MINDSET

GW™-BASIC

by Microsoft

Reference Manual

Personal
Computer
System

100213-001

Printed in U.S.A.
100213-001
A Tec-Ed manual.

Contents

Section 3
Programming Animation

Section 4
Starting GW-BASIC

Section 5
Editing Basic Programs

Section 6
GW-BASIC Commands, Statements, Functions, and Variables

Figures and Tables

Section 1

Introduction

Welcome to GW-BASIC. This manual describes GW-BASIC as implemented for the Mindset Personal Computer. This version of GW-BASIC includes powerful statements to support advanced graphics operations, including animation.

This reference manual is not intended to be used as instructional text. Readers unfamiliar with BASIC programming are encouraged to obtain one of the many introductory textbooks which describes programming techniques for BASIC.

You should read the Mindset Personal Computer Operation Guide before using GW-BASIC. The Operation Guide explains how to power up the system and how to insert cartridges, such as the one containing the GW-BASIC interpreter.

This section of the GW-BASIC Reference Manual describes the special features that are part of GW-BASIC. These features include graphics, sound and music, peripherals support, device-independent I/O, event trapping, and the use of the Alt key to enter commands.

Graphics

GW-BASIC enables users to create programs for graphics operation which can use color, draw various figures, and perform animation. The statements and functions that are used for graphics include:

ACTIVATE
ARRIVAL
CIRCLE
CLIP
COLLISION
COLOR
DEACTIVATE
DEF OBJECT

DIM OBJECT
DRAW
GET and PUT
LINE
OBJECT
ON ARRIVAL
ON CLIP
ON COLLISION
PAINT
PALETTE
PALETTE USING
POINT
PRESET
PSET
SCREEN
START OBJECT
STOP OBJECT

These statements and functions are described in Section 6, "GW-BASIC Commands, Statements, Functions, and Variables".

Sound and Music

The BEEP, PLAY, and SOUND statements enable use of the Mindset Personal Computer sound capability. These statements are described in Section 6, "GW-BASIC Commands, Statements, Functions, and Variables".

The BEEP statement operates through the internal beeper of the Mindset Personal Computer while the PLAY and SOUND statements use its external sound channel.

Input Peripherals Support

In addition to the keyboard, GW-BASIC supports a mouse and one or two joysticks as input peripherals. The statements and functions that support these input peripherals are described individually in Section 6. They include STICK and STRIG.

Device-Independent Input/Output

GW-BASIC provides device-independent input/output that works with the MS-DOS operating system.

The following statements, commands, and functions support device-independent I/O (see individual descriptions in Section 6, "GW-BASIC Commands, Statements, Functions, and Variables"):

BLOAD
BSAVE
CHAIN
CLOSE
ENVIRON
ENVIRON$
EOF
ERDEV
ERDEV$
FILES
GET
INPUT
INPUT$
IOCTL
IOCTL$
KILL
LINE
LIST
LLIST
LOAD
LOC
LOF
LPOS
LPRINT
MERGE
NAME
OPEN
OPEN COM
POS
PRINT
PRINT USING
PUT
RESET
RUN
SAVE
SHELL
WIDTH
WRITE

Event Trapping

Event trapping allows a program to transfer control to a specific program line when a certain event occurs. Control is transferred as if a GOSUB statement had been executed to the trap routine, starting at the specified line number. The trap routine, after servicing the event, executes a RETURN statement that causes the program to resume execution at the place where it was when the event trap occurred.

The events that can be trapped are receipt of characters from a communications port (ON COM), selected key activation (ON KEY), joystick trigger or mouse switch activation (ON STRIG), timer operation (ON TIMER), background music operation (ON PLAY), and animation operation (ON ARRIVAL, ON CLIP, and ON COLLISION).

This section gives an overview of event trapping. For more details on individual statements, see Sections 3 and 6.

Event trapping is controlled by the following types of statements:

<event specifier> ON to turn on trapping
<event specifier> OFF to turn off trapping
<event specifier> STOP to temporarily turn off trapping

where <event specifier> is one of the following:

COM (n) where n is the number of the communications channel. The COM channels are numbered 1 through 4.

Typically, the COM trap routine will read an entire message from the COM port before returning. The COM trap should not be used for single-character messages, because at high baud rates the overhead of trapping and reading for each character may allow the interrupt buffer for COM to overflow.

KEY (n) where n is a trappable key number. GW-BASIC supports 20 trappable keys (numbered 1 through 20), where numbers 1 through 10 are the softkeys F1 through F10; 11 through 14 are the cursor direction keys (as follows: 11 up, 12 left, 13 right, 14 down); and 15 through 20 are user-defined trappable keys (see Section 6, "GW-BASIC Commands, Statements, Functions, and Variables", for details). KEY (0) refers to the entire set of trappable keys.

Note that KEY (n) ON is not the same statement as KEY ON. KEY(n) ON sets an event trap for the specified key. KEY ON displays the values of all the function keys on the twenty-fifth line of the screen (see Section 6, "GW-BASIC Commands, Statements, Functions, and Variables").

When GW-BASIC is in direct mode (in other words, during input), function keys maintain their standard meanings.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY$ statements to find out which key caused the trap. If you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

PLAY (n)	where n is the number of music notes in the Background Music Queue. The event trap occurs whenever the PLAY statement causes the number of notes in the queue for voice 1 to go below n.
STRIG (n)	where n relates to the number of the joystick trigger. The range for n is 0 to 6. STRIG is also used to record the input from the button(s) on a mouse. For discussion of the STRIG function, see Section 6.
TIMER (n)	where n is the number of seconds between event traps. For a discussion of the TIMER function, see Section 6.
ARRIVAL (n)	where n is the number of the object being tracked to its destination. Section 6 describes both this statement and the ARRIVAL function.
CLIP (n)	where n is the number of the object being checked to see if it has reached the viewport boundary. Section 6 describes the CLIP statement and the corresponding CLIP function.
COLLISION (n[,m])	where n, m are the number of objects being checked for a possible collision. See Section 6 for a description of this statement and the COLLISION function.

ON...GOSUB Statement Type

The ON...GOSUB type of statement sets up a line number for the specified event trap. The format is:

ON <event specifier> GOSUB <line number>

A <line number> of 0 disables trapping for that event.

When trapping for a type of event is ON and 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 (for example, the joystick button has been pressed or a COM character has been received). When trapping for a type of event is OFF, no trapping takes place, and the event is not remembered even if it occurs.

When a type of event is stopped (<event specifier> STOP), no trapping takes place. However, the occurrence of an event is remembered so that an immediate trap will take place when an <event specifier> ON statement is executed.

When a trap is made for a particular event, the trap automatically causes a STOP on that event, 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.

Note that after an error trap takes place, all trapping is automatically disabled. You must explicitly re-enable trapping within the program before another trap can occur. In addition, event trapping will never occur when GW-BASIC is not executing a program.

RETURN Statement

When an event trap is in effect, a GOSUB statement will be executed as soon as the specified event occurs. For example, the statement:

```
ON KEY(5) GOSUB 1000
```

specifies that the program is to go to line 1000 as soon as function key F5 is pressed. A RETURN statement executed at the end of this subroutine will return program control to the statement following the one at which the trap occurred. When the RETURN statement is executed, its corresponding GOSUB return address is cancelled.

GW-BASIC also features the RETURN <line number> enhancement, which lets control return to a place in the program that is not necessarily the next statement after the occurrence of the event trap.

If not used with care, however, this capability may cause problems. Assume, for example, that your program contains:

```
10 ON KEY(5) GOSUB 1000
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE

200 REM PROGRAM RESUMES HERE

1000 'FIRST LINE OF SUBROUTINE
  .
  .
  .
1050 RETURN 200
```

If key F5 is pressed while the FOR/NEXT loop is executing, the subroutine will be performed, but program control will return to line 200 instead of completing the FOR/NEXT loop. The original GOSUB entry

will be cancelled by the RETURN statement, and any other GOSUB, WHILE, or FOR (for example, an ON STRIG statement) active at the time of the trap will remain active. However, the current FOR context will also remain active, and a "FOR without NEXT" error may result.

Keyword Entry Using the Alt Key

GW-BASIC enables the entry of many BASIC keywords by using only two keystrokes. A GW-BASIC keyword is entered by holding the Alt key down while pressing one of the alphabetic keys (A-Z). The following list describes the standard definitions for each key when used with the Alt key:

A - AUTO
B - BSAVE
C - COLOR
D - DELETE
E - ELSE
F - FOR
G - GOTO
H - HEX$
I - INPUT
J - (no key word)
K - KEY
L - LOCATE
M - (no key word)
N - NEXT
O - OPEN
P - PRINT
Q - (no key word)
R - RUN
S - SCREEN
T - THEN
U - USING
V - VAL
W - WIDTH
X - XOR
Y - (no key word)
Z - (no key word)

Section 2

General Information About GW-BASIC

This section describes the important elements of BASIC programs including syntax, characters, constants, variables, operators, and expressions. It also explains the two modes of GW-BASIC, the default disk drive, and the display pages for GW-BASIC output.

Syntax Notation

When commands are discussed in this document, the following notation will be followed:

[ ] Square brackets indicate that the enclosed entry is optional.

< > Angle brackets indicate user-entered data. For example, enter the name of your file when <filename> is shown in the format.

BOLD Bold lettering indicates text you must enter.

{ } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

| Within braces vertical bars separate choices. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.

... Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

Line Format

GW-BASIC program lines have the following format:

<nnnnn BASIC statement>[:<BASIC statement>...] <RETURN>

More than one GW-BASIC statement may be placed on a line, but each must be separated from the previous statement by a colon.

A GW-BASIC program line always begins with a line number and ends with a carriage return. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing. Line numbers must be in the range 0 to 65529. A line may contain a maximum of 255 characters including the line number.

You can extend a logical line over more than one physical line by pressing the CTRL and RETURN keys simultaneously. CTRL-RETURN lets you continue typing a logical line on the next physical line without entering a carriage return.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

Error Messages

If an error causes program execution to terminate, an error message is printed. For a complete list of GW-BASIC error codes and error messages, see Appendix A.

Modes of Operation

GW-BASIC may be used in either of two modes: direct mode or indirect mode.

In direct mode, statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for

later use, but the instructions themselves are lost after execution. Direct mode is useful for debugging and for using GW-BASIC as a calculator for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and may be stored in memory. The program stored in memory is executed by entering the RUN command.

Default Disk Drive

When a filespec is given (in commands or statements such as FILES, OPEN, KILL), the default disk drive is the one that was the default in MS-DOS before GW-BASIC was invoked.

Active and Visual (Display) Pages

Every command that reads from or writes to the screen is actually reading from or writing to the active page. The visual, or display, page is the active page that is shown on the screen.

The size of these pages is set by the SCREEN statement (see Section 6).

Character Set

The GW-BASIC character set consists of alphabetic characters, numeric characters, and special characters.

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

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

The following special characters and keys are recognized by GW-BASIC.

Character	**Action**
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign

Character	Action
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
"	Double quotation marks
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At sign
_	Underscore

Key	Action
BACK SPACE	Deletes last character typed (<backspace> appears as left arrow on the keyboard).
ESC	Erases the line containing the cursor from the screen and performs a carriage return. It does not delete existing program lines from memory.
TAB	Moves print position to next tab stop. Tab stops are set every eight columns.
CTRL-RETURN	Moves to next physical line.
RETURN	Terminates input of a line.

Constants

Constants are the values GW-BASIC uses during execution. There are two types of constants: string and numeric.

String Constants

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Examples:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

String constants which appear in a DATA statement do not have to be enclosed in double quotation marks unless they contain commas, colons, or significant leading or trailing spaces.

Example:

```
DATA This is string1, This is string2
```

This example shows two strings separated by a comma within a DATA statement.

Numeric Constants

Numeric constants are positive or negative numbers. They are stored with 7 digits of precision and printed with up to 6 digits of precision. GW-BASIC numeric constants cannot contain commas. There are six types of numeric constants:

1. Integer constants	Whole numbers between −32768 and 32767. Integer constants do not contain decimal points.
2. Fixed-point constants	Positive or negative real number constants are numbers that contain decimal points.

3. Floating-point constants

Positive or negative numbers represented by constants in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10^{-38} to 10^{+38}.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

(Double precision floating-point constants are denoted by the letter D instead of E.)

