PREFACE

This manual is a reference for all implementations of GW-BASIC* working on the operating system (MS-DOS*) of Canon AS-100 systems.

The manual is divided into three chapters plus some appendices.

Chapter 1 covers a variety of topics when you will need to know before you actually start programming. Much of the information pertains to data representation when using GW-BASIC.

Chapter 2 contains the syntax and semantics of every command and statement in GW-BASIC, ordered alphabetically.

Chapter 3 contains the syntax and semantics of every function and variable in GW-BASIC, ordered alphabetically.

The appendices contain other useful information, such as a list of error messages and codes, summary of GW-BASIC compiler, ASCII character code table, and hard copy.

*MS-DOS and GW-BASIC are registered trademarks of Microsoft, Inc.

CONTENTS

CHAPTER 2 GW-BASIC COMMANDS AND STATEMENTS

CHAPTER 3 GW-BASIC FUNCTIONS AND VARIABLES

Chapter I

GENERAL INFORMATION ABOUT GW-BASIC

Canon AS-100

CHAPTER 1

GENERAL INFORMATION ABOUT GW-BASIC

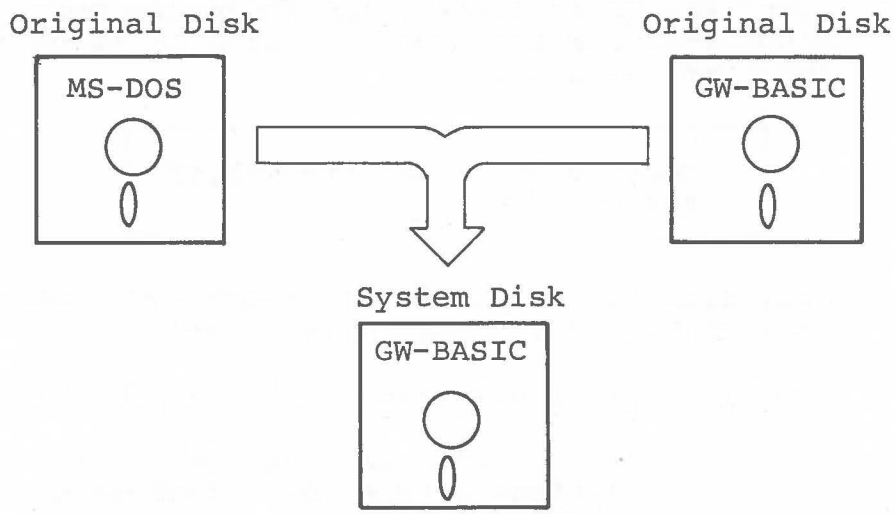
1.1 Activating GW-BASIC

1.1.1 System Disk Generation

To make use of GW-BASIC, it is necessary in the first place to generate a system disk for GW-BASIC from the original disk of MS-DOS and GW-BASIC.

The generation of a system disk for GW-BASIC (hereinafter referred to as system disk) is realized by copying all the system programs of MS-DOS all the modules of GW-BASIC into a new disk in which nothing is written.

Fig. 1.1 System Disk Generation



Generate a system disk by the following operations.

(1) Disk formatting for a system disk.

(2) Volume copy of the MS-DOS original disk.

(3) Copy GW-BASIC modules.

If an error occurs during operation, see the MS-DOS User's Manual.

(1) Disk Formatting

The formatting is required for a disk in which nothing is written, prior to use. This formatting is required for newly purchased disk (excluding the one in which software is stored) prior to use in AS-100.

The formatting consists of checking a fresh unused disk and dividing the disk into sections in a form so predetermined that it is allowed to read from and write into this disk in AS-100. For this disk formatting, the FORMAT command is used.

The operating procedure of the disk formatting is as follows:

1) Set the MS-DOS original disk in drive A.

2) In case the 8-inch floppy disk is used, turn on the power of the floppy disk unit.

3) Turn on the power of the display unit.

4) After MS-DOS is loaded, the following message will be displayed, and the operation will be ready for the input of the date.

```
Current date is XXX mm-dd-yyyy
Enter new date: _
```

According to the format of mm-dd-yyyy, enter the date. (mm: Month; dd: Date; yyyy: Year)

Example: Enter new date: [0][1][-][0][1][-][1][9][8][3][↵]

In the above example, [↵] indicates the carriage return key. Instead of [↵], [ENTER] can be used.

5) The following message will be displayed, and the operation will be ready for the input of the time:

```
Current time is hh:mm:ss.ss
Enter new time: _
```

According to the format of hh:mm:ss.ss, enter the time. (hh: Hour; mm: Minute; ss.ss: Second)

Example: Enter new time: [1][0][:][3][0][:][0][0][.][0][0][↵]

6) When "A > _" is displayed, enter [F] [O] [R] [M] [A] [T] [␣] [B] [:] [↵]

([␣] indicates the space bar.)

7) The following message will be displayed:

```
Diskette formatter Vx.xx

Insert new diskette for drive B:
and strike any key when ready _
```

8) Set a disk to be formatted in Drive B.

9) Strike any key whatever.
Here, if [CTRL] + [C] ([C] is depressed while [CTRL] is being depressed) is entered, the operation will return to the display of "A > _" without execution of disk formatting.

10) When disk formatting normally comes to an end, the following message will be displayed, and the operation will be ready for a key input.

```
xxxxxx bytes total disk space
xxxxxx bytes available on disk

Format another (Y/N)?_
```

Here, enter [N] [↵] . The operation will return to the display of "A > _".

Note: Do not open the doors of the disk drives absolutely during execution of disk formatting. For disk formatting for any purpose other than system disk generation, use the generated GW-BASIC system disk instead of the MS-DOS original disk.

(2) Volume Copy of MS-DOS Original Disk

The volume copy referrs to copying a disk. Here, the MS-DOS original disk is copied. For the volume copy of the MS-DOS original disk, the DISKCOPY command is used.

The operating procedure of the volume copy is as shown on the next page:

1) Continued from the disk formatting described in the preceding paragraph.

Drive A ··· MS-DOS original disk
Drive B ··· Formatted disk
Display ··· A > _

2) Enter [D] [I] [S] [K] [C] [O] [P] [Y] [␣] [A] [:] [␣] [B] [:] [↵] .

3) The following message will be displayed, and the operation will be ready for a key input:

```
Diskette copy Vx.x (C) CANON Inc.

Buffer size:  xxx K Bytes *2

Insert source diskette for drive A:
& insert target diskette for drive B:
and strike any key when ready _
```

4) Strike any key whatever.
Here, when [CTRL] + [C] is entered, the operation will return to the display of "A > _" without execution of volume copy.

5) When volume copy normally comes to an end, the following message will be displayed, and the operation will be ready for a key input:

```
Copy another disk (Y/N)?_
```

Here, enter [N] [↵] . The operation will return to the display of "A > _".

Note: It takes 2 minutes and 30 seconds for the volume copy of the MS-DOS original disk. Do not open the doors of the disk drives absolutely during execution of volume copy. In addition to this, the DISKCOPY command cannot be used between disks of different sizes (namely, between a 5-inch mini-floppy disk and an 8-inch floppy disk).

(3) Copying GW-BASIC modules

Then, transfer all the modules of GW-BASIC to that copy of the MS-DOS original disk which was prepared in the preceding paragraph.

Unlike the volume copy described in the preceding paragraph, the COPY command is used for this work to transfer all the modules of GW-BASIC to the MS-DOS system disk.

The operating procedure of the copying of GW-BASIC modules is as follows:

1) Continued from the volume copy described in the preceding paragraph.

   Drive A ... MS-DOS original disk
   Drive B ... MS-DOS system disk
   Display ... A > _

2) Take the MS-DOS original disk out of Drive A, and set the GW-BASIC original disk instead.

3) Enter [C] [O] [P] [Y] [␣] [A] [:] [*] [.] [*] [␣] [B] [:] [↵] .

4) When the copying normally comes to an end, "A > _" will be displayed.

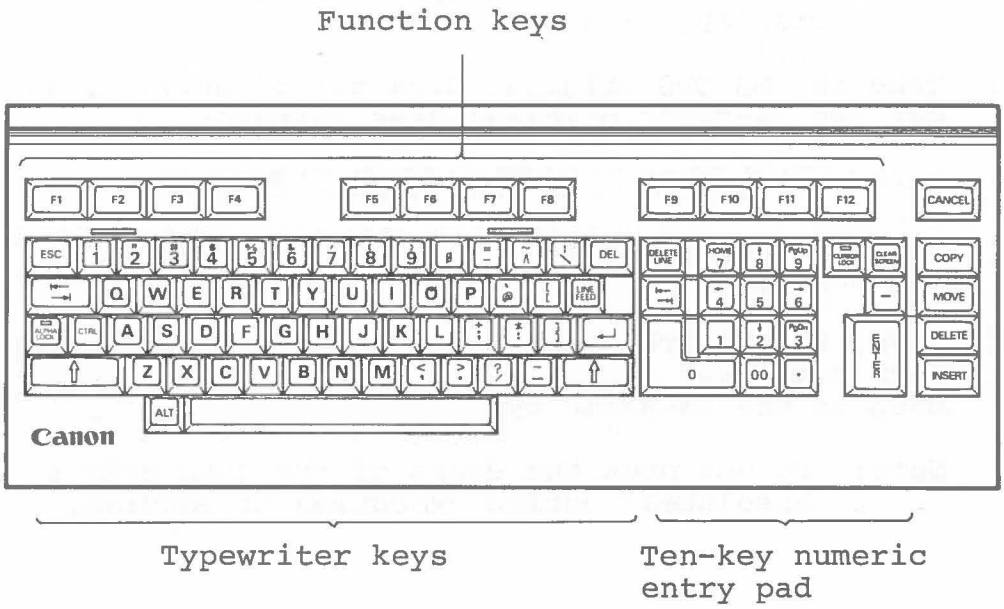
5) Thus, the entire work of system disk generation has been completed, and the disk set in Drive B will be used as the GW-BASIC system disk.

   Note: Do not open the doors of the disk drives absolutely during execution of copying.

1.1.2 Keyboard Configuration

The keyboard of AS-100 is configurated of a typewriter keys, a ten-key numeric entry pad and a function keys, as shown in the figure below.

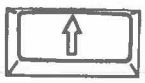
Fig. 1.2 Keyboard Configuration



(1) Typewriter Keys

The character engraved on each key top is entered in accordance with the condition of the key to select the following input mode:

(Alphabet lock key): In the ON state, it turns the keyboard into the alphabet lock mode. The ON state is changed over to/from the OFF state every time the key is once depressed.
In the ON state, the lamp inside the key lights up, and in the OFF state, it goes out.



(Shift key): When the key is depressed, the keyboard is turned into the shift mode.

a. Normal mode ··· When the alphabet lock key is in the OFF state and in the state where a

shift key is not depressed, alphabetical lower-case characters, numerical characters and symbols can be entered.

Example: [! / 1] ⇒ 1 [A] ⇒ a [+ / ;] ⇒ ;

b. Alphabet lock mode ··· Alphabetical upper-case characters, numerical characters and symbols can be entered.

Example: [! / 1] ⇒ 1 [A] ⇒ A [+ / ;] ⇒ ;

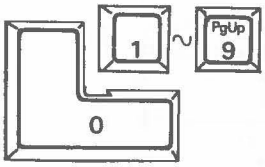
c. Shift mode ···· Alphabetical upper-case characters and symbols can be entered.

Example: [! / 1] ⇒ ! [A] ⇒ A [+ / ;] ⇒ +

(2) Ten-key Numeric Entry Pad

In the ten-key numeric entry pad, the entry of numeric values as well as the cursor control can be carried out.

(Delete line key): It deletes one entered line and returns the cursor to the top of the line.

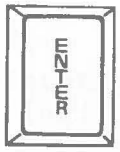


(Numeric key): It allows to enter a numeric of 0 through 9. In the cursor control mode, in addition, it provides the cursor control function.

(Two-zero key) : The entry same as that by depressing the [0] key two times is carried out.

(Decimal point key): It enters a deciman point.

(Minus key) : It enters the minus sign.



(Enter key) : It is depressed to bring entry to an end. The execution of each command is started by the input of this key. It is provided with the function same as the carriage return key in the typewriter keys.

(Cursor lock key): In the ON state, it turns the numeric key into the cursor control mode. The ON state is changed over to/from the OFF state every time the key is once depressed. In the ON state the lamp inside the key lights up, and in the OFF state it goes out.

In the cursor control mode, the cursor moves by one character to the direction indicated by any of the arrows at the [↓2], [←4], [→6] and [↑8] key tops. In addition to this, when the [HOME 7] key is depressed, the cursor will move to the home position.

(3) Function Keys

The function keys consist of twelve keys: F1 through F12. The user can define the contents of the keys by using KEY statement. When GW-BASIC is in activity, the contents of the twelve keys are defined as shown in the table below.

Key	Defined keyword	Key	Defined keyword
F1	LIST ␣	F7	TRON ↵
F2	RUN ↵	F8	TROFF ↵
F3	LOAD"	F9	LLIST ␣
F4	SAVE"	F10	EDIT ␣
F5	CONT ↵	F11	FILES ↵
F6	,"LPT1:" ↵	F12	CHR$(

(4) Special Keys

The description of the functions of special keys are given below:

CANCEL (Cancel key)	:	It interrupts the execution of a program or processing and turns the operation into a state ready for a command input.
CTRL (Control key)	:	It is used in combination with another key. For the function in this case, refer to page 10.
ALT (Alternate key)	:	It is used in combination with another key. For the function in this case, refer to page 12.
DEL (DEL key)	:	It deletes one character immediately before the cursor and moves the cursor backwords by one digit to the left, every time it is once depressed.
⇤ ⇥ (Tab key)	:	It moves the cursor by eight digits to the right.
LINE FEED (Line Feed key)	:	It moves the cursor to the top of the next line.
CLEAR SCREEN (Clear Screen key)	:	It clears the current display on the screen and moves the cursor to the home position (the left uppermost corner on the screen)
DELETE (Delete key)	:	It deletes one character in the cursor position, and shifts the character string following it by one digit to the left.

INSERT

(Insert key)

: When it is once depressed, the operation will be turned into the Insert mode and the character string entered in succession will be inserted in the cursor position. When it is depressed once more or when the cursor is moved by using the cursor control key, the Insert mode will be released.

- Function of CTRL key

Operation: Depress any of the following keys while depressing the control key.

Key	Function	Hexa. code	Equivalent function key
A	It displays the lastly entered command (within one line).	01	
B	It moves the cursor to the top of the word immediately before it.	02	⇧ ← 4 (Cursol control mode)
C	It interrupts the execution of a program.	03	
E	It deletes one line succeeding the cursor.	05	
F	It moves the cursor to the top of the next word.	06	⇧ → 6 (Cursor control mode)
G	It generates buzzer sound.	07	
H	It deletes one character immediately before the cursor and moves the cursor backwords by one character to the left.	08	DEL
I	It moves the cursor to the left by eight characters.	09	⇤

Key	Function	Hexa. code	Equivalent function key
J	It moves the cursor to the top of the next line.	0A	LINE FEED
K	It moves the cursor to the home position.	0B	HOME 7 (Cursor control mode)
L	It clears the current display on the screen and moves the cursor to the home position.	0C	CLEAR SCREEN
M	Carriage return.	0D	↵ , ENTER
N	It moves the cursor to the end of the line.	0E	
Q	It restarts the execution of a program that was suspended by depressing CTRL + S.	11	
R	It changes over the Insert mode to/from the normal mode.	12	INSERT
S	It brings the execution of a program to a temporary half. (Restart of execution: CTRL + Q)	13	
T	It displays the contents of the character string defined for the function keys in the lowermost line on the screen.	14	
U	It deletes the line in which the cursor is positioned.	15	DELETE LINE
W	It deletes one word immediately after the cursor.	17	
Z	It throughly deletes the display succeeding the cursor up to right lowermost corner on the screen.	1A	

Key	Function	Hexa. code	Equivalent function key
[|/]	It moves the cursor to the right by one character.	1C	[→6] (Cursor control mode)
[}]	It moves the cursor to the left by one character.	1D	[←4] (Cursor control mode)
[~^]	It moves the cursor upwards by one character.	1E	[↑8] (Cursor control mode)
[_]	It moves the cursor downwards by one character.	1F	[↓2] (Cursor control mode)

- Keyword of [ALT] key

Operation: Depress any of the following keys while depressing the alternate key.

Key	Keyword
[A]	AUTO
[B]	BSAVE
[C]	COLOR
[D]	DELETE
[E]	ELSE
[F]	FOR
[G]	GOTO
[H]	HEX$
[I]	INPUT
[J]	J
[K]	KEY
[L]	LOCATE
[M]	MERGE

Key	Keyword
[N]	NEXT
[O]	OPEN
[P]	PRINT
[Q]	Q
[R]	RETURN
[S]	SCREEN
[T]	THEN
[U]	USING
[V]	VAL
[W]	WIDTH
[X]	XOR
[Y]	Y
[Z]	Z

(5) Repeat Function and Click Sound

The keys for character input and numeric input as well as the cursor control key are entered in succession by continuously depressing them. This is referred to as Repeat function. Moreover, when each key is depressed, a click sound is generated, so that it can be acoustically confirmed that the key has been depressed.

(6) Leading Input Function

The leading input function refers to the function of receiving key input in advance even in a state where the characters entered from the keyboard have not yet displayed on the screen. Thus for instance, it is possible to enter the next instruction from the keyboard even while the CPU is executing a certain program. However, when [CANCEL] or [CTRL] + [C] is entered, the contents of the leading input that have been made so far will be cancelled.

However, if it is tried to perform leading input while a peripheral unit is in operation, the leading input will not normally be performed from time to time, so care should be exercised.

1.1.3 Activating GW-BASIC

GW-BASIC is activated by the following procedure:

1) Set the GW-BASIC system disk in Drive A.

2) In case the 8-inch floppy disk is used, turn on the power of the floppy disk unit.

3) Turn on the power of the display unit.

4) After MS-DOS is loaded, the following message will be displayed, and the operation will be ready for the input of the date.

```
Current date xxx mm-dd-yyy
Enter new date: _
```

According to the format of mm-dd-yyyy, enter the date. (mm: Month; dd: Date; yyyy: Year)

5) The following message will be displayed, and the operation will be ready for the input of the time.

```
Current time is hh:mm:ss.ss
Enter new time: _
```

According to the format of hh:mm:ss.ss, etner the time. (hh: Hour; mm: Minute; ss.ss: Second)

6) The following message will be displayed, and the operation will be ready for the input of a command.

```
Canon AS-100 MS-DOS Version x.xx
COPYRIGHT (C) by Microsoft 1982, all rights
reversed

BIOS (A) Vx.xx by Canon Inc.

A >_
```

7) Enter [G] [W] [B] [A] [S] [I] [C] [↵] .

8) GW-BASIC will be loaded, and the following message will be displayed.

```
Canon Personal Computer
Advanced BASIC-86 Version x.x
(C) Canon, Microsoft 1982
Created:  mm-dd-yy
xxxxxx Bytes free

Ok

_
```

9) In this state, thus, the activation of the GW-BASIC system has been completed, and it is possible to enter various kinds of commands.

1.1.4 Usable Characters

(1) Character Set

The GW-BASIC character set consists of alphabetic characters, numeric characters and special characters. These are the characters which GW-BASIC recognized. There are many characters which can be printed or displayed although they have no particular meaning to GW-BASIC.

The alphabetic characters in GW-BASIC are the uppercase and lowercase letters of the alphabet.

The numeric characters in GW-BASIC are the digits 0 through 9.

The following special characters have a special meaning GW-BASIC:

Character	Name
	blank
=	equal sign or assignment symbol
+	plus sign or concatenation symbol
-	minus sign
*	asterisk or multiplication symbol
/	slash or division symbol
\	backslash or integer division symbol
^	up arrow or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent
#	number (or pound) sign
$	dollar sign
!	exclamation point
&	ampersand
,	comma
.	period or decimal point
'	single quotation mark (apostrophe)
;	semicolon
:	colon
?	question mark

Character	Name
<	less than
>	greater than
"	double quotation mark
_	underline

(2) Reserved Words

Certain words have special meaning to GW-BASIC. These words are called reserved words. Reserved words include all GW-BASIC commands, statements, function names, and operator names. Reserved words may not be used as variable names. You should always separate reserved words from data or other parts of a GW-BASIC statement using spaces, or other special characters as allowed by the syntax. That is, the reserved words must be appropriately delimited so that GW-BASIC will recognize them.

The following is a list of all the reserved words in GW-BASIC.

Reserved Words

ABS	AND	ASC	ATN	AUTO
BASE	BEEP	BLOAD	BSAVE	CALL
CDBL	CHAIN	CHR$	CINT	CIRCLE
CLEAR	CLOSE	CLS	COLOR	COM
COMMON	CONT	COS	CSNG	CSRLIN
CVD	CVI	CVS	DATA	DATE$
DEF	DEFDBL	DEFINT	DEFSNG	DEFSTR
DELETE	DIM	DRAW	EDIT	ELSE
END	EOF	EQV	ERASE	ERL
ERR	ERROR	EXP	FIELD	FILES
FIX	FNxxxx	FOR	FRE	GET
GOSUB	GOTO	HEX$	IF	IMP
INKEY$	INP	INPUT	INPUT#	INPUT$
INSTR	INT	KEY	KILL	LEFT$
LEN	LET	LINE	LIST	LLIST
LOAD	LOC	LOCATE	LOF	LOG
LPOS	LPRINT	LSET	MERGE	MID$
MKD$	MKI$	MKS$	MOD	MOTOR

NAME	NEW	NEXT	NOT	OCT$
OFF	ON	OPEN	OPTION	OR
OUT	PALETTE	PAINT	PEEK	PEN
PLAY	POINT	POKE	POS	PRESET
PRINT	PRINT#	PSET	PUT	RANDOMIZE
READ	REM	RENUM	RESET	RESTORE
RESUME	RETURN	RIGHT$	RND	RSET
RUN	SAVE	SEG	SCREEN	SGN
SIN	SOUND	SPACE$	SPC	SQR
STEP	STICK	STOP	STR$	STRIG
STRING$	SWAP	SYSTEM	TAB	TAN
THEN	TIME$	TO	TROFF	TRON
USING	USR	VAL	VARPTR	WAIT
WEND	WHILE	WIDTH	WRITE	WRITE#
XOR				

1.2 Programming in GW-BASIC

1.2.1 Foundation of Programming

(1) Editor and Interpreter of GW-BASIC

GW-BASIC includes the eidtor for creating the program in the so-called BASIC language and the interpreter which interpretes and executes the program (source program) written by the editor.

Thus, it is that this BASIC is used in such a way that the editor is used to create a program and the interpreter is called to execute the program, though the commands (statements) for the editor and those for the interpreter are dealt with at the same level in this BASIC so that there is not so distinctive difference between them; rather it can be stated that, in this BASIC, the functions of the editor and interpreter are elaborately jointed together to allow everything to be processed in one processing system.

Although, therefore, such terms as BASIC, editor, interpreter, statement, command, etc. are used in the following chapters, they are to be used in the above-described sense, and it should be noted that they are not always used under strict definition; it is probable that sometimes the word "command" will be used to mean the same as the word "statement" and the term of BASIC will refer sometimes to BASIC as a program language and sometimes to the BASIC processing system.

(2) Command and Statement

The word of command is used to mean the instruction for the computer. The command is divided into the direct command which is used to command the computer to do something immediately (the direct mode), and the indirect command which is used in such a way that commands are written following line numbers and they are executed in the sequence of the line numbers (the programming mode).

The indirect command with a line number is referred to as statement, and a series of statements are called program. They are, for example, in a relation, as shown on the next page.

```
Ok
LIST                    <------ Command
10 FOR N=1 TO 10        <------ Statement  )
20 PRINT N;                                 } Program
30 NEXT N                                   )
Ok
RUN                     <------ Command
 1  2  3  4  5  6  7  8  9  10  <------ Execution result
Ok
```

LIST is a command to output the program list. And then, a group of statements (program) with line numbers are outputted in accordance with the command.

RUN is a command to execute this program. And the execution result is displayed in succession. When RUN command is given, the computer (the BASIC interpreter) interpretes the statements of the program from lower line numbers and executes them sequentially. After completion of the program execution, the operation is again ready for the input of a command.

(3) Syntax of GW-BASIC

The basic construction of so-called statements in GW-BASIC is as follows:

To the statements in BASIC, line numbers are always assigned. Normally, the statements are interpreted and executed from lower line numbers. In addition to this, the line number is used to designate the destination of conditional jump and unconditional jump, and moreover, the designation of the jump of a subroutine is designated by the line number. In GW-BASIC, the flow of a program is determined by the line number.

a. Line Format

For the line in BASIC, the format is determined as follows:

<Line No.> <Statement>:<Statement>: ··· :<Statement>

Statements are separated from one another by a colon (:). The line containing a plural number of statements is referred to as a line of multi-statements, though one line can contain one statement. A statement or statements constituted of up to 254 characters, including line number, space and colon (:), can be written in one line.

The statements in GW-BASIC is divided into two kinds: executable and non-executable statements. The executable statement is such an imperative statement as PRINT, INPUT, FOR-NEXT, etc. On the other hand, the non-executable statement is disregarded

and the control skips to the next statement. In the non-executable statements, the comment statement, such as REM statement, to facilitate writing a program, and the statement such as DATA statement, to store not instructions but data.

b. Line Number

As the line number attached to the top of a statement, an integer from 0 to 65529 is used. The BASIC interpreter executes statements from lower line numbers.

(4) Procedure of Programming

The procedure of programming consists of program preparation, correction and execution. It applies not only to BASIC but also to the whole programming in general. The preparation of a program is carried out by the editor in BASIC. In this case, if a program with no error can be written at a time, no correction of the program will be required. In actuality, however, that is rarely the case. Rather, it is the correction of a program (called debugging) that takes most time in programming. To facilitate this debugging, the function (such as TRON, TROFF, error message, etc.) to trace where errors (bugs) are located is provided.

The execution of a program is carried out by the RUN command.

1.2.2 Program Preparation

(1) Editor Function of GW-BASIC

The preparation and correction of a program are carried out by the editor. Since the editor is incorporated in BASIC, it can be used without being aware of it in particular.

Although the text editor of MS-DOS can be used to prepare a program in BASIC, which will be stored in a file and later be executed by BASIC, this method is inadvisable, since the editor of BASIC is generally more powerful and easier to use.

Any special procedure is not required to call the editor of BASIC. When the key input of a numeric (line number) is first made in the direct mode, a succeeding character string will be interpreted as a BASIC statement and stored in the memory.

For example, if the key input:

```
10 REM CANON ↵
```

(Note: This means that [1], [0], [Space], [R], [E], [M], [Space], [C], [A], [N], [O], [N], [↵] are entered in order. Hereinafter, [↵] indicates the carriage return key.)

is made, then the editor of BASIC stores in the memory the REM statement (which will be described in detail) of Line No. 10.

(2) Inputting Program

Try to make the key input of the following statement in the same way:

```
20 REM THIS IS SAMPLE PROGRAM ↵
30 REM WRITTEN BY TOM ↵
```

Thus, the BASIC statements of Line Nos. 10 through 30 have been stored in the memory. The BASIC statements stored in the memory can be listed on the screen by the LIST command.

```
LIST ↵
```

The result will be as follows:

```
10 REM CANON
20 REM THIS IS SAMPLE PROGRAM
30 REM WRITTEN BY TOM
```

The program input will be easier when the AUTO command is used.

The key input:

```
AUTO ↵
```

will result in the following:

```
Ok
AUTO
10
```

Make the key input of [R], [E], [M], [Space], [C], [A], [N], [O], [N], [↵] in the same way as the previous one.

```
Ok
AUTO
10 REM CANON
20
```

The next line number is automatically displayed, and the operation is ready for the input of the next statement. In the above case, the line number automatically increased by 10, though this increment can be set to any number by appropriate designation,

which will later be described in detail.

Every input in the case of the AUTO mode is interpreted as a BASIC statement. Therefore, it should be noted that, in the AUTO mode, direct execution of such commands as LIST, RUN, etc. is infeasible. To cancel the AUTO mode, depress [CTRL] + [C].

(3) Full Screen Editor

In GW-BASIC, the preparation and correction of a program can easily be carried out on the screen. This function is referred to as full screen editor function.

Thanks to this editor function, every line displayed on CRT can easily be changed or corrected by operating the cursor control key or other keys.

a. Change of Character

To change the character immediately before the cursor, depress the [DEL] key, and one character preceding the cursor will be deleted and the cursor will move to that position, so enter a new character.

To change a character apart from the cursor, move the cursor to the character to be changed, using any of [←4], [→6], [↑8], [↓2] in the cursor control mode, and enter a new character.

The cursor control key in the ten-key numeric entry pad can be used in cursor control mode.

b. Deletion of Character

After moving the cursor onto the character to be deleted, depress the [DELETE] key, and that character will be deleted, and the character string on the right of the cursor will shift to the left by one character and come to the position of the cursor, so as to fill in the blank. The position of the cursor will remain in the same position as before.

The [DEL] key is also used to delete the character. In this case, the character immediately before the cursor is deleted, and the character string on the right of the position of the cursor shifts by one character to the left. And when there is no character immediately before the cursor, that is, when the cursor has come to the left end, the characters on the right of the position of the cursor will be deleted in the same way as in the case of the [DELETE] key.

Thus, the function of the [DELETE] key and that of the [DEL] key differ from each other, though they appear to resemble each

other, so use them properly case by case.

c. Insertion of Character

Move the cursor onto the character immediately after the position in which a character is to be inserted, and then depress the [INSERT] key; the Insert mode will be realized, and the characters to be entered hereafter will be inserted in front of the cursor. To release the Insert mode, depress the [INSERT] key again, or enter either the cursor control key or the carriage return key.

To move from the line in which character change, deletion or insertion has been carried out to another line, enter the carriage return key. A corrected line will not be stored in the memory unless the carriage return key is depressed.

d. Editing Keys

The keys useful to edit are shown below. In addition to them, there are available the keys to be used in combination with the [CTRL] key. For further detail, refer to "1.1.2 Keyboard Configuration".

· To move the cursor by one word:

The input of [CTRL] + [F] will move the cursor onto the character at the top of the word on the right in that line. The input of [CTRL] + [B] will move the cursor to the left by one word. This is useful when the cursor is to be moved widely to the horizontal direction.

· To move the cursor to the home position:

The input of [CTRL] + [K] will move the cursor to the left uppermost position on the screen. In this case, the display will not be deleted.

· To delete characters from the position of the cursor to the end of the line:

The input of [CTRL] + [E] will delete the characters from the cursor position at that time to the end of that line. The cursor will not move.

· To delete a line:

The input of [CTRL] + [U] will delete all the characters in the line in which the cursor is located and move the cursor to the top of that line. This is the deletion on the screen, which differs from the deletion of the "line" of a BASIC program.

• To delete the characters succeeding the cursor line:

The input of [CTRL] + [Z] will delete the characters from the position in which the cursor is currently located to the right lowermost corner on the screen.

(4) Program Editing

The method of basic edit utilizing the full screen editor will be described by the help of examples in the following:

The statement of a BASIC program consists of a line number and statements succeeding it. For the format, the meaning of statements, etc., refer to Chapter 2.

In the following, mainly the method of the actual use of the editor will be described.

a. Commands for Program Editing

The commands in BASIC which are used for program editing are as follows:

• AUTO Command

Every time one program line is entered, it generates the next line number. It is used when a program is initially entered. The operation returns to the command level by the input of [CTRL] + [C].

• DELETE Command

It is used to delete several successive lines at a time.

• EDIT Command

It is used for the edit by displaying required line.

• LIST Command

It displays a program list. The list display is temporarily stopped by depressing [CTRL] + [S], and it is restarted by depressing [CTRL] + [Q]. Since the operation returns to a state ready for a command input by depressing [CTRL] + [C], the screen edit of the list on the screen is feasible.

• NEW Command

It deletes the program and the contents of variables, which are stored in the memory. It is used to enter a program anew.

• RENUM Command

It is used to renumber lines. In this case, all the line numbers that are used in statements will be renewed automatically.

b. Correction of Program

The method of program correction is as follows:

Example:
```
100 FOR I=1 TO 10
110     PRINT "BASIC"
120 NEXT I
```

To change Line 110 in the above example to:

```
110     PRINT "PERSONAL"
```

make a display of the list by means of the LIST command or the EDIT command. Then, move the cursor onto the character of B, depress keys [P], [E], [R], [S], [O], [N], [A], [L], ["] sequentially, and finally depress the [↵] key. If a display of the list is made again, it will be seen that the change has already been done.

Then, let us insert characters into the same line, as follows:

```
110     PRINT "PERSONAL COMPUTER"
```

Move the cursor to the second double quotation mark ("), depress [␣], [C], [O], [M], [P], [U], [T], [E], [R] sequentially after depressing the [INSERT] key, and lastly depress the [↵] key.

