrupt process declaration should be made by the ON TIMER statement.

The TIMER ON statement enables the timer interrupt. When a timer interrupt occurs after the statement is executed, the process routine specified by the ON TIMER statement is executed.

The TIMER STOP statement does not execute the interrupt process immediately when a timer interrupt occurs. However, up to one interrupt is stored. The interrupt process is executed just when the TIMER ON statement is executed later.

VAL

Function Converts a character string to a numeric value.

Format VAL (<character string>)

Explanation The opposite of the STR$ function, the VAL function converts a character string indicating the numeric value for a 1-byte or 2-byte character (ex. “456”) to a numeric value (ex. 456).

Spaces, tabs and line feeds placed at the beginning of the character string are ignored.

VAL (“␣-6”) is converted to the numeric value, -6.

When the character string is not a 1-byte character or 2-byte character, as in the case of a number, Kanji code, hexadecimal number or octal number, the VAL function provides “0”.

WHILE ~ WEND

Function

Repeats the statements included between the WHILE statement and the WEND statement for as long as the given conditions are satisfied.

Format

```
WHILE <expression>
  ⋮
WEND
```

Term

<expression>: The execution is repeated for as long as the <expression> (a numeric value or character) is satisfied.

Explanation

When <expression> is true (other than 0), the steps from WHILE statement to the WEND statement are executed. After the WEND statement is executed, the execution returns to the WHILE statement, and then <expression> is evaluated again. While <expression> is true, the above is repeated. When <expression> is false (0), the execution skips to the statement after the WEND statement.

If <expression> is false when it is first evaluated, between the WHILE statement and the WEND statement is not executed and the program moves on to the step following the WEND statement.

If the WHILE statement and the WEND statement are not paired, a “WHILE Without WEND” error or “WEND without WHILE” error occurs.

WRITE#

Function

Outputs data to a sequential file.

Format

WRITE# <file number>, <expression list>

Term

<file number>: The number assigned to the file by the OPEN statement.

<expression list>: Numeric expressions and character expressions are delimited by a comma (,), semicolon (;), and listed.

<format control character string>: The character string for which the format for outputting data to a file is specified in detail

Explanation

The WRITE# statement functions similar to the PRINT# statement. However, the WRITE# statement is different from the PRINT# statement in the following ways.

- A comma is inserted between expressions to be output
- When the expression is a character string, it is enclosed with double quotation marks (”).

The WRITE# statement does not output the unnecessary spaces. Therefore, area can be saved because the used area in the file is smaller than that of PRINT# statement.

The WRITE# statement does not insert a space for a sign if the value is positive.

The carriage return (CR) and line feed (LF) are output after <expression list> is output.

KEY ENTRY CODE LIST

Key Name	Keyboard Entry Code (Pressed only the key)	Keyboard Entry Code (Pressed together with the SHIFT key)
0	30H	B0H
1	31H	B1H
2	32H	B2H
3	33H	B3H
4	34H	B4H
5	35H	B5H
6	36H	B6H
7	37H	B7H
8	38H	B8H
9	39H	B9H
.	2EH	AEH
ENTER/PRINT	0DH	8DH
→	60H	E0H
←	61H	E1H
↑	62H	E2H
↓	63H	E3H
C	64H	E4H
FEED	65H	E5H
PAUSE	66H	E6H
RESTART	67H	E7H
FORMAT	68H	E8H
MODE	69H	E9H
QUANTITY	6AH	EAH
AMEND	6BH	EBH
F1	70H	F0H
F2	71H	F1H
F3	72H	F2H
SHIFT	–	–

RESTORE

Function

Returns the starting position where the DATA statement is read by the READ statement to the beginning of the program.

Format

RESTORE

Explanation

When the READ statement is placed after the RESTORE statement, the READ statement starts finding the DATA statement from the beginning of the program, and starts reading the first DATA statement.

SENDCMD

Function

Sends a command packet of character strings to the printer, and makes a response to an ACK or a NAK from the printer.

Format

SENDCMD <expression list>

Term

<expression list>: List of character expressions or character strings
(A character string should be enclosed in quotation marks (”).)

Explanation

This statement adds “STX” to the beginning of character strings, adds the command length, adds BCC data to the end of character strings, and then sends a command packet through COM1 (fixed). After a command packet is sent, this statement waits for an ACK from the printer. If it receives a NAK, the statement sends the command packet again. The max. length of command character string is 252 bytes.

Ex. 1

```
SENDCMD”{WU|}”
```

Ex. 2

```
A$=”{WU|}”
```

Ex. 3

```
CNT$=”0010”
SENDCMD”{XS;I,”+CNT$+”,0002C4101|}”
```

ERROR CODE TABLE

Code	Meaning
1	There is no FOR statement for the NEXT statement, or the number of NEXT statements does not match with (is more than) the number of FOR statements.
2	The program is not in accordance with the grammar. Statements which are not defined are included in the program.
3	The GOSUB statement and RETURN statement do not match each other properly.
4	Data to be read by the READ statement is not included in the DATA statement, or the amount of data is small.
5	The process to call a function is wrong in a statement or a function.
6	The input value or the calculation result exceeded the allowable range.
7	The memory capacity is short since the program is too long, or the array is too large.
8	The specified line number was not found. The line number is over the range specified.
9	The specified number of subscripts for the array variable exceeds the allowable max. number.
10	Attempted to define an array or user variable twice.
11	0 is specified for the divisor in division.
13	The data format such as the left and right part of an expression, and an argument for a function, does not match.
20	RESUME was used in a place other than the error routine.
22	The parameter required in the statement is not specified.
30	The number of WEND statements is more than the number of WHILE statements.
52	A file number which is not opened or cannot be opened, was specified.
53	Attempted to read a non-existent file.
54	Access was attempted in the unavailable mode.
55	Attempted to open an already-opened file.
57	Writing was performed in the write disable area.
61	There is no free space in the flash memory, however, creation or expansion of the file was attempted.
62	After all data in the file has been read, the INPUT statement for the file was executed.