4. Hex constants

Hexadecimal numbers, denoted by the prefix &H.

Examples:

```
&H76
&H32F
```

5. Octal constants

Octal numbers, denoted by the prefix &O or &.

Examples:

```
&O347
&1234
```

Single/Double Precision Form for Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant that has one of the following characteristics:

- Seven or fewer digits.
- Exponential form using E.
- A trailing exclamation point (!).

Examples:

46.8
−1.09E−06
3489.0
22.5!

A double precision constant is any numeric constant that has one of these characteristics:

- Eight or more digits.
- Exponential form using D.
- A trailing number sign (#).

Examples:

345692811
−1.09432D−06
3489.0#
7654321.1234

Variables

Variables are names used to represent values used in a GW-BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero (or null for a string variable).

Variable Names and Declaration Characters

GW-BASIC variable names may be any length. Up to 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special type declaration characters, described in this section, are also allowed.

A variable name may not be a reserved word; however, embedded reserved words are allowed. Reserved words include all GW-BASIC commands, statements, function names, and operator names. The complete list of reserved words is in Appendix D. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables may represent either a numeric value or a string. String variable names can be written with a dollar sign ($) as the last character; for example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single precision, or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of GW-BASIC variable names:

PI #	Declares a double precision value.
MINIMUM!	Declares a single precision value.
LIMIT%	Declares an integer value.
N$	Declares a string value.
ABC	Represents a single precision value.

Variable types may also be declared by including the GW-BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL in a program. These statements are described in detail in Section 6.

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,767.

Memory Space Requirements

The following list gives only the number of bytes occupied by the values represented by the variable names.

Variables	Type	Bytes
	Integer	2
	Single precision	4
	Double precision	8

Arrays	Type	Bytes
	Integer	2 per element
	Single precision	4 per element
	Double precision	8 per element

Type Conversion

When necessary, GW-BASIC will convert a numeric constant from one type to another. The following rules and examples apply to type conversions.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

   Example:

   ```
   10 A% = 23.42
   20 PRINT A%
   ```

   will yield

   ```
   23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

   Examples:

   ```
   10 D# = 6#/7
   20 PRINT D#
   ```

   will yield

   ```
   .8571428571428571
   ```

The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

```
10 D=6#/7
20 PRINT D
```

will yield

```
.857143
```

The arithmetic was performed in double precision, and the result was returned to D (single precision variable), rounded, and printed as a single precision value.

3. Logical operators (see "Logical Operators" later in this section) convert their operands to integers and return an integer result. Operands must be in the range −32768 to 32767 or an "Overflow" error occurs.

4. When a floating-point value is converted to an integer, the fractional portion is rounded.

   Example:

   ```
   10 C%=55.88
   20 PRINT C%
   ```

   will yield

   ```
   56
   ```

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E−8 times the original single precision value.

   Example:

   ```
   10 A=2.04
   20 B#=A
   30 PRINT A;B#
   ```

   will yield

   ```
   2.04 2.039999961853027
   ```

Expressions and Operators

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators. An expression always produces a single value.

Operators perform mathematical or logical operations on values. GW-BASIC operators may be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Each category is described in the following subsections.

Arithmetic Operators

The arithmetic operators, in order of evaluation, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
–	Negation	–X
*, /	Multiplication, Floating-point division	X*Y X / Y
\	Integer division	12\6 = 2 12\7 = 1
MOD	Modulus arithmetic	10.4 MOD 4 = 2
+, –	Addition, Subtraction	X+Y X–Y

You can change the order of evaluation by using parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of evaluation is maintained.

The following list gives some sample algebraic expressions and their GW-BASIC counterparts.

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
$X(-Y)$	X*(-Y)

Two consecutive operators must be separated by parentheses.

Integer Division and Modulus Arithmetic

In addition to the six standard operators (addition, subtraction, multiplication, division, negation, and exponentiation), GW-BASIC supports integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers before the division is performed, and the quotient is truncated to an integer. The operands must be in the range −32768 to 32767.

Examples:

```
10\4 = 2
25.68\6.99 = 3
```

Modulus arithmetic is denoted by the operator MOD. Modulus arithmetic yields the integer value that is the remainder of an integer division.

Examples:

```
10.4 MOD 4 = 2 (10/4 = 2 with the remainder 2)
25.68 MOD 6.99 = 5 (26/7 = 3 with the remainder 5)
```

Division By Zero and Overflow

If division by zero is encountered during the evaluation of an expression, a "Division by zero" error message is displayed. Machine infinity (the largest number that can be represented in floating-point format) with the sign of the numerator is supplied as the result of the division, and execution continues.

If the evaluation of an exponentiation operator results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, GW-BASIC displays an "Overflow" error message, supplies machine infinity with the algebraically correct sign as the result, and continues execution.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See the discussion of IF statements in Section 6.)

The relational operators are:

Operator	Relation Tested	Example
=	Equal to	X=Y
<>	Not equal to	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See the LET statement in Section 6.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression:

```
X+Y<(T-1) /Z
```

is true if the value of X plus Y is less than the value of T minus 1 divided by Z.

More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <>0 THEN K=K+1
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator performs bit-by-bit calculation and returns a result which is either "true" (not zero) or "false" (zero).

In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 2-1. The operators are listed in order of precedence.

Table 2-1: GW-BASIC Relational Operators Truth Table

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF statements in Section 6).

Examples:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range −32768 to 32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or −1, logical operators return 0 or −1. The given operation is performed on these integers bit by bit; in other words, each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16 = 16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110).
−1 AND 8 = 8	−1 = binary 1111111111111111 and 8 = binary 1000, so − 1 AND 8 = 8.

4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010 = 1010 (decimal 10).
−1 OR −2 = −1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X = −(X+1)	The two's complement of any integer is the bit complement plus one.

Functional Operators

When a function is used in an expression, it calls a predetermined operation that is to be performed on an operand. GW-BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All GW-BASIC intrinsic functions are described in Section 6.

GW-BASIC also allows "user-defined" functions that are written by the programmer. See "DEF FN Statement", Section 6.

String Operators

Strings may be concatenated by using the plus sign (+). For example:

```
10 A$ = "FILE" : B$ = "NAME"
20 PRINT A$ + B$
30 PRINT "NEW    "+A$ + B$
```

will yield

```
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=   <>   <   >   <=   >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples:

"AA"<"AB"
"FILENAME" = "FILENAME"
"X&">"X#"
"CL" >"CL"
"kg">"KG"
"SMYTH"<"SMYTHE"
B$<"9/12/78" where B$ = "8/12/78"

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

MS-DOS 2.0 File System

GW-BASIC includes commands which provide access to the file-management functions of MS-DOS. The following subsections describe this capability. These functions are available only if you are using GW-BASIC together with MS-DOS. If you use GW-BASIC without MS-DOS, these functions will give a syntax error.

Hierarchical File System

People naturally tend to think in hierarchical terms (for example, organization charts and family trees). It would be nice to allow users to organize their files on disk in a similar manner.

Consider this situation. In a particular business, both sales and accounting share a computer with a large disk, and the individual employees use it for preparation of reports and maintaining accounting information. One would naturally view the organization of files on the disk in this fashion:

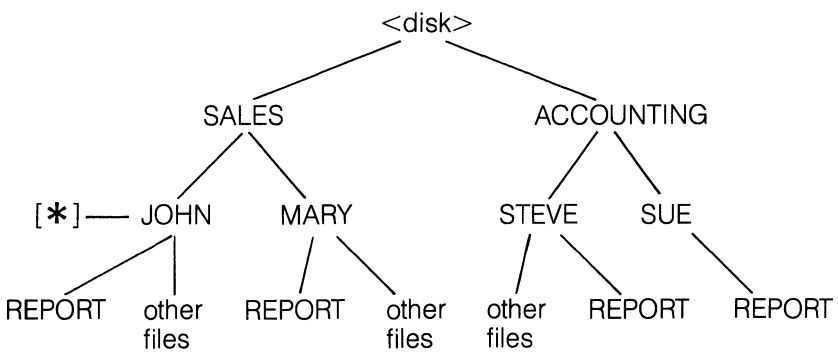


Figure 2-1: Disk file organization

Directory Paths

With GW-BASIC the user can organize a disk in such a manner that files not part of the current task do not interfere with that task.

MS-DOS 2.0 enables a directory to contain both files and subdirectories and to introduce the notion of the "current" directory.

To specify a filename, the user could use one of two methods: either specify a path from the root directory to the file, or specify a path from the current directory to the file.

A "Directory Path" is a series of directory names separated by '\' and ending with a filename. A path that starts at the root begins with a '\'.

There is a special directory entry in each directory, denoted by '..' that is the parent of the directory. The root directory's parent is itself.

Using a directory structure like the hierarchy above, and assuming that the current directory is at point [*] (directory JOHN), to reference the REPORT under JOHN, use any of the following paths:

```
REPORT
\SALES\JOHN\REPORT
..\JOHN\REPORT
```

To refer to the REPORT under MARY, use either of the following paths:

```
..\MARY\REPORT
\SALES\MARY\REPORT
```

To refer to the REPORT under SUE, use either of the following paths:

```
..\..\ACCOUNTING\SUE\REPORT
\ACCOUNTING\SUE\REPORT
```

There is no restriction on the depth of a tree (the length of the longest path from root to leaf), except in the number of allocation units available. The root directory will have a fixed number of entries, from 112 for floppy diskettes to many more for a large hard disk. For non-root directories, there is no limit to the number of files per directory except in the number of allocation units available.

Disks formatted with versions of MS-DOS earlier than version 2.0 will appear to MS-DOS 2.0 as having only a root directory containing files but no subdirectories.

File-Naming Conventions

Filenames follow MS-DOS naming conventions. All filenames may begin with a drive specification such as A: or B:. If no drive is specified, the current drive is assumed. If no period (.) appears in the filename and the filename is less than nine characters long, the default extension .BAS is appended to filenames used in the LOAD, SAVE, MERGE and RUN <filename> commands.

Examples:

```
RUN "NEWFILE.EXE"
RUN "A:NEWFILE.EXE"
SAVE "NEWFILE" (file is saved with a .BAS extension)
```

Several statements such as OPEN, KILL, FILES, MKDIR, CHDIR, and RMDIR allow a "directory path" to be given in addition to the standard filename and extension (See "Directory Paths" above).

<pathname> is used to denote when a directory path is legal in a statement.

A <pathname> may be complex as shown in "Directory Paths" above, or may simply be a filename (or <filespec>).

A <pathname> may not exceed 128 characters. Pathnames longer than 128 characters will cause a "Bad filename" error.

Note that <filespec> still refers to [device:] <filename> and may not contain directory names.

When the optional drive specification <d:> is included in a pathname, it must be first. If <d:> is omitted, the currently logged disk is assumed.

Example:

B:\SALES\JOHN\REPORT is legal, while:

\SALES\JOHN\B:REPORT is not.

Specifying a pathname where only a filespec is legal, or placing a drive specification elsewhere than at the beginning of the pathname will result in a "Bad filename" error.

Character Device Support

The general nature of the GW-BASIC file I/O system allows the user to take advantage of user-installed devices (see the MS-DOS 2.0 Device Drivers Manual for information on character devices).

Initially, character device drivers for LPT1, LPT2, and LPT3 are installed but may be replaced by the user. A line printer device may be opened with the statement:

OPEN "LPT1:" FOR OUTPUT AS #<filenumber>

or with the more general (preferred) statement:

OPEN "\DEV\LPT1" FOR OUTPUT AS #<filenumber>

If a user writes and installs a device called FOO, then the OPEN statement would appear as:

OPEN "\DEV\FOO" FOR (mode) AS #<filenumber>

The following rules apply to device drivers:

1. For user-installed devices, the second form of OPEN must be used (except LPT1:, LPT2:, and LPT3:).

   The reason is that BASIC knows about certain predefined devices (KYBD:, SCRN:, and so on) by the fact that the device name ends with a colon (':'). Only LPT1:, LPT2:, and LPT3: of the pre-defined devices may be replaced and new ones added.

   Such devices are OPENed and used in the same manner as disk files, except that characters are not buffered by BASIC as they are for disk files.

2. By default, the record length is set to 1 unless explicitly set to some other value by the LEN = <lrecl> option in the OPEN statement.