To delete one character, use the [DELETE] and [DEL] keys. To delete the characters of PERSONAL, either move the cursor onto the character of P and depress the [DELETE] key successively, or move the cursor after the character of L and depress the [DEL] key successively.

To delete one line of a BASIC program, enter

<Line No.> [↵]

and the line of that line number will be deleted.

To delete several continuous lines, use the DELETE command. For example,

DELETE 100-700 [↵]

will delete all lines from Line No. 100 through Line No. 700.

To insert a line additionally, enter a new line number. When lines of one and same line number are entered more than twice, the one entered lastly will be stored in the memory.

As for the instruction words of BASIC, the trouble of entering the full spell can be saved when the [ALT] key is used. (See "1.1.2 Keyboard Configuration".)

Moreover, the PRINT command can be entered by the key input of question mark (?) instead of PRINT. For example, if

```
110 ? "CANON" ↵
```

is entered in a program, "?" will be converted into "PRINT" in the display of a program list.

When the ↵ key is depressed only with a line number changed, the contents of the original line will be copied in the line of a new line number. For a long statement having contents similar to those of a certain line, if a copy of this line is modified instead of the detailed key input from the beginning of that statement, the time required for its entry will be shorted.

The full screen editor has a pointer indicating the line which is current aimed at. When the ↵ key is depressed after the input, correction, etc. of a certain line have been carried out, the pointer moves to that line. In addition to this, when the LIST command is executed, the pointer will be in the line displayed lastly.

The user can designate the line, in which this pointer is located, by a period '.' to make use of it in the LIST or EDIT command. For example, LIST. ↵ is depressed, the line which is aimed at that time will be displayed.

1.2.3 Program Execution and Debugging

(1) RUN and CONT

The RUN command is used to command the execution of a program. By RUN ↵ , a program is executed from the lowest line, but when a line number is specified following RUN, the program can be executed from any line.

Example: RUN 100 ↵

(When a program is to be executed from the Line 100)

This is used to check the operation of the part of higher line numbers of a program. If an error is found in the midst of a program, BASIC will stop the execution of the program and return to a state ready for a command input. Sometimes errors in the part of higher line numbers of a program are to be checked, disregarding the errors that have already occurred, since the detailed correction of errors will deteriorate the efficiency of debugging. In such a case, use the CONT command, as shown on the next page:

CONT ↵

This command can be used not only when the execution of a program has come to a stop due to the occurrence of an error, but also when the program is stopped by CTRL + C.

However, when the correction of a program has been carried out even once after the execution of the program was stopped, the CONT command cannot be used.

(2) Interruption of Program

When CTRL + C is entered during execution of a program, the operation will be broken, a line number will be displayed, and the control will be ready for a command input. In this case, since the values of variables, etc. are preserved, they can be referred to by a direct command.

By the CONT command, then, the execution can also be restarted from a next instruction.

By inserting a STOP statement into a program, the execution can be interrupted at the positon of that statement. Also in this case, the control will return to a state ready for a command input, and the execution will be restarted from a next instruction by the CONT command.

(3) TRON and TROFF (Trace)

The TRON instruction is used to trace the execution of a program. Normally, it is entered as a direct command, but it can also be used as a statement in a program.

After execution of TRON, the operation will be turned into a Trace mode, and the line numbers of the program that have already been executed will be displayed sequentially in brackets. Therefore, though the execution will be delayed for the display time, it will be possible to know what part of the program is currently being executed.

Bugs are discovered by interrupting the execution by CTRL + C and making sure of the values of variables in the direct mode.

When the STOP statement is used at the same time, more effective debugging may be possible. To release the Trace mode, enter the TROFF instruction.

(4) Debugging

It is very rare that an entered program can at once operate satisfactorily, and it is not too much to say that there are always errors in the program. These errors are generally referred to as bugs, and removing such bugs are called debugging. Bugs are classified into grammatical errors, input mistakes, mistakes of algorithm, etc., and since the BASIC interpreter displays grammatical errors and erroneous use of instructions as errors, they can easily be corrected as you get accustomed to them. However, the errors of algorithm, etc. do not appear as concrete errors.

As the methods of discovering this kind of bugs, the pre-described interruption and trace of the execution of a program are available. These methods will be helpful to early discovery of bugs.

(5) Error and EDIT Command

When an error occurs during execution of a program, the execution will come to a stop, and an error message will be displayed. In this case, the corresponding line can be displayed at once by entering LIST. [↵] . In the same way, the EDIT command displays that line by entering EDIT. [↵] and moves the cursor to the top of the line, so screen edit can promptly be carried out. The difference between the LIST command and the EDIT command in the case where one line is displayed can be regarded to lie only in the position of the cursor.

In this case, moreover, when [CTRL] + [A] is entered, the line indicated in the error message will be displayed in the same way as in the case where EDIT. [↵] is entered.

Especially in the case of a syntax error, BASIC automatically displays the line in which an error has been discovered, and moves the cursor to the top of the line.

(6) Error Message

When an error occurs during execution of a program, BASIC will display an error message and stop the execution of the program. The values of variables, etc. will remain as they are at the time of the interruption. Errors in the program cannot be detected before the execution moves to that line. Therefore, when there are a plural number of errors in the program, the errors in the later part cannot be detected unless either the first error is corrected or the program is executed with the line including that error skipped.

The error message is displayed in the following format:

<Error message> in <Line number>

Example: Syntax error in 30

This display means that there is a syntax error in the Line 30. For the error message, refer to "Error Messages" at the end of this manual. (For error processing, refer to 1.10.2)

1.2.4 Preservation and Reuse of Program

Since the program is prepared by consuming much time and labor and generally it is long, it is troublesome to enter it every time the power switch of the computer is turned on. Therefore, a debugged program is normally stored in a file for preservation. Stored programs can later be called and executed, or later be modified and improved. In addition to this, it is possible to joint a plural number of programs together to prepare a new program.

As the commands used for these purposes, the following are available:

SAVE: Stores a program in a file.

LOAD: Loads a program from a file.

MERGE: Joints the program in the memory with a program in a file.

The actual use of these commands will be described in a later chapter.

1.3 Constant and Variable

1.3.1 Constant

The constant refers to a numeric value or a character string directly used at the time of the execution of a program. Both are distinguished from each other by referring to the constant of the former as numeric constant and that of the latter as character constant.

The numeric constant is classified into floating decimal point type, integer type, etc., or from another point of view, it is grouped according to precision into double precision type and single precision type.

(1) Character Constant

The character string such as the following enclosed by double quotation marks (") is referred to as character constant:

```
"Thank you"
"1982-7-31"
```

It is used mainly in PRINT statements, etc., as follows:

```
Ok
PRINT "How are you?" ↵
How are you?
```

As a character constant, a character string of a length of 0 through 254 characters is allowed. The character constant of a length of 0 ("") is in particular referred to as null string.

(2) Types of Numeric Constant

The numeric constant is designated by +, -, period(.) and digits of 0 through 9. In addition to this, the notation of 40E-2 to express a power as well as the notation with % and # or in such a form as &H2AF0 to designate the type of constant is used.

a. Integer type

The range of integer type constants is from -32768 to 32767. Outside of that range are interpreted as constants of real number type. They can also be distinguished by adding to them at the end the type declaration character % which designates an integer type.

Example: 125, 2742%, -64, -872%

- Hexadecimal constant

The constant having &H at the top means a hexadecimal integer type constant.

Decimal number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal number	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

The range of hexadecimal constants is from -&HFFFF (-65535) to &HFFFF (65535).

Example: &H2AF0 → 10992
-&H1A → -26
&H100 → 256
&HFF+1 → 256 (255+1)

- Octal constant

The constant having &O at the top means an octal integer type constant.

Example: &O123 → 83
&O2000 → 1024
&O1777+1 → 1024 (1023+1)

The range of octal type constants is from -&O177777 (-65535) to &O177777 (65535).

b. Fixed-point type

It is a real number type constant with a decimal point, and a number from approximately 3.4×10^{39} to approximately 1.7×10^{38}.

Example: -364.7, 3682.0, 1876952.324

c. Floating-point type

It is a real number type constant of power notation, and its range is the same as that of the fixed-point type. It is a constant in the form that a fixed-point part is followed by E (Exponential) and then by an exponential part: e.g. 15E-2 = 15×10^{-2}.

Example: -365.42E10, 0.28E3, 1.2987E-7

The fixed-point part should be an integer with a fixed point, and the exponential part should be an integer of -39 through 38.

(3) Precision of Numeric Constant

The real number type constant is classified into single precision type and double precision type according to the precision. There is not such classification in the case of the integer type (including octal numbers and hexadecimal numbers).

a. Single precision type

The real number constant whose significant digits (fixed-point part) is less than 7 digits is regarded as single precision type, and it is displayed up to 6 digits. The precision is guaranteed up to 6 digits. Although it can be distinguished by adding the type declaration character "!" which designates the single precision type, the real number type constant up to 7 digits is regarded as single precision type, even if it has not the type declaration character.

Example: 16.427, 326.42!, 7!, 1021!

b. Double precision type

The real number type constant whose significant digits are 8 though 16 digits is regarded as double precision type. Moreover, the constant of less than 7 digits is regarded as a double precision constant by adding the type declaration character "#" at the end. In the case of the floating-point type constant, though the form of expression is same as that of the single precision type, "D" is used instead of "E" in the exponential part.

Example: 10#, 21.628763521, 116.421#

The symbol "#" is used in the following cases:

```
PRINT 10/7 ↵
1.42857                (Single precision)
PRINT 10#/7 ↵
1.428571428571429      (Double precision)
```

1.3.2 Variable

The variable is used to preserve values during the processing of data in the flow of a program. It can be said that the variable is a specific location in the memory to accommodate the values used in a program. And what is used to identify that specific location is a variable name.

(1) Rule of Variable Name

The variable names that can be used in GW-BASIC is restricted as follows:

1) The top should be an alphabetic character. The characters from the second can be any of alphabetic characters and numeric characters. No space should be inserted in the midst.

2) The variable name should be within 40 characters.

3) The same as reserved words (key words such as commands, statements, etc.) are not allowed. For example, the variable name such as GOTO or IF is not allowed.

4) A type declaration character can be attached to the end of a variable name. (It can be omitted.) A type declaration character (#, %, !, $) sbould not be situated at the top or in the midst of a variable name.

Examples of correct variable names:

```
ABC
DSCC2
E15
```

Examples of erroneous variable names:

```
1DATA   Beginning with a numeric character.
DA#SC   A type declaration character used in the midst.
GOTO    Same name as one of the reserved words.
        (GO is allowed.)
```

(2) Type of Variable

Variables are classified into several types in the same way as in the case of constants. They are classified in the first place into numeric variables and character variables, and the numeric variables are further classified into integer type, single precision type and double precision type.

As the character indicating the type of a variable, the type declaration characters same as those for constants are used, and a type declaration character such as %, !, #, and $ is added to a variable name at the end. It is used as follows: A$, BCD!, D%.

The type declaration character is a symbol which designates to what type the variable belongs.

	Type	Type declaration character	Amount of memory used	Significant digits	Example
Numeric value	Integer	%	2 bytes	-32768 ∿ 32767	A% ABC%
	Single precision	!	4 bytes	6 digits	B! TEST!
	Double precision	#	8 bytes	16 digits	C# XYZ#
Character value	Character	$	0 ∿ 255 bytes		D$ LMN$

The variable having no type declaration character in particular (such as ABC, ESCAP, etc.) is interpreted as a numeric variable of single precision type.

In the table given above, the amount of memory used is the number of bytes secured to store data in a variable. Another memory is required to identify the variable name. Therefore, if a long variable name is used, the program size will be increased.

1.3.3 Array

(1) Array

A group of variables can be ordered. As for variable name A, for example, the variable can be divided into A(0), A(1), A(2), ... Here, A(0) and A(1) can be dealt with as independent variables. Ordering a group of variables in this way is referred to as "setting in array".

To consider what merits are brought about by setting in array, assume the following three sample programs as shown on the next page.

Note: DIM is a statement declaring the size of array in advance.

Sample 1:

```
10 READ A0, A1, A2, A3, A4
20 PRINT A0, A1, A2, A3, A4
30 DATA 26, 33, 15, 22, 47
RUN
     26      33      15      22      47
Ok
```

Sample 2:

```
10 DIM A(4)
20 READ A(0), A(1), A(2), A(3), A(4)
30 PRINT A(0), A(1), A(2), A(3), A(4)
40 DATA 26, 33, 15, 22, 47
RUN
     26      33      15      22      47
Ok
```

Sample 3:

```
10 DIM A(4)
20 FOR I=0 TO 4
30 READ A(I)
40 PRINT A(I),
50 NEXT I
60 DATA 25, 33, 15, 22, 47
RUN
     26      33      15      22      47
Ok
```

Sample 1 is an example in which varaible A0 through A4 are used in an ordinary way. If the same program is written by making use of array, Sample 2 can be obtained. In these two programs, if the number of variables is to be increased, for example, to 20, the following troublesome change will be required:

```
10 READ A0, A1, ..., A20
20 PRINT A0, A1, ..., A20
```

On the other hand, if a FOR - NEXT loop is used as in the case of Sample 3, an extremely compact program can be written.

The method of giving an array name is same as in the case of variables. In addition to this, arrays are divided into types, and the same type declaration characters as those for variables are used.

The variable set in array has the following format:

```
ABC(15)
 ↑
Array name      Subscript
```

when 15 is designated as a subscript, as shown in the above example, there should be at least 16 arrays of ABC(0) through ABC(15). If the OPTION BASE instruction is used, the first array can be ABC(1) instead of ABC(0).

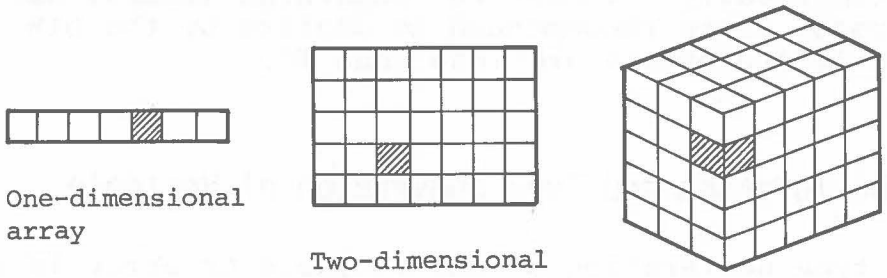
```
Ok
OPTION BASE 1 ↵
```

In this way, the array subscript will start from 1. If it is required to restore the array subscript so as to start from 0, write the following:

```
Ok
CLEAR ↵
```

(2) Multi-dimensional Array

Although the array described in the preceding paragraph was one-dimensional array, array can have any dimensions. The two-dimensional array can be ordered from the front and from the left. Further, the three-dimensional array can be ordered from the top in addition.



One-dimensional array

Two-dimensional Array

Three-dimensional array

Any number of dimensions of array can be allowed so far as the space of the memory admits, however high it may be.

(3) Array Declaration, Implicit Declaration and Memory Capacity

To use a variable to be set in array, the array should be first declared in principle. A few examples of the declaration of array are given below:

One-dimensional array	DIM A(5), B(10)
Two-dimensional array	DIM A(10, 10), B(5, 20)
Three-dimensional array	DIM A(25, 25, 25), B(6, 18, 23)

When array is declared, the BASIC processing system secures the amount of memory corresponding to the number of that array. For A(9, 9), 10×10 namely 100 real number variables of single precision type are secured. They correspond to 400 bytes since 4 bytes are secured per real number variables of single precision type. In the same way, for A(9, 9, 9), the memory of 10×10×10×4 ≒ 4 K bytes is secured. When high-dimensional array is used, attention should be paid to the memory capacity.

In the examples given above, the array only of the variable of single precision type was dealt with, but character type, integer type, double precision type, etc. can be set in array.

Example: A$(10), BDS%(20, 20), BAS#(5, 5, 5), CD!(20, 20)

To set in array, the array is in principle declared by the DIM statement, but in the case of subscripts of less than 10, array can be used without declaration. This implicit declaration of array can be admitted as to any dimensions if subscripts are less than 10.

It should be noted as to implicit declaration that, even if only one variable is used in such a form as A(9, 9, 9), the memory space (approximately 4 K bytes) for 1,000 variables will be secured automatically. Therefore, when high-dimensional array is to be used, it is recommended to declare by the DIM statement even if subscripts are less than 10.

1.3.4 Type Declaration and Type Conversion of Variable

In BASIC, the type declaration of the variable or array is automatically carried out when a variable name is given to it. As can be suspected from the description up to the preceding section, the type declaration has been carried out automatically by attaching a type declaration character to that variable name when the variable was first used.

When no type declaration character is attached in particular, the variable is regarded as a variable of single precision type. Therefore, even if you are not aware of the fact that type declaration is carried out, the type declaration of single precision type will be carried out automatically.

To convert the type of a variable which has once been declared (a variable which has once been used) into another type, an assignment statement and a new variable name are used.

Example: C%=C

Thus, the value of C will be rounded up and the integer part will be stored in C%. Subsequently, if C% is used, operation can be performed in the integer type. In addition to this, since the content of C remains as it is, care should be exercised so that C and C% may not be confused if the type conversion has been carried out as described above. The GW-BASIC interpreter interpretes C and C% as independent variables. Although few problems are posed when the integer type is substituted for the single precision type, it should be noted that sometimes sufficient precision cannot be obtained when the single precision type is converted into the double precision type and round-up is performed when the double precision type is converted into the single precision type.

Another method is available for the declaration of the type of a variable. The type declaration for the capital letter of a variable name can be carried out by declaration statements such as the DEFINT statement, DEFSNG statement, DEFDBL statement, DEFSTR statement. For further details of these declaration statements, refer to Chapter 2.

1.4 Expression and Operators

1.4.1 Expression

The expression is constituted of constants, variables, operators, functions, etc. in combination. The expression is used mainly in the assignment statement, IF statement and PRINT statement.

Example:

```
A = SIN(2) + 1

PRINT A/B

IF A$="Y" THEN 1000 ELSE 1500
```

The expression is classified into two kinds: the expression for the operation of numerical values, and that for the operation of characters. These two kinds of expressions should be distinguished when they are used. It is not allowed that character variables are confused with numeric variables nor that numeric operators are used for character variables.

Erroneous examples:

```
A+B+C$+1 ····· Numeric operation is confused with
               character operation.

B$+VAL(C$)···· VAL(C$) is a numeric function.

SQR(A$)····· The argument of the SQR function
             should be a numeric value.
```

The value of an expression is evaluated from the left to the right, and there is the following rule of priority.

1. Expression enclosed by parentheses.
2. Functions
3. Arithmetic operators (Addition, subtraction, multiplication, division, etc.)
4. Relational operation (Comparison of expression with each other)
5. Logical operation (Evaluation of the result of comparison)

Although operators and functions are mathematically equivalent, the computer distinguishes them from each other. The function has an operand (enclosed by parentheses) following a function name, while the operator is situated either between two operands or in front of one term. Since the operator is a reserved word, the symbols of operators (+, -, /, *, ...) cannot be used in a variable name.

1.4.2 Arithmetic Operators

The arithmetic operator is an operator used to perform the addition, subtraction, multiplication, division, etc. of numeric values, and the following arithmetic operators are provided in GW-BASIC:

Operator	Contents of operation	Example	Priority
∧	Exponentiation	B∧2	①
+, -	Signs	-2+B, B*-2	②
*, /	Multiplication, Floating Point Division	A*C, A/C	③
\	Integer Division	AB\C	④
MOD	Modulo Arithmetic	A MOD B	⑤
+, -	Addition, Subtraction	A+B, A-B	⑥

The rule of the priority of signs is somewhat complicated, so let us describe by the help of examples.

Example: A*(-B)*C → A*-B*C
A^{-3} *B → A^-3*B

When * or / is followed by + or - (normally, -), these + and - are interpreted as signs and have high priorities than those of * and /, as described in the above examples. In such a case, moreover, they have sometimes high priority than ^ (exponentiation), as described above.

When the operator MOD is used, spaces should be positioned before and behind MOD.

```
Erroneous use:  ?14MOD5
                 14  0
                ?14␣MOD5
                 14  0

Correct use:    ?14␣MOD␣5
                 4
```

MOD is an operator which returns a remainder or the residual of division.

The operator (\) performs division by rounding down the digits below the decimal point of the terms before and behind it, and further rounds down the digits below the decimal point of that result.

```
?27.45\4.62     27.45 / 4.62 ⇨ 27 / 4  ⇨ 6
                               Truncation  Truncation
```

1.4.3 Relational Operators

The relational operator compares numeric values as well as character strings. The result will be -1 if the result of operation is true, and 0 if it is false. The kinds of relational operators are as follows:

Operator	Relation Tested	Example
=	Equality	X=Y (X is equal to Y)
<>, ><	Inequality	X<>Y,X><Y (X is not equal to Y)
<	Less than	X<Y (X is less than Y)
>	Greater than	X>Y (X is greater than Y)
<=, =<	Less than or equal to	X<=Y,X=<Y (X is equal to Y or less than Y)
>=, =>	Greater than or equal to	X>=Y,X=>Y (X is equal to Y or greater than Y)

The expression containing relational opeators is used in the IF statement and an expression containing logical operators.

Example:
```
? 12>8, 10=9
-1       0

IF A<B THEN 300 ELSE 400
(If A<B, jump to Line 300, or else to Line 400.)

? 15>8 AND 8>12
 0
```

1.4.4 Logical Operators

They are a group of operators which perform the logical operation of the result of a relational expression. Six kinds of logical operators are available: AND, OR, NOT, XOR, IMP and EQV.

(1) Format of Logical Operators

The logical operator is used in the format as shown on the next page.

```
<Expression> ␣ <Logical operator> ␣ <Expression>
   or  <Logical operator> ␣ <Expression>
   ( ␣ indicates for a space.)
```

<Expression> is generally a relational expression. What contains logical operators can take place of <Expression>.

```
Example:  X<1 ␣ AND ␣ X<Y
          X<5 ␣ AND ␣ X<8 ␣ OR ␣ X=20
          (X<50 ␣ XOR ␣ Y>50) ␣ AND ␣ X*Y<20
```

More than one blank should always be inserted in the positions indicated by ␣ .

(2) Result of Logical Operation and IF Statement

The result of logical operation will be true (-1) or false (0). Therefore, the expression containing logical operators can be used in the IF statement, as follows:

```
IF ␣ <Expression of logical operation> ␣ THEN
     <Line No.> ELSE <Line No.>
```

```
Example:  IF X=1 OR X=2 THEN 150 ELSE 200
```

If the expression directly following IF is -1, then the operation will jump to the line number following THEN, and if it is 0, then the operation will jump to the line number following ELSE.

It is also possible to refer to a long expression of logical operation by the IF statement, using an assignment statement to put the result in one variable.

```
Example:  IF A>10 ␣ AND ␣ A>B THEN 100 ELSE 150

          X=A>10 ␣ AND ␣ A>B
          IF X THEN 100 ELSE 150
```

These two IF statements have the same function.

(3) Logical Operators

a. AND

The logical operator AND is used on the basis of the following truth table. It is an operator which is true (-1) only when both of the terms (X and Y) before and behind it are true.

X	Y	X AND Y
True (-1)	True (-1)	True (-1)
False (0)	True (-1)	False (0)
True (-1)	False (0)	False (0)
False (0)	False (0)	False (0)

b. OR

The logical operator OR is used on the basis of the following truth table. It is an operator which is true (-1) if either of the terms before and behind it is true (-1).

X	Y	X OR Y
True (-1)	True (-1)	True (-1)
False (0)	True (-1)	True (-1)
True (-1)	False (0)	True (-1)
False (0)	False (0)	False (0)

c. NOT

The logical operator NOT is an operator which acts upon only one term, and it negates the term.

X	NOT X
True (-1)	False (0)
False (0)	True (-1)

d. XOR

The logical operator XOR is an operator which means exclusive logical sum, and it is used on the basis of the following truth table:

X	Y	X XOR Y
True (-1)	True (-1)	False (0)
False (0)	True (-1)	True (-1)
True (-1)	False (0)	True (-1)
False (0)	False (0)	False (0)

e. EQV

The logical operator EQV is an operator which judges equivalence. It is equivalent to the negaton of XOR.

X	Y	X EQV Y
True (-1)	True (-1)	True (-1)
False (0)	True (-1)	False (0)
True (-1)	False (0)	False (0)
False (0)	False (0)	True (-1)

f. IMP

The logical operator IMP is an operator which performs the logical operation of implication. It is used on the basis of the following truth table:

X	Y	X IMP Y
True (-1)	True (-1)	True (-1)
False (0)	True (-1)	True (-1)
True (-1)	False (0)	False (0)
False (0)	False (0)	True (-1)

(4) Combination of Logical Operators

Expression having various kinds of logic can be formed by combining logical operators.

Example: (NOT ␣ A) ␣ OR ␣ (NOT ␣ B)

NOT (X>=15 ␣ AND ␣ X<>Y)

It should be noted that logical operators are used in principle to act upon a relational expression as well as a variable for which a relational expression is substituted.

(5) Internal Construction of Logical Operator

Logical operation is performed by carrying out bit invert or arithmetic operation after converting the numeric value of the term (X or Y) before or behind it into the two's complement of 16 bits. Therefore, 23 AND 7, for example, means a certain numeric value. Several examples are given below:

```
-1␣OR␣0  =-1                          16␣EQV␣6  =-23
  -1 = (1111 1111 1111 1111)₂           16 = (0000 0000 0001 0000)₂
   0 = (0000 0000 0000 0000)₂            6 = (0000 0000 0000 0110)₂

23␣AND␣7 =7                           12␣IMP␣5  =-9
  23 = (0000 0000 0001 0111)₂           12 = (0000 0000 0000 1100)₂
   7 = (0000 0000 0000 0111)₂            5 = (0000 0000 0000 0101)₂

16␣XOR␣6  =22                         NOT␣15  =-16
  16 = (0000 0000 0001 0000)₂           15 = (0000 0000 0000 1111)₂
   6 = (0000 0000 0000 0110)₂
```

In the above examples, the operator NOT, for example, performs the arithmetic operation:

```
NOT   X = -(X+1)
```

(6) IF Statement and Expression of Logical Operation

The IF statement is used as follows:

```
IF␣<Expression>␣THEN␣<Line No.>␣ELSE
  <Line No.>
```

Example: IF A>1 THEN A=A+1 ELSE 20

Normally, a relational expression, an expression containing logical operation, or an expression for which they are substituted is used as <Expression>. However, the IF statement functions for other expression. In other words, the IF statement executes the part following THEN when the value of <Expression> is -1 and the part following ELSE in the other cases (for other values than -1).

(7) Priority of Operators

The priority of the operators described above is as follows:

Priority	Operator
1	∧
2	+,- (sign)
3	*,/
4	\
5	MOD
6	+,-
7	=,<>,><,>,<,<=,=<,>=,=>

Priority	Operator
8	NOT
9	AND
10	OR
11	XOR
12	IMP
13	EQV

1.4.5 Operation of Character String

The operation dealing with character strings is as follows:

(1) Concatenation

Joining two character strings together is called concatenation. Character strings are concatenated using the plus symbol (+). For example:

Example:
```
10 A$= "CANON" + " AS-100"
20 PRINT A$
RUN
CANON AS-100

10 B$= " AS-100"
20 A$= "CANON" + B$
30 PRINT A$
RUN
CANON AS-100
```

(2) Comparison of Character Strings

In the comparison of character strings, character strings are turned into corresponding character codes one by one, and the codes are compared from the top character by character. If all characters are equal, the equal sign holds, whereas if the number of characters, namely character codes are different, comparison as to the size will be carried out according to the codes. The characters whose character codes under comparison are less, or the characters on one side that have come to an end during comparison are judged to be less.

The details of comparison are as shown on the next page.

1) Corresponding characters are sequentially compared from the top of two character strings.

2) For the comparison of characters, they are converted into character codes and the size of these codes are judged.

3) Characters are sequentially compared from the top, and if character codes differing from each other are found, the size of the entire character string is judged by the size of the charactor codes.

4) When the same character string appears in succession, the character string that has first come to an end is judged to be less.

5) The null character is regarded as one of characters.

As relational operators, >, <, =, ><, >=, =< can be used. If a relational expression is true, it means -1, and if it is false, it means 0.

For the comparison of character strings, refer to the table of character codes.

The size relation of character strings can be learned simply by the following:

1) Numeric characters < Alphabetic upper-case characters < Alphabetic lower-case characters.

2) The lower number is less. (0<1<2 ... <9)

3) The lower alphabetic character is less. (A<B<C ... <Z)

The above rule is sufficient for every case unless special characters are used.

1.4.6 Function

(1) Numeric Function

BASIC is provided with such functions as sine, cosine, log, etc., which can be used in such expressions of an assignment statement, etc.

SQR(X)

Function name Argument

Example: SQR(2), LOG(A/B)

For further details of individual functions, refer to Chapter 3.

(2) User Defined Function

Functions can be used by user's definition by means of the DEF FN statement.

```
DEF FNxxx (A, B, ..., Z) = SQR (A) * B/C + ...
```

Function name Dummy argument Definition of function

(3) Character Function

As the functions which perform the operation of character strings, RIGHT$, LEFT$, STRING$, etc. are available. For further details of these functions, refer to Chapter 3. In the following, only typical functions will be described:

- LEFT$ function

  Returns the leftmost n characters of the character string.

```
10 A$= "CANON AS-100"
20 B$=LEFT$(A$,5)
30 C$=LEFT$(A$,12)
40 PRINT B$
50 PRINT C$
60 END
RUN
CANON
CANON AS-100
```

- RIGHT$ function

  Returns the rightmost n characters of the character string.

```
10 A$= "CANON AS-100"
20 B$=RIGHT$(A$,6)
30 C$=RIGHT$(A$,12)
40 PRINT B$
50 PRINT C$
60 END
RUN
AS-100
CANON AS-100
```

The preceding two functions take out the leftmost and the rightmost of the first argument as many as the number designated by the second argument. The following relation holds:

```
LEFT$("CANON", 2) = "CA"
```

Character string Number of characters

As the functions to deal with character strings, MID$, STR$, STRING$, CHR$, etc. are available in addition. Those having $ at the end of a function name are functions having any relation with character strings. Moreover, such numeric functions as ASC, LEN and VAL are related with the processing of character strings.

1.5 Data Input/Output

1.5.1 Assignment Statement

Suppose the following program:

```
10 A$= "Personal Computer"
20 PRINT A$
RUN
Personal Computer
```

The result is same as above for all that PRINT is followed by A$. This is because the "assignment" of A$= "Personal Computer" is carried out in Line 10. In other words, as dimly seen from the form of the expression, PRINT A$ resulted in the same as PRINT "Personal Computer" because the constant of "Personal Computer" was assigned to A$. The statement such as A$= "Personal Computer" in Line 10 is referred to as assignment statement.

The assignment statement as to a numeric value can also be dealt with in the same way. Here, it should be noted that "assignment" (=) differs from the "sign of equality" in the mathematical sense.

Suppose the following program:

```
10 A=10
20 PRINT A
30 A=A+1
40 PRINT A
RUN
 10
 11
```

The feature of the assignment statement is typically seen in the expression in Line 30. Such an expression cannot mathematically hold, and the sign of "=" used in the assignment statement is not the sign of equality used in mathematics.

The variable indicates a specific part in the memory of the computer. And the area of the memory is identified by a variable name such as A or A$. The sign of "=" for assignment means: Store the result calculated by the expression following the sign of "=" of the assignment statement in the specific memory area.

It is seen from the above description that the left side of an assignment should always be only one variable. In other words, the following assignment statements as shown on the next page do not exist.

```
A+3=B
A*2=A+2
```

While the above two expressions are mathematically allowed, they are not allowed in the assignment statement. This is because the meaning of assignment differs from the sign of equality.

The right side of an assignment statement must be an expression using a variable, constant, operator and built-in function.

1.5.2 Output Statement

(1) PRINT Statement / LPRINT Statement

The PRINT statement displays characters and numeric values on the screen, and the LPRINT statement prints out them on the printer.

PRINT can be replaced with the question mark(?).

Example: PRINT 5

The numeric value 5 is displayed.

PRINT 3/2

The answer 1.5 of 3/2 is displayed.

PRINT 2*A

The value obtained by doubling the content of numeric variable A is displayed.

PRINT "BASIC"

The character string of BASIC is displayed.

PRINT B$

The content of character variable B$ is displayed.

PRINT "CANNON" + " AS-100"

CANON AS-100 is displayed.

To display a plural number of data, arrange data, dividing them by commas (,) and semicolon (;). BASIC divides one line into areas each for 14 characters. When data is divided by a comma, the next area will be displayed from the beginning. When it is divided by a semicolon, it will be displayed directly following what has just been displayed.

Example:

```
100 PRINT "A=",3
110 PRINT "$";100
120 PRINT "DATA",-1
130 PRINT "DATA",
140 PRINT 33
150 PRINT "NO.";
160 PRINT 1
170 PRINT 3,4,2
180 PRINT 3;4;2
RUN
A=            3
$ 100
DATA         -1
DATA          33
NO. 1
 3            4            2
 3 4 2
Ok
```

Pay attention to Lines 130 and 140 as well as Lines 150 and 160. Also when the PRINT statement is divided in this way, the same result will be obtained.

(2) PRINT USING/LPRINT USING

PRINT USING is an instruction to display data by format control. It is used, for example, when numeric values are displayed with digits aligned. In addition to the designation of data same as that of the PRINT statement, a format control character string is contained in the statement, and the format is controlled by it.

Numeric Display and Format Control

- The symbol "#" indicates for a digit, and "." for the position of a decimal point. For example, when data is displayed in the format of an integer part of 3 digits and a fraction part of 2 digits, the control character string will be as follows:

  "###.##"

When the integer part of a numeric value in this case is less than 3 digits (including the sign), it will be displayed from the right end. When the number of digits of the fraction part is less than 2, 0 will be inserted. When it is larger than 2, on the contrary, the digits in the third place of decimals will be rounded. In other words, when this designated is made, the fraction part will always be displayed in 2 digits.

- For a numeric value whose integer part exceeds 3 digits, "%" is displayed immediately before that numeric value, and the

whole of it is shifted to the right, though every digit of the integer part will be displayed.

- If spaces are inserted between "#" and """, data will be separated in accordance with the number of spaces when it is divided into several parts and displayed. To output several numeric data line by line in different formats, use the following:

```
"   ###.###     ####.###   "
```

Thus, the first data will be displayed in the left format, and after placing spaces, the second data will be displayed in the right format. The control is carried out in the way that the third is displayed in the left format, the fourth in the right format, and so on.

- To display a numeric value in the exponential form, and "^^^^" after the digit designation, as follows:

```
"   ###.###^^^^   "
```

- To display the signs (plus and minus) of data, add "+" before or behind the digit designation. When it is added before the digit designation, the sign will be displayed at the top of the displayed data, and when it is added behind the digit designation, the sign will be displayed immediately after the data.

```
"+###.##"
"###.#+"
```

- If "-" is added after the digit designation, the minus sign will be displayed after a numeric value when the numeric value is negative. It is not allowed to add "-" before the digit designation.

- To display the dollar sign($) at the top of displayed data, add "$$" in the following way:

```
"$$#####"
```

An area for 2 digits is secured by "$$", though the area for one digit is used to display the dollar sign ($).

- If "**" is used in place of "$$", the blank part where a numeric value does not fulfill the designated number of digits will be filled with "*" in the display.

```
"**#####"
```

An area for 2 digits is also secured by "**".

- If "**$" is added before the digit designation, "$" will be displayed at the top of a numeric value, and the blank part will be filled with "*". An area for 3 digits is secured by "**$", and "$" corresponds to one of the 3 digits.

  "**$####.#"

  It is not allowed to use "$$" for the format designation in the exponential form.

To display these format control characters as characters, add "_" in front of the characters. One character following "_" is displayed as a character.

Character Display and Format Control

- To output only one character at the beginning of the character data to be displayed, use the following format control character string:

  "!"

- To display a character string for 5 characters from the top of character data, use the following:

  "\ \"

  That is to say, the characters for the number of digits including "\" will be displayed. If given data is longer than the designated number of characters, excessive characters will be disregarded, and on the contrary, if given data is shorter than it, characters will be displayed from the left end.

Example:
```
100 PRINT USING "! "; "BASIC", "AS-100"
110 PRINT USING "\  \"; "CANON"
120 PRINT USING "\   \"; "TEST"
RUN
B A
CANO
TEST
Ok
```

(3) PRINT#/PRINT# USING

The instruction having the mark "#" such as PRINT# is used to carry out some operation for a file. The following description will be concerned with the case where the screen, printer or keyboard is designated as a file. For a disk file or a communication file, refer to another paragraph.

The PRINT# statement is an instruction to output a numeric value or a character string for a designated file. Although the screen or printer can be used as an output file by opening it, normally the PRINT statement or the LPRINT statement is used.

The PRINT# USING statement is an instruction to carry out format control in the same way as the PRINT USING statement, and it allows an output to a designated file.

1.5.3 Input Statement

The following input statements are available, and main input statements will be described in the following. (For further details, refer to Chapter 2.)

```
READ       INPUT$
DATA       INPUT#
RESTORE    LINE INPUT
INPUT      LINE INPUT#
INKEY$     GET
```

(1) READ/DATA/RESTORE Statements

The READ statement assigns the data prepared in a program by the DATA statement to a designated variable. As a connected instruction, the RESTORE statement is available.

Example:
```
100 READ A,B$
110 PRINT A,B$
120 END
130 DATA 492, "CANON"
RUN
 492      CANON
Ok
```

(2) INPUT Statement

The INPUT statement is an instruction which assigns entered statements or numeric values to variables, waiting for an input from the keyboard. When the operation is ready for an input, a question mark (?) is displayed. An input is made by depressing the carriage return key.

The type and number of the variable designated by the INPUT statement should agree with those of input data. If numeric values are entered from the keyboard when character variables are used, they will be substituted as a character string for those variables. If characters are entered when numeric

variables are used, or if the number of input data does not agree with the number of variables, the message

```
?Redo from start
```

will be printed out, and the operation will be ready for an input again.

A message can be displayed by the INPUT statement. The message is a prompt statement enclosed by double quotation marks ("), and it is placed in front of a variable. When it is separated from a variable by a semicolon (;), and question mark (?) will be displayed after the message. When it is separated by a comma (,), no question mark will be displayed.

In addition to this, if a semicolon is placed between INPUT and a prompt statement, the cursor will not move after data has been entered, and it will remain in the position at the time when the carriage return key is depressed.

(3) INKEY$ System Variable

INKEY$ referred to as system variable and dealt with as one of variables, so it can be used together with other instruction statements.

Unlike the INPUT statement, it does not stop the operation, waiting for a key input, and if there is a key input at that time, it will give that character, while if there is no key input, it will give a null string. Moreover, a key input will not displayed on the screen.

Since the existence of a key input is only momentarily checked, it is used normally in a loop.

Example:
```
100 A$=INKEY$
110 IF A$= "E" THEN END ELSE 100
```

When this program is executed, the operation will continue to go through the loop until "E" is depressed. However, if the [CANCEL] or [CTRL] + [C] key is depressed, the operation will be broken and return to the command level. Therefore, the codes of [CANCEL] and [CTRL] + [C] cannot be assigned to variables by this instruction.

1.6 Program Execution Control

The following commands are used for controlling the execution sequence of a program:

```
GOTO
ON...GOTO
GOSUB
ON...GOSUB
IF...THEN...ELSE
FOR...NEXT
WHILE...WEND
```

1.6.1 GOTO Statement

The GOTO statement is a command to perform an unconditional jump. When this command is used, the BASIC program jumps to Line No. given immediately after "GOTO". If the Line No. giving the specified jump destination is not found at this time, an error occurs. The contents of the jump destination line of the GOTO statement may be a non-executable statement (such as a REM statement).

The use of the GOTO statement is to facilitate formation of a loop (repeated operation), but too frequent use of the GOTO statement may complicate the entire structure of the program.

Example:
```
100 PRINT "LOOP"
110 GOTO 100
```

When the above program is run, characters "LOOP" are displayed continuously on the screen. To stop the display, it is necessary to depress [CTRL] + [C]. However, such a continuous loop should be used in some case. For instance, a loop is continued without any meaning in order to produce a waiting and idling state. Once an interrupt is made, control escapes from this perpetual loop and proceeds to a separate operation such as each interrupt processing.

1.6.2 IF...THEN...ELSE Statement

The IF statement is a command to perform conditional judgement. If the value of an expression succeeding IF is "0," judgement will be "false," and if the value is not "0," judgement will be "true."

If the judgement is true, control will execute the statement following THEN or jump to the line having the Line No. which

follows THEN. If the judgement is false, control will execute the statement following ELSE or jump to the Line No. If ELSE and the rest are omitted and the expression following IF is false, commands from the next line onward will be executed.

By making statements following THEN into a multistatement, several commands can be executed, if judgement is true. It is also possible to make multiplex judgement by inserting an IF statement into another IF statement. In such case, all the statements should be accommodated on one line.

For the expression following IF, a logical expression is usually employed. The logical expression consists of a logical operator and an expression, and takes a value of -1 at true judgement and a value of 0 at false judgement.

1.6.3 FOR...NEXT Statement

The FOR...NEXT statement forms a loop between FOR and NEXT. The number of loopings is set in the FOR statement as follows:

```
FOR I=1 TO 20 STEP 2
```

In the above statement, the value of I is incremented by 2 starting from 1 when operation enters a loop, and a program between the FOR and NEXT statements is repeated until the value of I exceeds 20. This I is called the "loop variable," and it is assumed that I=1 is an initial value, 20 is an end value, and 2 is an increment. Namely, a loop from 1, 3, 5, ..., 19 is repeated 10 times. If you want to form simply a 10-time loop, however, the following will do:

```
FOR I=1 TO 10
```

(STEP, if omitted, will be 1.) or

```
FOR I=10 TO 1 STEP -1
```

Variables in the FOR statement in many cases are used in the loop program. For instance, a program which uses angles at every 10° from 0° to 180° is written as follows:

```
FOR I=0 TO 180 STEP 10
```

A FOR...NEXT statement may enclose another FOR...NEXT statement. Pay attention at this time to the positions of the corresponding two NEXT statements. The following program is not allowed:

Erroneous Example:

```
FOR K=1 TO 5
FOR T=2 TO 10
     :
     :
NEXT K
NEXT T
```

Correct Example;

```
FOR K=1 TO 5
FOR T=2 TO 10
     :
     :
NEXT T
NEXT K
```

As shown above, a FOR···NEXT statement must be completely enclosed in another FOR···NEXT statement. Even if a FOR···NEXT statement is in the state of all the conditions being satisfied from the start, the program in the loop is executed once. You will understand this fact, when you run the following program:

```
10 FOR R=1 TO 1
20 PRINT "LOOP"
30 NEXT R
40 PRINT R
```

Pay attention to the fact that the value of R has become "2" upon completion of a single loop. The value of a loop variable upon completion of a single FOR···NEXT loop has become "end value + increment".

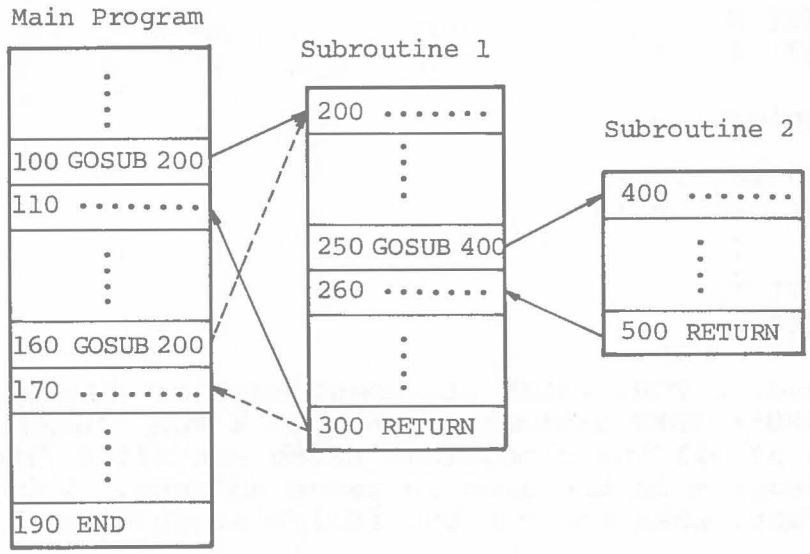
1.6.4 GOSUB...RETURN Statements

The GOSUB statement is a command to pass the control to a subroutine and used in a pair with the RETURN statement.

The GOSUB statement, when executed, causes control to jump to the specified line as with the GOTO statement and to execute the program sequentially starting from the line. When the RETURN statement comes, control returns to the command next to the GOSUB statement and resumes execution starting from the command.

A subroutine can call and enclose another subroutine. This procedure is called the "nesting of subroutine." The number of usable nestings is determined by the size of the stack area, in which return destinations from the subroutine are stored.

Subroutine and Nesting



1.6.5 ON...GOTO/ON...GOSUB Statements

These statements determine the lines of branching destinations according to the values of an expression following ON . Namely, these statements constitute a command which combines the IF statement and the GOTO or GOSUB statement. For instance, in the following program:

```
ON I GOTO 700, 800, 900
```

control jumps to Lines 700, 800, and 900, when the value or variable I is 1, 2, and 3 respectively. If the value of variable I is a negative number, an error will occur; if the value of varaible I is other than those mentioned above, control ignores this command and executes the succeeding command.

1.7 File Handling

1.7.1 Files

BASIC employs a concept of a file for exchanging information with I/O devices. The term "file" is a collection of information having meanings, which specifies the name of the file and the device at which it is stored, with the help of a file descriptor.

1.7.2 File Descriptor

File descriptor is composed of the following character string:

"[<Device Name>][<File Name>]"

The file descriptor is ordinarily expressed as a character string surrounded by double quotation marks as shown above or by a character variable. The device and file names may sometimes be omitted.

I/O device name	Device name	Input	Output	Remarks
Keyboard	KYBD:	○	×	
Screen	SCRN:	×	○	
Centronics I/F	LPT1:	×	○	
	LPT2:	×	○	Optional
	LPT3:	×	○	
	LPT4:	×	○	
Floppy Disk Drives	A:	○	○	
	B:	○	○	
	C:	○	○	Optional
	D:	○	○	
RS232C I/F	COM1:	○	○	
	COM2:	○	○	Optional
	COM3:	○	○	
	COM4:	○	○	

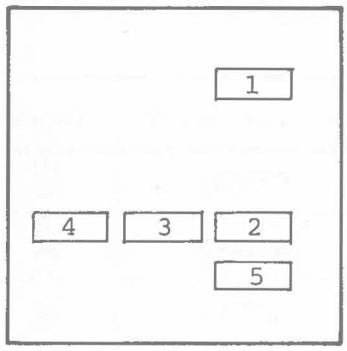
Specify a following device name when the peripheral devices are defined.

No.	I/O connector	Device name
1	Centronics I/F	LPT1:
2	RS232C I/F or Centronics I/F	COM2: or LPT2:
3	"	COM3: or LPT3:
4	"	COM4: or LPT4:
5	RS232C I/F	COM1:

Note: Centronics I/F can be added to only one of the connector 2 to 4.

The numbers correspond to those in the figure below.

Rear view of display unit



(1) Device name

Device name is allocated to each I/O device and defines the device in which the file specified by the file descriptor in question is stored.

The <device name> in BASIC is allocated to each I/O device as shown in the table on the preceding page. If the <device name> is omitted, the floppy disk drive which is currently selected is specified.

(2) File Name

File name is used to differentiate a great number of files stored in the I/O device specified by the device name.

The <file name> is composed of "8 characters for file name + 3 characters for extension" at the maximum. The file name is punctuated from the extension by a period (.). If the extension exceeds 3 characters, the overflowing characters are omitted. Characters which can be used for the <file name> are limited to the following:

Alphabet (from A to Z)
Numerals (from 0 to 9)
$
@
- (hyphen)

Example: "B:DATA.DAT"

File named "DATA.DAT" in drive B.

"DATA.DAT"

File named "DATA.DAT" in the currently selected drive.

"KYBD:"

Represents the keyboard.

"F$"

When expressed by a character variable, F$ should naturally be defined beforehand.

The file name and extension can be replaced by an asterisk respectively, which can be used in place of any given file name or extension.

The question mark can substitute a single character in the file name and extension and be used when a part of spelling is unknown.

Example: FILES "B:*.*"

Displays all the file names in drive B

KILL "PROG.*"

Files called "PROG" in all the extension in current drive are deleted.

FILES "*. BAS"

Displays all the file names having extension called "BAS."

LOAD "PROG??. BAS"

The symbol "?" represents any given single character.

1.7.3 Program File

BASIC is provided with several commands for handling files on the disk which stores programs and data. Here description is given to several important commands for handling program files and to the usage of such commands.

Main commands are given below. For their detailed commentaries, refer to Chapter 2.

FILES: Find out the name of file stored in the disk.

SAVE : Save on the disk the BASIC program which is currently operating.

LOAD : Load the BASIC program file stored on the disk.

RUN : Load and run the BASIC program file stored on the disk.

KILL : Delete the file stored on the disk.

NAME : Change the name of the file stored on the disk.

MERGE: Merge the program currently stored in memory and that stored on the disk and store the merged program in memory.

(1) FILES (Check the Existence of File)

The FILES command is used to find out the name of the file stored on the disk.

While the machine is in the state of waiting for a BASIC command (while "Ok" is displayed), key-in the following:

```
FILES ↵
```

Then the names of all files stored on the disk in the currently-selected drive, together with extension, will be displayed on the screen.

If the existence of a file having a specific name is to be checked, key-in the file name after the FILES command as follows:

```
FILES "PROG.COM" ↵
FILES "*. BAS" ↵
```

If the specified file exists, the file name will be displayed on the screen. The example on the second line above shows that the specification of the file name is substituted by an asterisk.

Names of all the files stored on the disk which have an extension ".BAS"(BASIC program file) will be displayed.

If files on the disk in the drive which is not currently selected are to be found out, key-in the file name preceded by a drive name as follows:

```
FILES "B: MYPROG.BAS" ↵
FILES "B: *.*" ↵
```

In the latter example shown above, names of all the files stored on the disk in drive B: will be displayed on the screen.

(2) SAVE (Save the Program)

If you switch OFF power supply to the machine or restore the MS-DOS level by the SYSTEM command, the BASIC program which you have produced with much pains will disappear from memory and cannot be listed and run again. It becomes necessary, therefore, to save the BASIC program on external memory units such as a disk.

Let's consider how to save on the disk your program which you have just produced or are in the course of producing. Denoting the name of your program file by, say, "MYPROG-1.BAS," key-in the following:

```
SAVE "MYPROG-1" ↵
```

Then the program will be saved on the disk in the currently-selected drive. The portion of the extension ".BAS" is added automatically by BASIC, even if you do not write it.

If a file named "MYPROG-1.BAS" already exists on the disk, the contents of the original file will be rewritten into the newly-saved program. This process is time-saving and convenient, while you are producing a program in the course of its development by frequently revising it, but you may destroy the valuable file by mistake. Before you use the SAVE command, therefore, you must check existing files by using the above-mentioned FILES command and pay careful attention to naming files.

Programs saved by the SAVE command are stored on the disk usually in a compressed binary format. Beside the above, there is a storing method in ASCII format. This method saves program characters as they are and can be implemented by specifying the A-option in the SAVE command as follows:

```
SAVE "PROG-2", A ↵
```

The program file stored in the ASCII format has advantages that the program contents can be seen by means of the TYPE command of MS-DOS and the program file can be comparatively easily handled as a data file for I/O purposes. Also program files to be handled in the MERGE and CHAIN statements must be this ASCII format file.

Whereas the file saved in the binary form has an advantage of a smaller file size than that of the ASCII form.

(3) LOAD (Load the Program)

To call the program stored on the disk, the LOAD command is used as follows:

```
LOAD "A:PROG-1" ↵
```

If the above command is keyed-in, the file "PROG-1.BAS" will be searched out and loaded into internal memory. If the extension is omitted, ".BAS" will be automatically added as with the SAVE command. When a non-existent file name is specified, a message "File not found" is displayed and no loading is performed.

Upon executing the LOAD command, the program which remains in the memory of the machine is lost. If you want to save the program, do so before you load the next program.

Files saved by the SAVE command are divided into two distinct types, i.e., the ASCII format and compressed binary format, but in the LOAD command, there is no need of specifying such distinction, because program files, regardless of whichever form they may use, are loaded after being automatically identified.

Upon completion of program loading, the machine usually returns to the state of waiting for command input, and "Ok" is displayed. Upon specifying the R-option of the LOAD command, the loaded program is immediately run.

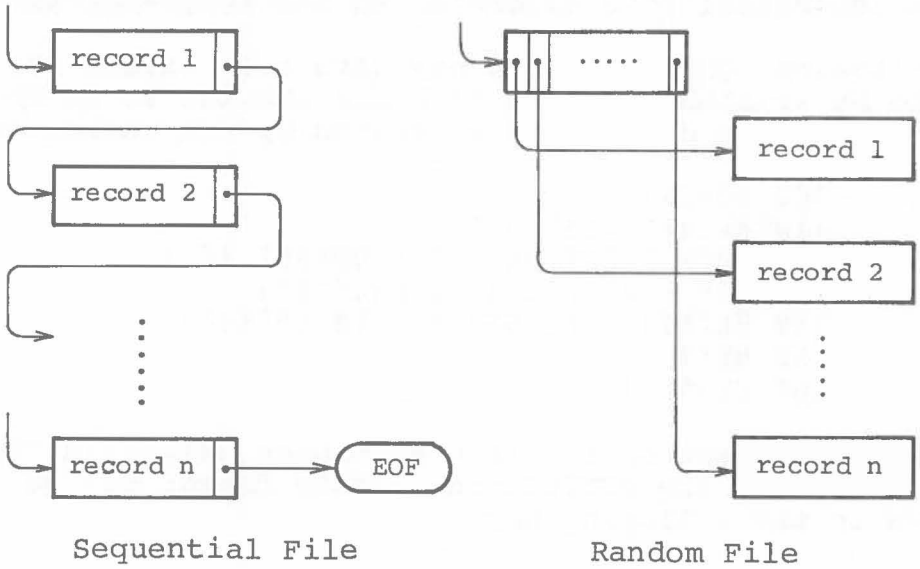
1.7.4 Handling of Data Files

Files which become the object of I/O operation in the BASIC program include two types, i.e., the sequential file and random file.

In the sequential file, records can be accessed one by one starting from the leading record in the file, whereas in the random file, a record in an arbitrary position can be directly

accessed. On the other hand, handling of the random file becomes more complex than that of the sequential file. The file constitution is determined according to the purposes of use of files.

Fig. 7.1 Sequential File and Random File



(1) I/O Operation of Sequential File

Input/output to and from the sequential file are performed by the following statements:

```
INPUT#
LINE INPUT#
PRINT#
PRINT#USING
WRITE#
INPUT$
```

For the details of the above statements, refer to Chapter 2.

The sequential file is usually accessed according to the following sequence:

1. Open the object file with the OPEN statement. Specify the OUTPUT or APPEND mode for outputting and the INPUT mode for inputting.

2. Input/output the file by using the I/O statement. Specify the file to be the object of I/O operation by means of File Nos. allocated by the OPEN STATEMENT. It is impossible, at this time, out to the file which has been opened by the

INPUT mode or conversely to input form the file in the OUTPUT mode.

3. When all the necessary I/O operation are completely executed, close the file with the CLOSE statement.

Let's see an example of the program which performs I/O operation of the sequential file according to the above-mentioned sequence:

The following SQNC-1 forms a new data file called "TEST.DAT." Data to be written into the file are assumed to be the character code and the characters expressed by the character code.

```
100'SQNC-1
110'create TEST.DAT
120 OPEN "TEST.DAT" FOR OUTPUT AS 1
130 FOR I=ASC("A") TO ASC("E")
140 PRINT#1, I, STRING$(10,CHR$(I))
150 NEXT
160 CLOSE 1
```

The OPEN statement on Line 120 allocates data file "TEST.DAT" to File No.1 in the OUTPUT mode. This format may be also written in the following way:

```
OPEN "O", 1, "TEST.DAT"
```

Data are formed by the FOR-NEXT loop on Lines 130 to 150, and stored in the file. Since the character code of "A" is 65 and that of "E" is 69, variable I changes from 65 to 66, ..., and 69, as it loops. The PRINT# statement on Line 140 prints out this character code and a continuation of 10 respective characters.

The CLOSE statement on Line 160 closes the file having File No. 1. <u>After this CLOSE statement, File No. 1 can be used for opening other files.</u>

The execution of this SQNC-1 produces a data file on the disk. Check to see, by using the FILES command, if the file has been formed.

Next, let's input the data from this data file. Program SQNC-2 to be used for this purpose and the results of its execution are shown below.

```
100'SQNC-2
110'read TEST.DAT
120 OPEN "TEST.DAT" FOR INPUT AS 1
130 INPUT#1, I, A$
140 PRINT I, A$
150 IF NOT EOF(1) THEN 130
160 CLOSE 1
Ok
RUN
 65        AAAAAAAAAA
 66        BBBBBBBBBB
 67        CCCCCCCCCC
 68        DDDDDDDDDD
 69        EEEEEEEEEE
Ok
```

First, data file "TEST.DAT" is opened in the INPUT mode on Line 120. This OPEN statement can be also written in the following way:

```
OPEN "I", 1, "TEST.DAT"
```

In the INPUT# statement on Line 130, data are inputted to variable I and A$ starting from data file No.1. The form, number and sequence of variables in the INPUT# statement should match those of the data file; otherwise it is impossible to input the data correctly. In this case, the character code (numeric variable I) and character string (character variable A$) are inputted sequentially as written in the program SQNC-1.

The PRINT statement on Line 140 displays on the screen the contents inputted to the variables.

The IF-THEN statement on Line 150 checks to see, by using the EOF function, whether or not the file record has reached the End of File. When the file record reaches the End of File, the EOF function returns "-1." Since this negative is taken up in the conditional formula, operation returns to the INPUT statement on Line 130 and continues inputting, if EOF is not valid. When the file record is exhausted, operation proceeds to Line 160 and closes the date file.

If the file is opened in the APPEND mode when data are stored into the file, the contents of the previous file remains and the newly-written contents are added thereto. An example of storing data in the APPEND mode to the data file formed by SQNC-1 mentioned earlier is given in SQNC-3 as shown on the next page.

```
100'SQNC-3
110'append to TEST.DAT
120 OPEN "TEST.DAT" FOR APPEND AS 1
130 FOR I=ASC("F") TO ASC("G")
140 PRINT#1, I, STRING$(10,CHR$(I))
150 NEXT
160 CLOSE 1
Ok
RUN
Ok
RUN "SQNC-2"
 65        AAAAAAAAAA
 66        BBBBBBBBBB
 67        CCCCCCCCCC
 68        DDDDDDDDDD
 69        EEEEEEEEEE
 70        FFFFFFFFFF
 71        GGGGGGGGGG
Ok
```

SQNC-3 newly appends data "F" and "G". After executing SQNC-3, SQNC-2 is loaded and executed by the RUN command in order to check the contents of TEST.DAT.

Beside the EOF function used in SQNC-2, a number of functions related to file I/O operation are provided.

The LOC function returns the number of records read or written after the opening of the file, if the function is used in sequential I/O operation. Here a single record consists of a data block 128 bytes long.

The LOF function expresses the number of bytes allocated to the file, namely, the file size. The number of bytes returned by the LOF function is a multiple of 128 bytes.

In the I/O operation of the sequential file, reading/writing is performed sequentially one data after another starting from the leading item. Since the sequential file is comparatively simpler in its procedure than the random file as shown above, the former is more widely used in ordinary cases.

In addition to the I/O statement used in the 3 program examples mentioned above, statements such as PRINT# USING, WRITE#, and LINE INPUT# can be used as and when necessary. For the details of formats of these statements, refer to Chapter 2.

(2) I/O Operation of Random File

The I/O operation of the random file requires a slightly more complex sequence than that of the sequential file. To make up

for this shortcoming, any records in the random file can be accessed directly and randomly.

In the random file, a single memory unit is called a "record," and records are controlled by affixing Nos. to them. A record length can be declared in the OPEN statement (the record length, if omitted, will be 128 bytes).

These two file types also differ in the recording format on the disk. The sequential file is recorded in the ASCII format, whereas the random access file is recorded in the packed binary format. At the time of I/O operation particularly of numeric data, therefore, format converting operation becomes necessary.

A statement related to I/O operation of the random file is given below.

```
FIELD
GET
LSET
PUT
RSET
```

As functions for format conversion, the following are used:

```
CVI/CVS/CVD
MKI$/MKS$/MKD$
```

Data are stored to the random file in the following procedure:

1. Open the file in the random access mode by using the OPEN statement. If necessary, declare the size of a record. Omission of this declaration means that the record size is 128 bytes long.

2. Allocate the random buffer area to variables to be outputted to the file, by using the FIELD statement.

3. Set output data (constants and variables) to the random buffer by using LSET and RSET statements. At this time it is necessary to convert numeric variables into character-type data by using the MKI$/MKS$/MKD$ function.

4. Output the contents of the random buffer to the record specified by the file, using the PUT statement.

5. Upon completion of all I/O operations, close the file by using the CLOSE statement.

When data are read out of the random file, Items 3 and 4 will be changed as shown on the next page.

3. Input the specified record to the random buffer by using the GET statement.

4. Inputted data can be cited in the program. If the data are the numeric type, they are stored in the character format at the buffer area, and therefore, they must be returned to the original format by using the CVI/CVS/CVD function.

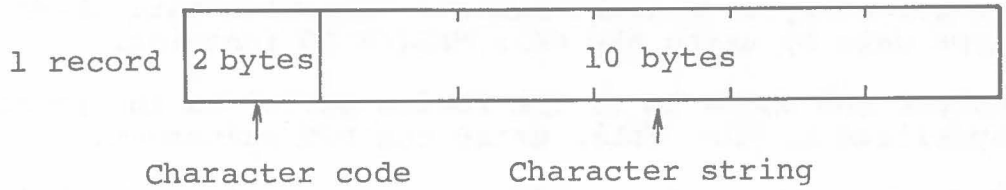
The random file once opened permits a direct access to an arbitrary record and can read and write them. During a period of one time of OPEN to CLOSE, operation of reading out a record, correcting it and outputting it, as it is, can be executed for any number of times.

Let's form a random file according to this procedure. A program example is shown below.

```
100'Randm-1
110 OPEN "TEST.DAT" AS 1 LEN=12
120 FIELD 1, 2 AS N$, 10 AS A$
130 NO=ASC("A")-1
140 FOR N=1 TO 26
150 LSET N$=MKI$(N+NO)
160 LSET A$=STRING$(10,CHR$(N+NO))
170 PUT 1, N
180 NEXT N
190 CLOSE 1
```

The above program produces character codes from A to Z (from 65 to 90) and forms a random file string which has, as data, character strings consisting of 10 such characters arranged continuously.

A single record in this example is composed of 2 bytes for the character code (integer) and 10 bytes for the character string, 12 bytes in total.



This record from A to Z, that is, 26 records in total are outputted.

The OPEN statement on Line 110 opens data file "TEST.DAT" in the random access mode. The record length is declared to be 12

bytes long. This line also can be written in the following format:

```
OPEN "R",1, "TEST.DAT",12
```

In the FIELD statement on Line 120, the leading 2 bytes in the 12-byte random buffer area is allocated as variable N$ and the remaining 10 bytes as A$.

Do not use variables, which were specified in the FIELD statement, on the left-hand side of the assignment statement or in the INPUT statement in the same program, that is, no assigning operation should be made except in LSET and RSET statements. If you make this assigning operation, the variable in question will be removed from the random buffer area and return to be an ordinary variable.

At every looping on Lines 140 to 180, data of a single record is outputted. Here "N" represents Record No.

On Line 150, the character code in the integer form is converted into the literal form by using MKI$ function and set to M$ (2 bytes) in the random buffer. On line 160, a character string of 10 characters is similarly formed by using STRING$ function and set to A$ in the random buffer.

Upon completion of setting to the random buffer, the PUT statement on Line 170 is used to output the contents of the buffer to the N-th record.

The above-mentioned outputting operation is sequentially repeated from record 1 to record 26, and then the file is closed by the CLOSE statement on Line 190.

In the preceding program example, data were stored sequentially from the leading record in order to prepare the data file easily. Naturally random access is also possible in which data can be stored starting from the last record, for example.

Now let's see the example of random access. The next program example reads out an arbitrary record in the random file which has been prepared a short while ago and displays its contents.

When the program is run, the machine asks for Record No. When you give Nos. 1 to 26, the program reads out the records from the file and displays them on the screen. When No. below 0 is given, the program is terminated. The execution example of the program is also shown after the list.

```
100'Randm-2
110 OPEN "TEST.DAT" AS 1 LEN=12
120 FIELD 1, 2 AS N$, 10 AS A$
130 INPUT "Record";R
140 IF R =0 THEN
150 GET 1, R
160 PRINT CVI(N$), A$
170 GOTO 130
180 CLOSE 1
RUN
Record?1
 65        AAAAAAAAAA
Record?2
 66        BBBBBBBBBB
Record?26
 90        ZZZZZZZZZZ
Record?0
Ok
```

The OPEN statement on Line 110 opens the data file which was formed a short while ago. The FIELD statement on Line 120 allocates the field of the random buffer. The names of variables allocated may be different from those used in "Random-1," but the record length of the OPEN statement and the allocated size and sequence of the FIELD statement must coincide with those used in "Random-1"; otherwise data cannot be inputted correctly.