   When the LEN = <lrecl> option is used, BASIC will buffer that many characters before sending them to the driver.

3. BASIC sends only a <CR>(carriage return = &H0D) as the end-of-line indicator. If the device requires a <LF> (line feed = &H0A), the driver must provide it.

4. When writing device drivers, remember that BASIC will want to read and write control information.

   Writing and reading of device control data is handled by the BASIC IOCTL statement and IOCTL$(f) function. (See "IOCTL statement" in Section 6 for the format of IOCTL control data strings.)

5. Necessary control functions that your device driver must provide are:

   a. Set a maximum line width as requested by the BASIC OPEN statement.

   b. Return the current maximum line width when BASIC asks for it.

   c. To close a sequential input file open to a device driver, have the input devices return an "End of File" indicator to BASIC (used by the EOF(f) statement). If BASIC attempts to read past the end of the device input stream, the driver should return a CTRL-Z. BASIC uses this code to give the "Input past end" error.

For more information, see the discussions in Section 6 about the IOCTL statement, the IOCTL$ function, and the device error variables ERDEV and ERDEV$.

Assembly Language Subroutines

You may call assembly language subroutines from your GW-BASIC program with the USR function or the CALL or CALLS statement. The USR function enables you to call an assembly language subroutine to return a value in the same way you call GW-BASIC intrinsic functions. However, for most machine language programs, you should use the CALL or CALLS statement. These statements produce more readable source code and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than is the USR function.

Memory Allocation

Memory space must be set aside for an assembly language subroutine before the subroutine can be loaded. To set aside memory space, use the /M: switch during start-up. The /M: switch sets the highest memory location to be used by GW-BASIC.

In addition to the GW-BASIC code area, GW-BASIC uses up to 64K of memory beginning at its data segment (DS).

If more stack space is needed when an assembly language subroutine is called, you can save the GW-BASIC stack and set up a new stack for use by the assembly language subroutine. The GW-BASIC stack must be restored, however, before you return from the subroutine.

The assembly language subroutine can be loaded into memory in one of two ways. First, you may use the DEBUG facility provided with your MS-DOS operating system. Also, you may read your linked output file (EXE) with random input and poke it into memory.

When loading a subroutine, observe these guidelines:

- Make sure the subroutines do not contain any long references.
- Skip over the first 512 bytes of the linker's output file (.EXE), and then read in the rest of the file.

The CALL Statement

The CALL statement is the recommended way of interfacing machine language subroutines with GW-BASIC. Do not use the USR function unless you are running previously written subroutines that already contain USR functions.

The syntax of the CALL statement is:

CALL <variable name> [(<argument list>)]

where <variable name> contains the segment offset that is the starting point in memory of the subroutine being called.

<argument list> contains the variables or constants, separated by commas, to be passed to the subroutine.

Invoking the CALL statement causes the following events to occur:

- For each argument in the argument list, the two-byte offset of the argument's location within the data segment (DS) is pushed onto the stack.
- The GW-BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.
- Control is transferred to the subroutine with a long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

Figure 2-2 illustrates the state of the stack at the time the CALL statement is executed. Figure 2-3 illustrates the condition of the stack during execution of the called subroutine.

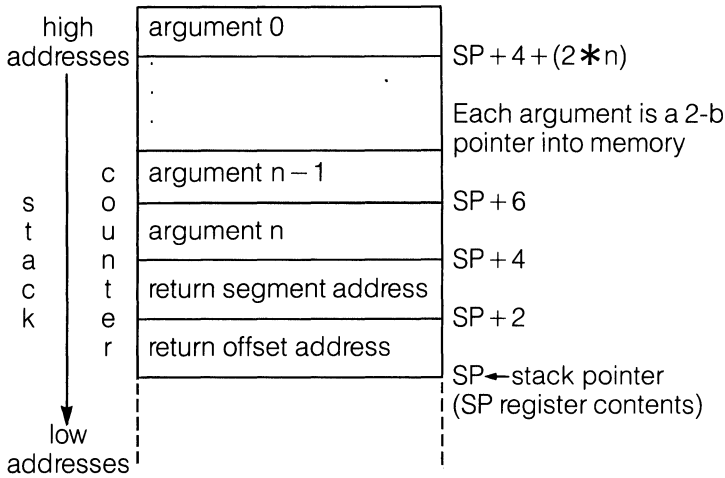


Figure 2-2: Stack layout when CALL statement is activated

After the CALL statement has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

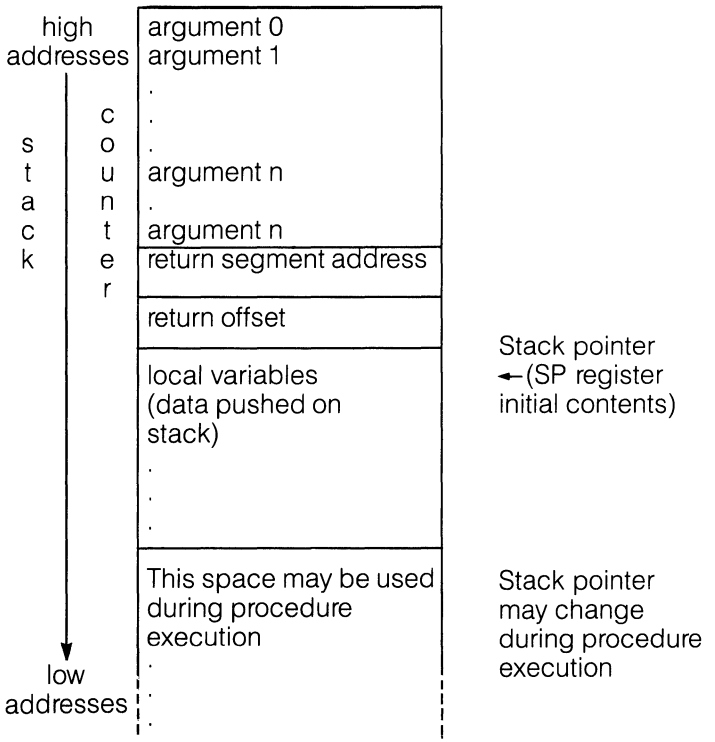


Figure 2-3: Stack layout during execution of a CALL statement

Observe the following rules when coding a subroutine:

1. The called routine must preserve segment registers DS, ES, SS, and BP. If interrupts are disabled in the routine, they must be re-enabled before exiting. The stack must be cleaned up when the called routine exits.

2. The called routine must know the number and length of the arguments passed. The following assembly language routine shows an easy way to refer to arguments:

   ```
   PUSH BP
   MOV BP,SP
   ADD BP, (2*number of arguments)+4
   ```

   Then:

   ```
   argument 0 is at BP
   argument 1 is at BP-2
   argument n is at BP-2*n

   (number of arguments = n+1)
   ```

3. Variables may be allocated either in the code segment (CS relative) or on the stack. Be careful not to modify the return segment and offset stored on the stack.

4. The called subroutine must perform a RET <I> statement (where <I> is two times the number of arguments in the argument list). The purpose of the RET <I> statement is to adjust the stack to the start of the calling sequence.

5. Values are returned to GW-BASIC by including in the argument list the name of the variable that will receive the result.

6. If the argument is a string, the argument's offset points to three bytes which, as a unit, are called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2 are, respectively, the lower and upper 8 bits of the string starting address in the string space.

   Warning: If the argument is a string literal in the program, the string descriptor will point to program text, which can alter or destroy your program. To avoid unpredictable results, add + "" to the string literal in the program. For example, use:

   ```
   20 A$ = "BASIC" + ""
   ```

This addition forces the string literal to be copied into the string space. Then the string may be modified without affecting the program.

7. The contents of a string may be altered by user routines, but the descriptor **must not** be changed. Do not write past the end-of-string. GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

8. Data areas needed by the routine must be allocated either in the code segment of the user routine or on the stack. It is not possible to declare a separate data area in the user assembler routine.

Example of a CALL statement:

```
100 DEF SEG = &H8000
110 FOO = &H7FA
120 CALL FOO(A,B$,C)
.
.
.
```

Line 100 sets the segment to 8000 Hex. FOO is set to &H7FA by line 110 so the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence in assembly language demonstrates access to the arguments passed. The returned result is stored in the variable 'C'.

```
PUSH    BP              ;Set up pointer to arguments.
MOV     BP,SP
ADD     BP,(4+2*3)
MOV     BX,[BP-2]       ;Get address of B$ descriptor.
MOV     CL,[BX]         ;Get length of B$ in CL.
MOV     DX,1[BX]        ;Get address of B$ text in DX.
          .
          .
          .
MOV     SI,[BP]         ;Get address of 'A' in SI.
MOV     DI,[BP-4]       ;Get pointer to 'C' in DI.
MOVS    WORD            ;Store variable 'A' in 'C'.
POP     BP
RET     6               ;Restore stack, return.
```

Important: The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction MOVS WORD, copies only two bytes. This is fine if variables A and C are integers. You would have to copy four bytes if the variables were numbers in floating-point format.

The CALLS Statement

The CALLS statement should be used to access subroutines that were written using MS-FORTRAN calling conventions. CALLS works just like CALL; however, with CALLS, the arguments are passed as segmented addresses, rather than as unsegmented addresses.

Because MS-FORTRAN routines need to know the segment value for each argument passed, the segment is first pushed and then the offset is pushed. For each argument, four bytes are pushed rather than two, as in the CALL statement. Therefore, if your assembler routine uses the CALLS statement, n in the RET <n> statement is four times the number of arguments.

The USR Function

Although using the CALL statement is the recommended way of calling assembly language subroutines, the USR function is also available for this purpose. This availability ensures compatibility with older programs that contain USR functions.

USR[<digit>][(<argument>)]

<digit> is from 0 to 9. <digit> specifies which USR routine is being called. If <digit> is omitted, USR0 is assumed.

<argument> is any numeric or string expression. Arguments are discussed in detail in the following paragraphs.

In the GW-BASIC Interpreter, a DEF SEG statement **must** be executed prior to a USR function call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR function call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value that specifies the type of argument given. The value in AL may be one of the following:

Value in AL	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Floating-point number

If the argument is a number, the BX register pair points to the Floating-Point Accumulator (FAC) where the argument is stored.

If the argument is a string, the DX register pair points to three bytes which, as a unit, are called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2 are, respectively, the lower and upper 8 bits of the string starting address in the GW-BASIC data segment.

Warning: If the argument is a string literal in the program, the string descriptor will point to program text, which can alter or destroy the program.

Usually, the value returned by a USR function has the same type (integer, string, or floating-point) as the argument that was passed to it.