Line 130 inputs No. of the record to be read out; Line 150 stores the prescribed record from the file into the random buffer.

The character-type data can be cited as it is, but numeric-type data is converted into the original format, becore it is cited. In this case, N contains a character code (integer), and therefore, it is returned to its original format by using the CVI function.

The above-mentioned input operation is repeated until Record No. below "0" is specified. At the end, the file is closed by the CLOSE statement.

The next table shows the relation between the form of I/O data, used quantity of the random buffer and function used for type conversion.

Data type	Type converting function	Occupied quantity of random buffer
Character	⟶ ⟵	(Number of characters)
Integer	MKI$ ⟶ ⟵ CVI	2 bytes
Single precision real	MKS$ ⟶ ⟵ CVS	4 bytes
Double precision real	MKD$ ⟶ ⟵ CVD	8 bytes

When the LOC function is used in the random access mode, the record No. which was accessed immediately before is returned.

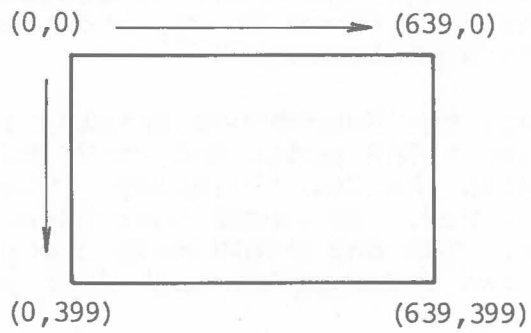
The LOF function can check the size of the file as in the case of the sequential file.

1.8 Graphics

1.8.1 Coordinates

The display capacity of the AS-100 CRT screen is 80 characters × 25 lines, containing 640 × 400 dots. Display on the ordinary CRT screen is given in units of characters, whereas display on the screen which employs the graphic function is performed in units of dots.

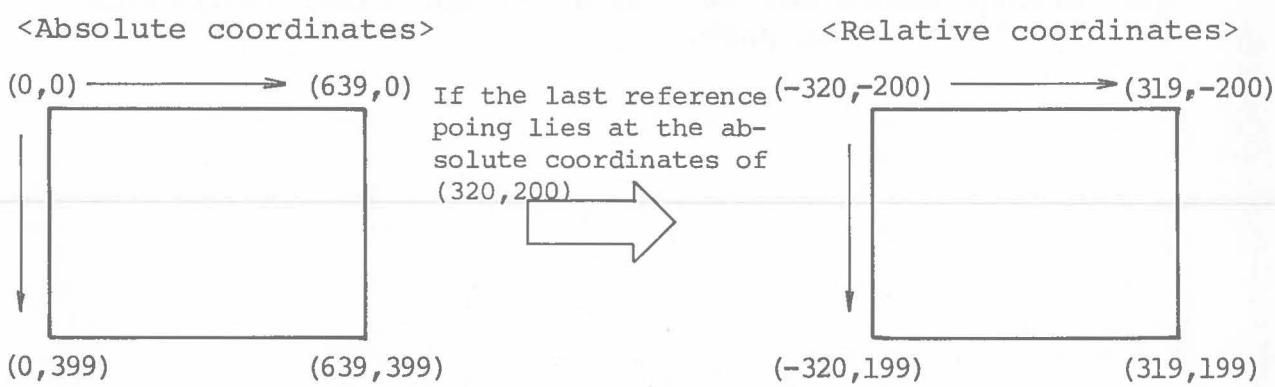
The graphic command requires information concerning the position on the screen where the user is going to draw a figure. The user gives this position in the form of coordinates which are set at 0 to 639 in the X-axis (horizontal) direction and at 0 to 399 in the Y-axis (vertical) direction. These coordinates correspond to dot positions on the screen and are called the "absolute coordinates."



Against this coordinate specification by the absolute coordinates, there is a method for specifying coordinates by relative coordinates. In the latter method, the point which was used last is employed as an origin, and a new point is displayed on the basis of this origin. The format used in the method is given below.

STEP (<X-offset>,<Y-offset>)

Namely, the vertical and horizontal offsets from the point used last are employed to specify new coordinates.



1.8.2 Palettes and Color Specification

Selection of the color (attribute) in graphic processing is made through palettes. In the following, description is given to the palette and color (attribute).

The AS-100 Color Display can give 27 colors including black (non-display). These 27 colors correspond to Color Nos. 0 to 26. For instance, if a command to draw a circle on the screen is to be executed, specify the palette No. which red is defined and a red circle will be drawn on the screen.

The AS-100 Color Display is provided with 8 palettes, which correspond to Palette Nos. 0 to 7. Further, information about what palette was used in drawing the display is stored in respect of all displayed figures on the screen, even after the figures have been drawn. It is possible, therefore, to change the displayed color by changing the color specification of the palette. Assume that the above-mentioned red circle was drawn by using Palette No. 4 and change the color definition of Palette No.4 to white after the figure was drawn, and then the red circle will be changed to a white one.

On the other hand, the Monochrome Display is divided into one V-RAM model and two V-RAM model and provided with palettes in the same way as with the Color Display. These palettes can be defined with Color Nos. Provided that these Color Nos. actually mean attribute. The one V-RAM model is provided with two palettes and the two V-RAM model with four palettes.

(1) PALETTE Statement

The PALETTE statement is a command to define Color No. to the palette used in the COLOR statement or graphic command. Its format is shown below.

```
PALETTE <Palette No.>,<Color No.>
```

Palette Nos. mean numerals 0 to 7 to be used for specifying each palette.

The correspondence between Color No. and color (attribute) is shown on the next page.

• For Color Display

Color No.	Color						Remarks
	r	R	g	G	b	B	
0	0	0	0	0	0	0	Black
1	0	0	0	0	0	1	Blue
2	0	0	0	0	1	1	
3	0	0	0	1	0	0	Green
4	0	0	0	1	0	1	Cyan
5	0	0	0	1	1	1	
6	0	0	1	1	0	0	
7	0	0	1	1	0	1	
8	0	0	1	1	1	1	
9	0	1	0	0	0	0	Red
10	0	1	0	0	0	1	Magenta
11	0	1	0	0	1	1	
12	0	1	0	1	0	0	Yellow
13	0	1	0	1	0	1	White
14	0	1	0	1	1	1	
15	0	1	1	1	0	0	
16	0	1	1	1	0	1	
17	0	1	1	1	1	1	
18	1	1	0	0	0	0	
19	1	1	0	0	0	1	
20	1	1	0	0	1	1	
21	1	1	0	1	0	0	
22	1	1	0	1	0	1	
23	1	1	0	1	1	1	
24	1	1	1	1	0	0	
25	1	1	1	1	0	1	
26	1	1	1	1	1	1	
27	1	1	1	1	1	1	
28	1	1	1	1	1	1	

r: Half brightness red R: Red
g: Half brightness green G: Green
b: Half brightness blue B: Blue

• For Monochrome Display

Color No.	Attribute			Remarks
	B	H	S	
0	0	0	0	Non-display
1	0	0	1	Standard brightness
2	0	1	0	High brightness
3 ∿ 26	0	0	1	Standard brightness
27	1	0	1	Standard brightness blinking
28	1	1	0	High brightness blinking

When GW-BASIC is started, palettes are defined with the following Color Nos. as initial values:

• For Color Display

Palette No.	0	1	2	3	4	5	6	7
Color No. (Color)	0 (Black)	1 (Blue)	3 (Green)	4 (Cyan)	9 (Red)	10 (Magenta)	12 (Yellow)	13 (White)

• For Monochrome Display (One V-RAM model)

Palette No.	0	1
Color No. (Attribute)	0 (Non-display)	1 (Standard brightness)

• For Monochrome Display (Two V-RAM model)

Palette No.	0	1	2	3
Color No. (Attribute)	0 (Non-display)	2 (High brightness)	27 (Standard brightness blinking)	1 (Standard brightness)

Colors which have been defined to the respective palettes by the PALETTE statement will be effective until the palettes are redefined by the next PALETTE statement.

(2) COLOR Statement

The COLOR statement is a command to specify palette Nos. which define the colors of the foreground and background on the screen, and written in the following format.

```
COLOR [<foreground>][,<background>]
```

The <foreground> consists of Palette Nos. (0 to 7) which define the colors of the foreground; the <background> consists of Palette Nos. to define the colors of the background. If <background> is omitted, Palette No.0 will be automatically defined.

Initial values of definitions of palettes in Color Dispaly are set at Palette No.7 for the foreground and Palette No.0 for the background. The initial values of palette definitions in Monochrome Display are set at Palette No.1 (one V-RAM model) or 3 (two V-RAM model) for the foreground and at Palette No.0 for the background.

(3) PAINT Statement

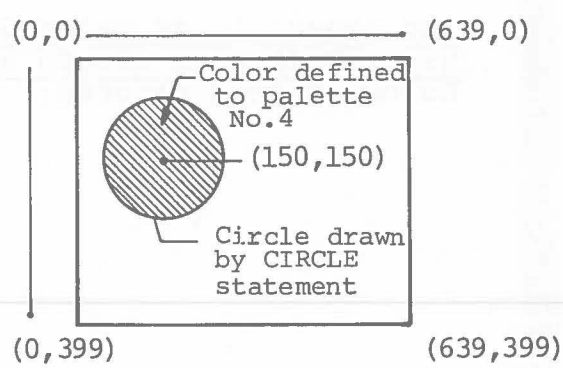
The PAINT statement is a command to paint completely with specified colors the closed areas of various figures drawn by the DRAW and CIRCLE statements, etc., and written in the following format:

```
PAINT | (<x-coordinate>,<y-coordinate>)  | [,<color>
      | STEP (<x-offset>,<y-offset>)     | [,<boundary color>]]
```

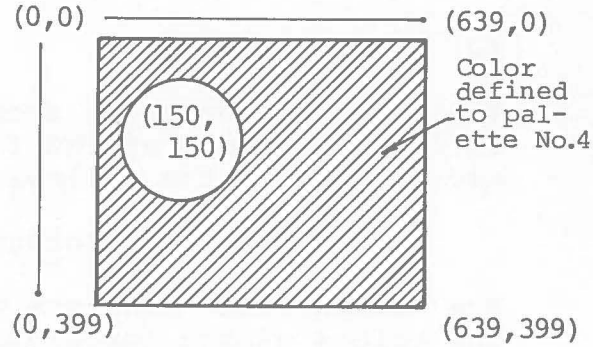
Example-1:
```
100  CLS
110  CIRCLE(150,150),100,4
120  PAINT(150,150),4,4
```

In Example-1, the interior of a circle drawn by the CIRCLE statement on Line 110 is completely painted by the PAINT statement. The execution result is shown in the figure on the right.

Example-2:
```
100 CLS
110 CIRCLE(150,150),100,4
120 PAINT(300,300),4,4
```



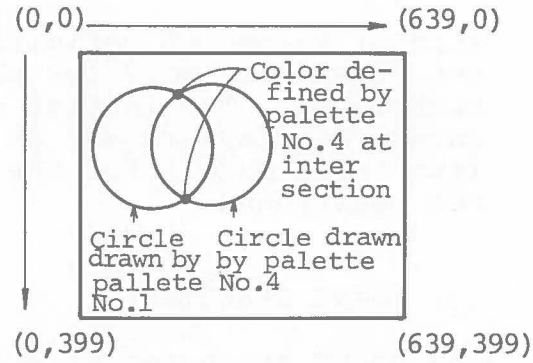
In Example-2, the exterior of a circle drawn by the CIRCLE statement on Line 110 is completely painted by the PAINT statement. The execution result is shown in the figure on the right.



Example-3:

```
100 CLS
110 CIRCLE (150,150),
    100,1
120 CIRCLE (250,250),
    100,4
```

Intersections of two circles drawn by the program in Example-3 are displayed with a color defined by Palette 4. Therefore, the circle drawn by Line 110 is not a closed area. If the following line is executed here, the entire screen will be completely painted.




```
130 PAINT(150,150),1,1
```

As can be seen from this example, the intersections of the figure are displayed in the color of the figure drawn last.

If you omit the specification of the <boundary color> which is one of the operands of the PAINT statement, the same Palette No. as that of the <color> will be defined. For instance, Line 120 in Example-1 specified as follows:

```
120 PAINT(150,150),4,4
```

The same execution result will be obtained, if the following specification is made.

```
120 PAINT(150,150),4
```

<u>The object to be painted completely, when a PAINT statement is used, must be a closed area.</u> Otherwise, the whole screen will be completely painted.

1.8.3 I/O Operation of Graphic Pattern on Screen

A graphic pattern drawn inside a rectangle having two specified points as the apexes of 2 diagonal lines is inputted to an array variable (by GET statement), or the contents of the array variable are outputted on the screen (by PUT statement). The formats of the GET and PUT statements are given below.

```
GET |(<x1-coordinate>,<y1-coordinate>)|-
    |STEP (<x1-offset>,<y1-offset>)   |

    |(<x2-coordinate>,<y2-coordinate>)|,<Array variable name>
    |STEP (<x2-offset>,<y2-offset>)   |

PUT |(<x-coordinate>,<y-coordinate>)  |,
    |STEP (<x-offset>,<y-offset>)     |

     <Array variable name> [,<conditions>]
```

* For the details of operands, refer to Chapter 2.

The array variable specified by the <Array variable name> must be a one-dimensional numeric-type variable. Also before starting the execution of the GET command, the necessary memory area must be declared by the DIM statement. The value of the suffix to the array which is specified by the DIM statement is obtained by the following calculation formula.

```
<Value of suffix specified by DIM statement>
        = INT ((4+INT((x+7)/8)*y*3)/n)+1
```

where x is the number of dots in the X-axis direction, y is the number of dots in the Y-axis direction, and n takes the following forms depending upon the form of arrays:

- Integer-type array : n = 2
- Single-precision-type array: n = 4
- Double-precision-type array: n = 8

Example:

```
10 DIM IMAGE(267)
20 CLS
30 CIRCLE (100,100), 25
40 GET (75,75) - (125,125),IMAGE
     :
     :
80 PUT (300,300),IMAGE,PSET
```

In the above example, the CIRCLE statement on Line 30 draws a circle with a center (100,100) and a radius of 25, and the GET statement inputs a graphic pattern inside a rectangle, which has apexes (75,75) and (125,125) of 2 diagonal lines, into a

real-type array called "IMAGE." Further, the PUT statement on Line 80 inputs the contents of array "IMAGE" to the area, which has coordinates (300,300) at the upper left corner, according to the conditional expression PSET.

The value of suffix specified by the DIM statement will be 256 by assigning x = 51, y = 51, and n = 4 into the calculation formula on the previous page, since IMAGE is a real-type array.

```
<Value of suffix> = INT((4+INT((x+7)/8)*y*3)/n)+1
                  = INT((4+INT((51+7)/8)*51*3)4)+1
                  = 267
```

1.9 Machine Language Subroutines

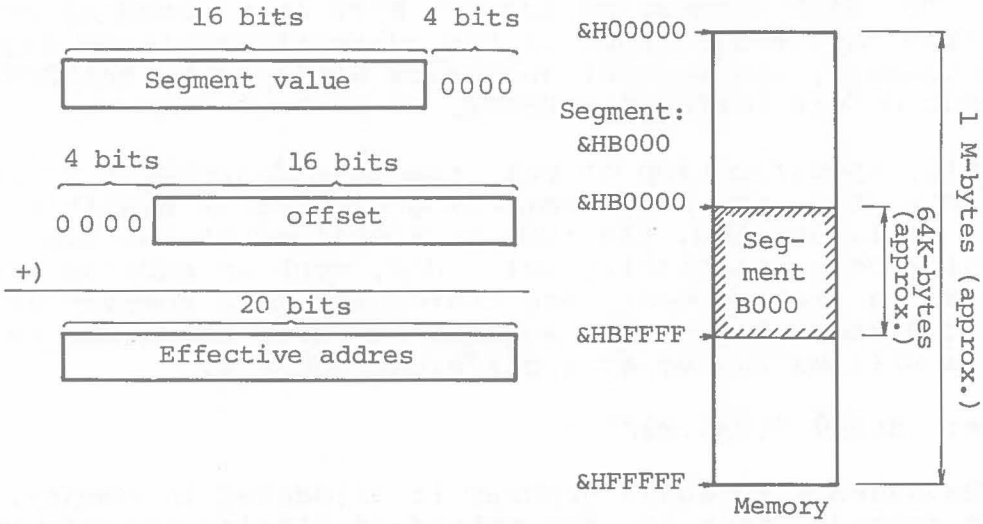
BASIC is provided with an interface function to the machine language program. In the following, description is given to the method of accessing from the BASIC program to the machine language program.

For the details of the machine language program of the 8088 CPU, refer to other reference books.

1.9.1 Address Generation

The 8088 CPU, which is used in the machine, has 20-bit addresses, and therefore, the CPU has an address space of 2^{20} = 1M-bytes (approx.). The 8088 CPU does not directly handle the total area of this wide address space, but handles the address space by dividing it into small sections called "segments." This segment is a sort of a window peeping into a part of the wide space and has a size of 64K-bytes.

The following diagram shows the concrete method of address generation:



The segment value (16-bit) and offset (16-bit) are added up as shown in the figure above to obtain a 20-bit effective address. The carry generated by the adding-up is ignored.

Four registers which reserve segment values, i.e., CS (code segment), DS (data segment), SS (stack segment) and ES (extra seg-

ment), are provided. By setting a value to each register, each segment can be allocated to an arbitrary position (in steps of 16 bytes) in the 1M-byte address space. Once the segment is set up, addresses in the same segment can be accessed simply by changing the offset value.

Addresses of memory specified by BLOAD and BSAVE commands and POKE and CALL statements are not made effective addresses as they are, but used as offset values in the process of effective address calculation. Segment values can be changed by the DEF SEG statement.

1.9.2 Load and Save of Machine Language Subroutine

BLOAD and BSAVE commands are provided in order to load and save not only the machine language program but also memory image data.

In saving memory image data, specification of the address at which saving is commenced and the size of saving, in addition to that of the file descriptor, is required.

Example: BSAVE "CORE.MEN", 0, &H1000

In this example, 4K-bytes from the leading part of the segment are saved by the name of "CORE.MEM" in the currently-selected drive. The SAVE commencing address here is treated as an offset value from the leading part of the currently-declared segment. When necessary, the segment should be declared by the DEF SEG statement before performing BSAVE.

For BLOAD, specification of only the file descriptor is required. Specification of the LOAD commencing address is possible, but even if it is omitted, the address specified at the time of BSAVE will be automatically set. Now, such an address is an offset value in the segment, and therefore, if a segment is set which is different from the segment in which BSAVE was performed, the data will be loaded at a different address.

Example: BLOAD "CORE.MEM"

When the machine language program is allocated in memory, there are two methods, that is, the method of placing the machine language program in the BASIC segment and that of placing it outside the BASIC segment.

In allocating the machine language program in the BASIC segment, operation such as segment alteration is unnecessary, but the working area of BASIC should be restricted in order to prevent from destroying the machine language program. This restriction

is performed by "/M:option," which is an option of GWBASIC command at the time of BASIC start or by employing the CLEAR statement after the BASIC start.

Example: At BASIC Start:

```
GWBASIC /M:&HC000
```

After BASIC Start:

```
CLEAR, &HC000
```

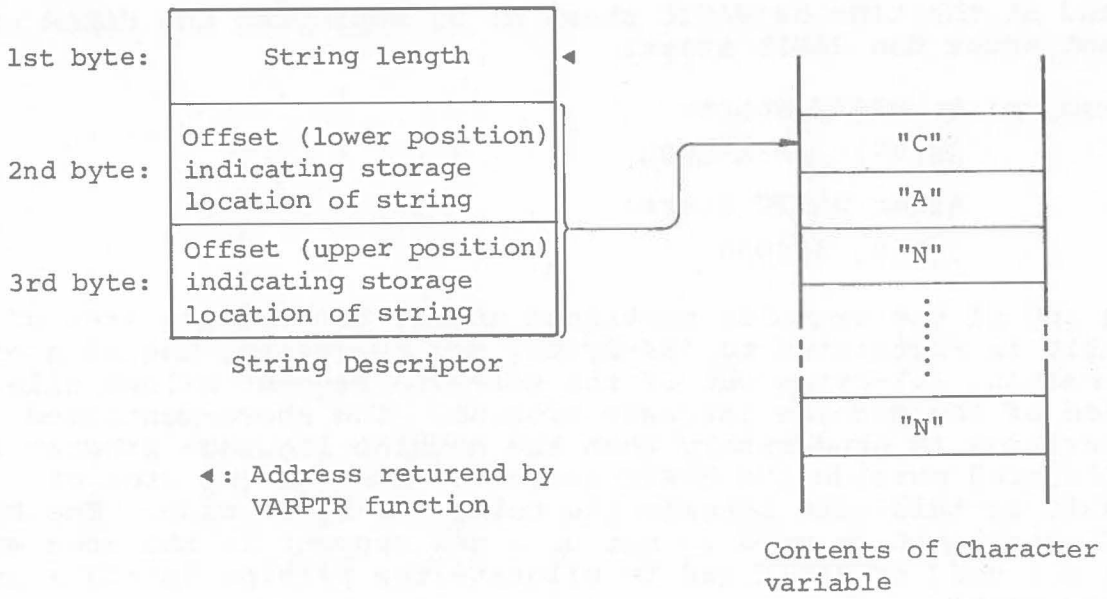
In any of the examples mentioned above, the working area of BASIC is restricted to 48K-bytes, and therefore, the area of remaining 16K-bytes out of the 64K-byte segment allows allocation of the machine language program. The above-mentioned operation is unnecessary when the machine language program is allocated outside the BASIC segment. The working area of BASIC at this time becomes the total 64K-bytes wide. The DEF SEG statement is used to set up a new segment in the area which is not used by BASIC and to allocate the machine language program there.

1.9.3 Storage Method of Variables

(1) Storage of Character String

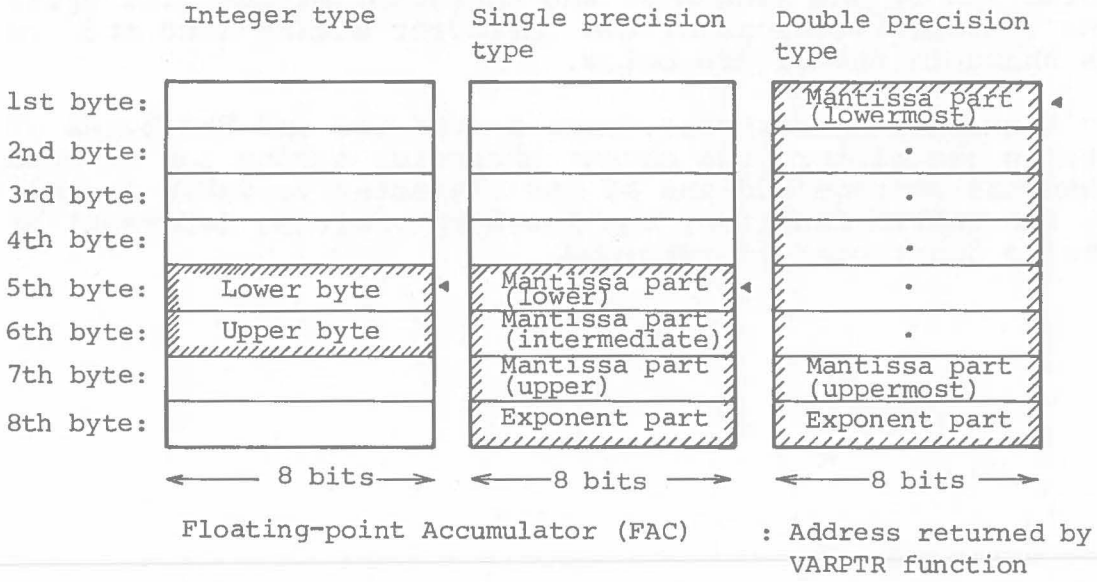
Character-type variables are stored by the following 2-step constitution: For a single character variable, a 3-byte area called the "string descriptor" is provided, which contains information of the length of the character string (1st byte) and the storage locations of the character string (2nd and 3rd bytes) as shown in the figure below.

By tracing the addresses given by the 2nd and 3rd bytes of the string descriptor, the object character string can be obtained. When the storage address of the character variable is checked by the VARPTR function, the 1st-byte position (address) of the string descriptor is returned.



(2) Numerals Handling

The contents of a numeric variable are stored and calculated in an 8-byte area called the "floating-point accumulator (FAC)." This FAC area is handled differently depending upon the integer-type, single precision real-type and double precision real-type data.



For the integer-type variable, the lower 8 bits out of the 16-bit value is stored at the 5th byte of FAC and the upper 8 bits at the 6th bit by the formation of the two's complement.

For the single precision real-type variable, the part from the 5th byte onward of FAC is used. In the exponent part at the 8th byte, the value of (exponent - 128) is stored. At the 7th byte, the upper 7 bits of the mantissa part are stored. The remaining one bit (the uppermost bit in the 7th byte) represents the sign of the mantissa parts, "0" showing positive and "1" negative. The decimal point is treated to have been placed at the left side of the uppermost bit of the mantissa part, namely, at the left side of the 7th bit of the 7th byte. The 5th and 6th bytes store the lower 8 bits and intermediate 8 bits of the mantissa part respectively.

For the double precision real-type variable, the entire area of the floating-point accumulator is used. The 1st to 7th bytes in this case are used for storing the mantissa part. The 5th to 8th bytes store the exponent part and the upper 3 bits of the mantissa part in the same formats as those for the single precision real-type variables. The 1st to 4th bytes store the lower 4 bytes of the mantissa part.

1.9.4 Caution concerning Machine Language Subroutine

For calling a machine language subroutine from BASIC, two methods are used, i.e., the method using the USR function and that using the CALL statement. At this time, the execution commencing address and argument are transferred between BASIC and the machine language program, and in both cases of the USR function and CALL statement, attention should be given to the following rules:

- At the instant when the program enters the machine language subroutine, the value at the segment of in the BASIC data area is set to all the 3 segment registers DS, ES, and SS.

- At the CS (code segment) when the program enters the machine language subroutine, a value is set which was specified by the DEF SEG statement immediately before. When the DEF SEG statement is not executed or the operand of the DEF SEG statement immediately before is omitted, the CS register is set to a segment value which is the same as those of other 3 segment registers.

- When the argument transferred from BASIC to the machine language subroutine is of the character type, what is actually transferred is the leading address (offset) value of the string description (3-byte).

- It is allowed to change the contents of the character string indicated by the string descriptor at the machine language subroutine side, but alteration of the string descriptor itself should be avoided.
- At the instant when the machine language subroutine enters execution, SP (stack pointer) indicates the stack area of 16 bytes (8 words) which has been prepared for the use of the machine language subroutine. If a stack area exceeding this area is required, it is necessary to set up an independent stack segment and stack pointer in the machine language subroutine.
- When the program returns from the machine language subroutine, the segment register and stack pointer which have undergone internal operation should be returned to their original values.
- For the return to the BASIC program, an Inter-segment Return instruction should be used.
- When an interrupt is inhibited in the machine language subroutine, the interrupt should be enabled before the operation returns to BASIC.

1.9.5 USR Function (Calling Machine Language Subroutine - 1)

The format of the USR function is as follows:

USR [<No.>](<argument>)

Here the <No.> means any No. (0 to 9) allocated by the DEF USR statement to the machine language subroutine to be called. If the <No.> is omitted, it is assumed that "USR0" has been specified.

For the <argument> , the name of the variable to be transferred to the machine language subroutine is written. Arguments of both the numeric and character types can be specified. The argument should be specified by all means. Even if the argument is not required on the machine language subroutine side, a dummy argument should be written.

The execution commencing address of the machine language subroutine is defined by using the DEF USR statement as the offset value in the segment which was declared at the time of calling.

When the machine language subroutine is called by the USR function, the AL register stores a value which represents the type of the argument. This value corresponds to each type of the argument as shown in the table on the next page.

Value of AL-register	Type of argument
2	Integer (2 bytes)
3	Character
4	Single precision real
8	Double precision real

If the argument is of the character type, the leading address in the string descriptor (3-byte) will be stored in the DX register as an offset in the BASIC data segment. The actual character string is stored starting from the address specified by the 2nd and 3rd bytes of the string descriptor. Information concerning the length of the character string is stored at the 1st byte.

If the argument is of the integer type, single precision real type or double precision real type, the BX register points to the 5th byte in the 8-byte area of FAC. In this case, the address also is transferred as an offset value in the BASIC data segment. If access is made to the using area corresponding to any of the respective types of arguments, it is possible to receive the argument.

1.9.6 CALL Statement (Calling Machine Language Subroutine - 2)

The format of the CALL statement is as follows:

```
CALL <variable name>[(<argument>[,<argument>...])]
```

To the variable specified by the <variable name>, the execution commencing address of the machine language subroutine is assigned beforehand. The execution commencing address is represented by the offset value in the segment which was declared by the DEF SEG statement immediately before.

The <argument> is followed by the names of variables in continuation to be transferred to the machine language subroutine. It is not allowed to write constants, both numeric and character types, at this position.

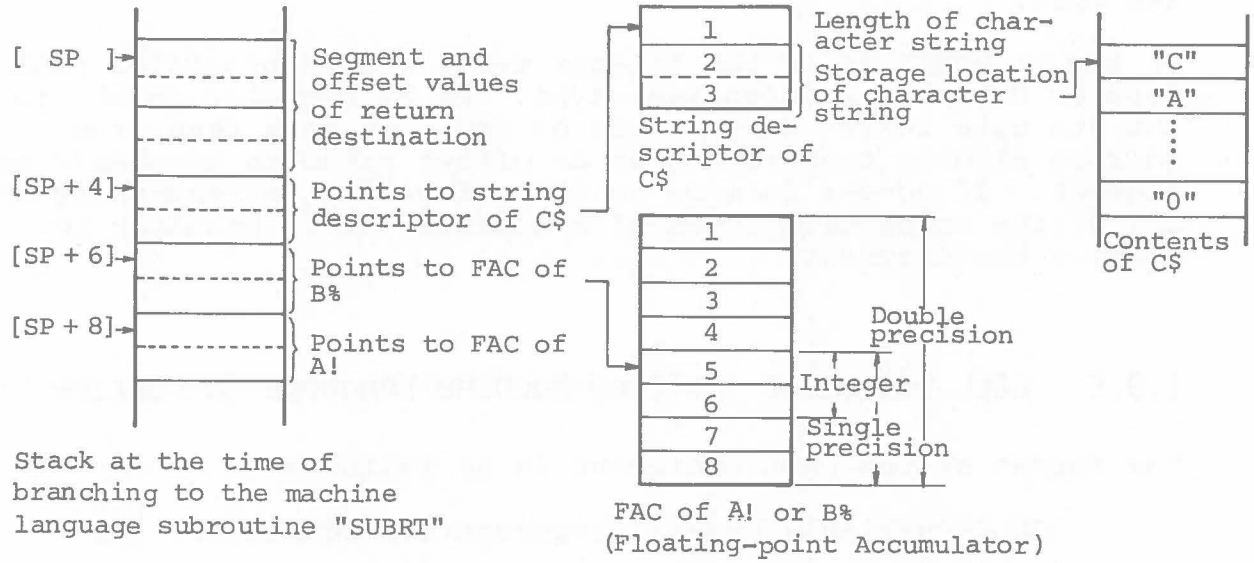
In the CALL statement, the machine language subroutine is called by the following procedure:

1. The storage locations of individual arguments are expressed as 2-byte offset values in the BASIC data segment and are piled up at the stack according to the sequence written in the row following the <argument>.

2. The segment and offset values of the return destination from the machine language subroutine are piled up at the stack.

3. Control is transferred to the macnine language subroutine at the address to be determined by the current segment value and by the offset value which is stored in the <variable> of the CALL statement.

When CALL SUBRT (A!, B%, C$) is executed, for instance, the stack will become as shown in the figure below.



CALL SUBRT (A!, B%, C$)

1.10 Others

1.10.1 RS232C Communication Ports

(1) Ports

BASIC is subject to control by the communication ports called "RS232C" which meet the non-synchronizing serial data transfer standard specification.

Four communication ports at the maximum can be installed and have the respective device names, i.e., COM1: to COM4:.

These ports can be used for communication to other machine (such as a computer) which has RS232C ports.

To use the communication ports, first the file should be opened as in the case of other I/O devices.

The OPEN "COM statement is used to open the communication file. The OPEN "COM statement obtains correspondence between the communication ports (Circuit No.) and File No. and sets the communication baud rate (number of bits communicated during 1 sec), parity, data length and stop-bit width. The default value at the time of port omission is as follows: 300 bauds, even parity, a character of 7-bits in length and stop bit 1.

(2) Input Interrupt

An exclusive use instruction is provided to permit inputting to the RS232C communication ports by using an interrupt. The ON COM(n) GOSUB statement defines the Commencing Line No. of the interrupt processing routine with respect to the input interrupt from the communication circuit. The symbol "n" is not File No., but Circuit No. The interrupt is disabled when Commencing Line No. is defined as "0" or the COM OFF statement is executed.

The interrupt occurs when the COM(n) ON is executed and the data is inputted to the port in question.

When the interrupt occurs, operation branches to Line No. which has been defined beforehand. During the processing routine, the COM(n) STOP statement is in the state of being executed and no further interrupt will be given, but if there is input in the port, such a state will be stored and an interrupt will be effective immediately after operation returns from the interrupt processing routine.

Functions EOF, LOC, and LOF check the state of the input buffer. The EOF function gives "-1" when the buffer is empty and gives "0" when the buffer is not empty.

1.10.2 Error Processing

When an error occurs during program execution, BASIC displays an error message and suspends execution. If the generated error is recoverable, the error is recovered by the error recovery processing routine provided at the branching destination of the ON ERROR GOTO statement, and the execution of the original program will be continued. The following statements and system variables are provided for error processing:

```
ON ERROR GOTO
RESUME
ERROR
ERR
ERL
```

An error which requires program correction or which has been caused by an anomaly of hardware cannot be recovered in the program.

(1) Declaring Error Processing Routine

For performing error recovery processing, first the execution commencing address of the error processing routine is specified by the ON ERROR GOTO statement.

Example: `ON ERROR GOTO 500`

If an error occurs hereafter, operation proceeds to Line 500. This is called an error interrupt. The ON ERROR GOTO statement is a sort of declaration statement approving an error interrupt. This declaration is released by

```
ON ERROR GOTO 0
```

or by the execution of the CLEAR command.

The declaration "ON ERROR GOTO 0," when it is executed in the ordinary program, releases the declaration of the error processing routine, but when it is executed in the error processing routine, it suspends error processing and displays an error message.

An error generated by erroneous keying-in of data during the execution of the INPUT statement causes no error interrupt and

thus cannot be processed by the error processing routine. At this time, a message "? Redo from Start" is displayed and the machine will wait for keying-in.

(2) Investigating Error Information

System variables ERR and ERL are provided to facilitate error recovery processing in the error processing routine.

Variable ERR indicates an error code corresponding to the error occurrence. For the meaning of error codes, refer to Appendix A.

Variable ERL indicates Line No. of the line on which an error has occurred. When an error occurs in the command mode, 65535 is assigned.

By using these two vriables, operation branches to the recovery processing routine corresponding to the location and type of the error which has occurred. When the value of ERL is to be examined, write Line No. to the right of the equal sign. Otherwise, the portion at which Line No. is re-assigned by the RENUM command will not be corrected.

Example: IF ERL=20 THEN 520

(3) Return from Error Processing Routine

After the error recovery processing is completed and the continued execution of the program has become possible, operation returns from the processing routine by executing the RESUME statement.

RESUME (or RESUME 0)

Execution is resumed starting from the line on which the error occurred.

RESUME NEXT

Execution is resumed starting from the line next to the line on which the error occurred.

RESUME 100

Execution is resumed starting from Line 100.

(4) Error Simulation

Error generation can be simulated by using the ERROR statement.

ERROR <error code>

When any one of error codes from 0 to 255 is specified in this format, an error which corresponds to the code is generated. If no error processing routine has been declared, the corresponding error message will be displayed and program execution stopped.

If the error processing routine has been declared, Line No. of the line on which this ERROR statement has been written will be assigned to ERL, a specified error code will be assigned to ERR, and the error processing routine will be executed.

Example:
```
ERROR 11
Division by zero
Ok
```

When an error code is undefined, a message "Unprintable error" is displayed. An error code which is not used in BASIC can be used by the user.

If a processing routine with respect to the error code defined by the user is provided beforehand in the error processing routine, specification of this code in the ERROR statement will make it possible to execute the user's processing routine, as if the BASIC error processing subroutine were called.

(5) Program Example

The program shown below examines the size of the specified disk file. The name of the file to be examined is keyed-in. The error in mis-specification of the file name which is liable to occur frequently can be recovered by the provision of an error recovery routine. The contents of this error recovery processing consists simply of displaying a caution message for making it possible to re-input the file name.

```
100 ON ERROR GOTO 200
110 INPUT "FILENAME"; N$
120 OPEN "N$" AS 1
130 PRINT LOF(1)
140 CLOSE
150 GOTO 110
200 IF ERR=53 THEN PRINT "NOT FOUND":RESUME 110
210 IF ERR=64 THEN PRINT "BAD FILE NAME":RESUME 110
220 ON ERROR GOTO 0
```

Line 100 declares that the error processing routine begins on Line 200. Line 110 inputs the name of the file for examining the file size; Line 120 opens the file.

The LOF function on Line 130 examines the file size. For the details of the LOF function, refer to Chapter 3.

The part from Line 200 onwards constitutes the error processing routine.

Lines 200 and 210 display error messages for 2 types or recoverable errors and resume execution starting from the entry of the file name on Line 110. Since these 2 errors are liable to occur only in the OPEN statement on Line 120, judgement by ERL is omitted.

When other types of errors occur, execution of Line 220 suspends the error recovery processing, displays the error message and suspends the program run.

Chapter II

GW-BASIC COMMANDS AND STATEMENTS

Canon AS-100

CHAPTER 2

GW-BASIC COMMANDS AND STATEMENTS

All of the GW-BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format : Shows the correct format for the instruction. See below for format notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

—— Format Notation ——

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.

2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.

3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

Format : AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at line number and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing [CTRL] + [C]. The line in which [CTRL] + [C] is typed is not saved. After [CTRL] + [C] is typed, GW-BASIC returns to command level.

Example: AUTO 100,50 — Generates line numbers 100, 150, 200 .....

AUTO — Generates line numbers 10, 20, 30, 40 .....

2.2 BEEP

Format : BEEP

Purpose: To beep the speaker.

Remarks: The BEEP statement sounds the speaker at 800 Hz tone for 1/4 of a second. Both BEEP and PRINT CHR$(7); have the same effect.

Example:

```
20 IF X < 20 THEN BEEP
```

This example specifies a beep when X is greater than 20.

2.3 BLOAD

Format : BLOAD <filename>[,<offset>]

where:

<filename> is a string expression containing the device and filename.

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address at which loading is to start in the segment declared by the last DEF SEG statement.

Purpose: Load a memory image file into memory.

Remarks: The BLOAD statement loads data anywhere in user memory. Any program that has been assembled and is a memory image file can be loaded with this command. A memory image file is the final executable form of a file, in which all definitions and references have been resolved. BLOAD is often used to load machine language programs, but it is not restricted to machine language; Pascal or FORTRAN programs, for example, can be BLOADed after they have been assembled.

BLOAD observes the following rules:

1. If device is omitted, the current drive is assumed.

2. If the device is omitted and the filename is less than 1 character or longer than 8 characters, a "Bad file name" error is issued and the load is aborted.

3. If the device is specified and the filename is omitted, the next memory image file encountered on the disk is loaded.

4. If offset is omitted, the offset specified at BSAVE is assumed. Therefore, the file is loaded into the same location it was saved from.

5. If offset is specified, a DEF SEG statement must be executed before the BLOAD. When offset is given, GW-BASIC assumes the user wants to BLOAD at an address other than the one saved. The last known DEF SEG address will be used.

CAUTION

BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over the GW-BASIC stack, an GW-BASIC program, or a variable area.

Example:
```
10 'Load screen buffer
20 DEF SEG=&HB800 'Point segment at screen buffer
30 BLOAD"PICTURE",0 ' Load file PCITURE into
screen buffer
```

Note that the DEF SEG statement in Line 20 and the offset of 0 in 30 guarantee that the correct address will be used.

The BSAVE example in the next section illustrates how PICTURE was saved.

NOTE : BLOAD is not restricted to machine language programs. Any segment may be specified as the source or target for BLOAD by using the DEF SEG statement. This provides a useful way of saving and displaying graphic images by allowing the video screen buffer to be read or written from disk.

2.4 BSAVE

Format : BSAVE <filename>,<offset>,<length>

where:

<filename> is a string expression containing the device and filename.

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset address to start saving from in the segment declared by the last DEF SEG statement.

<length> is a numeric expression returning an unsigned integer in the range 1 to 65535. This is the length is bytes of the memory image file to be save.

Purpose: Allows portions of memory to be saved on disk.

Remarks: The BSAVE statement allows portions of memory to be saved on disk. BSAVE is often used to save machine language programs, but is not restricted to machine language; for example, Pascal or FORTRAN programs that have been assembled can be BSAVED.

The following rules are observed:

1. If device is omitted, the current disk is assumed.

2. If the filename is less than 1 character or longer than 8 characters, a "Bad file name" error is issued and the save is aborted.

3. If offset is omitted, a "Bad file name" error is issued and the save is aborted. A DEF SEG statement must be executed before the BSAVE. The last known DEF SEG address will be used for the save.

4. If length is omitted, a "Bad file name" error is issued and the save is aborted.

Example:

```
10 'Save the screen buffer
20 DEF SEG=&HB800 'Point segment at screen buffer
30 BSAVE"PICTURE",0,16384 'Save screen buffer in
file PICTURE
```

In this example, the DEF SEG statement sets the segment address as the start of the screen buffer. An offset of 0 and length 16384 specifies that the entire 16K screen buffer is to be saved.

2.5 CALL

Format : CALL <numvar>[(<arg 1>[,<arg 2>]...)]

where:

<numvar> is a numeric variable or value that is the offset in memory of the subroutine. The segment value is determined by the DEF SEG statement.

<arg n> are the arguments that are to be passed to the subroutine.

Purpose: To call an machine language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function.)

The CALL statement generates the same calling sequence used by GW-BASIC compilers.

Example:
```
110 MYROUT=&HD000
120 DEF SEG=&HC000
130 CALL MYROUT(I,J$,K(0))
 :
 :
```
This example will call a subroutine at C000:D000 and pass variables I, J$, and array K as parameters.

2.6 CHAIN

Format : CHAIN [MERGE]<filespec>[,[<line>][,[ALL][,DELETE <range>]]]

where:

<filespec> is the name of the program that is to be chained.

<line> is the line number or an expression that evaluates to a line number in the called program where execution is to begin. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. <line> is not affected by a RENUM command.

Purpose: To call a program and pass variables to it from the current program.

Remarks: The COMMON statement may be used to pass variables.

Example 1:
```
10 REM This program demonstrates chaining using
common to pass variables.
20 REM Save this module on disk as "PROG1"
using the A option.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2)="VALUES BEFORE CHAINING."
70 B$(1)="": B$(2) =""
80 CHAIN "PROG2"
90  PRINT: PRINT B$(1): PRINT: PRINT B$(2): PRINT
100 END
```

Example 2:
```
10 REM The statement "DIM A$(2),B$(2)"
may only be executed once.
20 REM Hence, it does not appear in this module.
30 REM Save this module on the disk as "PROG2"
using the A option.
40 COMMON A$(),B$()
50 PRINT: PRINT A$(1);A$(2)
60 B$(1)="NOTE HOW THE OPTION OF SPECIFYING
A STARTING LINE NUMBER"
70 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
STATEMENT IN 'PROG1'."
80 CHAIN "RPOG1",90
90 END
```

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed.

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGED.

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Example 3:
```
10 REM This program demonstrates chaining using
the MERGE and ALL options.
20 REM Save this module on the disk as "MAINPRG".
30 A$="MAINPRG"
40 CHAIN MERGE "OVERLAY1",1010,ALL
50 END

1000 REM Save this module on the disk as
"OVRLAY1" using the A option.
1010 PRINT A$; " HAS CHAINED TO OVRLAY1."
1020 A$="OVRLAY1"
1030 B$="OVRLAY2"
1040 CHAIN MERGE "OVRLAY2",1010,ALL,
DETELE 1000-1050
1050 END

1000 REM Save this module on the disk as
"OVRLAY2" using the A option.
1010 PRINT A$; " HAS CHAINED TO ";B$:"."
1020 END
```

The line numbers if <range> are affected by the RENUM command.

Note : The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

The CHAIN statement leaves the files open during CHAINing.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

2.7 CIRCLE

Format : CIRCLE (<xcenter>,<ycenter>),<radius>[,<color> [,<start>,<end>[,<aspect>]]]

where:

<xcenter> is the x coordinate for the center of the circle.

<ycenter> is the y coordinate for the center of the circle.

<radius> is the size of the radius of the circle.

<color> is the numeric value of the color (0-7).

<start> is the start angle in radians.

<stop> is the stop angle in radians.

<aspect> is the ratio of the x dimension to the y dimension.

Purpose: To draw a circle or ellipse on the screen.

Remarks: The CIRCLE statement draws a circle or ellipse with a center and radius as indicated by the first of its argu-

ments. The default color is the foreground color. The start and end angle parameters are radian arguments between -2*PI and 2*PI which allow you to specify where drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (Note that this is different than adding 2*PI). The start angle may be less than the end angle.

The aspect ratio describes the ratio of the X radius to the Y radius. The default aspect ratio is 90/100 and will produce a 'round' circle on the standard monitor.

If the aspect ratio is less than one, then the radius is given in X-pixels. If it is greater than one, the radius is given in Y-pixels. The standard relative notation may be used to specify the center point.

The last point referenced after a circle is drawn is the center.

Example: Draw Pack man in the middle of the screen.

```
10 CIRCLE(320,200),100,7,-.4,-5.5
20 CIRCLE(300,150),10,7,,,1.5
30 PAINT(300,200),6,7
```

2.8 CLEAR

Format : CLEAR [,[<ds>][,<stack>]]

where:

<ds> is the size of the data segment, which if specified, sets the highest location available for use by GW-BASIC.

<stack> sets aside stack space for GW-BASIC. The default is 512 bytes or one-eighth of the vailable memory, whichever is smaller.

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory (size of data segment) and the amount of stack space.

Remarks: CLEAR frees up all memory used for data without erasing the program text. After a CLEAR, arrays are

undefined; numeric variables have a value of zero; string variables have a value of null; and any information set with any DEF statement is lost.

Restricting the size of the data segment is usually done to reserve space for machine language subroutines that are to be called by a basic program. Defining additional stack space would be useful while PAINTing very complex shapes or if very deep nesting of GOSUBs or FOR..NEXT loops are used.

GW-BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for GW-BASIC to use.

If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 512 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files.
Clears all COMMON and user variables.
Resets the stack and string space.
Releases all disk buffers.

Example: CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000

2.9 CLOSE

Format : CLOSE [[#]<filenum>[,[#]<filenum>]...]

where:

<filenum> is the number used on an OPEN statement.

Purpose: To conclude I/O to a disk file or device.

Remarks: The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file. A CLOSE with no arguments closes all open files.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: 100 CLOSE 1, 3

Causes the files and devices associated with file numbers 1 and 3 to be closed.

2.10 CLS

Format : CLS

Purpose: To clear the screen.

Remarks: The CLS statement clears the text and graphics data from the screen. The WIDTH statement will also force a screen clear if the new screen mode is different from the current screen mode. The screen may also be cleared by pressing the "CLEAR SCREEN" key.

Example: 10 CLS 'Clear the screen

2.11 COLOR

Format : COLOR [<foreground>][,<background>]

where:

<foreground> is the number of the color (0-7) that the characters will be written with. It is also the color that will be used as the default by the graphics routines.

<background> is the number of the color (0-7) that the characters will be written on.

Purpose: To select the colors that will be used to display the text and graphics information on the screen.

Remarks: The COLOR statement selects the foreground and background colors for the text and graphics display.

The colors (attributes) for the initial palette are shown below.

- For Color Display

Palette No.	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Yellow
7	White

- For Monochrome Display (one V-RAM model)

Palette No.	Attribute
0	Non-display
1	Standard brightness

- For Monochrome Display (two V-RAM model)

Palette No.	Attribute
0	Non-display
1	High brightness
2	Standard brightness blinking
3	Standard brightness

These colors will be used by the PSET, PRESENT, CIRCLE and LINE statement.

Any parameters outside the numeric ranges specified will result in an "Illegal function call" error. In this case, previous values are retained.

Foreground color may be the same as the background color, thus making displayed characters invisible.

Any parameter can be omitted. If a parameter is omitted, the previous value is retained.

The COLOR statement may not end with a comma (,). If it does, a "Syntax error" will result.

Initial values of definitions of palettes in Color Display are set at Palette No. 7 for <foreground> and Palette No. 0 for <background>.

Initial values of definitions of palettes in Monochrome Display are set at Palette No. 1 (one V-RAM model) or 3 (two V-RAM model) for <foreground> and at Palette No.0 for <background>.

Example: COLOR 4,6

2.12 COM(n)

Format : COM(n) ON
COM(n) OFF
COM(n) STOP

where:

n is the number of the COM channel (1-4)

Purpose: To enable or disable trapping of communications activity on the indicated communications channel.

Remarks: When an event is ON and if a non-zero line number is specified in the ON GOSUB statement, every time GW-BASIC starts a new statement it will check to see if the specified event has occurred (e.g., a COM character has come in). When an event is OFF, no trapping takes place, and the event is not remembered even if it takes place.

When a COM(n) STOP is executed, no trapping takes place on channel n, but the occurrence of an event is remembered so that an immediate trap will take place when a COM(n) ON statement is executed.

When a communications trap is detected on channel n, the trap automatically causes a COM(n) STOP for that channel, so recursive traps can never occur. A return from the trap routine automatically executes an ON statement unless an explicit OFF has been performed inside the trap routine.

A line number of zero disables trapping for that channel.

When an error trap takes place, all trapping is automatically disabled.

Event trapping will never occur when GW-BASIC is not executing a program.

A user can use the following statement:

```
RETURN  <line number>
```

to return to the GW-BASIC program at a fixed line number while still eliminating the GOSUB entry that the trap created. Note that this type of RETURN must be used with care. Any other GOSUB, WHILE, or FOR that was active at the time of the trap will remain active. If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT with FOR" error.

2.13 COMMON

Format : COMMON <variable>[,<variable>]...

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
  :
  :
```

Array variables used in a COMMON statement must be declared in a preceding DIM statement.

2.14 CONT

Format : CONT

Purpose: To continue program execution after [CTRL] + [C] has been typed or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example:

```
10 PRINT 10
20 END
30 PRINT 30
RUN
 10
Ok
CONT
 30
Ok
```

2.15 DATA

Format : DATA <constant>[,<constant>]...

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s).

Remarks: DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a

line or where the lines are placed in the program.

Constant may be numeric constants in any format; i.e., fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

Example: See "READ" Statement

2.16 DEF FN

Format : DEF FN<name>[(<arg 1>[<arg 2>]...)]=<definition>

where:

<name> must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<arg n> is a list of variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

<definition> is an expression that performs the operation of the function. It is limited to one line.

Purpose: To define and name a function that is written by the user.

Remarks: Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A varaible name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
.
:
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
:
:
```

Line 410 defines the function FNAB.
The function is called in line 420.

2.17 DEF -INT,-SNG,-DBL,-STR

Format : DEF<type> <letter>[-<letter>][,<letter>[-<letter>]]

where:

<type> is INT, SNG, DBL, or STR

Purpose: To declare varaible types as integer, single precision, double precision, or string.

Remarks: Any variable names beginning with the letter(s) specified will be considered the type of variable specified in the type portion of the statement. However, a type declaration character (%,!,# or $) always takes precedence over a DEF <type> statement.

If no type declaration statements are encountered, GW-BASIC assumes all variables without declaration characters are single precision variables.

Example: 10 DEFDBL L-P — All variables beginning with the letters L, M, N, O, and P will be double precision variables.

10 DEFSTR A — All variables beginning with the letter A will be string variables.

10 DEFINT I-N,W-Z — All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.18 DEF SEG

Format : DEF SEG[=<address>]

where:

<address> is a numeric expression returning an unsigned integer in the range 0 to 65535.

Purpose: To define the current "segment" of storage.

Remarks: The address specified is saved for use as the segment required by the PEEK, POKE, and CALL statements or the USR function. Assigns the current value to be used by a subsequent CALL or USR function call.

Any value outside the address range will result in an "Illegal function call" error. The previous value will be retained.

If the address is omitted, the segment to be used is set to the GW-BASIC data segment (DS). This is the default value.

If the address is given, it should be a value based on a 16-byte boundary. For the PEEK, POKE, or CALL statements, or for the USR function, the value is shifted left 4 bits to form the Code Segment address for the subsequent call instruction. GW-BASIC does not perform any checking to assure that the resulting segment + offset value is valid.

Example:
```
10 DEF SEG=&HB800 'Set segment to screen buffer
20 DEF SEG 'Restore segment to GW-BASIC DS
```

Note : DEF and SEG must be separated by a space.

2.19 DEF USR

Format : DEF USR[<digit>]=<offset>

where:

<digit> is any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If digit is omitted, DEF USR0 is assumed.

<offset> is the starting address of the USR routine. See "Machine Language Subroutines" section of this manual.

Purpose: To specify the starting address of a machine language subroutine that will be called via the USR function.

Remarks: The value of offset is added to the current segment value to obtain the actual starting address of the USR routine. See the DEF SEG statement in this chapter.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
 .
 :
190 DEF SEG =&H4000
200 DEF USR0=&H5000
210 X=USR0(Y^2/2.89)
 :
 .
```

This example will call absolute memory location 45000H.

2.20 DELETE

Format : DELETE [<first>][-<last>]

where:

<first> is the first line number to be deleted.

<last> is the last line number to be deleted.

Purpose: To delete program lines.

Remarks: GW-BASIC always returns to command level after a DELETE is executed. If the first or last line specified does not exist, an "Illegal function call" error occurs.

A period (.) may be used in place of the line number to indicate the current line.

Example:

DELETE 40	Deletes line 40.
DELETE 40-100	Deletes lines 40 through 100, inclusive.
DELETE -40	Deletes all lines up to and including line 40.

2.21 DIM

Format : DIM <variable>(<subscripts>)[,<variable>(<subscripts>)]...

where:

<variable> is a legal variable name.

<subscripts> are the maximum number of elements for each dimension of the array. There can be up to 255 subscripts but the maximum size of the array cannot exceed the amount of memory available.

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of numeric arrays to an initial value of zero. String arrays elements are all variable length and are initialized to null (0 length).

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
 :
 :
```

2.22 DRAW

Format : DRAW <string>

where:

<string> is one of the subcommands described below.

Purpose: Draws lines as indicated by the subcommands described below.

Remarks: The DRAW statement combines many of the capabilities of the other graphics statements into an easy to use object definition language called Graphics Macro Language. A GML command is a single character within a string, optionally followed by one or more arguments.

Each of the following subcommands begins movement from the "current graphics position." This is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen when a program is run.

The following commands move one unit if no argument is supplied.

U [n] Move up (scale factor * n) points
D [n] Move down
L [n] Move left
R [n] Move right
E [n] Move diagonally up and right
F [n] Move diagonally up and left
G [n] Move diagonally down and left
H [n] Move diagonally down and right

M x,y move absolute or relative. If x is preceded by a "+" or "-", x and y are added to the current graphics position and connected with the current position by a line. Otherwise, a line is drawn to point x, y from the current cursor position.

The following prefix commands may preceded any of the movement commands:

B — Move but don't plot any points.

N — Move but return to original position when done.

An — Set angle n. n may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so they will appear the same size as with 0 or 180 degrees on a monitor screen with the standard aspect ratio of 4/3.

Cn — Set foreground color n. n may range from 0 to 7.

Sn — Set scale factor. n may range from 1 to 255. The scale factor multiplied by the distances given with U, D, L, R, or relative M commands gives the actual distance traveled.

X<string>; — Execute substring (not supported by GW-BASIC Compiler). This powerful command allows you to execute a second substring from a string, much like GOSUB in GW-BASIC. You can have one string execute another, which executes another, and so on.

Numeric arguments can be constants like "123" or "=variable" where variable is the name of a variable.

Example: The following statements will draw a blue diamond with green lines emanating from the vertices.

```
10 CLS
20 A$ = "X B$ ; X C$"
30 B$ = "C1;F10"
40 C$ = "C2;ND10"
50 FOR I = 0 TO 3
60 DRAW "A=I ; XA$;"
70 NEXT
```

2.23 EDIT

Format : EDIT <line>

where:

<line> is the line number of a existing program line.

Purpose: To display a line for editing by the screen editor.

Remarks: The EDIT statement is used with the Full Screen Editor to display a specified line and position the cursor at the beginning of that line. The line may then be modified using the subcommands described in the section on the "Full Screen Editor." If there is no such line, an "Undefined line number" error results.

If you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter the Full Screen Editor at the current line. "." refers to the last line referenced by an EDIT statement, LIST statement, or error message.

Example: EDIT 20

displays Line 20 and places the cursor at the beginning of the line. The Full Screen Editor subcommands may now be used to edit the line.

```
20 PRINT "ADDRESS:"
EDIT .
```

displays Line 20 so that the line can be edited by the Full Screen Editor.

2.24 END

Format : END

Purpose: To terminate program execution, close all files, and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END closes all open files or devices, and END does not cause a "Break in line nnnnn" message to be printed. An END statement at the end of a program is optional. GW-BASIC always returns to command level after an END is executed.

Example: 520 IF K=1000 THEN END ELSE GOTO 20

2.25 ERASE

Format : ERASE <arrayname>[,<arrayname>]...

where:

<arrayname> is the name of an existing array.

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Duplicate definition" error will occur.

If the array specified in <arrayname> does not exist, an "Illegal function call" error will occur.

Example:
```
10 A=5 : A(1)=6
20 ERASE A
30 DIM A(4,5)
40 PRINT A
50 END
RUN
 5
Ok
```

2.26 ERROR

Format : ERROR <n>

where:

<n> is the error code to simulate.

Purpose: To simulate the occurrence of a BASIC error, or to allow error codes to be defined by the user.

Remarks: The value of <n> must be greater than 0 and less than 255. If the value of <n> equals an error code already in use by BASIC, the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed.

To define your own error code, use a value that is greater than any used by GW-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to GW-BASIC.) This user-defined error code may then be conveniently handled in an error handling routine.

Execution of an ERROR statement for which there is no error handling routine causes an error message to be printed and execution to halt. If an ERROR statement specifies a code for which no error message has been defined, GW-BASIC will use the "Unprintable error" error message.

Example:

```
LIST
10 S=10
20 T=5
30 ERROR S+T
40 END
Ok
RUN
String too long in line 30


Ok
ERROR 15            (You type this line.)
String too long     (BASIC types this line.)
Ok
```

Example:

```
 :
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B=5000 THEN ERROR 210
 :
400 IF ERR=210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL=130 THEN RESUME 120
 :
```

2.27 FIELD

Format : FIELD [#] <file number>,<field width> AS <string variable>[,<field width> AS <string variable>,...]

where:

<file number> is the number under which the file was OPENed.

<field width> is the number of characters to be allocated to <string variable>.

<string variable> is a string variable which will be used for random file access.

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example:

```
10 OPEN "R",#1,"RANDFILE.TST",25
20 FIELD#1,10 AS A$, 15 AS B$
 :
```

Note : Do not use a FIELDed varaible name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer.

If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.28 FILES

Format : FILES [<filespec>]

where:

<filespec> is a string expression for the file specification.

Purpose: To display the names of files residing on a disk.

Remarks: If <filespec> is omitted, all the files on the currently selected drive will be listed.

All files matching the filename are displayed. The filename may contain question mark (?). A question mark will match any character in the name or extension. An asterisk (*) as the first character of the name or extension will match any name or any extension.

If a drive is specified as part of <filespec>, then files which match the specified filename on the disk in that drive are listed. Otherwise, the current drive is used.

Example: FILES

This will display all files on the current drive.

FILES "*.BAS"

This will display all files with an extension of .BAS on the current drive.

FILES "B:*.*"

This will display all files on drive B.

2.29 FOR...NEXT

Format : FOR <variable>=x TO y [STEP Z]

```
:
NEXT [<variable>][,<variable>...]
```

where:

<variable> is used as a counter.

x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR ...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most

recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:
```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:
```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:
```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

2.30 GET (Files)

Format : GET [#]<file number>[,<record number>]

where:

<file number> is the number under which the files was OPENed.

<record number> is the number of the reocrd to be read in the range 1 to 32767.

Purpose: To read a record from a random file into a random buffer.

Remarks: If <record number> is omitted, the next record (after the last GET) is read into the buffer.

After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

GET may also be used for communications files. In this case, <record number> is the number of bytes to read from the communication buffer.

Example:
```
10 OPEN,"R",#1,"ADDRESS",50
20 FIELD #1, 20 AS NAME$,30 AS ADDR$
30 GET #1
40 PRINT NAME$, ADDR$
```

2.31 GET(Graphics)

Format : GET (x1,y1)-(x2,y2),<array>

where:

xn is the x coordinate (0-639)

yn is the y coordinate (0-399)

<array> is the name of the array to transfer the screen image to.

Purpose: To read a block of points from an area of the screen into an array.

Remarks: The PUT and GET statements are used to transfer graphics images to and from the screen. PUT and GET make possible animation and high-speed object motion in either graphics mode.

The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ",B" option.

The array is used simply as a place to hold the image and can be of any type except string. It must be

dimensioned large enough to hold the entire image. The contents of the array after a GET will be meaningless when interpreted directly (unless the array is of integer type - see below)

1. Animation of an object is usually performed as outlined below.

2. PUT the object(s) on the screen.

3. Recalculate the new position of the object(s).

4. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).

5. Go to step one, this time PUTting the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

The storage format in the array is as follows: The data for each row of pixels is left justified on a byte boundary, so if there are less than a multiple of 8 bits stored, the rest of the byte will be filled out with zeros. The required array size in bytes is:

```
4+INT((x+7)/8)*y*3
```

Remember that BASIC assigns storage to arrays as follows:

2 for integer
4 for single precision
8 for double precision

It is possible to examine the x and y dimensions and even the data itself if an integer array is used.

The information from the screen is stored in the array as follows:

1. 2 bytes giving the x dimension in bits

2. 2 bytes giving the y dimension in bits

3. the data itself

The x dimension is in bytes 0 1 of the array, and the y dimension is found in bytes 2 3. It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

Example: 4+INT((x+7)/8)*y*3 say you want to GET a 10 by 12 pixel image into an integer array. The number of bytes required is 4+INT((10+7)/8)*12*3. So, you would need an integer array with at least 38 elements.

```
10  DIM DAT%(38)
 :
120 GET (0,0)-(9,11),DAT%
```

2.32 GOSUB...RETURN

Format : GOSUB <line number>

```
 :
RETURN
```

where:

<line number> is the first line of the subroutine.

Purpose: To branch to and return from a subroutine.

Remarks: A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.33 GOTO

Format : GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
?Out of data in 10
Ok
```

2.34 IF...THEN...ELSE, IF...GOTO...ELSE

Format : IF <expression>[,] THEN <clause>[[,]ELSE<clause>]

IF <expression>[,] GOTO <line number>[[,]ELSE<clause>]

where:

<clause> may be a BASIC statement or sequence of statements, or it may be simply the number of a line to branch to.

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example,

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

Note : When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1: 200 IF I THEN GET#1,I

This statement GETs record number I if I is not zero.

Example 2:
```
100 IF(I<20) and (I>10) THEN DB=1983-1:GOTO 300
110 PRINT "OUT OF RANGE"
 :
 :
 :
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3: 210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$

This statement causes printed output to go either to the screen or the printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the printer, otherwise output goes to the screen.

2.35 INPUT

Format : INPUT[;][<"promt string">;]<variable>[,<variable>]...

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDAY",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of varaibles in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Example 1:
```
10 INPUT X
20 PRINT X;"SQUARED IS";X^2
30 END
RUN
?  5       (The 5 was typed in by the user
            in response to  the question mark)
 5 SQUARED IS 25
Ok
```

Example 2:
```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946
Ok
```

2.36 INPUT#

Format : INPUT#<file number>,<variable>[,<variable>]...

where:

<file number> is the number used when the file was OPENed for input.

<variable> is the name of a variable that will have an item in the file assigned to it.

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: The sequential file may reside on disk, if may be a sequential data stream from a communications adapter, or it may be the keyboard (KYBD:).

The type of data in the file must match the type specified by the variable name. Unlike INPUT, no question mark is printed with INPUT#.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example:
```
10 OPEN "I",#1,"ADDRESS"
20 INPUT#1,COD$,ADDR$,TEL$,NAM$
30 PRINT "CODE",COD$
40 PRINT "ADDRESS",ADDR$
50 PRINT "TELEPHONE",TEL$
60 PRINT "NAME",NAM$
70 END
```

2.37 KEY

Format :
```
KEY n,x$
KEY LIST
KEY ON
KEY OFF
```

where:

n is the function key number (1-12)

x$ is the text assigned to the specified key.

Purpose: Assigns softkey values to function keys and displays the values.

Remarks: The KEY statement allows function keys to be designated for special "softkey" functions. Each of the twelve function keys may be assigned a 15-byte string which, when that key is pressed, will be input to GW-BASIC.

Initially, the softkeys are assigned the following strings:

F1 - LIST	F7 - TRON↵
F2 - RUN↵	F8 - TROFF↵
F3 - LOAD"	F9 - LLIST
F4 - SAVE"	F10- EDIT
F5 - CONT↵	F11- FILES↵
F6 - ,"LPT1:"↵	F12- CHR$(

Once softkeys have been designated, they can be displayed with the KEY ON, KEY OFF, and KEY LIST statements.

KEY ON causes the Soft Key values to be displayed on the 25th line on the CRT screen. When the screen width is 40 charaters, four of the twelve softkeys are displayed; when the width is 80, eight softkeys are displayed. In either screen width, only the first 7 characters of each key are displayed.

KEY OFF erases the Soft Key display from the 25th line, making that line available for program use. It does not disable the function keys. OFF is the default state for the Soft Key display.

KEY LIST displays all twelve softkey values on the screen, with all 15 characters of each key displayed.

If the function key number is not in the range 1-12, an "Illegal function call" error is produced, and the previous key string expression is retained.

Assigning a null string (string of length 0) to a softkey disables the function key as a softkey.

When a softkey is assigned, the INKEY$ function returns one character of the softkey string per invocation.

Example:

```
50 KEY ON 'Displays the softkey on 25th line
60 KEY OFF ' Erases softkey display
70 KEY 1,"MENU"+CHR$(13) ' Assigns the string
"MENU" followed by a carriage return
to softkey 1.
```

Such assignments might be used to speed data entry.