Example of a USR function:

```
110 DEF USR0 = &H8000 'Assumes user gave /M:32767
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument passed.

Section 3

Programming Animation

This section explains the basic techniques for using the GW-BASIC animation statements and functions. A sample program included in this section illustrates the use of the animation statements and functions described later in Section 6, "GW-BASIC Commands, Statements, Functions, and Variables."

Before introducing the sample program, this section briefly describes some of the important animation features of GW-BASIC for the Mindset Personal Computer. Then the section divides the sample program into modules. Each module contains a description followed by a listing of the program statements in the module. A complete listing of the sample program appears at the end of the section.

Animation Features

GW-BASIC simplifies the task of programming animation by performing many routine tasks automatically. For example, as an object moves across the screen, GW-BASIC saves the background patterns as they are obscured by the object in the foreground. As the background is uncovered behind the object, GW-BASIC replaces the original background patterns.

Animation features of GW-BASIC include programmable motion and viewing priority, multiple views of an object, statements that control animation events, functions that control animation events, event control statements, and statements that activate the animation object.

Programmable Motion and Priority

GW-BASIC for the Mindset Personal Computer uses an OBJECT statement to define the motion for an object. This statement determines the initial location for an object, the destination for the object, and the speed

at which the object moves to its destination. (For information on other animation parameters in this statement, see the OBJECT statement in Section 6.)

The number of each object determines its viewing priority. When two objects overlap, the object with the higher number is drawn on top of the other object.

Multiple Object Views

An animation object in GW-BASIC can include up to eight different views of the object. As the object proceeds across the screen, GW-BASIC changes the view as often as specified in an OBJECT statement for that object.

To define an object, the application program can first draw each view of the object on the screen and then store the view in its own array, using the graphics version of the GET statement. You can use the BSAVE statement to save object arrays on disk and recall them later with a BLOAD statement. You can use this technique to increase the effective size of an animation program by creating and saving objects in a separate program. You may also use any image generated external to GW-BASIC, provided the format corresponds to an array produced by the GET statement.

The DIM OBJECT statement simplifies the sizing of arrays for object views. It enables you to specify the rectangle containing the view rather than requiring you to calculate the number of words required for the array. The SCREEN statement must establish the screen mode before the DIM OBJECT statement can accurately calculate the number of words required to contain the array.

After the object views are stored in arrays, the DEF OBJECT statement logically links the arrays together to define an object. The object views appear in the order listed in the DEF OBJECT statement. An array may be used more than once in the definition for a single object. An array may also be used in the definition of more than one object.

Animation Event Statements

Perhaps the most powerful feature of GW-BASIC animation is the ability to set an object in motion and then continue with other unrelated statements while the object moves across the screen. GW-BASIC for the Mindset Personal Computer includes ON ARRIVAL, ON CLIP, and ON COLLISION statements. These call subroutines to handle events that occur as a result of object motion.

The ON ARRIVAL statement calls a subroutine when one or more objects reach their destinations. This subroutine then redirects the objects, changes their definition, or performs any other desired processing.

The ON CLIP statement calls a subroutine when one or more specified objects reach the boundary of the screen (or the current window). The ON COLLISION statement calls a subroutine when two or more specified objects collide.

Animation Event Functions

In addition to the animation event *statements* ON ARRIVAL, ON CLIP, and ON COLLISION, GW-BASIC for the Mindset Personal Computer includes ARRIVAL, CLIP, and COLLISION *functions*. These functions determine whether the corresponding events have occurred since the last time they were checked.

There are three forms of the ARRIVAL function (and three forms of the CLIP and COLLISION functions as well): ARRIVAL(−1), ARRIVAL(0), and ARRIVAL(<object number>).

The first form of the ARRIVAL function, ARRIVAL(−1), returns −1 if one or more objects have reached their destinations; it returns 0 otherwise.

This form of the function also latches the arrival status for the objects which have arrived, providing a "snapshot" of the arrival status. This snapshot enables the subroutines handling all arrived objects to be executed without being confused by arrival events occurring after the ARRIVAL(−1) function is executed.

The second form of the ARRIVAL function, ARRIVAL(0), is used after the ARRIVAL(−1) function has latched the arrival status. It returns the number of the lowest-numbered object which has arrived.

After the ARRIVAL(0) function returns the number of an object, it clears the arrival status for that object. As a result, the next ARRIVAL(0) function can return the number of the next object which had arrived when the ARRIVAL(−1) function was executed. The ARRIVAL(0) function returns 0 if there are no more objects which arrived when the ARRIVAL(−1) function was last executed.

The ARRIVAL(<object number>) form of the ARRIVAL function tests for the arrival of a specific object. This function returns the object number if the specified object has arrived; it returns 0 otherwise.

Consider the following example of the ARRIVAL function, where objects 15 and 2 are the only objects on the screen that have arrived at their assigned destinations. In this example, you might define a subroutine to:

1. Call the ARRIVAL(−1) function to check for arrivals and to latch the arrival status of any objects that have arrived. Here, the ARRIVAL(−1) function returns −1, indicating that some objects have arrived at their destinations.

2. Call the ARRIVAL(0) function to determine the number of the lowest-numbered object which has arrived. The ARRIVAL(0) function returns a value of 2 and clears the arrival status of object 2.

3. Execute a routine to handle the arrival of object 2.

4. Call the ARRIVAL(0) function to determine the number of the next arrived object. The ARRIVAL(0) function returns a value of 15 and clears the arrival status of object 15. (Alternately, you could use the ARRIVAL(15) form of the function to determine only if object 15 had arrived.)

5. Execute a routine to handle the arrival of object 15.

6. Call the ARRIVAL(0) function to determine the number of the next arrived object. The ARRIVAL(0) function returns a value of 0, because the arrival status of all objects has been cleared.

7. Return to the main program.

Animation Event Control Statements

Nine statements control the operation of the event-related statements. These control statements enable, disable, or temporarily suspend the testing for animation events. The arrival statements and functions can illustrate how animation control statements work.

The ARRIVAL ON statement enables the operation of the ON ARRIVAL statement and the ARRIVAL function. GW-BASIC does not test for object arrivals until the ARRIVAL ON statement is executed.

The ARRIVAL OFF statement disables the operation of the ON ARRIVAL statement and the ARRIVAL function.

The ARRIVAL STOP statement temporarily suspends the operation of the ON ARRIVAL statement. Object arrivals that occur while ARRIVAL STOP is in effect are remembered, but no action is taken until the ARRIVAL ON statement is executed.

Suppose one or more arrivals took place while ARRIVAL STOP was in effect. After the ARRIVAL ON statement is executed, the ON ARRIVAL statement operates as if any remembered arrivals had just taken place. Also, the ARRIVAL function will then return a value other than 0 for the remembered arrivals.

To prevent recursive arrival traps from occurring, GW-BASIC performs an implicit ARRIVAL STOP when the ON ARRIVAL statement calls a subroutine. When the subroutines for all arrived objects have been executed, GW-BASIC performs an implicit ARRIVAL ON. If you wish, you can include an explicit ARRIVAL OFF statement in the arrival-handling subroutine to override the implicit ARRIVAL ON performed by GW-BASIC.

Object Activation Statements

After an object is defined with a DEF OBJECT statement and the object's origin, destination, and speed are defined with an OBJECT statement, the ACTIVATE statement causes the object to appear at its origin and travel toward its destination.

The complement of the ACTIVATE statement is the DEACTIVATE statement. DEACTIVATE removes one or more objects from the screen and prevents them from being involved in arrivals, clips, or collisions.

Overview of Animation Programming

The following steps describe one of several ways to cause an object to move.

1. Select the graphics display mode desired for animation. This step must occur before using the DIM OBJECT statement.
2. Dimension the arrays which are to hold the object data.
3. Clear the screen and draw the views for each object on the screen.
4. Store the views in the arrays dimensioned in Step 2.
5. Clear the screen again, if necessary.
6. Define the group of views which comprise each object.
7. Set up the ON ARRIVAL, ON CLIP, and ON COLLISION event traps (unless you plan to use animation event functions instead).

8. Define the procedures for each event trap.

9. Enable the event traps at the appropriate points in the program.

10. Define the object parameters such as origin, destination, and speed.

11. Activate the desired objects to begin animation.

The following subsections describe each of these steps and include the appropriate sections of the sample animation program.

Sample Animation Program Description

The sample program uses five objects to demonstrate the animation statements and functions. When the program begins operation, two faces begin moving from their initial positions at the left side of the screen. As the faces travel across the screen, they open and close their mouths.

Each face follows a predefined path from the left side of the screen to the right. At the right side of the screen, the faces are replaced with words enclosed in a circle. These words then travel back to the positions where the faces first appeared and the cycle begins again. The program continues until any key is pressed.

The first face travels from left to right across the middle of the screen. When it arrives at the right side of the screen, the word "DONE" replaces the face and travels back to the left side of the screen. This procedure demonstrates the use of the DIM OBJECT, DEF OBJECT, ON ARRIVAL, ARRIVAL ON, OBJECT, ACTIVATE, and DEACTIVATE animation statements.

The second face follows a zigzag path which intersects the path of the first face. When the second face reaches the right side of the screen, the word "CLIP" replaces the face and travels back to the left side of the screen. In addition to the statements used with the first face, this procedure demonstrates the use of the ON CLIP and CLIP ON statements and the OBJECT function.

When the faces collide, they stop moving and the word "HIT" enclosed within a circle overlays the point of collision between the two objects. After a brief pause, the HIT circle disappears and the objects continue

moving across the screen. This procedure demonstrates the use of the ON COLLISION and COLLISION ON statements and how an object overlays other objects with numerically lower object numbers.

In the sample program, note that comments are used liberally to describe its operation. Although using frequent comments is a commendable programming practice, it is inadvisable to use this many in a real program because each comment reduces the amount of memory available for the program and its objects.

Initializing the System

Initialize the system by clearing the screen with the CLS statement and then by selecting one of the graphics modes with the screen statement.

```
10  '       ******** SAMPLE ANIMATION PROGRAM *****
20  '
30  '       ***** INITIALIZE SYSTEM *****
40  '
50  CLS
60  SCREEN 2
65  KEY OFF      'Erase function key display
70  '
```

Dimensioning the Object Arrays

Use the DIM or DIM OBJECT statements to dimension the arrays which hold the data for each view of an object. DIM requires the use of a formula to determine the number of elements required in each array. DIM OBJECT is easier to use because it takes the horizontal and vertical dimensions of the object and computes the number of required array elements.

The sample program uses six views which appear in five objects. Table 3-1 lists the array names used in this program. The array names indicate the object views which the arrays will contain.

Table 3-1: Sample Program Array Names

ARRAY NAME	VIEW
COLLIDED%	Circle containing the word "HIT".
AGAPE%	Face showing a mouth opened wide.
OPENED%	Face showing a slightly opened mouth.
CLOSED%	Face showing a closed mouth.
ARRIVED%	Circle containing the word "DONE".
CLIPPED%	Circle containing the word "CLIP".

```
80  '     ***** DIMENSION OBJECT ARRAYS *****
90  '
100 DIM OBJECT COLLIDED%(41,19)          ' View for object 6
110 DIM OBJECT AGAPE%(41,19)             ' View 1 for objects 2 and 3
120 DIM OBJECT OPENED%(41,19)            ' View 2 for objects 2 and 3
130 DIM OBJECT CLOSED%(41,19)            ' View 3 for objects 2 and 3
140 DIM OBJECT ARRIVED%(41,19)           ' View for object 5
150 DIM OBJECT CLIPPED%(41,19)           ' View for object 4
160 '
```

Generating the Views for Each Object

Use graphics statements and print statements to generate the views for each object so they can be stored and linked to create objects. This section of the program generates the six views listed in Table 3-1.

```
170 '     ***** DRAW OBJECTS *****
180 '
190 PRINT:PRINT:PRINT:PRINT " CLIP    HIT     DONE":LOCATE 13,9,0
200 '  Spaces for line 190:      s     sss     ssss
210 CIRCLE (22,9),20,,,-.7,-5.75         ' Face with mouth agape
220 CIRCLE (22,3),2                      ' Eye for face with mouth agape
230 CIRCLE (76,9),20,,,-.3,-5.9          ' Opened mouth face
240 CIRCLE (83,4),2                      ' Eye for opened mouth face
250 CIRCLE (134,9),20,,,-.05,-6.2        ' Closed mouth face
260 CIRCLE (141,5),2                     ' Eye for closed mouth face
270 CIRCLE (22,27),20                    ' Circle to hold 'CLIP' word
280 CIRCLE (76,27),20                    ' Circle to hold 'HIT' word
290 CIRCLE (134,27),20                   ' Circle to hold 'DONE' word
300 '
```

Storing the Object Views in Arrays

Use the graphics GET statement to store the generated object views in the arrays you dimensioned with the DIM or DIM OBJECT statements earlier. The syntax of the graphics GET statement is very similar to that of the LINE statement. Substituting the LINE statement with the B option can be used to draw a visible box around a view to verify that the specified dimensions capture the expected portion of the screen.

```
310 '     ***** STORE OBJECT VIEWS IN ARRAYS *****
320 '
330 GET (2,0)-(42,18),AGAPE%             ' Store agape face
340 GET (56,0)-(96,18),OPENED%           ' Store opened face
350 GET (114,0)-(154,18),CLOSED%         ' Store closed face
360 GET (2,19)-(42,36),CLIPPED%          ' Store CLIP circle
370 GET (56,19)-(96,36),COLLIDED%        ' Store HIT circle
380 GET (114,19)-(154,36),ARRIVED%       ' Store DONE circle
390 '
```

Defining the Views Comprising an Object

The DEF OBJECT statement defines the object views that are part of an object. The OBJECT statement, described later, includes a parameter which determines how long each view is visible before the next view appears. The object views appear in the order they are listed in each DEF OBJECT statement.

```
400 '     ***** DEFINE VIEWS FOR EACH OBJECT *****
410 '
420 DEF OBJECT 6 AS COLLIDED%                                  ' 1 view
430 DEF OBJECT 2 AS CLOSED%,OPENED%,AGAPE%,OPENED%             ' 4 views
440 DEF OBJECT 3 AS CLOSED%,OPENED%,AGAPE%,OPENED%             ' 4 views
450 DEF OBJECT 4 AS CLIPPED%                                   ' 1 view
460 DEF OBJECT 5 AS ARRIVED%                                   ' 1 view
470 '
```

Setting Up the Event Traps

There are three event trap statements associated with animation: ON ARRIVAL, ON CLIP, and ON COLLISION. Each event trap statement defines the line number of the subroutine which handles that event. You can have up to 17 arrival event traps, 17 clip event traps, and 17 collision event traps in effect concurrently.

```
480 '     ***** SET UP ANIMATION EVENT TRAPS *****
490 '
500 ON ARRIVAL (2) GOSUB 10000      ' Subroutine for object 2 arrival
510 ON ARRIVAL (3) GOSUB 11000      ' Subroutine for object 3 arrival
520 ON ARRIVAL (4) GOSUB 12000      ' Subroutine for object 4 arrival
530 ON ARRIVAL (5) GOSUB 13000      ' Subroutine for object 5 arrival
540 ON CLIP (3) GOSUB 14000         ' Subroutine for object 3 clip
550 ON COLLISION GOSUB 15000        ' Subroutine for object collisions
560 '
```

Enabling the Event Traps

Animation event detection is turned on with the ARRIVAL ON, CLIP ON, and COLLISION ON statements. To turn off the event detection, use the ARRIVAL OFF, CLIP OFF, and COLLISION OFF statements.

```
570 '     ***** ENABLE EVENT TRAPPING *****
580 '
590 ARRIVAL ON                      ' Enable arrival event trapping
600 CLIP ON                         ' Enable clip event trapping
610 COLLISION ON                    ' Enable collision event trapping
```

Defining Object Parameters and Activating the Object

The OBJECT statement specifies the attributes of an object. For each object, this statement defines the initial position, the final position, the length of time which each view is visible, the initial view, and the speed at which the object travels to its final position.

The OBJECT statement also includes offset and transparency parameters. The offset parameter determines which point in the object is placed at the location specified by the position parameters. An offset of (0,0) causes the OBJECT position parameter to refer to the center of the rectangle containing the object. The offset for an object which looks like an arrow could be set to represent the point on the arrow.

In the example program, object 3 moves in a zigzag path. Unlike object 2 which has a single destination at the right side of the screen, object 3 receives a new destination, (X3,Y3), each time it moves 100 logical units toward the right. A direction flag, DIRFLAG, indicates whether the object should go up or down to its next destination. After object 3 goes up, the program toggles the direction flag to indicate that the object should go down to its next destination.

In addition to the object view itself, GW-BASIC stores the background contained in the rectangle defined by the GET statement. A value of 0 for the transparency parameter in the OBJECT statement causes the background in the rectangle to be transparent. A value of 1 causes the background to be displayed.

The ACTIVATE statement causes an object to appear at its initial position and causes it to move toward its final position. If the speed parameter in the OBJECT statement is 0, the object will remain stationary.

```
630 '     ***** SET OBJECTS 2 AND 3 IN MOTION *****
640 '
650 OBJECT 2,1=20,2=100,3=609,4=100,5=.4,10=50      ' Go left to right
660 X2 = 75: Y2 = 20                                ' Initial destination
670 DIRFLAG = -1                                    ' Set direction flag
680 OBJECT 3,1=20,2=180,3=X2,4=Y2,5=.2,10=120   ' Move up to peak
690 ACTIVATE 2,3                                    ' Make 2 and 3 move
700 '
```

Including the Background Program

The background program provides a way for the user to end the main program. The background program runs at the same time that GW-BASIC performs the animation defined by the OBJECT statement. This program simply waits for the user to press a key to terminate program operation.

```
710  ' *** WAIT FOR USER TO PRESS A KEY TO END PROGRAM ***
720  '
730  ' Cursor is at line 13, column 9
740  PRINT "Press any key to terminate program."
750  A$ = INKEY$: IF A$ = "" THEN 750      ' Test for input key
760  DEACTIVATE 'Remove objects from screen
770  END
9990 '
9999 '
```

Using ARRIVAL Subroutines

This subsection illustrates the sample subroutines called by the ON ARRIVAL statements described earlier.

The OBJECT 2 ARRIVAL SUBROUTINE assumes control when object 2 reaches its final position, at the right side of the screen. The subroutine replaces object 2 with object 5 (the circle containing the word DONE) and sends object 5 back to the original position for object 2.

The OBJECT 3 ARRIVAL SUBROUTINE assumes control when object 3 reaches its final position, which is 60 units above or below the horizontal line traversed by object 2. This subroutine causes object 3 to follow a zigzag path from left to right.

The OBJECT 4 ARRIVAL SUBROUTINE assumes control when object 4 reaches its final position, which is the same as the initial position of object 3. This subroutine replaces object 4 with object 3 and restarts object 3 on its original zigzag path from left to right across the screen.

The OBJECT 5 ARRIVAL SUBROUTINE assumes control when object 5 reaches its final position, which is the same as the initial position of object 2. This subroutine replaces object 5 with object 2 and restarts object 2 on its straight path from left to right across the screen.

```
10000  '     ***** OBJECT 2 ARRIVAL SUBROUTINE *****
10020  '
10030  DEACTIVATE 2                                    ' Vanish object 2
10040  OBJECT 5,1=609,2=100,3=20,4=100,9=1,10=100' Set up object 5
10050  ACTIVATE 5                                      ' Make object 5 move
10060  RETURN
10070  '
11000  '     ***** OBJECT 3 ARRIVAL SUBROUTINE *****
11010  '
11020  X3 = OBJECT(3,1) + 100                          ' Move object 3 right 100 units
11030  ' Go up if direction flag is positive; go down otherwise
11040  IF DIRFLAG > 0 THEN Y3 = 20 ELSE Y3 = 180
11050  OBJECT 3,3=X3,4=Y3                              ' Set up next move for 3
```

continued on next page

```
11060  DIRFLAG = DIRFLAG * (-1)              ' Toggle direction flag
11070  RETURN
11080  '
12000  '     ***** OBJECT 4 ARRIVAL SUBROUTINE *****
12010  '
12020  DEACTIVATE 4                                  ' Vanish object 4
12030  X3 = 75: Y3 = 20                              ' Reset destination
12040  OBJECT 3,1=20,2=180,3=X3,4=Y3                 ' Set up object 3
12050  ACTIVATE 3                                    ' Send 3 on its way
12060  RETURN
12070  '
13000  '
13010  '     ***** OBJECT 5 ARRIVAL SUBROUTINE *****
13020  '
13030  DEACTIVATE 5                                  ' Vanish object 5
13040  OBJECT 2,1=20,2=100,3=609,4=100               ' Set up object 2
13050  ACTIVATE 2                                    ' Send 2 on its way
13060  RETURN
13070  '
```

Using CLIP Subroutines

This subsection illustrates a sample subroutine which is called by the ON CLIP statement in line 540. The OBJECT 3 CLIP SUBROUTINE assumes control when object 3 reaches the right boundary of the screen. The subroutine replaces object 3 with object 4 (the circle containing the word CLIP) and sends object 4 back to the original position for object 3.

```
14000  '     ***** OBJECT 3 CLIP SUBROUTINE *****
14010  '
14020  DEACTIVATE 3                          ' Vanish object 3
14030  X4 = OBJECT(3,1)                      ' Initial X for 4 = last X for 3
14040  Y4 = OBJECT(3,2)                      ' Initial Y for 4 = last Y for 3
14050  ' Set new initial location for object 4
14060  OBJECT 4,1=X4,2=Y4,3=20,4=180,5=1,10=170
14070  ACTIVATE 4                    ' Send 4 on its way to home of 3
14080  RETURN
14090  '
```

Using COLLISION Subroutines

This subsection illustrates a sample subroutine which is called by the ON COLLISION statement in line 550. The OBJECT COLLISION SUBROUTINE assumes control when any two objects collide. The subroutine stops the objects and overlays the point of collision between the two objects with object 6 (the circle containing the word HIT). After moving one unit to the right very slowly (effectively a pause), this subroutine erases object 6 and restarts the objects on their predefined paths.

```
15000  '     ***** OBJECT COLLISION SUBROUTINE *****
15010  '
16000  STOP OBJECT                          ' Freeze objects
16010  A = COLLISION(0)                     ' Get lowest object which collided
16020  B = COLLISION(A)                     ' Find who it collided with
16030  ' Calculate average x and y positions of collided objects
16030  XC=(OBJECT(A,1)+OBJECT(B,1))/2
16040  YC=(OBJECT(A,2)+OBJECT(B,2))/2
16050  ' Place object 6 at average position calculated above
16060  OBJECT 6,1=XC,2=YC,3=XC+1,4=YC,10=2,9=1     ' Move slowly
16070  COLLISION OFF: ACTIVATE 6                   ' Make it visible
16080  START OBJECT 6                              ' Start it moving
16090  WHILE (((ARRIVAL(-1))*ARRIVAL(6))=0): WEND ' Wait for it to arrive
16100  DEACTIVATE 6 :COLLISION  ON                 ' Make it disappear
16110  START OBJECT                         ' Restart the other objects
16120  RETURN
```

The complete program listing is shown below.

```
10  '       ****** SAMPLE ANIMATION PROGRAM  ******
20  '
30  '     ***** INITIALIZE SYSTEM *****
40  '
50  CLS
60  SCREEN 2
65  KEY OFF                                  'Erase function key display
70  '
80  '     ***** DIMENSION OBJECT ARRAYS *****
90  '
100   DIM OBJECT COLLIDED%(41,19)            ' View for object 6
110   DIM OBJECT AGAPE%(41,19)               ' View 1 for objects 2 and 3
120   DIM OBJECT OPENED%(41,19)              ' View 2 for objects 2 and 3
130   DIM OBJECT CLOSED%(41,19)              ' View 3 for objects 2 and 3
140   DIM OBJECT ARRIVED%(41,19)             ' View for object 4
150   DIM OBJECT CLIPPED%(41,19)             ' View for object 5
160   '
170   '     ***** DRAW OBJECTS *****
180   '
190   PRINT:PRINT:PRINT:PRINT "  CLIP     HIT      DONE":LOCATE  13,9,0
200   '   Spaces for line 190:        s       sss      ssss
210   CIRCLE (22,9),20,,,-.7,-5.75    ' Face with mouth agape
220   CIRCLE (22,3),2                 ' Eye for face with mouth agape
230   CIRCLE (76,9),20,,,-.3,-5.9     ' Opened mouth face
240   CIRCLE (83,4),2                 ' Eye for opened mouth face
250   CIRCLE (134,9),20,,,-.05,-6.2   ' Closed mouth face
260   CIRCLE (141,5),2                ' Eye for closed mouth face
270   CIRCLE (22,27),20               ' Circle to hold 'CLIP' word
280   CIRCLE (76,27),20               ' Circle to hold 'HIT' word
290   CIRCLE (134,27),20              ' Circle to hold 'DONE' word
300   '
```

continued on next page