```
80 KEY 1,""  'Disables softkey 1
```

The following routine initializes the first 5 softkeys:

```
10 KEY OFF 'Turns off key display during
     initialization
20 DATA KEY1,KEY2,KEY3,KEY4,KEY5
30 FOR I=1 TO 5
40 READ SOFTKEY$(I)
50 KEY I,SOFTKEY$(I)
60 NEXT I
70 KEY ON 'Displays new softkeys
```

2.38 KEY(n)

Format :

```
KEY(n) ON
KEY(n) OFF
KEY(n) STOP
```

where:

n is the key number (1-16).

Purpose: To enable or disable trapping of the specified key in a BASIC program.

Remarks: When a KEY(n) ON statement has been executed and a non-zero line number is specified, every time GW-BASIC starts a new statement it will check to see if the specified event has occurred (e.g., a key has been depressed).

When a OFF, no trapping takes place, and the event is not remembered even if it takes place.

When a STOP is executed, no trapping takes place, but the occurrence of an event is remembered so that an immediate trap will take place when a ON statement is executed.

When a trap is detected, the trap automatically causes a STOP, so recursive traps can never occur. A return from the trap routine automatically executes an ON statement unless an explicit OFF has been performed inside the trap routine.

A line number of zero disables trapping.

- When an error trap takes place, all trapping is automatically disabled.

Event trapping will never occur when GW-BASIC is not executing a program.

A user can use the following statement:

```
RETURN <line number>
```

to return to the GW-BASIC program at a fixed line number while still eliminating the GOSUB entry that the trap created. Note that this type of RETURN must be used with care. Any other GOSUB, WHILE, or FOR that was active at the time of the trap will remain active.
IF the trap comes out of a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT without FOR" error.

Example:

```
10  ON KEY(1) GOSUB 90
20  KEY(1) ON
30  CLS
40  R=RND(1)*8
50  C=INT(R)
60  COLOR C
70  M=R*20:IF M<32 THEN 40 ELSE PRINT CHR$(M);
80  GOTO 40
90  'INTERRUPT ROUTINE
100 CLS
110 FOR I=1 TO 23
120     PRINT SPC(I*2);"CANON AS-100"
130 NEXT I
140 CLS
150 RETURN
```

2.39 KILL

Format : KILL <filename>

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

2.40 LET

Format : [LET] <varaible>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
     :
     :
     :
     :
    or

110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
     :
     :
     :
     :
```

2.41 LINE

Format : LINE [[STEP](x_1,y_1)]-[STEP](x_2,y_2)[[,[<color>][,B[F]]

where:

xn are the horizontal coordinates in the range of 0 to 639.

yn are the vertical coordinates in the range of 0 to 399.

(xn,yn) describes a point on the screen. (0,0) is the upper left hand corner of the screen.

<color> is the numeric (0-7) representation of the desired color that the line (box) will be drawn with.

B is the optional box parameter.

F is the optional box fill parameter.

Purpose: To draw a line or box on the screen.

Remarks: LINE is one of the most powerful graphics statement. It allows a group of pixels to be controlled with a single statement.

The simplest form of line is:

```
LINE -(x2,y2)
```

This will draw from the last point drawn to the point (x2,y2) using the foreground color.

We can include a starting point also:

```
LINE (0,0)-(639,399) 'draw diagonal line down
screen
LINE (0,100)-(639,100) 'draw bar across screen
we can indicate the color to draw the line in:
LINE (10,10)-(20,20),2   'draw in color 2
```

The final argument to line is ",B" -- box or ",BF" filled box. The syntax indicates we can leave out color and include the final argument as follows:

```
LINE (0,0)-(100,100),,B 'draw box in foreground
```

or include it:

```
LINE (0,0)-(200,200),2,BF 'filled box color 2
```

The ",B" tells BASIC to draw a rectangle using the points (x1,y1) and (x2,y2) as upper left and lower right corners respectively. This is a convenient abbreviation of the following four LINE commands:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

The ",BF" means draw the same rectangle as ",B" but also fill in the interior points with the selected color.

When out of range coordinates are given the line command the coordinate which is out of range is given the closest legal value. In other words, negative values become zero, y values greater than 399 become 399 and x values greater than 639 become 639.

Another form of the LINE command allows the user to specify the coordinates of a point as an offset from the previous point. The STEP(xoffset,yoffset) form can be used wherever a coordinate is used. Note that all of the graphics statements and functions update the "more recent point used". In a line command if the relative form is used on the second coordinate it is relative to the first coordinate. The only other way "the most recently used" point is changed is that CLS initialize it to be the point in the moddle of the screen (320,200).

Example: Draw lines forever using random color --

```
10 CLS
20 LINE -(RND*319,RND*199),RND*4
30 GO TO 20
```

Draw vertical line pattern - line on line off

```
10 FOR X=0 TO 639
20 LINE (X,0)-(X,399),X AND 1
30 NEXT
```

Draw random size boxes filled with random colors

```
10 CLS
20 LINE -(RND*639,RND*399),RND*2,BF
30 GO TO 20
```

The following program segment as shown on the next page will print the time in the upper left hand corner of the screen and then simulate the second hand of a clock.

```
10 CLS
20 T$=TIME$ : LOCATE 1,1 : PRINT T$
30 SEC = VAL(MID$(T$,7,2))
40 S = 3*6.28/4+SEC*6.28/60
50 X=200*COS(S) : Y=200*SIN(S)
60 LINE (320,200)-STEP(X,Y),7
70 IF SEC = VAL(MID$(TIME$,7,2)) GOTO 70
80 LINE (320,200)-STEP(X,Y),0
90 GOTO 20
```

2.42 LINE INPUT

Format : LINE INPUT[;][<"prompt string">;]<string variable>

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing [CTRL] + [C] BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See example in the next section, "LINE INPUT# Statement".

2.43 LINE INPUT#

Format : LINE INPUT#<file number>,<string variable>

where:

<file number> is the number under which the file was OPENed.

<string variable> is the variable name to which the line will be assigned.

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential data file to a string variable.

Remarks: LINE INPUT# reads all characters in the sequential file up to a carriage return. If then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES     234,4    MEMPHIS
LINDA JONES    234,4    MEMPHIS
Ok
```

2.44 LIST

Format : LIST [<line>[-[<line>]]][,<filespec>]

where:

<line> is a valid line number in the range of 0 to 65529.

<filespec> is a string expression for the device or file specification.

Purpose: To list the program currently in memory on the screen or other specified device.

Remarks: LIST allows a program or a range of lines to be listed to the screen, disk file, or other specified device. If the optional device specification is omitted, the specified lines are listed to the screen.

If the line range is omitted, the entire program is listed.

When the hyphen(-) is used in a line range, three options are available

1. If only the first number is given, that line and all higher numbered lines are listed.

2. If only the second number is given, all lines from the beginning of the program through the given line are listed.

3. If both numbers are given, the inclusive range is listed.

Example:	LIST,"LPT1:"	Lists program on the printer
	LIST 10-20	Lists lines 10 through 20 on the screen
	LIST 10-,"SCRN:"	Lists lines 10 through last on the screen
	LIST -200,"FILE1.BAS"	Lists first through 200 lines to file "FILE1.BAS" on default disk drive.

2.45 LLIST

Format : LLIST [<line>[-[<line>]]]

where:

<line> is a valid line number in the range of 0 to 65529.

Purpose: To list all or part of the program currently in memory on the printer.

Remarks: BASIC always returns to command level after LLIST is executed.

Example: LLIST Prints a listing of the entire program

LLIST 10-30 Prints lines 10 through 30

LLIST 100- Prints all lines from 100 through the end of the program

LLIST -300 Prints first through lines 300

2.46 LOAD

Format : LOAD <filespec>[,R]

where:

<filespec> is a string expression of the device and filename of the program to be loaded into memory.

R is an optional switch that causes the program to be run after it is loaded.

Purpose: To load a program from the specified device into memory, and optionally run it.

Remarks: The LOAD statement causes a GW-BASIC program to be loaded into memory.

<filespec> consists of either a device name or a file descriptor. If specified, the non-disk device name must be four characters. The file descriptor consists of

[<drive>:]<filename>[.<extension>]

<drive> specifier is the one letter identifier associated with the disk drive. <filename> may be one to eight characters. <extension> may be one to three characters.

If the ,R option is included, the program will be run after loading. LOAD <filespec>,R is equivalent to RUN <filespec>.

Example: LOAD "MENU" Loads program "MENU" but does not run it.

LOAD "TEST",R Loads program "TEST" and runs it

2.47 LOCATE

Format : LOCATE [<row>][,[<col>][,[<cursor>]]]

where:

<row> is a line number (vertical) on the screen. Row should be a numeric expression returning an unsigned integer in the range 1 to 25.

<col> is the column number on the screen. It should be a numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending on screen width.

<cursor> is a numeric expression representing the desired color of the cursor.

Purpose: The LOCATE statement moves the cursor to a specified position on the screen. The optional cursor parameter may be used to select the color of the nonblinking cursor.

Remarks: Any value outside the specified ranges will result in an "Illegal function call" error. In this case, previous values are retained.

Any parameter may be omitted from the statement. If a parameter is omitted, the previous value is assumed.

Note that setting the cursor to the background color makes the cursor invisible.

Example:

```
10 LOCATE 1,1
```

Moves the cursor to upper left corner of the screen.

```
20 LOCATE ,,1
```

Makes the cursor visible; position remains unchanged.

2.48 LPRINT, LPRINT USING

Format : LPRINT [<list of expressions>][;]

LPRINT USING <string exp>;<list of expressions>[;]

Purpose: To print data on the printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the printer. See "PRINT Statement" and "PRINT USING Statement".

Example:
```
10 LPRINT "ABC"
20 END
```

2.49 LSET, RSET

Format : LSET <string variable> = <string expression>

RSET <string variable> = <string expression>

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions.

Note : LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines.

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

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

Example:
```
100 LSET A$=MKS$(AMT)
```

2.50 MERGE

Format : MERGE <filespec>

where:

<filespec> is a string expression for the file specification.

Purpose: Searches the disk for the spcified file and incorporates it into the program in memory. After the files have been merged, the new program resides in memory.

Remarks: The MERGE statement incorporates the lines from an ASCII file on disk into the program already in memory. The program being merged must have been saved in ASCII format. If it was not, a "Bad file mode" error results the program in memory remains unchanged.

If any of the line numbers in the file to be merged match those in memory, the lines from the file to be merged will replace those in memory.

Example: 10 MERGE "SUBRTN"

Incorporates all lines of the program SUBRTN into those of the file in memory.

2.51 MID$

Format : MID$(<string expl>,n[,m])=<string exp2>

where:

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

<string expn> is string expression.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

Example:
```
10 A$="KANSAS CITY, MO"
20 MIDS(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string.

2.52 NAME

Format : NAME <old filename> AS <new filename>

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:
```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.53 NEW

Format : NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

2.54 ON COM(n) GOSUB

Format : ON COM(n) GOSUB <line>

where:

n is the number of the communications adapter (1 - 4).

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

Remarks: A COM(n) ON statement must be executed to "activate" this statement for adapter n. After COM(n) ON, if a non-zero line number is specific in the ON COM(n) statement then every time BASIC starts a new statement, it will check to see if any characters have come in to the specified communications adapter. If so, it will perform a GOSUB to the specified line.

If COM(n) OFF was executed, no trapping takes place for the adapter and the event is not remembered even if it does take place.

If a COM(n) STOP statement has been executed, no trapping can take place for the adapter, but if a character is received, this is remembered so an immediate trap will take place when COM(n) ON is executed.

When the trap occurs, an automatic COM(n) STOP is executed so that recursive traps can never take place. The RETURN from the trap routine will automatically do a COM(n) ON unless an explicit COM(n) OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place, this automatically disables all trapping (including ERROR, COM and KEY).

Using a line of 0(zero) disables trapping of communications activity.

Typically, the communications trap routine will read an entire message from the communications line before returning. It is not recommended to use the communications trap for single character messages since at high baud rates, the overhead of trapping and reading for each individual character may cause the communications buffer to overflow.

You may use RETURN line if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example:
```
150 ON COM(1) GOSUB 500
 :
 :
499 STOP
500 REM Event trap subroutine for Communications
channel  1
```

```
510 C$=INPUT$(LOC(1),#1) : REM read all avaialble
characters
520 RETURN
```

2.55 ON ERROR GOTO

Format : ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If line number does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note : If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example: 10 ON ERROR GOTO 1000

2.56 ON...GOSUB , ON...GOTO

Format : ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of expression is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of expression is negative or greater than 255, an "Illegal function call" error occurs.

Example: 100 ON L-1 GOTO 150,300,320,390

2.57 ON KEY(n) GOSUB

Format : ON KEY(n) GOSUB <line>

where:

n is a numeric expression in the range of 1 to 16 indicating the key to be trapped, as follows:

1-12 Function keys F1 TO F12
13 Cursor up key
14 Cursor left key
15 Cursor right key
16 Cursor down key

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Remarks: A KEY(n) ON statement must be executed to "activate" this statement. After KEY(n) ON, if a non-zero line number is specified in the ON KEY(n) statement, then every time BASIC starts a new statement, it will check to see if the specified key was pressed. If so, it will perform a GOSUB to the specified line.

If a KEY(n) OFF statement is executed, no trapping takes place for the specified key and the event is not remembered even if it does take place.

If a KEY(n) STOP statement has been executed, no trapping can take place for the specified key, but if the key is pressed, it is remembered so that an immediate trap will take place when KEY(n) ON is executed.

When the trap occurs, an automatic KEY(n) STOP is executed so that recursive traps can never take place.

The RETURN from the trap routine will automatically do a KEY(n) ON unless an explicit KEY(n) OFF has been performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place, it automatically disables all trapping (including ERROR, COM, and KEY).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

If line is 0, this disables trapping of the specified key.

You may use RETURN line if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(n) ON has no effect on whether the Soft Key values are displayed at the bottom of the screen.

Example:

```
100 ON KEY(5) GOSUB 200
110 GOTO 110
120 REM CONTINUE HERE AFTER FK5 PRESSED
 :
 :
190 STOP
200 REM FUNCTION KEY 5 EVENT HANDLER SUBROUTINE
 :
 :
290 RETURN 120
```

2.58 OPEN

Format :

```
OPEN [<dev>]<filename>[FOR <mode 1>]
     AS [#]<file number>[LEN=<reclen>]

OPEN <mode 2>,[#]<file number>,[<dev>]
     <filename>[,<reclen>]
```

where:

<dev> is optionally part of the file name string and may be one of the following.

A: - D: Disk
KYBD: Keyboard - Input Only
SCRN: Screen - Output Only
LPT1: Printer 1
COM1: RS232C Communications 1

<filename> is a valid string literal or variable optionally containing <dev>. If <dev> is omitted, current disk is assumed.

<mode 1> determines the initial positioning within the file and the action to be taken if the file does not exist. The valid mode and actions taken are:

INPUT Position to the beginning of an existing file. A "File not found" error is given if the file does not exist.

OUTPUT Position to the beginning of the file. If the file does not exist, one is created.

APPEND Position to the end of the file. If the file does not exist, one is created.

If the FOR <mode 1> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the Random I/O mode. That is, records may be read or written at will at any position within the file.

<mode 2> is a string expression whose first character is one of the following.

"O" specifies sequential output mode
"I" specifies sequential input mode
"R" specifies random input/output mode

<file number> is an integer expression returning a number in the range 1 through 15. The number is used to associate an input/output buffer with a disk file or device. This association exists until a CLOSE statement is executed.

<reclen> is an integer expression in the range 1 to 32767. This value sets the record length to be used for random files (see the FIELD statement). If omitted, the record length defaults to 128 byte records.

Purpose: To allow input/output operations to take place to/from a file or to device.

Remarks: When a disk file is OPENed FOR APPEND, the position is initially at the end of the file and the record number is set to the last record of the file (LOF(x)/128). PRINT#, WRITE#, or PUT will then extend the file. The program may position elsewhere in the file with a GET statement. If this is done, the mode is changed to random and the position moves to the record indicated.

Any values entered outside of the ranges given will result in an "Illegal function call" error. The file is not opened.

If the file is opened as INPUT, attempts to write to the file will result in a "Bad file mode" error.

If the file is opened as OUTPUT, attempts to read the file will result in a "Bad file mode" error.

At any one time, it is possible to have a particular disk filename OPEN under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, to use different file numbers for different modes of access. Each file number has a different buffer, so several records from the same file may be kept in memory for quick access.

A file may NOT be opened FOR OUTPUT, however, on more than one file number at a time.

Example:

```
10 OPEN "DATA" FOR OUTPUT AS #1
```

or

```
10 OPEN "O",#1,"DATA"


30 OPEN "B:TEST" AS 1 LEN=256
```

or

```
30 OPEN "R",1,"B:TEST",256
```

2.59 OPEN "COM

Format : OPEN "COMn:[<speed>][,<parity>][,<data>][,<stop>]
[,BIN][,LF]" AS [#]<filenum>

where:

n is the number of the communication adapter (1 - 4).

<speed> is the baud rate of the device in bits per second (bps). Valid speeds are 110, 150, 300, 600, 1200, 2400, 4800, and 9600. The default is 300 bps.

<parity> is a one-character constant specifying the parity for transmit and receive. The possible entries are:

O - ODD: Odd transmit parity, odd receive parity checking.

E - EVEN: Even transmit parity, even receive parity checking.

N - NONE: No transmit parity, no receive parity checking.

The default is EVEN(E).

<data> is an integer constant indicating the number of transmit/receive data bits. Valid entries are 7 or 8 bits per byte. The default is 7.

<stop> is a constant indicating the number of stop bits. Valid value are 1 or 2. The default for baud rate greater than 110 is 1; for 110 baud rate is 2.

BIN specifies that binary input/output will e performed.

LF sends a linefeed following each carriage return.

Purpose: To open (and initialize) a communications channel for input/output.

Remarks: The OPEN "COM statement must be executed before a device can be used for RS232C communication.

NOTE - only the following combinations are valid:

Data Bits	Parity	Stop Bits
7	E	1
7	E	2
7	O	1
7	O	2
8	E	1
8	O	1
8	N	1
8	N	2

The OPEN "COM statement allocates a 128 byte buffer for input in the same manner as OPEN for disk files.

Any syntax errors in the OPEN "COM statement will result in a "Bad filename" error. The incorrect parameter will not be shown.

A "Device timeout" error will occur if Data Set Ready (DSR) is not detected.

LF is intended for those using communication files to print to a printer. When LF is specified, a linefeed character (0AH) is automatically sent after each carriage return character (0CH). (This includes the carriage return sent as a result of the width setting.) Note that INPUT# and LINE INPUT#, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed. The LF option is superseded by the BIN option.

In the BIN mode, tabs are not expended to spaces, a carriage return is not forced at the end-of-line, and Control-Z is not treated as end-of-file. When the channel is closed, Control-Z will not be sent over the RS232C line if the BIN option has been used. The BIN option supersedes the LF option.

Speed, parity, data, and stop must be listed in the order shown; LF, and BIN may be listed in any order.

Example: 10 "COM1:9600,N,8,1,BIN" AS #2

Will open communications channel 1 at a speed of 9600 baud with no parity bit, 8 data bits and 1 stop bit. Input/output will be in the binary mode. Other lines in the program may now access channel 1 as dev #2.

2.60 OPTION BASE

Format : OPTION BASE n

where:

n is 0 or 1.

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base on 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is one.

2.61 OUT

Format : OUT n,m

where:

n is the range of 0 to 65535.

m is the range of 0 to 255.

Remarks: The integer expression n is the port number, and the integer expression m is the data to be transmitted.

OUT is the complementary statement to the INP function. Refer to "INP function".

Example:

```
10 FOR I=&H2000 TO &H20FF
20    OUT I, 0
30 NEXT I
40 END
```

2.62 PAINT

Format : PAINT (<xstart>,<ystart>)[,<paint color>[,<boader color>]]

where:

<xstart>,<ystart> are the coordinates where painting should begin. Painting should always start on a non-border point.

<paint color> is the color to be placed in the filled area of the figure, in the range 0 to 7.

<border color> specifies the border color of the figure to be filled in, in the range 0 to 7.

Purpose: Fills a graphics figures with the color specified.

Remarks: The PAINT statement will fill in a graphics figure with the specified color (attribute). If the paint attribute is not given in the command, it will default to the foreground color. If the border attribute is not given in the command, it will default to the paint attribute, which is required.

This command can be used to fill any figure, but painting jagged edges or very complex figures may result in an "Out of memory" error. If this happens, the CLEAR statement must be given to increase the amount of stack space available.

Example:

```
10 CLS
20 CIRCLE (100,100),100,2
30 PAINT (100,100),2,2
```

The PAINT statement in Line 30 will fill in the circle drawn in Line 20 with color 2.

2.63 PALETTE, PALETTE USING

Format : PALETTE

PALETTE <palno>,<color>

PALETTE USING <array>

where:

<palno> is palette number in the range 0 to 7.

<color> is color number in the range 0 to 28.

<array> is an integer array. This form specifies a new value for each palette entry.

Purpose: To allow the user to access the hardware palette to select the actual colors (attributes) to be displayed.

Remarks: This statement lets the user select which colors (attributes) of the possible 28 color combinations will be used. These colors (attributes) are then used by all the other statements that refer to color (attribute).

The PALETTE statement without any parameters cause colors (attributes) to be set to their initial values. See the COLOR statement for these values.
The 28 valid hardware colors (attributes) are defined as follows.

• For Color Display:

Color No.	Color						Remarks
	r	R	g	G	b	B	
0	0	0	0	0	0	0	Black
1	0	0	0	0	0	1	Blue
2	0	0	0	0	1	1	
3	0	0	0	1	0	0	Green
4	0	0	0	1	0	1	Cyan
5	0	0	0	1	1	1	
6	0	0	1	1	0	0	
7	0	0	1	1	0	1	
8	0	0	1	1	1	1	
9	0	1	0	0	0	0	Red
10	0	1	0	0	0	1	Yellow
11	0	1	0	0	1	1	
12	0	1	0	1	0	0	Magenta
13	0	1	0	1	0	1	White
14	0	1	0	1	1	1	
15	0	1	1	1	0	0	
16	0	1	1	1	0	1	
17	0	1	1	1	1	1	
18	1	1	0	0	0	0	
19	1	1	0	0	0	1	
20	1	1	0	0	1	1	
21	1	1	0	1	0	0	
22	1	1	0	1	0	1	
23	1	1	0	1	1	1	
24	1	1	1	1	0	0	
25	1	1	1	1	0	1	
26	1	1	1	1	1	1	
27,28	1	1	1	1	1	1	

R: Red r: Half brightness red
G: Green g: Half brightness green
B: Blue b: Half brightness blue

• For Monochrome Display (one V-RAM model)

Color No.	Attribute
0	Non-display
1	Standard brightness
2 - 28	Standard brightness

• For Monochrome Display (two V-RAM model)

Color No.	Attribute			Remarks
	Blinking	High brightness	Standard brightness	
0	0	0	0	Non-display (black)
1	0	0	1	Standard brightness
2	0	1	0	High brightness
3 - 26	0	0	1	Standard brightness
27	1	0	1	Standard blinking
28	1	1	0	High brightness blinking

Eample:

```
10 FOR PN = 0 TO 7
20 READ PAL%(PN)
30 NEXT PN
40 PALETTE USING PAL%(7)
50 STOP
60 DATA 3,5,9,11,22,24,26,28
```

This program segment would read in 8 colors from the data statement and assign them to the PAL% array. Line 40 is the equivalent of:

```
41 PALETTE 0,PAL%(0)
42 PALETTE 1,PAL%(1)
43 PALETTE 2,PAL%(2)
44 PALETTE 3,PAL%(3)
45 PALETTE 4,PAL%(4)
46 PALETTE 5,PAL%(5)
47 PALETTE 6,PAL%(6)
48 PALETTE 7,PAL%(7)
```

2.64 PLAY

Format : PLAY <string>

where:

<string> is a string expression consisting of music commands as explained below.

Purpose: To play music as specified by 'string'.

Remarks: PLAY uses a concept similar to that used in the DRAW statement by embedding a Music Macro language into one statement. A set of subcommands, used as part of the PLAY statement itself, specifies the particular action to be taken.

The single character commands in PLAY are:

A-G Plays a note in the range A-G. # or + after the note specifies sharp; - specifies flat.

L[n] Sets the length of each note. L4 is a quarter note, L1 is a whole note, etc. n may be in the range 1 through 64.

The length may also follow the note when a change of length only is desired for a particular note. In this case, A16 is equivalent to L16A.

MF Sets music (PLAY statement) and SOUND to run in the foreground. That is, each subsequent note or sound will not start until the previous note or sound has finished. This is the default setting.

MB Sets music (PLAY statement) and SOUND to run in the background. That is, each note or sound is placed in a buffer allowing the GW-BASIC program to continue executing while the note or sound plays in the background. Up to 32 notes or rests can be played in the background at one time.

MN Sets "music normal" so that each note will play 7/8 of the time determined by the length (L).

ML Sets "music legato" so that each note will play the full period set by length (L).

MS Sets "music staccato" so that each note will play 3/4 of the time determined by the length (L).

N[n] Plays note n. n may range from 0 through 84 (in the 7 possible octaves, there are 84 notes). n = 0 means a rest.

O[n] Sets the current octave. There are seven octaves, numbered 0 through 6.

P[n] Specifies a pause, ranging from 1 through 64.

T[n] Sets the "tempo," or the number of L4's in one second. n may range from 32 through 255. The default is 120.

. A period after a note causes the note to play 3/2 times the length determined by L multiplied by T (tempo). Multiple periods may appear after a note. The period is scaled accordingly; for example, A. is 3/2, A.. is 9/4, A... is 27/8, etc. Periods may appear after a pause (P). In this case, the pause length may be scaled in the same way notes are scaled.

X Executes a substring. (not available with GW-BASIC Compiler)

Because of the slow clock interrupt rate, some notes will not play at higher tempos (L64 at T255, for example).

Example:
```
10 A$ = "BB-C"
20 B$ = "O4XA$:'
30 C$ = "L1CT50N3N4N5N6"
40 PLAY "P2XA$;XB$;XC$;"
```

2.65 POKE

Format : POKE n,m

where:

n must be in the range 0 to 65535.

m must be in the range 0 to 255.

Purpose: To write a byte into a memory location.

Remarks: The integer expression n is the address of the memory location to be POKEd. The integer expression m is the data to be POKEd.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. Refer to "PEEK function".

POKE and PEEK are useful for efficient data storage, loading machine language subroutines, and passing arguments and results to and from machine language subroutines.

BASIC does not do any checking on the address. So do not go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Example: 10 POKE &H5A00, &HFF

2.66 PRESET

Format : PRESET (<xcoordinate>,<ycoordinate>)[,<attribute>]

where:

(<xcoordinate>,<ycoordinate>) specifies the point on the screen to be colored.

<attribute> is the number of the color to be used.

Purpose: Sets the color of a specified point on the screen.

Remarks: RESET works exactly like PSET except that if the attribute is not specified, the background color is selected.

If an out-of-range coordinate is given, no action is taken, nor is an error message given.

Coordinates can be shown as absolutes, as in the above syntax, or the STEP option can be used to reference a point relative to the most recent point used. The syntax of the STEP option is:

STEP (<xoffset>,<yoffset>)

For example, if the most recent point referenced were (0,0), STEP (10,0) would reference a point at offset 10 from x and 0 from y.

Example:

```
10 FOR I=0 TO 100
20 PRESET (I,I)
30 NEXT 'draw diagonal line to (100,100)
40 FOR I=100 TO 0 STEP -1
50 PRESET (I,I),0
60 NEXT
```

clears out the line by setting each pixel to attribute 0.

2.67 PRINT

Format : PRINT [<list of expressions>][;]

? [<list of expressions>][;]

Purpose: To display data on the screen.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are displayed on the screen. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer

digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is out as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:
```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next printtzone.

Example 2:
```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:
```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.68 PRINT USING

Format : PRINT USING <string exp>;<list of expressions>[;]

Purpose: To print strings or numbers using a specified format.

Remarks: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;B$
30 RPINT USING "\  \";A$;B$
40 PRINT USING "\    \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Example:
```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

"#" A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRING USING "##.##   ";10.2,5.3,66.789,.234
10.2     5.30   66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

"+" A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

"-" A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40   +55.60     -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-   22.45     7.01-
```

"**" A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4   *-0.9   765.1
```

"$$" A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

"**$" The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

"," A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50
PRINT USING "####.##,";1234.5
1234.50,
```

"^^^^" Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";888888
 .8889E+06

PRINT USING "+.##^^^^";123
 +.12E+03
```

"_" An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!"12.34
!12.34!
```

The literal character itself may be an underscore by placing "_" in the format string.

"%" If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.69 PRINT# , PRINT# USING

Format : PRINT#<file number>, [USING<string exp>;]<list of exps>

where:

<file number> is the number used when the file was OPENed for output.

<string exp> is a string expression comprised of formatting characters as described in the previous section, "PRINT USING Statement."

<list of exps> is a list of the numeric and/or string expressions that will be written to the file.

Purpose: To write data to a sequential disk file.

Remarks: PRINT# does not compress data on the file. An image of the data is written to the file, just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the file, so that it will be input correctly from the file.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

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

would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

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

The image written to the file is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

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

would write the following image to the file:

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

and the statement

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

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the file, write double quotes to the file image using CHRS(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

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

and the statement

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

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

See also "WRITE# statement" in this chapter.

2.70 PSET

Format : PSET (<xcoordinate>,<ycoordinate>)[,<attribute>]

where:

(<xcoordinate>,<ycoordinate>) specifies the point on the screen to be colored.

<attribute> is the number of the color to be used.

Purpose: Sets the color of a specified point on the screen.

Remarks: When GW-BASIC scans coordinate values, it will allow them to be beyond the edge of the screen. However, values outside the integer range -32768 to 32767 will cause an "Overflow" error.

The coordinate (0,0) is always the upper left corner of the screen. The bottom right corner of the screen, therefore, is (639,399).

PSET allows the attribute to be left off the command line. If it is omitted, the default is the foreground attribute.

Example:
```
10 FOR I=0 TO 100
20 PSET (I,I)
30 NEXT 'draw diagonal line to (100,100)
40 FOR I=100 TO 0 STEP -1
50 PSET (I,I),0
60 NEXT
```

clears out the line by setting each pixel to 0.

2.71 PUT(Files)

Format : PUT [#]<file number>[,<record number>]

where:

<file number>	is the number under which the file was OPENed.
<record number>	is the record number for the record to be written.

Purpose: To write a record from a random buffer to a random disk file.

Remarks: IF <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

PUT can be used for a communications file. In that case <record number> is the number of bytes to write to the communications file.

Example:
```
10 OPEN "R",#1,"TESTPUT DAT"
20 PRINT #1,"ABC"
30 PUT #1
40 CLOSE #1
50 END
```

Note : PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.72 PUT(Graphics)

Format : PUT (x1, y1),<array>[,<action>]

where:

(x1, y1) is the coordinates of the top left corner of the image to be transferred.

<array> is the name of the array to transfer the screen image to.

<action> is one of these verb: PSET, PRESET, AND, OR, XOR

Purpose: To transfer the contents of an array to the screen.

Remarks: The PUT and GET statements are used to transfer graphics images to and from the screen. PUT and GET make possible animation and high-speed object motion in either graphics mode.

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. An "Illegal function call" error will result if the image to be transferred is too large to fit on the screen.

The action verb is used to interact the transferred image with the image already on the screen.

PSET transfers the data onto the screen verbatim.

PRESET is the same as PSET except that a negative image (black on white) is produced.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode often used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background twice, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action mode is XOR.