```
310  '     ***** STORE OBJECT VIEWS IN ARRAYS *****
320  '
330  GET (2,0)-(42,18),AGAPE%                    ' Store agape face
340  GET (56,0)-(96,18),OPENED%                   ' Store opened face
350  GET (114,0)-(154,18),CLOSED%                ' Store closed face
360  GET (2,19)-(42,36),CLIPPED%                  ' Store CLIP circle
370  GET (56,19)-(96,36),COLLIDED%                ' Store HIT circle
380  GET (114,19)-(154,36),ARRIVED%               ' Store DONE circle
390  '
400  '     ***** DEFINE VIEWS FOR EACH OBJECT *****
410  '
420  DEF OBJECT 6 AS COLLIDED%                                    ' 1 view
430  DEF OBJECT 2 AS CLOSED%,OPENED%,AGAPE%,OPENED%               ' 4 views
440  DEF OBJECT 3 AS CLOSED%,OPENED%,AGAPE%,OPENED%               ' 4 views
450  DEF OBJECT 4 AS CLIPPED%                                     ' 1 view
460  DEF OBJECT 5 AS ARRIVED%                                     ' 1 view
470  '
480  '     ***** SET UP ANIMATION EVENT TRAPS *****
490  '
500  ON ARRIVAL (2) GOSUB 10000          ' Subroutine for object 2 arrival
510  ON ARRIVAL (3) GOSUB 11000          ' Subroutine for object 3 arrival
520  ON ARRIVAL (4) GOSUB 12000          ' Subroutine for object 4 arrival
530  ON ARRIVAL (5) GOSUB 13000          ' Subroutine for object 5 arrival
540  ON CLIP (3) GOSUB 14000             ' Subroutine for object 3 clip
550  ON COLLISION GOSUB 15000            ' Subroutine for object collisions
560  '
570  '     ***** ENABLE EVENT TRAPPING *****
580  '
590  ARRIVAL ON                          ' Enable arrival event trapping
600  CLIP ON                             ' Enable clip event trapping
610  COLLISION ON                        ' Enable collision event trapping
620  '
630  '     ***** SET OBJECTS 2 AND 3 IN MOTION *****
640  '
650  OBJECT 2,1=20,2=100,3=609,4=100,5=.4,10=50    ' Go left to right
660  X2 = 75: Y2 = 20                              ' Initial destination
670  DIRFLAG = -1                                  ' Set direction flag
680  OBJECT 3,1=20,2=180,3=X2,4=Y2,5=.2,10=120     ' Move up to peak
690  ACTIVATE 2,3                                  ' Make 2 and 3 move
700  '
710  '     ** WAIT FOR USER TO PRESS A KEY TO END PROGRAM **
720  '
730  ' Cursor is at line 13, column 9
740  PRINT "Press any key to terminate program."
750  A$ = INKEY$: IF A$ = "" THEN 750          ' Test for input key
760  DEACTIVATE   ' Remove objects from screen
770  END
9990  '
9999  '
```

```
10000  '     ***** OBJECT 2 ARRIVAL SUBROUTINE *****
10020  '
10030  DEACTIVATE 2                                    ' Vanish object 2
10040  OBJECT 5,1=609,2=100,3=20,4=100,9=1,10=100' Set up object 5
10050  ACTIVATE 5                                      ' Make object 5 move
10060  RETURN
10070  '
11000  '     ***** OBJECT 3 ARRIVAL SUBROUTINE *****
11010  '
11020  X3 = OBJECT(3,1) + 100                  ' Move object 3 right 100 units
11030  ' Go up if direction flag is positive; go down otherwise
11040  IF DIRFLAG > 0 THEN Y3 = 20 ELSE Y3 = 180
11050  OBJECT 3,3=X3,4=Y3                      ' Set up next move for 3
11060  DIRFLAG = DIRFLAG * (-1)                ' Toggle direction flag
11070  RETURN
11080  '
12000  '     ***** OBJECT 4 ARRIVAL SUBROUTINE *****
12010  '
12020  DEACTIVATE 4                                    ' Vanish object 4
12030  X3 = 75: Y3 = 20                                ' Reset destination
12040  OBJECT 3,1=20,2=180,3=X3,4=Y3                   ' Set up object 3
12050  ACTIVATE 3                                      ' Send 3 on its way
12060  RETURN
12070  '
13000  '
13010  '     ***** OBJECT 5 ARRIVAL SUBROUTINE *****
13020  '
13030  DEACTIVATE 5                                    ' Vanish object 5
13040  OBJECT 2,1=20,2=100,3=609,4=100                 ' Set up object 2
13050  ACTIVATE 2                                      ' Send 2 on its way
13060  RETURN
13070  '
14000  '     ***** OBJECT 3 CLIP SUBROUTINE *****
14010  '
14020  DEACTIVATE 3                            ' Vanish object 3
14030  X4 = OBJECT(3,1)                        ' Initial X for 4 = last X for 3
14040  Y4 = OBJECT(3,2)                        ' Initial Y for 4 = last Y for 3
14050  ' Set new initial location for object 4
14060  OBJECT 4,1=X4,2=Y4,3=20,4=180,5=1,10=170
14070  ACTIVATE 4                  ' Send 4 on its way to home of 3
14080  RETURN
14090  '
15000  '     ***** OBJECT COLLISION SUBROUTINE *****
15010  '
16000  STOP OBJECT                             ' Freeze objects
16010  A = COLLISION(0)                        ' Get lowest object which collided
16020  B = COLLISION(A)                        ' Find who it collided with
16030  ' Calculate average x and y positions of collided objects
16030  XC=(OBJECT(A,1)+OBJECT(B,1))/2
16040  YC=(OBJECT(A,2)+OBJECT(B,2))/2
16050  ' Place object 6 at average position calculated above
```

continued on next page

```
16060  OBJECT 6,1=XC,2=YC,3=XC+1,4=YC,10=2,9=1   ' Move slowly
16070  COLLISION OFF: ACTIVATE 6                 ' Make it visible
16080  START OBJECT 6                            ' Start it moving
16090  WHILE (((ARRIVAL(-1))*ARRIVAL(6))=0): WEND ' Wait for it to arrive
16100  DEACTIVATE 6 :COLLISION ON                ' Make it disappear
16110  START OBJECT                              ' Restart the other objects
16120  RETURN
```

Section 4

Starting GW-BASIC

This section explains installing GW-BASIC, starting GW-BASIC, redirecting standard input and output, and exiting from GW-BASIC.

GW-BASIC is available on either a ROM cartridge or a diskette. This section describes the procedures for getting started with either the cartridge or the diskette version of GW-BASIC.

The Mindset GW-BASIC cartridge can work with or without MS-DOS. MS-DOS is not required to use BASIC files stored in NVRAM cartridges, but it is required to use disk files.

The MS-DOS diskette for the Mindset Personal Computer includes a program which links the GW-BASIC cartridge to MS-DOS when you enter the BASIC command line described in this section under "Starting GW-BASIC with MS-DOS".

Starting GW-BASIC Without MS-DOS

If you are using the cartridge version of GW-BASIC, you may run GW-BASIC without MS-DOS.

Read the *Mindset Personal Computer System Operation Guide* if you are unfamiliar with operating the Mindset configuration screen and using cartridges.

To use the GW-BASIC cartridge without MS-DOS, enter the Mindset configuration screen and make Cart 1 the first load priority. Next, place the Mindset GW-BASIC cartridge in port 1 and press the ALT-RESET keys to start GW-BASIC.

If you are using a RAM cartridge to store BASIC programs and data, you must insert the RAM cartridge in port 2 before pressing the ALT-RESET keys.

When you press the ALT-RESET keys, the screen shown in Figure 4-1 appears.

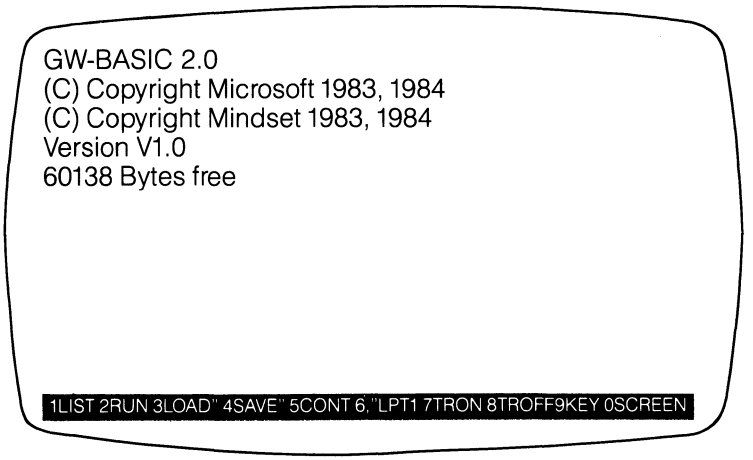


Figure 4-1: GW-BASIC start-up screen

The number of free bytes of memory displayed on this screen varies depending on the amount of memory installed in your system.

GW-BASIC Operating Environment Without MS-DOS

When using GW-BASIC without MS-DOS, you cannot specify any command switches or use BASIC statements that require MS-DOS such as CHDIR, SHELL, or SYSTEM. Using these statements will result in an error message.

When using GW-BASIC without MS-DOS, these parameters are in effect. The maximum number of open files for a RAM cartridge is six. The maximum record size for files on RAM cartridges is 128 bytes. The transmit buffer for each RS-232-C module is 256 bytes long and the receive buffer is 128 bytes long.

Starting GW-BASIC With MS-DOS

You may run GW-BASIC with MS-DOS using either the cartridge or diskette version of GW-BASIC.

To use the GW-BASIC cartridge with MS-DOS, enter the Mindset System configuration screen and make Disk the first load priority. Next, place the Mindset MS-DOS diskette in the left disk drive and place the Mindset GW-BASIC cartridge in cartridge port 1. Then press the ALT-RESET keys to load MS-DOS.

To start the diskette version of GW-BASIC, you must first load MS-DOS. Refer to the *Introductory Guide to MS-DOS* or the *MS-DOS Reference Manual* for instructions on how to load MS-DOS. After you load MS-DOS, place the GW-BASIC diskette in the default drive.

The procedures for running GW-BASIC from either a cartridge or a diskette are now the same. Both versions use the same command line options as explained below.

There are eight options you can specify when GW-BASIC is started from MS-DOS. These options, explained below, include device names, file names, and option switches. An option switch always begins with a forward slash (/) character.

You do not have to include any options to run most BASIC programs.

BASIC Command Line Syntax

The format of a GW-BASIC command line is:

```
BASIC [[ <stdin ] [ >stdout ]]
[<filename>]
[/C:<buffer size>]
[/F:<number of files>]
[/I]
[/M:[<highest memory location>] [,<maximum block size>]]
[/S: <lrecl>]
```

Command Line Options

Command line options refer to the parameters you can specify to configure GW-BASIC to meet the requirements of a particular application program.

<stdin

BASIC input is redirected to come from the file specified by stdin. When present, this syntax must appear before any switches.

>stdout

BASIC output is redirected to the file specified by stdout. When present, this syntax must appear before any switches.

File Options

To specify the filename of a BASIC program file, include this option:

filename

If <filename> is present, GW-BASIC proceeds as if a RUN <filename> command were given after GW-BASIC initialization is complete. If no extension is included with the filename, GW-BASIC uses .BAS as the extension.

This file specification enables you to run BASIC programs in batch by putting this form of the command line in a file with a .BAT extension. Programs run in this manner must exit with the SYSTEM statement (discussed later in this section) to allow the next command from the batch file to be executed.

Option Switches

You can use these option switches with the BASIC command.

/F: <number of files>

File number switch. This switch is ignored unless the /I switch, described below, is also specified on the command line.

If the /F and the /I switches are present, then the maximum number of files that may be open simultaneously during the execution of a BASIC program is set to <number of files>. Each file requires 62 bytes for the File Control Block (FCB), plus 128 bytes for the data buffer. The data buffer size may be altered with the /S: option switch, described below. If the /F: option is omitted, the number of files is set to 3.

The number of open files that MS-DOS supports depends upon the value of the FILES = parameter in the CONFIG.SYS file. For GW-BASIC, the parameter should be FILES = 10. Keep in mind that the first three are taken by stdin, stdout, stderr, stdaux, and stdprn. One additional file is needed by BASIC for LOAD, SAVE, CHAIN, NAME, and MERGE. This leaves six for BASIC file I/O; thus /F:6 is the maximum supported by MS-DOS when FILES = 10 appears in the CONFIG.SYS file.

Attempting to OPEN a file after all the file handles have been exhausted will result in a "Too many files" error.

/S: <lrecl>

Record size switch. This switch is ignored unless the /I switch, described below, is also specified on the command line.

If this switch and the /I switch are present, the maximum record size allowed for use with random files is set to <lrecl>.

Note: The record size option to the OPEN statement cannot exceed this value. If the /S: option is omitted, the record size defaults to 128 bytes.

/C: <buffer size>

Communications switch. If present, this switch controls RS-232-C communications. If RS-232-C modules are present, /C:0 disables RS-232-C support. Any subsequent I/O attempts will result in a "Device unavailable" error.

Specifying /C:<n> allocates space for communications buffers. GW-BASIC allocates <n> bytes for the receive buffer and 128 bytes for the transmit buffer for each RS-232-C module present. If the /C: option is omitted, GW-BASIC allocates 256 bytes for the receive buffer and 128 bytes for the transmit buffer of each module present. GW-BASIC ignores the /C: switch when RS-232-C modules are not present.

/I

Static file space allocation switch. GW-BASIC is able dynamically to allocate space required to support file operations. For this reason, GW-BASIC does not support the /S and /F switches.

However, certain applications have been written in such a manner that some BASIC internal data structures must be static. To provide compatibility with these BASIC programs, GW-BASIC will statically allocate space required for file operations based on the /S and /F switches when the /I switch is specified.

/M: [<highest memory location>] [,<maximum block size>]

Memory location switch. When present, this switch sets the highest memory location that will be used by BASIC. BASIC will attempt to allocate 64K of memory for the data and stack segments. If machine language subroutines are to be used with BASIC programs, use the /M: switch to set the highest location that BASIC can use. When omitted or 0, BASIC attempts to allocate all the memory it can, up to a maximum of 65536 bytes.

If you intend to load code or data above the highest location that BASIC can use, enter the optional parameter <maximum block size> to preserve space for them. This parameter is necessary if you intend to use the SHELL statement. Failure to use this parameter will result in COMMAND being loaded on top of your routines when a SHELL statement is executed.

<maximum block size> must be in paragraphs (byte multiples of 16). When omitted, &H1000 (4096) is assumed. This default size allocates 65536 bytes (65536 = 4096 x 16) for BASIC's data and stack segments. If you wanted 65536 bytes for BASIC and 512 bytes for machine language subroutines, then enter /M:,&H1010 (4096 paragraphs for BASIC plus 16 paragraphs for your routines).

The /M option can also be used to shrink the BASIC block to free more memory for SHELLing other programs. /M:,2048 says: "Allocate and use 32768 bytes maximum for data and stack". /M:32000,2048 allocates 32768 bytes maximum; BASIC will use only the lower 32000, leaving 768 bytes of extra memory.

Note that <number of files>, <lrecl>, <buffer size>, <highest memory location>, and <maximum block size> are numbers that may be decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

BASIC Command Examples