AND, OR and XOR have the following effects on color:

AND

array attrb	screen attribute 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1
2	0	0	2	2	0	0	2	2
3	0	1	2	3	0	1	2	3
4	0	0	0	0	4	4	4	4
5	0	1	0	1	4	5	4	5
6	0	0	2	2	4	4	6	6
7	0	1	2	3	4	5	6	7

OR

array attrb	screen attribute 0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	1	3	3	5	5	7	7
2	2	3	2	3	6	7	6	7
3	3	3	3	3	7	7	7	7
4	4	5	6	7	4	5	6	7
5	5	5	7	7	5	5	7	7
6	6	7	6	7	6	7	6	7
7	7	7	7	7	7	7	7	7

XOR

array attrb	screen attribute 0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Animation of an object is usually performed as outlined below:

1. PUT the object(s) on the screen.

2. Recalculate the new position of the object(s).

3. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).

4. Go to step one, this time PUTting the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at the time.

If it is not important to preserve the background, animaton can be performed using the PSET action verb. The idea is to leave a border around the image when it is first gotten as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points. This method may be somewhat faster than the method using XOR described above since only one PUT is required to move an object (although you must PUT a larger image).

The storage format in the array is as follows:

2 bytes giving X dimension in BITS
2 bytes giving Y dimension in BITS
The array data itself

The data for each row of pixels is left justified on a byte boundary, so if there are less than a multiple of 8 bits stored, the rest of the byte will be filled out with zeros. The required array size in bytes is:

```
4+INT((x+7)/8)*y*3
```

It is possible to modify the X and Y dimensions and even the data itself (it is best to use an integer array for this purpose). The X dimension is in element 0 of the array, and the Y dimension is found in element 1. Changing the X or Y dimensions in the array does not affect any storage allocations; it only changes the apparant size (as seen by the PUT statement). It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

The bytes per element of an array are: 2 for integer 4 for single precision 8 for double precision.

Example: The following program segment will produce a blue ball bouncing around inside a white box.

```
10  DIM B%(55) : CLS
20  CIRCLE(7,7),7
30  PAINT(7,7),1,7
40  GET (0,0)-(14,14),B%
50  CLS
60  LINE(0,0)-(639,399),7,B
70  VX=4 : VY=4 : X=12 : Y=12
80  PUT (X,Y),B%,XOR
90  IF X=620 OR X=ABS(VX) THEN VX=-VX
100 IF Y=380 OR Y=ABS(VY) THEN VY=-VY
110 PUT (X,Y),B%,XOR
120 X=X+VX : Y=Y+VY
130 GOTO 80
```

2.73 RANDOMIZE

Format : RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3 (user types 3)
 .88595  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user types 4
for new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
```

```
RUN
Random Number Seed (-32768 to 32767)? 3 (same sequence
as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

2.74 READ

Format : READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables.

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in list of variables exceeds the number of elements in the DATA statement(s), an "Out of data" error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

See "DATA statement" and "RESTORE statement" in this chapter.

Example:

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE         ZIP
DENVER,       COLORADO      80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.75 REM

Format : REM [<remark>]

'[<remark>]

where:

<remark> may be any sequence of characters.

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of REM or :REM.

Example:

```
  .
  .
  .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
  .
  .
  .
```

or

```
  .
  .
  .
120 FOR I=1 TO 20   'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
  .
  .
  .
```

Note : Do not use a single quotation mark instead of :REM in a DATA statement as it would be considered legal data.

2.76 RENUM

Format : RENUM [[<new number>][,[<old number>][,<increment>]]]

where:

<new number> is the first line number to be used in the new sequence. The default is 10.

<old number> is the line in the current program where renumbering is to begin. The default is the first line of the program.

<increment> is the increment to be used in the new sequence. The default is 10.

Purpose: To renumber program lines.

Remarks: RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB statements and ERL to reflect the new line numbers. If a non-existent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

Example:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

Note : RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

2.77 RESET

Format : RESET

Purpose: To close all disk files and flush the system buffer.

Remarks: If all open files are on disk, then RESET is the same as CLOSE with no file numbers after it.

Example:
```
10 OPEN "TEST1.DAT" FOR OUTPUT AS #1
20 PRINT #1,"ABC"
30 RESET
40 OPEN "TEST1.DAT" FOR INPUT AS #2
50 INPUT #2,A$
60 PRINT A$
70 RESET
80 END
```

2.78 RESTORE

Format : RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:
```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
 :
 :
```

2.79 RESUME

Format : RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number> .

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
 :
 :
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 90
 :
 :
```

2.80 RETURN

Format : RETURN [<line number>]

Purpose: To bring you back from a subroutine.

Remarks: Although you can use RETURN <line number> to return from any subroutine, this enhancement was added to allow non-local returns from the event trapping routines. From one of these routines you will often want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. Use of the non-local RETURN must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example:

```
10  FOR I=1 TO 5
20  GOSUB 100
30  NEXT I
40  END
100 PRINT "SUBROUTINE",I
110 RETURN
```

2.81 RUN

Format : RUN [<line number>]

RUN <filename>[,R]

where:

<filename> is the name used when the file was SAVEd.

Purpose: To begin execution of a program.

Remarks: The first form begins exectuion of the program currently in memory. If line number is specified, execution begins with the specified line number. Otherwise, execution begins at the lowest line number.

The second format loads a file from disk into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example:

```
10 PRINT 1/7
RUN
.1428571

10 PI=3.141593
20 PRINT PI
RUN 20
 0

RUN "B:NEWFIL",R
```

2.82 SAVE

Format : SAVE <filespec> [,A]

SAVE <filespec> [,P]

where:

<filespec> is a string expression for the file specification.

Purpose: To save a BASIC program file on disk.

Remarks: The SAVE statement allows a GW-BASIC program to be saved on disk.

The ,A option saves the program in ASCII format. If this option is not used, the file is saved in compressed binary format.

The ,P option saves the file in compressed binary format. When such a "protected" program is later run or loaded, any attempt to LIST or EDIT it will result in an "Illegal function call" error.

If the filename is less than 1 or more than 8 characters, a "Bad file name" error results and the file is not saved.

Files that are to be MERGEd must be saved with the ,A option. Attempts to merge binary programs will result in a "Bad file mode" error.

Example:

```
10 SAVE "INVENT" 'Saves program on the default
disk drive as file "INVENT.BAS"

20 SAVE "PROG",A 'Saves PROG in ASCII format

30 SAVE "SECRET",P 'Saves SECRET as protected file
which cannot be altered
```

2.83 SOUND

Format : SOUND <freq>,<duration>

where:

<freq> is the desired frequency in Hertz. A numeric expression in the range 37 to 3950.

<duration> is the desired duration in clock ticks. A numeric expression in the range of 0 to 255.

Purpose: The SOUND statement is used to control the sound generated by the speaker.

Remarks: The SOUND statement generates sound through the speaker. If the duration is zero, any current SOUND statement that is running will be turned off. If no SOUND statement is currently running, a SOUND statement with a duration of zero will have no effect.

Example: 30 SOUND RND*1000+37,2

This statement creates random sounds.

2.84 STOP

Format : STOP

Purpose: To terminate program execution and return to BASIC's command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is displayed:

Break in nnnnn

where nnnnn is the line number where the STOP occurred.

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution may be resumed by issuing a CONT command.

Example:

```
10 INPUT A,B,C
20 X=A*3 : Y=B*5
30 STOP
40 Z=C*2
50 PRINT Z
RUN
? 1,2,3
Break in 30
Ok
PRINT Y
 10
Ok
CONT
 6
Ok
```

2.85 SWAP

Format : SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two varaibles must be of the same type or a "Type mismatch" error occurs.

Example:

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

2.86 SYSTEM

Format : SYSTEM

Purpose: To exit GW-BASIC and return to MS-DOS.

Remarks: SYSTEM closes all files before it returns to MS-DOS.

2.87 TRON , TROFF

Format : TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40]  1  10  20
[50][60][30][40]  2  20  30
[50][60][70]
Ok
TROFF
Ok
```

2.88 WAIT

Format : WAIT <port number>, n[,m]

where:

<port number> is the port number, in the range 0 to 65535.

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is

exclusive OR'ed with the integer expression m, and then AND'ed with n. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If m is omitted, it is assumed to be zero.

Example: 100 WAIT 32,2

Note : It is possible to enter an infinite loop with the WAIT statement. You can do a CTRL Break or a System Reset to exit the loop.

2.89 WHILE...WEND

Format : WHILE <expression>

```
:
[<loop statements>]
:
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example.

```
80  'BUBBLE SORT ARRAY A$
90  FLIPS=1 'FORCE ONE PASS THRU LOOP
100 WHILE FLIPS
110       FLIPS=0
120       FOR I=1 TO J-1
130             IF A$(I)>A$(I+1) THEN
                        SWAP A$(I),A$(I+1):FLIPS=1
140       NEXT I
150 WEND
```

2.90 WIDTH

Format : WIDTH <size>

WIDTH <filenum>,<size>

WIDTH <dev>,<size>

where:

<size> is a numeric expression in the range 0 to 255 specifying the new width.

<filenum> is a numeric expression in the range 1 to 15. This is the number of the file OPENed to one of the devices listed below.

<dev> is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, COM1:, COM2:, COM3:, or COM4:

Purpose: The WIDTH statement sets the printed line width, in characters, for the screen, printer, or communications channel.

Remarks: Depending upon the syntax that is used, the following actions will be performed.

WIDTH <size> or WIDTH "SCRN:", <size>

Sets the screen width. Only 40 or 80 character column width is allowed.

WIDTH "LPT1:",<size>

Stores a width assignment for the printer, but without changing the current setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS #n will use this value for width while the file is open.

WIDTH <file number>,<size>

If the file is open to LPT1:, immediately changes the printer width to the size specified. This allows the width to be changed at will while the file is open. This form of WIDTH is used only with LPT1:, COM1:, COM2:, COM3:, or COM4:.

Valid widths for the screen are 40 and 80. Valid width for the printer is 1 to 255. Any value outside these ranges will result in an "Illegal function call" error, and the previous value will be retained.

Width has no effect for the keyboard.

Specifying WIDTH 255 for the printer (LPT1:) disables line folding. This has the effect of infinite width.

WARNING:

Changing the screen width causes the screen to be cleared.

Example:

```
10 WIDTH 40
15 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
 :
 :
60 WIDTH #1,40
```

In this example, line 10 sets the screen width to 40 characters per line. Line 15 stores a printer width of 75 characters. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 60 changes the current printer width to 40 characters.

2.91 WRITE

Format : WRITE [<list of expressions>]

Purpose: To output data on the screen.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the screen. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, GW-BASIC inserts a carriage return/line feed.

WRITE outputs values in a manner similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

Example:
```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

2.92 WRITE#

Format : WRITE#<file number>,<list of expressions>

where:

<file number> is the number under which the file was OPENed for output.

Purpose: To write data to a sequential file.

Remarks: The expressions in the list are string or numeric expressions, and they may be separated by commas or semicolons.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

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

would input "CAMERA" to A$ and "93604-1" to B$.

Chapter III

GW-BASIC FUNCTIONS AND VARIABLES

Canon AS-100

CHAPTER 3

GW-BASIC FUNCTIONS AND VARIABLES

The intrinsic functions and variables provided by GW-BASIC are presented in this chapter. The functions and variables may be called from any program without further definition.

Arguments to functions and variables are always enclosed in parentheses. In the formats given for the functions and variables in this chapter, the arguments have been abbreviated as follows:

m and n ··· Represent integer expressions

x and y ··· Represent any numeric expressions

x$ and y$ ··· Represent string expressions

If a floating point value is supplied where an integer is required, GW-BASIC will round the fractional portion and use the resulting integer.

Note: With the GW-BASIC interpreter, only integer and single precision results are returned by functions and variables. Double precision functions and variables are supported only by the GW-BASIC Compiler.

3.1 ABS

Format : ABS(x)

Purpose: Returns the absolute value of the expression x.

Remarks: The absolute value of a number is always positive or zero.

Example:
```
Ok
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Format : ASC(x$)

Purpose: Returns the ASCII code for the first character of the string x$.

Remarks: If x$ is null, an "Illegal function call" error is returned.

The CHR$ function is the inverse of the ASC function, and is used to convert from the ASCII code to a character.

See "ASCII character Codes Table" in Appendix.

Example:
```
Ok
10 X$="TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

3.3 ATN

Format : ATN(x)

Purpose: Returns the arctangent of x in radians,

Remarks: The result of the ATN function is a value in radians in the range -PI/2 to PI/2, where PI=3.141593. The expression x may be any numeric type, but the evaluation of ATN is always performed in single precision.

If you want to convert radians to degrees, multiply by 180/PI.

Example:
```
Ok
10 INPUT X
20 PRINT ATN(X)
RUN
?3
 1.249046
Ok
```

3.4 CDBL

Format : CDBL(x)

Purpose: To convert x to a double precision number.

Remarks: Refer to the CINT and CSNG functions for converting numbers to integer and single precision.

Example:
```
Ok
10 A=454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

3.5 CHR$

Format : CHR$(n)

where n is in the range 0 to 255.

Purpose: To convert an ASCII code to its character equivalent.

Remarks: CHR$ function is commonly used to send a special character to the screen or printer. For instance, the BEL character could be sent CHR$(7) as a preface to an error message, or a form feed could be sent CHR$(12) to clear a screen and return the cursor to the home position.

Look under "ASC function" in this chapter, to convert a character back to its ASCII code.

Example:
```
Ok
PRINT CHR$(66)
B
Ok

KEY 12, "AUTO"+CHR$(13)
Ok
```

Sets function key F12 to the string "AUTO" joined with the carriage return. This is a good way to set the function keys so the carriage return is automatically done for you when you press the function key.

3.6 CINT

Format : CINT(x)

Purpose: To convert x to an integer.

Remarks: x is converted to an integer by rounding the fractional portion. If x is not in the range -32768 to 32767, an "Overflow" error occurs.

See the FIX and INT functions, both of which also return integers. See also the CDBL and CSNG functions for converting numbers to single or double precision.

Example:
```
Ok
PRINT CINT(45.67)
 46
Ok
```

3.7 COS

Format : COS(x)

Purpose: Returns the cosine of x in radians.

Remarks: The calculation of COS(x) is performed in single precision.

To convert from degrees to radians, multiply by PI/180, where PI=3.141593.

Example:
```
Ok
10 X=2*COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

3.8 CSNG

Format: CSNG(x)

Purpose: Converts x to a single precision number.

Remarks: See the CINT and CDBL functions for converting numbers to integer and double precision.

Example:
```
Ok
10 A#=975.3421222#
20 PRINT A#;CSNG(A#)
RUN
 975.3421222  975.3421
Ok
```

3.9 CSRLIN

Format : x=CSRLIN

Purpose: Returns the vertical coordinates of the cursor.

Remarks: The CSRLIN variable returns the current line (row) position of the cursor on the current screen. The value returned will be in the range of 1 to 25.

x=POS(0) will return the column location of the cursor. Refer to the "POS" function in this chapter.

Refer to the "LOCATE" statement to see how to set the cursor line.

Example:

```
10 Y=CSRLIN'Record current line
20 X=POS(0)'Record current column
30 LOCATE 24, 1: PRINT "HELLO"
40 LOCATE X,Y 'Restore position to old line
and column
```

3.10 CVI , CVS , CVD

Format : CVI(<2-byte string>)

CVS(<4-byte string>)

CVD(<8-byte string>)

Purpose: To convert string values to numeric values.

Remarks: Numeric values that are read in from a random file must be converted from strings back into numbers.
CVI converts a 2-byte string to an integer.
CVS converts a 4-byte string to a single precision number.
CVD converts an 8-byte string to a double precision number.

See "MKI$, MKS$, MKD$ Functions" in this chapter.

Example:

```
.
.
.
60 OPEN "TEST.DAT" AS #1 LEN=16
70 FIELD #1, 4 AS N$, 12 AS B$
80 GET #1
90 PRINT CVS(N$),B$
.
.
.
```

3.11 DATE$

Format : DATE$=x$

y$=DATE$

where:

mm : = two digit value for the month (01 - 12)

dd : = two digit value for the day (01 - 31)

yy : = two digit value for the year (80 - 77)

yyyy: = four digit value for the year (1980 - 2077)

x$ is the date in one of the following forms:

mm - dd - yy

mm - dd - yyyy

mm / dd / yy

mm / dd / yyyy

y$ is a 10 character string of the form mm - dd - yyyy.

Purpose: Set or retrieve the current date.

Remarks: Retrieves the current date and assigns it to the string variable if DATE$ is the expression in a LET or PRINT statement. The date that is retrieved will be calculated from the last date assigned with the DATE$=string statement. If DATE$ has never been assigned a value, then the value supplied by MS-DOS will be used.

If x$ is not a valid date string, a "Type mismatch" error will result. In this case, previous values are retained.

If any of the values are out of range or are missing, an "Illegal function call" error results and the previous date is retained.

Example:
```
10 DATE$="03/01/1983"
20 PRINT DATE$
```

Sets the current date to March 1, 1983 and prints "03/01/1983".

3.12 EOF

Format : EOF (<file number>)

where:

<file number> is the file number specified on the OPEN statement.

Purpose: To test for end of file condition.

Remarks: The EOF function returns -1 (true) if end of file has been reached on the specified file. A zero(0) will be returned if end of file has not been reached. The file may be a sequential access file or it may be a communications file. A -1 for a communications file means the buffer is empty.

Example:
```
      .
      .
      .
 50 OPEN "DATA.DAT" FOR INPUT AS #1
 60 C=0
 70 IF EOF(1) THEN 200
 80 INPUT #1, M(C)
 90 C=C+1
100 GOTO 70
      .
      .
      .
```

3.13 ERR , ERL

Format : x = ERR

y = ERL

Purpose: To return the error code and line number associated with that error.

Remarks: When an error handling routine is entered, the

variable ERR contains the error code for the error and the variable ERL contains the line number of the line in which the error was detected.
The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine.

IF the statement that caused the error was a direct mode statement, ERL will contain 65535.
To test whether an error occurred in a direct statement, use

```
IF ERL=65535 THEN ...
```

Otherwise, use

```
IF ERR=error code THEN ...

IF ERL=line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

Example:

```
Ok
10 ON ERROR GOTO 50
20 ERR=20 'this will cause a syntax error (ERR=2)
30 ERR=30 'this will cause a syntax error (ERR=2)
50 IF ERR=2 AND ERL=20 THEN RESUME NEXT
60 ON ERROR GOTO 0 'BASIC will now take over.
RUN
Syntax error in 30
Ok
```

3.14 EXP

Format : EXP(x)

Purpose: Calculates the exponential function.

Remarks: Returns *e* to the power of x. x must be ≤ 88.02968. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
Ok
10 X=2
20 PRINT EXP(X-1)
RUN
 2.718282
Ok
```

3.15 FIX

Format : FIX(x)

Purpose: Truncates x to an integer.

Remarks: FIX(x) is equivalent to SGN(x)*INT(ABS(x)). The major difference between FIX and INT is that FIX does not return the next lower number for negative x.

Example:
```
Ok
PRINT FIX(58.75)
 58
Ok
PRINT FIX(-58.75)
-58
Ok
```

3.16 FRE

Format : FRE(x)

FRE(x$)

Purpose: Returns the number of bytes in memory not being used by GW-BASIC.

Remarks: Arguments to FRE are dummy arguments.

FRE("") forces a garbage collection before returning the number of free bytes. GW-BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:
```
Ok
PRINT FRE(0)
 14542
Ok
```

3.17 HEX$

Format : HEX$(n)

Purpose: Returns a string which represents the hexadecimal value of the decimal arguments.

Remarks: n is rounded to an integer before HEX$(n) is evaluated.

See the OCT$ function for octal conversion.

Example:
```
Ok
PRINT HEX$(32)
 20
Ok
```

3.18 INKEY$

Format : INKEY$

Purpose: To read a character from the keyboard.

Remarks: Returns either a one-character string containing a character read from the keyboard or a null string if no character is pending at the keyboard.
No characters will be echoed and all characters are passed through to the program except for Control - C, which terminates the program.

Example:
```
Ok
10 'stop program until a key is pressed
20 PRINT "Press any key to continue"
30 X$=INKEY$:IF X$=""THEN 30
     .
     .
     .
```

3.19 INP

Format : INP(n)

where n must be in the range 0 to 65535.

Purpose: Returns the byte read from port n.

Remarks: INP is the complementary function to the OUT statement.

Example:

```
.
.
.
100 A=INP(255)
.
.
.
```

3.20 INPUT$

Format : INPUT$(n[,[#]m])

where m is the file number used on the OPEN statement.

Purpose: Returns a string of n characters, read from the keyboard or from file number m.

Remarks: If the keyboard is used for input, no characters will be displayed on the screen and all control characters are passed through except Control - C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
10 'List the contents of a sequential file in
   hexadeimal
20 OPEN "I",1,"DATA.DAT"
30 IF EOF(1) THEN 60
40 PRINT HEX$(ASC(INPUT$(1,1)));
50 GOTO 30
60 PRINT
70 END
```

Example 2:

```
.
.
.
100 PRINT "Type P to proceed or S to stop"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

3.21 INSTR

Format : INSTR([n,]x$,y$)

where:

n is a numeric expression in the range 1 to 255.

x$, y$ may be string variables, string expression or string constants.

Purpose: Searches for the first occurrence of string y$ in x$ and returns the position at which the match is found.

Remarks: Optional offset n sets the position for starting the search. If n > LEN(x$) or if x$ is null or if y$ cannot be found, INSTR returns 0. If y$ is null, INSTR returns n or 1.

If n is out of range, an "Illegal function call" error will be returned.

Example:

```
Ok
10 X$="ABCDEB"
20 Y$="B"
30 PRINT INSTR(X$,Y$); INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

3.22 INT

Format : INT(x)

Purpose: Returns the largest integer which is less than or equal to x.

Remarks: See the FIX and CINT functions which also return integer values.

Example:

```
Ok
PRINT INT(99.89)
 99
Ok
PRINT INT(-12,11)
-13
Ok
```

3.23 LEFT$

Format : LEFT$(x$,n)

where n must be in the range 0 to 255.

Purpose: Returns a string comprised of the leftmost n characters of x$.

Remarks: If n is greater than LEN(x$), the entire string (x$) will be returned. If n = 0, the null string is returned.

See the MID$ and RIGHT$ functions.

Example:
```
Ok
10 A$="CANON AS-100"
20 B$=LEFT$(A$,5)
30 PRINT B$
RUN
CANON
Ok
```

3.24 LEN

Format : LEN(x$)

Purpose: Returns the number of characters in x$.

Remarks: Unprintable characters and blanks are counted.

Example:
```
Ok
10 X$="CANON AS-100"
20 PRINT LEN(X$)
RUN
 12
Ok
```

3.25 LOC

Format : LOC(<file number>)

Purpose: Returns the current position in the file.

Remarks: With random files, LOC returns the record number of the last record read from or written to a random file.

With sequential files, LOC return the number of records read from or written to the file since it was OPENed. When a file is OPENed for sequential input, GW-BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file occurs.

For a communications file, LOC(x) returns the number of characters in the input buffer waiting to be read. The input buffer can hold 128 characters. If there are more than 128 characters in the buffer, LOC(x) returns 128.

Example: 200 IF LOC(1)>50 THEN STOP

3.26 LOF

Format : LOF(<file number>)

Purpose: Returns the number of bytes allocated to the file or communications buffer.

Remarks: For disk files, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned.

For communications, LOF returns the amount of free space in the input buffer. That is, 128 - LOC(x). Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose.

Example:

```
.
.
.
70 OPEN "TEST.DAT" FOR INPUT AS #1
80 PRINT LOF(1)
90 CLOSE
.
.
.
```

3.27 LOG

Format : LOG(x)

Purpose: Returns the natural logarithm of x.

Remarks: x must be greater than zero. The natural logarithm is the logarithm to the base *e*.

Example:
```
Ok
PRINT LOG(45/7)
 1.860752
Ok
```

3.28 LPOS

Format : LPOS(x)

Purpose: Returns the current position of the print head within the printer buffer.

Remarks: Does not necessarily give the physical position of the print head. x is a dummy argument.

Example:
```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.29 MID$

Format : MID$(x$,n[,m])

where n and m must be in the range 1 to 255.

Purpose: Returns a string of length m characters from x$ beginning with the n-th character.

Remarks: If m is omitted or if there are fewer than m characters to the right of the n-th character, all rightmost characters beginning with the n-th character are returned. If n > LEN(x$), MID$ returns a null string.

If either n or m is out of range, an "Illegal function call" error will be returned.

See also the LEFT$ and RIGHT$ functions.

Example:
```
Ok
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
Ok
```

3.30 MKI$, MKS$, MKD$

Format : MKI$(<integer expression>)

MKS$(<single precision expression>)

MKD$(<double precision expression>)

Purpose: To convert numeric type values to string type value.

Remarks: Any numeric value that is placed in a random file buffer within an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

See also "CVI,CVS,CVD Functions" in this chapter.

Example:

```
      .
      .
      .
50 AMT=K+T
60 FIELD #1, 4 AS D$, 20 AS N$
70 LSET D$=MKS$(AMT)
80 LSET N$=A$
90 PUT #1
      .
      .
      .
```

3.31 OCT$

Format : OCT$(n)

Purpose: Returns a string which represents the octal value of the decimal argument.

Remarks: n is rounded to an integer before OCT$(n) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

3.32 PEEK

Format : PEEK(n)

where n is an integer in the range 0 to 65535.

Purpose: Returns the byte read from the indicated memory position.

Remarks: The returned value will be an integer in the range 0 to 255. n is the offset from the current segment as defined by the DEF SEG statement.

PEEK is the complementary function to the POKE statement.

See DEF SEG and POKE statements in Chapter 2.

Example: A=PEEK(&H5A00)

3.33 POINT

Format : POINT(x,y)

Purpose: Returns the color of the specified point on the screen.

Remarks: The POINT function allows the user to read the attribute value of a pixel from the screen. If the point given is out of range the value -1 is returned.

Example:

```
10 CLS
20 FOR C=0 TO 7
30   PSET (C*10,10),C
40   IF POINT(C*10,10)<>C THEN PRINT "Broken!"
50 NEXT C
```

3.34 POS

Format : POS(n)

Purpose: Returns the current cursor column position.

Remarks: The current column position of the cursor is returned. n is a dummy argument.

The returned value will be in the range 1 to 80 or

1 to 40, depending on the current WIDTH setting.

CSRLIN can be used to find the row position of the cursor.

Also see the LPOS function.

Example: IF POS(0)>60 THEN PRINT CHR$(13)

3.35 RIGHT$

Format : RIGHT$(x$,n)

Purpose: Returns the rightmost n characters of string x$.

Remarks: If n is greater than or equal to LEN(x$), then x$ is returned. If n is zero, the null string is returned.

Also see the MID$ and LEFT$ functions.

Example:
```
Ok
10 A$="CANON AS-100"
20 B$=RIGHT$(A$,6)
30 PRINT B$
RUN
AS-100
Ok
```

3.36 RND

Format : RND[(x)]

Purpose: Returns a random number between 0 and 1.

Remarks: The same sequence of random number is generated each time the program is RUN unless the random number generator is reseeded. This is most easily done using the RANDOMIZE statement. You may also reseed the generator when you call the RND function by using x where x is negative. This will always generate the particular sequence for the given x. This sequence is not affected by RANDOMIZE, so if you want it to generate a different sequence each time the program is run, you must use a different value for x each time.

If x is positive or omitted, RND(x) generates the next random number in the sequence.

RND(0) repeats the last number generated.

Example:
```
Ok
10 FOR I=1 TO 5
20  A=INT(RND*100)
30  PRINT A;
40 NEXT I
RUN
 12  65  86  72  79
Ok
```

3.37 SCREEN

Format : SCREEN(<row>,<column>[,Z])

where:

<row> is a valid numeric expression returning an unsigned integer in the range 1 to 25.

<column> is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80 depending upon the width.

Z is a valid numeric expression returning a boolean result.

Purpose: Returns the ASCII code (0 - 255) for the character displayed on the screen at the specified row and column position.

Remarks: The ASCII code for the character at the specified coordinates is stored in the numeric variable. If the optional parameter Z is given and non-zero, the color attribute for the character is returned instead.

Any values entered outside of these ranges will result in an "Illegal function call" error.

Example:
```
100 X=SCREEN(10,10)
```

If the character at (10,10) is "A", then X will be 65.

```
200 X=SCREEN(1,1,1)
```

Returns the color attribute of the character in the upper left hand corner on the screen.

3.38 SGN

Format : SGN(x)

Purpose: Returns the mathematical signum function.

Remarks: If x > 0, SGN(x) returns 1.

If x = 0, SGN(x) returns 0.

If x < 0, SGN(x) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300

Branches to 100 if x is negative, 200 if x is 0 and 300 if x is positive.

3.39 SIN

Format : SIN(x)

Purpose: Returns the sine of x in radians.

Remarks: SIN(x) is calculated in single precision.

If you want to convert degrees to radians, multiply by PI/180, where PI=3.141593.

Example:
```
Ok
PRINT SIN(1.5)
 .9974951
Ok
```

3.40 SPACE$

Format: SPACE$(n)

where n must be in the range 0 to 255.

Purpose: Returns a string of spaces of length n.

Remarks: Also see the SPC function in this chapter.

Example:
```
Ok
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
  2
   3
    4
     5
Ok
```

3.41 SPC

Format : SPC(n)

where n must be in the range 0 to 255.

Purpose: Prints n spaces on the screen or the printer.

Remarks: SPC function may only be used with PRINT,LPRINT and PRINT# statements. A ';' is assumed to follow the SPC(n) function.

Also see the SPACE$ function in this chapter.

Example:
```
Ok
PRINT "OVER"SPC(10) "THERE"
OVER          THERE
Ok
```

3.42 SQR

Format : SQR(x)

Purpose: Returns the square root of x.

Remarks: x must be greater than or equal to zero.

Example:
```
Ok
10 FOR I=10 TO 25 STEP 5
20  PRINT I, SQR(I)
30 NEXT I
```

```
RUN
 10          3.162278
 15          3.872984
 20          4.472136
 25          5
Ok
```

3.43 STR$

Format : STR$(x)

Purpose: Returns a string representation of the value of x.

Remarks: The STR$ function is the inverse of VAL.

Also see VAL function in this chapter.

Example:
```
10 'Arithmetic for kids
20 INPUT "Type a number";N
30 ON LEN(STR$(N)) GOSUB 30,100,200,300
     .
     .
     .
```

3.44 STRING$

Format : STRING$(n,m)

STRING$(n,x$)

where n and m must be in the range 0 to 255.

Purpose: Returns a string of length n whose characters all have ASCII code m or the first character of x$.

Example:
```
Ok
10 X$=STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.45 TAB

Format : TAB(n)

where n must be in the range 1 to 255.

Purpose: Tabs to position n.

Remarks: If the current print position is already beyond space n, TAB goes to position n on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one.

TAB may only be used in PRINT,LPRINT and PRINT# statement.

Example:
```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
RUN
NAME                      AMOUNT

G.T. JONES                $25.00
Ok
```

3.46 TAN

Format : TAN(x)

Purpose: Returns the tangent of x in radians.

Remarks: TAN(x) is calculated in single precision.

To convert degrees to radians, multiply by PI/180, where PI=3.141593.

Also see the SIN,COS and ATN function in this chapter.

Example:
```
Ok
10 INPUT X
20 Y1=SIN(X)
30 Y2=COS(X)
40 Y3=TAN(X)
50 Y4=ATN(X)
```

```
 60 PRINT "X=";X
 70 PRINT "SIN(X)"; Y1
 80 PRINT "COS(X)"; Y2
 90 PRINT "TAN(X)"; Y3
100 PRINT "ATN(X)"; Y4
RUN
? 1
X=1
SIN(X)= .841471
COS(X)= .5403023
TAN(X)=1.557408
ATN(X)= .7853983
Ok
```

3.47 TIME$

Format : TIME$=x$

y$=TIME$

where:

x$ is a string expression indicating the time to be set. Valid forms of this string are explained below.

Purpose: To set or retrieve the current time.

Remarks: Retrieves the current time and assigns it to the string variable if TIME$ is the expression in a LET or PRINT statement. The current time that is retrieved is calculated from the last time set with the TIME$=x$ statement.

If x$ is not a valid string, a "Type mismatch" error will result.

For y$=TIME$,TIME$ returns an 8-character string in the form hh:mm:ss, where hh is the hour (00 to 23), mm is minutes (00 to 59), and ss is seconds (00 to 59).

For TIME$=x$; x$ may be in one of the following forms:

hh (sets the hour; minutes and seconds default to 00)

hh:mm (sets the hour and minutes; seconds default to 00)

hh:mm:ss (sets the hour, minutes, and seconds)

If any of the values are out of range, an "Illegal function call" error results. In the case, the previous value is retained.

Example:
```
10 TIME$="15:30"
20 PRINT TIME$
```

3.48 USR

Format : USR[<digit>](x)

where:

<digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for the desired routine.

Purpose: To call the indicated machine language subroutine with the argument x.

Remarks: If <digit> is omitted, USR0 is assumed.

The CALL statement is another way to call a machine language subroutine.

See Chapter 1, "9. Machine Language Subroutines" for complete information on using machine language subroutines.

Example:
```
     .
     .
     .
40 B=T*SIN(Y)
50 C=USR(B/2)
60 D=USR(B/3)
     .
     .
     .
```

3.49 VAL

Format : VAL(x$)

Purpose: Returns the numerical value of string x$.

Remarks: The VAL function strips leading blanks, tabs, and linefeeds from the argument string. For example:

```
VAL("        -3")
```

returns -3.

If x$ is not numeric, then VAL(x$) will return zero(0).

See the STR$ function for numeric to string conversion.

Example:

```
10 INPUT "Input hexadecimal value:",H$
20 IF H$=" " THEN END
30 PRINT "&H";H$;"=";VAL("&H"+H$)
40 GOTO 10
RUN
Input hexadecimal value:4F
&H4F=79
Input hexadecimal value:32
&H32=50
Input hexadecimal value:FF
&HFF=255
Input hexadecimal value:
Ok
```

3.50 VARPTR

Format : VARPTR(<file number>)

VARPTR(<variable>)

Purpose: Returns the address in memory of the variable or file control block.

Remarks: The second format returns the address of the first byte of data identified with the variable. A value must be assigned to the variable prior to the call to VARPTR, or an "Illegal function call" error will result. Any type variable name may be used (numeric, string, array)

Note: All simple variables should be assigned before calling VARPTR for an array, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to a machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest addressed element of the array is returned.

For files, the VARPTR function returns the address of the first byte of the File Control Block (FCB) for the opened file. This is not the same as the MS-DOS file control block. The file must be OPENed before the call to VARPTR.

Offsets to information in the FCB from the address returned VARPTR is as follows:

Offset	Length	Contents
0	1	The mode in which the file was opened: 1 - Input only 2 - Output only 4 - Random I/O 16 - Append only 32 - Internal use 64 - Future use 128 - Internal use
1	38	Disk File Control Block. Refer to MS-DOS User's Manual for contents.
39	2	Number of sectors read or written for sequential access. For random access, it contains the last record number +1 read or written.
41	1	Number of bytes in sector when read or written.
42	1	Number of bytes left in input buffer.
43	3	Reserved for future expansion.
46	1	Device Number: 0-9 - Disk A: through J: 248 - LPT3: 249 - LPT2: 250 - COM2: 251 - COM1:

Offset	Length	Contents
		252 - 253 - LPT1: 254 - SCRN: 255 - KYBD:
47	1	Device width.
48	1	Position in buffer for print.
49	1	Internal use during LOAD/SAVE not used for data files.
50	1	Output position used during tab expansion.
51	128	Physical data buffer (BUFFER). Used to transfer data between MS-DOS and GW-BASIC. Use this offset to examine data in sequential I/O mode.
179	2	Variable length record size (VRECL). Default is 128. Set by length option in OPEN statement.
181	2	Current physical record number.
183	2	Current logical record number.
185	1	Future use.
186	2	Disk files only. Output position for PRINT#,INPUT# and WRITE#.
188	n	Actual FIELD data buffer. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine file data in random I/O mode.

Example:

```
10 OPEN "DATA.FIL" AS #1
20 FCBADR=VARPTR(#1)'set FCBADR to start of FCB.
30 DATADR=FCBADR+188'DATADR contains address
40                  'of data buffer.
50 A$=PEEK(DATADR)  'A$ contains 1st byte in
60                  'data buffer.
```

APPENDIX A

Summary of Error Messages and Codes

Error Message	Code	Contents
Bad file mode	54	You tried to use PUT or GET with a sequential file or a closed file, to MERGE a non-ASCII file, or to execute an OPEN with a file mode other than input, output, append, or random.
Bad file name	64	An invalid form is used for the filename with BLOAD,BSAVE,KILL, OPEN, NAME, or FILES(e.g., a filename starting with a period).
Bad file number	52	A statement references a file with a file number that is not OPEN or is out of the range of possible file numbers which was specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename was too long or invalid.
Bad record number	63	In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
Can't continue	17	You tried to use CONT to continue a program that: 1. has halted due to an error. 2. has been modified during a break in execution, or 3. does not exist.
Communication buffer overflow	69	A communication input statement was executed but the input buffer was already full. You should use an ON ERROR statement to retry the input when this condition occurs. Subsequent inputs will attempt to clear this fault unless characters continue

Error Message	Code	Contents
		to be received faster than the program can process them. If this is the case there are several things you might do: 1. Increase the size of the communications buffer. 2. Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. 3. Use a lower baud rate to transmit and receive.
Device Fault	25	Indicates a hardware error indication returned by an interface adapter.
Device I/O Error	57	An error occurred on a device I/O operation. MS-DOS cannot recover from the error.
Device Timeout	24	BASIC did not receive information from an input/output device within a predetermined amount of time.
Device Unavailable	68	You tried to OPEN a file to a device which doesn't exist. Either you do not have the hardware to support the device (such as printer adapters for a second or third printer), or you have disabled the device.
Direct statement in file	66	A direct statement was encountered while LOADing or CHAINing to an ASCII format file. The LOAD or CHAIN is terminated. The ASCII file should consist only of statements preceded by line numbers. This may occur because of a line feed character in the input stream.
Disk full	61	All disk storage space is in use. Files will be closed when this error occurs.

Error Message	Code	Contents
Disk Media Error	72	The controller attachment card has detected a hardware or media fault. Usually this means the disk has gone bad. Copy any existing files to a new disk and re-format the bad disk. If formatting fails, the disk should be discarded.
Disk not Ready	71	The disk drive door is open or a disk is not in the drive. Place the correct disk in the drive and continue the program.
Disk write protected	70	You tried to write to a disk that is write protected.
Division by zero	11	In an expression you tried to divide by zero, or you tried to raise zero to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the exponentiation, and execution continues.
Duplicate Definition	10	You tried to define the size of the same array twice. This may happen in one of several ways: 1. two DIM statements are given for the same array. 2. a DIM statement is given for an array after the default dimension or 10 has been established for that array. 3. an OPTION BASE statement has been encountered after an array has been dimensioned, either by a DIM statement or by default.
FIELD overflow	50	A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the

Error Message	Code	Contents
		FIELD buffer was encountered while doing sequential I/O (PRINT#,WRITE#,INPUT#,etc.) to a random file.
File already exists	58	The filename specified in a NAME statment is identical to a filename already in use on the disk.
File already open	55	You tried to OPEN a file for sequential output or append, and the file is already OPEN. Or, you tried to KILL a file that is open.
File not found	53	A LOAD,KILL,NAME,FILES, or OPEN references a file that does not exist on the disk in the specified drive.
For without NEXT	26	A FOR was encountered without a matching NEXT. That is, a FOR was active when an END,STOP, or RETURN was encountered.
Illegal direct	12	A statement that is invalid in direct mode is entered as a direct mode command. For example, DEF FN.
Illegal function call	5	A parameter that is out of range is passed to a system function. The error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative mantissa with a non-integer exponent 3. a call to a USR function for which the starting address has not yet been given 4. a negative record number on GET or PUT 5. an improper argument to a function or statement 6. an attempt to list or edit a protected BASIC program

Error Message	Code	Contents
Input past end	62	This is an end-of-file error. An input statement was executed for a null (empty) file, or after all the data in a sequential file was already input. To avoid this error, use the EOF function to detect the end of file. This error also occurs if you try to read from a file that was opened for output or append.
Internal error	51	An internal malfunction has occurred in BASIC. Report to your computer dealer the conditions under which the message appeared.
Line buffer overflow	23	You tried to enter a line that has too many characters.
Missing operand	22	An expression contains an operator, such as * or OR, with no operand following it.
NEXT without FOR	1	A variable in a NEXT statement does not correspond to any previously executed and unmatched FOR statement variable.
No RESUME	19	The program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (An END,STOP, or RETURN was found before a RESUME statement.)
Out of DATA	4	A READ statement is executed when there are no DATA statements with unread data remaining in the program.
Out of memory	7	A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex PAINTing. You may want to use

Error Message	Code	Contents
		CLEAR at the beginning of your program to set aside more stack space or memory area.
Out of Paper	27	The printer is out of paper, or the printer is not turned on. You should insert paper (if necessary), verify that the printer is properly connected and that the power is on. Then continue the program.
Out of string space	14	BASIC will allocate string space dynamically, until it runs out of memory. This message means that string variables have caused BASIC to exceed the amount of free memory remaining after doing housecleaning.
Overflow	6	The magnitude of a number is too large to be represented in BASIC's number format. Integer overflow will cause execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues. Note: If a number is too small to be represented in BASIC's number format, then we have an underflow condition. If this occurs, the result is zero and execution continues without an error.
Rename across disks	74	You tried to rename across disks by using NAME command.
RESUME without error	20	A RESUME statement is encountered before an error trapping routine is entered.
RETURN without GOSUB	3	A RETURN statement is encountered for which there is no previous unmatched GOSUB statement.

Error Message	Code	Contents
String formula too comples	16	A string expression is too long or too complex. The expression should be broken into smaller expressions.
String too long	15	You tried to create a string more than 255 characters long.
Subscript out of range	9	An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts. You may have put a subscript on a variable that is not an array. Or you may have incorrectly coded a built-in function.
Syntax error	2	A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
Too many files	67	An attempt is made to create a new file (using SAVE or OPEN) when all directory entries on the disk are full, or the file specification is invalid.
Type mismatch	13	You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP single and double precision values, etc.
Undefined line number	8	A line reference in a statement or command is to a nonexistent line.
Underfined user function	18	You called a function before defining it with the DEF FN statement.
Unprintable error	21 28	An error message is not available for the error condition

Error Message	Code	Contents
	31 - 49 56 59 - 60 65 75-255	which exists. This is usually caused by an ERROR statement with an undefined error code.
WEND without WHILE	30	A WEND was encountered before a matching WHILE was executed.
WHILE without WEND	29	A WHILE statement does not have a matching WEND. That is, a WHILE was still active when an END,STOP, or RETURN statement was found.

APPENDIX B

GW-BASIC Compiler

B.1 Outline

The GW-BASIC is provided with a compiler. Similar to an interpreter, the compiler is a program that translates a source program of BASIC into a machine language program. While the interpreter translates and executes line by line in executing a program, the compiler translate the whole of source program into relocatable machine language before executing the program and prepares an object program.

Thus, translation into machine language is performed at the time of compiling, and therefore, translation of the source program is not made at the time of program execution. Further, branch address of variable, GOTO statement and GOSUB statement is referred by absolute address at the time of program execution, it is not necessary to examine a variable list or line number during program execution. Accordingly, execution speed is faster than execution by an interpreter. Brief explanation on the GW-BASIC compiler will be given below.

Module structure of GW-BASIC compiler is as follows.

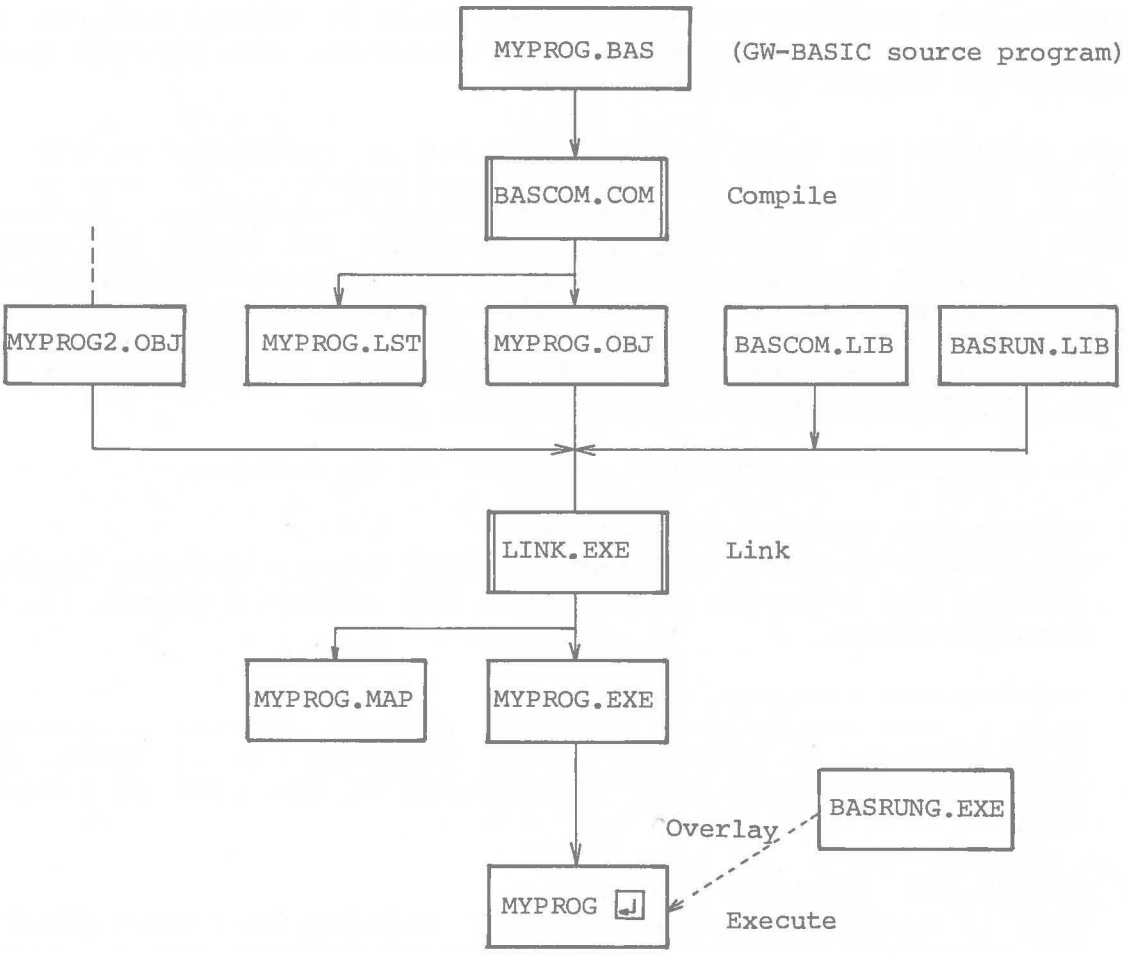
(1) BASCOM.COM (GW-BASIC compiler)
Translates a source program of BASIC into a machine language program and prepares relocatable and linkable object file (xxxxxxxx.OBJ).

(2) BASRUNG.EXE (runtime module)
This is the module that includes runtime routine necessary for execution of prepared object program. It is loaded on the memory together with the program at the time of execution.

(3) BASRUN.LIB (runtime library)
This is a library module used for calling of a user program and interface with runtime routine of BASRUNG.EXE. Connected with the user program at the time of linking.

(4) BASCOM.LIB (runtime library)
This is a group of object modules including routine having nearly the same function with BASRUNG.EXE. It is used to prepare a file (xxxxxxxx.EXE) executable without runtime module.

The flow of processes from compiling a source program of BASIC to execution is shown below. The source program to be compiled must be a file of ASCII format. Accordingly, a source program prepared by using an editor of BASIC interpreter must be saved in ASCII format.



B.2 How to operate the compiler

There are two ways of operation of the compiler. One is to specify parameter by interactive type according to prompt of the compiler without specifying parameter when inputting compiler calling command. The other is to enumerate parameter when inputting compiler calling command.

(1) Method of specifying parameter by interactive type

1) Enter [B][A][S][C][O][M] [↵] when OS mode ("A>_" is displayed) to call the compiler.

2) The following message is displayed.

```
Microsoft BASIC Compiler
Version x.xx
(C) Copyright Microsoft Corp 1982

Source filename [.BAS]: _
```

3) Input source file name of GW-BASIC to be compiled. When the extension is omitted, .BAS is automatically added. For instance, when a source file named MYPROG.BAS is to be compiled, input [M][Y][P][R][O][G] [↵].

```
Source filename [.BAS]: MYPROG ↵
Object filename [MYPROG.OBJ]: _
```

4) Input object file name to be prepared. When the extension is omitted, .OBJ is added automatically. However, if the file name displayed in square bracket will do, input [↵] key only.

```
Source filename [.BAS]: MYPROG ↵
Object filename [MYPROG.OBJ]: ↵
Source listing [NUL.LST]: _
```

5) Input source list file name.

* When it is not necessary to prepare a source list file ... [↵]
* When it is required to output a source list on the screen ... [C][O][N] [↵]
* When it is required to output a source list to a printer ... [L][P][T][1] [↵]

* When it is required to output a source list to a disk file ... <filename> ↵

In specification of file name when outputting a source list, if the extension is omitted, .LST is added automatically. ↵

Specification of compilation switch which will be mentioned later is applicable to any of file name specification of source file, object file and source list file.

(2) Method that enumerates the parameter

Specify parameter according to following form.

A > BASCOM <source file name>,<object file name>,<source list file name>↵

Method of specification of file name of each parameter and method of omitting are the same with the case where specification is made according to prompt of the compiler described in (1). The same applies to specification of compilation switch.

* When file name specification is deficient, prompt corresponding to it is returned.

  Example: A > BASCOM MYPROG ↵
  As only source file name is specified, prompt for object file and source list file is returned.

* When parameter specification is terminated with semicolon, file at the time of omitting is supposed for succeeding parameter. Prompt is not returned.

  Example: A > BASCOM MYPROG; ↵
  MYPROG.OBJ is supposed as an object file, and NUL.LST is supposed as a source list file.

* When parameter is omitted after a comma(,) which is an end of parameter specification, source file name and the extension at the time of omitting are supposed as the parameter.

  Example 1: A > BASCOM MYPROG., ↵
  MYPROG.OBJ is supposed as an object file and MYPROG.LST is supposed as a source list file. However, prompt of
  Source listing MYPROG.LST : _
  is returned.

Example 2: A BASCOM MYPROG,,; ↵
MYPROG.OBJ is supposed as object file and MYPROG.LST is supposed as a source list file. Prompt is not returned.

B.3 Compilation switch

Compilation switches can be used for extension and optimization of compiling function. Two or more switches can be used at a time. However, each switch must start with a slash(/).

Example 1: A > BASCOM MYPROG/A,,NUL ↵

Example 2: A > BASCOM MYPROG2,/O; ↵

Example 3: A > BASCOM MYPROG3
Object filename MYPROG3.OBJ : /X/D ↵
Source listing NUL.LST : LST: ↵

A table of compilation switches is shown in Table B.1.

Table B.1 Table of compilation switches

	Explanation
/E	Specified when ON ERROR GOTO and RESUME line number are included in the program.
/X	Specified when ON ERROR GOTO and RESUME, RESUME NEXT, RESUME 0 are included in the program.
/V	Test of event trapping is performed for each statement.
/W	Test of event trapping is performed for each line.
/4	Use Microsoft 4.51 scanning conversions (not allowed together with /N).
/T	Use Microsoft 4.51 execution conversions.
/A	Outputs to a source list file inclusive of a listing of object code.

	Explanation
/C:combuf	Sets the size of communication buffer.
/D	Generates debug code for error checking when program runs.
/N	Specified when making the line number to optional order or removing. /N is a convenient switch when taking text of other file into the source program using INCLUDE metacommand.
/R	Stores multi-dimensional array in the memory making a line in a unit.
/S	Outputs a character string enclosed by quotation marks on an object file (.OBJ) on a disk instead of data domain in the memory.
/O	Specified when BASRUNG.EXE runtime module is not used.

Notice: /O switch
When /O switch is not specified, the program automatically overlays BASRUNG.EXE runtime module and uses at the time of execution. Even when the size of the program itself is small, memory of about 30K byte becomes necessary for BASRUNG.EXE. When /O switch is specified, load of the program and speed of execution becomes greater as BASRUNG.EXE is not overlaid, and the rate of occupancy becomes small. However, it is not possible to own jointly the data between programs connected by COMMON statement. As the possibility of incorporation of similar execution module in all execution files is high, storing efficiency of the disk become poor.

B.4 How to operate the linker

The linker converts the object file (xxxxxxxx. OBJ) prepared by the compiler into excutable program linking with execution library, and prepares an execution file. The linker can be operated by two different ways. The first method does not specify the parameter when inputting linker calling command, and specifys the parameter by interactive type according to prompt of the linker. The second method enumerates and specifies the parameter when inputting the linker calling command. For details of the linker (LINK) refer to "MS-DOS user's manual."

(1) Method of specifying the parameter by interactive type

1) Input [L][I][N][K] [↵] when OS mode to call the linker.

2) Following message is displayed.

```
Microsoft Object Linker Vx.xx
(C) Copyright 1981 by Microsoft Inc.

Object Modules [.OBJ]: _
```

3) Input the name of object file (xxxxxxxx.OBJ) to be linked. When plural object files are to be linked, mark off each file with space or a plus mark(+). When a plus mark is inputted immediately before pressing the carriage return key, the same prompt is returned again. For instance, to link two object files, i.e. MYPROG.OBJ and MYPROG2.OBJ, input [M][Y][P][R][O][G][+][M][Y][P][R][O][G][2] [↵].

```
Object Modules [.OBJ]: MYPROG+MYPROG2↵
Run File [MYPROG.EXE]: _
```

4) Input the name of execution file. If only the carriage return key is depressed, the head object file name .EXE inputted in 3) is supposed.

```
Object Modules [.OBJ]: MYPROG+MYPROG2↵
Run File [MYPROG.EXE]: ↵
List File [NUL.MAP]: _
```

5) Input the name of list file. If only the carriage return key is depressed, the list file is not prepared.

```
Object Modules [.OBJ]: MYPROG+MYPROG2↵
Run File [MYPROG.EXE]: ↵
List File [NUL.MAP]: ↵
Libraries [.LIB]: _
```

6) Input the name of library file (BASRUN.LIB or BASCOM.LIB). When the extension is omitted, .LIB is added automatically.

(2) Method that enumerates the parameter

Specify the parameter according to the following format.

A > LINK <object file list>,<execution file>,<list file>
, < library file list> ↵

Method of file specification and omitting of each parameter are the same with the case when specification is made according to prompt of the linker mentioned in (1).

* When file specification is deficient, corresponding prompt is returned.

* When parameter specification is terminated with a semicolon(;), file at the time of omission is supposed for succeeding parameters. Prompt is not returned.

B.5 Execution of program

Inputting of the program name only is required for execution of compiled and linked program irrespective of presence of /O switch at the time of compiling. In this case, omission of the extension (.EXE) is possible.

Example: A > M Y P R O G ↵

When /O switch is not specified at the time of compiling, load BASRUNG.EXE runtime module when executing the program. It is also possible to execute another program out of programs.

Example 1: 10 RUN "MYPROG2"
MYPROG2.EXE is loaded and executed.

Example 2: 10 CHAIN "MYPROG2"
MYPROG2.EXE is loaded and executed.

In the case of Example 1, BASRUNG.EXE is reloaded. When the program is executed using CHAIN statement as in Example 2, BASRUNG.EXE is not loaded again.

B.6 Instruction for exclusive use of compiler(metacommand)

Metacommand is effective only for the compiler. To distinguish it from BASIC command, "$" mark is added to the head of the command. Metacommand is used to control operation of the compiler, and normally used in REM statement. By this way no problem occurs when operating by the interpreter.

Example: REM $LINESIZE:132

Table of metacommand is shown in Table B.2.

Table B.2 Table of metacommand

Metacommand	Explanation
$INCLUDE : 'filename'	Inputs source file to the position of $INCLUDE command from the file specified by 'filename'. 'filename' must be ASCII file.
$LINESIZE: n	The length of one line of the source list is specified. n is integer in the range of 41 through 255. n = 80 is supposed when $LINESIZE command is omitted.
$LIST+ $LIST-	Presence of output of the source list is specified. Succeeding source list is outputted and when $LIST-, output of succeeding source list is stopped. When NUL.LST is specified as the source list file at the time of compiling, the source list is not outputted even if LIST+ is specified.
$OCODE+ $OCODE-	In the case of $OCODE+, the code address and operation mnemonic for each line of succeeding source list are outputted, and when $OCODE-, succeeding output is stopped. If NUL.LST is specified as the source list file at the time of compiling, the code address and operation mnemonic are not outputted even when $OCODE+ is specified. When at least one of the switches is specified, $OCODE- is not effective.
$PAGE	Page of source list output is renewed just after $PAGE command.
$PAGEIF: n	Page is renewed when remaining number of printable lines is less than n (1-255).
$PAGESIZE: n	Number of lines per page is specified. n is integer in the range of 41 through 255, and if $PAGESIZE command is omitted, n = 66 is supposed.
$SKIP: n	n (1-255) line is renewed when $SKIP is detected.
$SUBTITLE: 'string'	A subtitle specified by 'string' is printed under the title of each page.
$TITLE: 'string'	A title specified by 'string' is printed at the top of each page. 'string' is a row of characters less than 60 characters.

B.7 Difference between GW-BASIC Compiler and Interpreter

(1) Operation Difference

The compiler interacts with the console only to read compiler command. These specify what files are to be compiled. There is no "direct mode," as with the GW-BASIC interpreter. Commands that are usually issued in the direct mode with the GW-BASIC interpreter are not implemented on the compiler.

The following statements and commands are not implemented and will generate an error message.

AUTO	BLOAD	BSAVE	CONT	DELETE
EDIT	LIST	LLIST	LOAD	MERGE
NEW	RENUM	SAVE		

Because there is no direct mode for typing in programs or edit mode for editing programs, use the GW-BASIC interpreter for creating and editing programs. If you use the interpreter, be sure to save the file with the A (ASCII format) option.

The compiler cannot accept a physical line that is more than 253 characters in length. A logical statement, however, may contain as may physical lines as desired. Use line feed to start a new physical line within a logical statement.

To reduce the size of the compiled program, there are no program line numbers included in the object code generated by the compiler unless the /D, /X, or /E switch is set in the compiler command. Error messages, therefore, contain the address where the error occurred, instead of a line number. The complier listing and the map generated by LINK are used to identify the line that has an error. It is GW-BASIC interpreter before attempting to compile them.

(2) Language difference

Most programs that run on the GW-BASIC interpreter will run on the GW-BASIC compiler with little or no change. However, it is necessary to note differences in the use of the following program statements and function:

1. CALL
   The variable name field in the CALL statement must contain an external symbol, i.e., one that is recognized by LINK as a global symbol. This routine must be supplied by the user as a machine language subroutine.

2. CHAIN and RUN
   The CHAIN statement is used to chain a new program overlay using the runtime module. The RUN statement is to be used to execute any executable file.

3. CLEAR
   The CLEAR statement is only supposed in compiled programs using the runtime module.

4. COMMON
   The COMMON statement must appear before any executable statement.

5. DEFINT/SNG/DBL/STR
   The compiler does not "execute" DEFxxx statements; it reacts to the static occurrence of these statements, regardless of the order in which program lines are executed. A DEFxxx statement takes effect as soon as its line is encountered. Once the type has been defined for a given variable, it remains in effect until the end of the program or until a different DEFxxx statement with that variable takes effect.

6. DIM
   The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. This is, DIM takes effect when its line is encountered. If the default dimension (10) has already been established for an array variable and that variable is later encountered in a DIM statement, a "Redimensioned array" error results.

   Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables,a arithmetic expressions, or floating point values. For example,

```
DIM A1(I)
DIM A1(3+4)
```

   are both illegal.

7. END
   During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" products proper program termination.

8. FOR/NEXT and WHILE/WEND
   FOR/NEXT and WHILE/WEND loops must be statically nested.

9. ON ERROR GOTO/RESUME line number
   If a program contains ON ERROR GOTO and RESUME line number statements, the /E compilation switch must be used. If the RESUME NEXT, RESUME, or RESUME 0 form is used, the /X switch must also be included.

10. REM
    REM statements or remarks starting with a single quotation mark do not take up time or space during execution, and so may be used as freely as desired.

11. STOP
    The STOP statement is identical to the END statement. Open files are closed and control returns to the operating system.

12. TRON/TROFF
    In order to use TRON/TROFF, the /D compilation switch must be used. Otherwise, TRON and TROFF are ignored and a warning message is generated.

13. USR Function
    USR function are significantly difference from the interpreter versions. The argument to the USR function is ignored and an integer result is returned in the HL registers. It is recommended that USR function be replaced by the CALL statement.

(3) Expression evaluation

During expression evaluation, the operands of each operator are converted to the same type, that of the most precise operand. For example,

```
QR=J%+A!+Q#
```

causes J% to be converted to single precision and added to A!. This results is converted to double precision and added to Q#.

The compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter the following program

```
I%=20000
J%=20000
K%=-30000
M%=I%+J%+K%
```

yield 10000 for M%. That is, it adds I% to J% and, because the number is too large, it converts the result into a floating point number. K% is then converted to floating point and subtracted. The result of 10000 is found, and is converted back to integer and saved as M%.

The compiler, however, must make type conversion decisions during compilation. It cannot defer until the actual values are known. Thus, the compiler would generate code to perform the entire operation in integer mode. If the /D switch were set,

the error would be detected. Otherwise, an incorrect answer would be produced.

In order to produce otpimum efficiency in the compiled program, the compiler may perform any number of valid algebraic transformations before generating the code. For example, the program

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

could produce an incorrect result when run. If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds j%, an overflow will occur. The compiler follows the rules for operator precedence and parenthetic modification of such precedence, but no other guarantee of evaluation order can be made.

(4) Integer variables

In order to produce the fastest and most compact object code possible, make maximum use of integer variables. For example, this program

```
FOR I=1 to 10
   A(I)=0
NEXT I
```

can execute approximately 30 times faster by simply substituting "I%" for "I". It is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

APPENDIX C

ASCII Character Code Table

The following table lists all the ASCII codes (in hexadecimal) and their associated characters.

n \ m	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	(null)	◀	(space)	0	@	P	`	p		┌		É			∛	┌
1	⊥⊥	±	!	1	A	Q	a	q	☺	└	§	Å			π	└
2	⊥	≥	"	2	B	R	b	r	☻	┘	Ä	å			×	┘
3	↕	≤	#	3	C	S	c	s	♥	¬	Ö	₧			÷	┐
4	↨	≈	$	4	D	T	d	t	♦	│	Ü	Φ			Σ	│
5	■	▽	%	5	E	U	e	u	♣	_	ä	ƒ			↑	─
6	□	½	&	6	F	V	f	v	♠	┴	ö	ll			↓	┼
7	(beep)	2	'	7	G	W	g	w	●	├	ü	ij			→	├
8	(back-space)	3	(	8	H	X	h	x	◘	┤	ß	ò			←	┤
9	(tab)	4	)	9	I	Y	i	y	○	┬	à	ì			µ	┬
A	(line feed)	1	*	:	J	Z	j	z	◙	┴	°				σ	┴
B	(home)	2	+	;	K	[	k	{	♂	↖	Ç				φ	├
C	(cls)	(cursor right)	,	<	L	\	l	|	♀	↵	é				α	─
D	(carrige return)	(cursor left)	-	=	M	]	m	}	♪	½	ù				β	┤
E	◆	(cursor up)	.	>	N	^	n	~	♫	¼	è				γ	═
F	∟	(cursor down)	/	?	O	_	o	▉	▌	▌	ē				£	▉

These characters can be displayed using PRINT CHR$(n), where n is the ASCII code.

APPENDIX D

Hard Copy

Outputting the contents displayed on the screen to a printer is called "to take a hard copy". Only the CANON dot impact printer A-1200 and the CANON color printer A-1210 can take hard copies.

D.1 Loading of handler

To take a hard copy, it is necessary to load various handlers on the memory before starting GW-BASIC. Handlers to be loaded are as follows:

a. When A-1200 is used:

A > [G][R][H][N][D] [↵]

A > [A][1][2][0][0][/][H][C][O][P][Y] [↵]

b. When A-1210 is used:

A > [G][R][H][N][D] [↵]

A > [A][1][2][1][0][/][H][C][O][P][Y] [↵]

For details of handlers refer to "MS-DOS User's manual".

D.2 How to take a hard copy

Input the following keys when taking of a hard copy is required.

[CTRL] + [⇧] + [CLEAR SCREEN]

(Depress the clear screen key while depressing the control key and shift key.)

When this operation is made during execution of the program, execution of the program is stopped, and the contents displayed on the screen is outputted to the printer. When hard copy output is completed, execution of the program is resumed. When outputting the contents displayed on the Color Display to the printer A-1210, a color which is the nearest to the color displayed on the screen is selected and outputted. Accordingly, it sometimes happens that the contents displayed by different colors on the screen is outputted to the printer in the same color. No problem occurs when only basic colors of the printer (black, blue, green, cyan, red, magenta, yellow, white) are

used. However, the contents displayed in white on the screen is outputted to the printer in black, and the contents displayed in black on the screen is outputted to the printer in white.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X