```
A>BASIC PAYROLL
```

Uses 64K of memory and 3 files to load and execute PAYROLL.BAS.

```
A>BASIC INVENT/F:6/I
```

Uses 64K of memory and 6 files to load and execute INVENT.BAS.

```
A>BASIC /C:0/M:32768
```

Disables RS-232-C support and uses only 32K of data memory. The 32K above that is free for the user.

```
A>BASIC /I/F:4/S:512
```

Uses 4 files and allows a maximum record length of 512 bytes.

```
A>BASIC TTY/C:512
```

Uses 64K of memory and 3 files, allocates 512 bytes to RS-232-C receive buffers and 128 bytes to transmit buffers, and loads and executes TTY.BAS.

Redirection of Standard Input and Standard Output

GW-BASIC can be redirected to read from standard input and write to standard output by providing the input and output filenames on the command line:

BASIC program_name [<input_file] [[>] >output_file]

Rules for Redirecting Input and Output

1. When redirected, all INPUT, LINE INPUT, INPUT$, and INKEY$ statements will read from input_file, instead of from the keyboard.

2. All PRINT statements will write to output_file instead of to the screen.

3. Error messages go to standard output.

4. File input from "KYBD:" still reads from the keyboard.

5. File output to "SCRN:" still directs output to the screen.

6. BASIC will continue to trap keys from the keyboard when the ON KEY (n) statement is used.

7. The printer echo key will not cause LPT1: echoing if standard output has been redirected.

8. Typing CTRL-BRK will cause GW-BASIC to close any open files, issue the message "Break in line <line_number>" to standard output, exit GW-BASIC, and return to MS-DOS.

9. When input is redirected, BASIC will continue to read from the new input source until a CTRL-Z is detected. This condition may be tested with the EOF function. If the file is not terminated by a CTRL-Z, or if a BASIC file input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and BASIC returns to MS-DOS.

Example of I/O Redirection

BASIC MYPROG >DATA.OUT

Data read by INPUT and LINE INPUT will continue to come from the keyboard. Data output by PRINT will go into the file DATA.OUT.

BASIC MYPROG <DATA.IN

Data read by INPUT and LINE INPUT will come from DATA.IN. Data output by PRINT will continue to go to the screen.

```
BASIC MYPROG <MYINPUT.DAT >MYOUTPUT.DAT
```

Data read by INPUT and LINE INPUT will now come from the file MYINPUT.DAT and data output by PRINT will go into MYOUTPUT.DAT.

```
BASIC MYPROG <\SALES\JOHN\TRANS.DAT
>>\SALES\SALES.DAT
```

Data read by INPUT and LINE INPUT will now come from the file \SALES\JOHN\TRANS.DAT. Data output by PRINT will be appended to the file \SALES\SALES.DAT.

Returning to MS-DOS from GW-BASIC

To exit from GW-BASIC, enter the command:

```
SYSTEM
```

This command closes all files and returns to MS-DOS if MS-DOS is available.

Section 5

Editing BASIC Programs

GW-BASIC provides three ways to enter and edit text: you can use the line editing capabilities, issue an EDIT command to place you in edit mode, or use the full screen editor.

Line Editing

If the cursor is currently on a line, you can make the following changes. If you are entering a line in response to an INPUT statement, you can use only the first two items in this list:

- Delete an incorrect character from the line that is being typed. The BACK SPACE key moves the cursor back one space and deletes a character. The DEL key erases the character under the cursor. Use the cursor movement keys to reach the character to be deleted.
- Delete the entire line that is being typed by pressing the ESC key. A carriage return is executed automatically after the line is deleted.
- Correct program lines for a program that is currently in memory by retyping the line, using the same line number. GW-BASIC will automatically replace the old line with the new one.
- Delete the entire program currently residing in memory by entering the NEW command. NEW is usually used to clear memory prior to entering a new program. See Section 6 for more information about the NEW command.

Edit Command

The EDIT command places the cursor on a specified line so that changes can be made to the line. See Section 6 for a description of the EDIT command.

Full Screen Editor

The full screen editor gives you immediate visual feedback, so that program text is entered in a "what you see is what you get" manner. You can enter program lines, then edit an entire screen before recording the changes. This time-saving capability is made possible by special keys for cursor movement, character insertion and deletion, and line or screen erasure. Specific functions and key assignments are discussed in the following sections.

With the full screen editor, you can move quickly around the screen, making corrections where necessary. The changes are entered by placing the cursor on the first line changed and pressing RETURN at the beginning of each line.

Writing Programs

You are using the full screen editor any time between the GW-BASIC "Ok" prompt and the execution of a RUN command. Every line of text that is entered is processed by the editor. Every line of text that begins with a number is considered a program statement.

It is possible to extend a logical line over more than one physical line by using the CTRL-RETURN sequence to produce a linefeed. A linefeed causes subsequent text to start on the next line, without a carriage return. A carriage return signals the end of the logical line; when a carriage return is entered, the entire logical line is passed to GW-BASIC.

Program statements are processed by the editor in one of the following ways:

- A new line is added to the program. Addition of a line occurs if the line number is valid (0 through 65529) and at least one non-blank character follows the line number.

- An existing line is replaced. Replacement of a line occurs if the line number matches that of an existing line in the program. The existing line is replaced with the text of the new line.

- An existing line is deleted. Deletion of a line occurs if the line number matches that of an existing line and the new line contains only the line number.

- The statements are passed to the command scanner for interpretation (that is, the line that has been entered contains no errors and is processed normally).

- An error is produced. If an attempt is made to delete a non- existent line, an "Undefined line" error message is displayed. If program memory is exhausted, and a line is added to the program, an "Out of memory" error is displayed and the line is not added.

More than one statement may be placed on a line. If this is done, the statements must be separated by a colon (:). The colon need not be surrounded by spaces.

The maximum number of characters allowed in a logical program line, including the line number, is 250.

Editing Programs

Use the LIST command to display an entire program or range of lines on the screen so that the lines can be edited with the full screen editor. Text can then be modified by moving the cursor to the place where the change is needed and then performing one of the following operations:

- Typing over existing characters
- Deleting characters to the right of the cursor
- Deleting characters to the left of the cursor
- Inserting characters
- Appending characters to the end of the logical line

These actions are performed by special keys assigned to the various full screen editor functions (see the next section, "Control Functions and Editor Keys").

Changes to a line are recorded when the RETURN key is pressed while the cursor is anywhere on that line. The RETURN key enters all changes for that logical line, no matter how many physical lines are part of the logical line and no matter where the cursor is located within the logical line.

As lines are being re-entered to record changes, the cursor may occasionally be positioned on a line that contains a message issued by GW-BASIC, such as "Ok". When this happens, the line is automatically erased. (GW-BASIC recognizes its own messages because they are terminated by FF Hex to distinguish them from user text.) This is provided as a courtesy to the programmer. If GW-BASIC did not erase

the line and the RETURN key was pressed, the message would be passed to GW-BASIC as a direct mode statement and a syntax error would result.

Control Functions and Editor Keys

Table 5-1 lists the hexadecimal codes for the GW-BASIC control characters and summarizes their functions. The control-key sequence normally assigned to each function is also listed. These sequences conform as closely as possible to ASCII standard conversions.

Table 5-1: GW-BASIC Control Functions

Hex Code	Decimal Code	Key	Function
02	02	B	Move cursor to start of previous word
05	05	E	Truncate line (clear text to end of logical line)
06	06	F	Move cursor to start of next word
0B	11	K	Move cursor to home position
0C	12	L	Clear screen
0E	14	N	Move to end of line
12	18	R	Toggle insert/typeover mode
1C	28	\	Cursor right
1D	29	]	Cursor left
1E	30	^	Cursor up
1F	31	_	Cursor down (underscore)
7F	32	DEL	Delete character at cursor

Enter the ASCII control function by pressing the key listed while holding down the CTRL key.

The CHR$ function may be used to perform these control functions. For example, printing CHR$(30) moves the cursor up one line.

BASIC Editor Function Keys

The following keys and key sequences perform the indicated editor functions.

Previous word—CTRL-◀ moves the cursor left to the previous word. A word is made up of the characters A-Z, a-z, or 0-9 and is delimited by a space.

Truncate—CTRL-END deletes characters from the cursor position to the end of the logical line.

Next word—CTRL-▶ moves the cursor right to the next word. A word is made up of the characters A-Z, a-z, or 0-9, and is delimited by a space.

Linefeed—CTRL-RETURN moves the cursor to the next line but does not insert a carriage return, so that the logical line can be continued.

The linefeed occupies one space on a line, always to the right of the last character typed on a physical line. If there is no room on the physical line, the linefeed is wrapped. The linefeed can be deleted, but it cannot be typed over.

Cursor home—The HOME key moves the cursor to the upper left corner of the screen. The screen is not blanked.

Clear window—CTRL-HOME moves the cursor to home position and clears the entire window, regardless of where the cursor is positioned when the key is pressed.

Carriage return—The RETURN key places a carriage return at the current cursor position. A carriage return ends the logical line and sends it to GW-BASIC.

Append—The END key moves the cursor to the end of the line, without deleting the characters passed over. All characters typed from the new position until the RETURN key is pressed are appended to the logical line.

Insert—The INS key is a toggle switch for insert mode. When insert mode is on, characters are inserted at the current cursor position. Characters to the right of the cursor move right as new ones are inserted. Line wrap is observed.

When insert mode is off, typed characters will overstrike existing characters on the line.

Clear logical line—Pressing the ESC key anywhere in the line causes the entire logical line to be erased.

▶ —The ▶ moves the cursor one position to the right. Line wrap is observed.

◀ —The ◀ key moves the cursor one position to the left. Line wrap is observed.

▲—The ▲ key moves the cursor up one physical line in the current column.

▼—The ▼ key moves the cursor down one physical line in the current column.

Logical Line Definition with INPUT

Normally, a logical line consists of all the characters on each of the physical lines that make up the logical line. During execution of an INPUT or LINE INPUT statement, however, this definition is modified slightly to allow for forms input. When either of these statements is executed, the logical line is restricted to characters actually typed or passed over by the cursor.

The insert and delete modes move only characters that are within that logical line. Delete mode decrements the size of the line.

Insert mode increments the logical line except when the characters moved will write over non-blank characters that are on the same physical line, but not part of the logical line. In this case, the non-blank characters not part of the logical line are preserved and the characters at the end of the logical line are thrown out. This preserves labels that existed prior to the INPUT statement.

Editing Lines Containing Variables

When a syntax error is encountered during program execution, GW-BASIC enters edit mode at the line where the error occurred. You can correct the error, enter the change, and re-execute the program. If the corrected line contains variables, however, the values of these variables will be lost when the line is re-entered. If you wish to examine the contents of the variables before making the syntax correction, pressing the BREAK key or moving the cursor will return to direct mode without destroying any of the variables. Then the variables can be printed before the line is edited and the program re-executed.

Section 6

GW-BASIC Commands, Statements, Functions, and Variables

This section describes the GW-BASIC commands, statements, functions, and variables in alphabetical order. Briefly, these elements can be defined as:

Command: An instruction that returns control to the user after the instruction has been performed. For example, LIST and MERGE are commands.

Statement: An instruction that is entered as part of a program source line. For example, LET and LINE are statements.

Function: A function converts a value into some other value according to a fixed formula. The functions described in this section are built-in functions, or "intrinsic" to GW-BASIC. These functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the syntax given for the functions in this section, the arguments have been abbreviated as follows:

Argument	**Meaning**
X and Y	Represent numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions
n	Represents an integer value

If a floating-point value is supplied where an integer is required, GW-BASIC rounds the fractional portion and uses the resulting integer.

See Appendix B for information about mathematic functions that are not intrinsic to GW-BASIC.

Variable: The variables described in this section are "system variables". The names of these system variables are reserved words just as are the names of statements, commands, and functions. GW-BASIC stores information in these variables for use in a BASIC program. An example of a system variable is the TIME$ string variable which always contains the time from the system's time-of-day clock.

Each description in this section is formatted as follows:

The first line indicates the name of the instruction or function.

PURPOSE:
Tells what the instruction or function is used for.

SYNTAX:
Shows the correct syntax for the instruction or function. See the introduction to this manual for syntax notation.

When the term "filespec" is used as an option in the syntax, it refers to a combination of device name, filename, and extension in the correct format for the operating system.

NOTES:
Describe in detail how the instruction or function is used.

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

ABS FUNCTION

PURPOSE:
Returns the absolute value of the expression X.

SYNTAX:
Y = ABS(X)

EXAMPLE:
The function:

```
PRINT ABS(7*(-5))
```

will yield:

```
35
```

ACTIVATE/DEACTIVATE STATEMENTS

PURPOSE:
ACTIVATE starts the motion of one or more objects. DEACTIVATE stops the motion of one or more objects and removes them from the screen.

SYNTAX:
ACTIVATE [<object number 1>[,<object number 2>...]]
DEACTIVATE [<object number 1>[,<object number 2>...]]

<object number 1> and <object number 2> are the numbers of the objects being set into motion by the ACTIVATE statement or being stopped and removed from the screen by the DEACTIVATE statement.

NOTES:
The objects specified in an ACTIVATE statement must be previously defined with a DEF OBJECT statement.

If no object numbers are specified, ACTIVATE sets all objects into motion and DEACTIVATE stops and removes all objects from the screen.

See also the START OBJECT and STOP OBJECT statements in this section.

EXAMPLES:
The statement:

```
10 ACTIVATE 7,2,5
```

sets objects 7, 2, and 5 into motion. The statement:

```
20 DEACTIVATE
```

stops the motion of all objects on the screen and removes them from view.

ARRIVAL FUNCTION

PURPOSE:
Returns the arrival status of an object.

SYNTAX:
X = ARRIVAL(−1)
X = ARRIVAL(0)
X = ARRIVAL(<object number>)

<object number> is the number of a specific object to be tested for possible arrival at its destination.

NOTES:
The ARRIVAL function enables a program to test for the arrival of an object at its destination. The operation of the ARRIVAL function is similar to that of the CLIP and COLLISION functions.

The ARRIVAL function always returns 0 if the ARRIVAL ON statement has not been executed.

When an object arrives at the destination set for it in an OBJECT statement, GW-BASIC sets an internal arrival flag for that object. The ARRIVAL function tests the arrival flag of one or more objects depending on which form of the function is used.

The ARRIVAL(−1) form of the ARRIVAL function latches the current state of the arrival flags for all objects which have arrived since the last time the arrival flags were latched. If any of the arrival flags for active objects are set, then the ARRIVAL(−1) function returns −1, makes a copy of the affected arrival flags, and clears the arrival flags so that they can be set again. If no object has arrived at its destination since the last time the arrival status was latched, the ARRIVAL(−1) function returns 0.

After using the ARRIVAL(−1) function, the ARRIVAL(0) function can be used to return the lowest numeric object which has the copy of its arrival flag set. After returning this number, GW-BASIC clears the copy of the arrival flag. The ARRIVAL(0) form of the ARRIVAL function returns 0 if the copies of all arrival flags set by the ARRIVAL(−1) function have been cleared by a corresponding number of ARRIVAL(0) functions.

The third form of the ARRIVAL function, ARRIVAL (<object number>), is also used after using the ARRIVAL(−1) function to make copies of all currently set arrival flags. This form of the function returns the object number if the copy of that object's arrival flag is set (it also clears the copy of the flag). Otherwise, the ARRIVAL (<object number>) function returns a value of 0.

The ON ARRIVAL statement automatically latches the arrival flags and makes copies for testing with the ARRIVAL(0) and ARRIVAL(<object number>) functions before performing a GOSUB. Thus, the ARRIVAL(−1) function is not required before testing the copies of the arrival flags within such a subroutine.

EXAMPLES:
The following program uses line 100 to latch the arrival flags for all objects. Line 110 obtains the number of the lowest numbered object which has arrived at its destination. Line 120 checks the copies of the arrival flags to see if any are still set. Line 130 selects the subroutine corresponding to the object number returned in line 110.

If both objects 1 and 2 have arrived since the last time their flags were latched, then line 110 places a value of 1 in X and clears the copy of the arrival flag for object 1. After executing the subroutine at line 1000 and returning to line 110, the ARRIVAL(0) function places a value of 2 in X because the arrival flag for object 1 has been cleared. Line 130 then transfers control to the subroutine at line 2000.

```
100 IF ARRIVAL(−1) = 0 THEN 150
110 X = ARRIVAL(0)
120 IF X = 0 THEN 150
130 ON X GOSUB 1000,2000
140 GOTO 110
150 'Remainder of main program
.
.
.
990 END
1000 'Subroutine for arrival of object 1
.
.
.
1990 RETURN
2000 'Subroutine for arrival of object 2
.
.
.
2990 RETURN
```

Alternately, the ON ARRIVAL statement could be used to accomplish the same results as follows (see the ARRIVAL statement for more information):

```
100 ON ARRIVAL(1) GOSUB 1000
110 ON ARRIVAL(2) GOSUB 2000
120 ARRIVAL ON
130 'Remainder of main program
.
.
.
990 END
1000 'Subroutine for arrival of object 1
.
.
.
1990 RETURN
2000 'Subroutine for arrival of object 2
.
.
.
2990 RETURN
```

ARRIVAL STATEMENT

PURPOSE:
The ARRIVAL ON statement enables arrival testing. An arrival occurs when an object arrives at the destination specified in an OBJECT statement.

The ARRIVAL OFF statement disables arrival testing.

The ARRIVAL STOP statement suspends arrival testing until the next ARRIVAL ON statement is executed.

SYNTAX:
ARRIVAL ON
ARRIVAL OFF
ARRIVAL STOP

NOTES:
The ARRIVAL ON statement enables arrival testing by an ON ARRIVAL statement (see "ON ARRIVAL Statement" later in this section) as well as polling of the arrival status using the ARRIVAL function. While the ARRIVAL ON statement is in effect, and if a non-zero line number has been specified in the ON ARRIVAL statement, GW-BASIC checks after the execution of every statement to see if any of the specified arrival events has occurred.

The ARRIVAL OFF statement disables arrival testing by an ON ARRIVAL statement. An arrival event that occurs after an ARRIVAL OFF statement has been executed is not discovered.

The ARRIVAL STOP statement suspends arrival testing. An arrival event that occurs after the execution of this statement is remembered and trapped after the next ARRIVAL ON statement is executed.

EXAMPLES:
The statement:

```
10 ARRIVAL ON
```

enables arrival testing as specified in one or more ON ARRIVAL statements. The statement:

```
20 ARRIVAL OFF
```

disables arrival testing. The statement:

```
30 ARRIVAL STOP
```

suspends arrival testing. Arrival events which occur after this ARRIVAL STOP statement are trapped after the next ARRIVAL ON statement is executed.

ASC FUNCTION

PURPOSE:
Returns a numeric value that is the ASCII code for the first character of the string X$. (See Appendix C, "ASCII Character Codes", for ASCII codes.)

SYNTAX:
Y = ASC(X$)

NOTES:
If X$ is null, an "Illegal function call" error is returned.

EXAMPLE:
The function:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
```

will yield:

```
84
```

See "CHR$ function" later in this section for details on ASCII-to-string conversion.

ATN FUNCTION

PURPOSE:
Returns the arctangent of X, where X is in radians. The result is in the range $-pi/2$ to $pi/2$ radians.

SYNTAX:
Y = ATN(X)

NOTES:
The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

EXAMPLE:
The function:

```
10 INPUT X
20 PRINT ATN(X)
```

will yield:

```
? 3
1.249046
```

AUTO COMMAND

PURPOSE:
Generates line numbers automatically during program entry.

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

NOTES:
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 existing line and generate the next line number.

If the cursor is moved to another line on the screen, numbering will resume there.

Automatic line numbering is terminated by pressing the BREAK key.

EXAMPLE:

```
AUTO 100,50
```

generates line numbers 100, 150, 200....

```
AUTO
```

generates line numbers 10, 20, 30, 40....

BEEP STATEMENT

PURPOSE:
Sounds the speaker at 800 Hz for 1/4 second.

SYNTAX:
BEEP

NOTES:
The BEEP statement sounds the ASCII bell code. This statement has the same effect as PRINT CHR$(7).

EXAMPLE:

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

This example executes a beep when X is less than 20.

BLOAD STATEMENT

PURPOSE:
Loads a specified memory image file into memory from disk or RAM cartridge.

SYNTAX:
BLOAD <filespec> [,<offset>]

The device designation portion of the filespec is optional. The filename may be from 1 to 8 characters long.

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

NOTES:
The BLOAD statement allows a program or a block of data that has been saved as a memory image file to be loaded anywhere in memory. A memory image file is a byte-for-byte copy of what was originally in memory. See "BSAVE Statement" in this section for information about saving memory image files.

If the offset is omitted, the segment address and offset contained in the file (that is, the address specified by the BSAVE statement when the file was created) are used. Therefore, the file is loaded into the same location from which it was saved.

If offset is specified, the segment address used is the one given in the most recently executed DEF SEG statement. If no DEF SEG statement has been given, the GW-BASIC data segment will be used as the default (because it is the default for DEF SEG).

BLOAD can be used with VARPTR to initialize an array (such as a view of an animated object) with data from disk.

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 operating system.

EXAMPLE:

```
10 'Load subroutine at 600:F000
20 DEF SEG = &H600 'Set segment to 600 Hex
30 BLOAD"PROG1",&HF000 'Load PROG1
```

This example sets the segment address at 600 Hex and loads PROG1 at F000.

BSAVE STATEMENT

PURPOSE:
Saves the contents of the specified area of memory as a disk or RAM cartridge file.

SYNTAX:
BSAVE <filespec>,<offset>,<length>

The device designation portion of the filespec is optional. The filename may be from 1 to 8 characters long.

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535. BSAVE begins storing the file contents at the offset address 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 integer is the length in bytes of the memory image file to be saved.

NOTES:
The BSAVE statement enables data or programs to be saved as memory image files on disk. A memory image file is a byte-for-byte copy of what is in memory.

The <filespec>, <offset>, and <length> are required in the syntax.

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

If length is omitted, a "Syntax error" is issued and the save is aborted.

BSAVE can be used with VARPTR to write an array (such as a view of an animated object) to disk. See the description of the GET statement, which includes a formula for calculating the required array size for graphic objects.

EXAMPLE:

```
10 'Save PROG1
20 DEF SEG = &H6000
30 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes, starting at 6000:F000 in the file PROG1.

CALL STATEMENT

PURPOSE:
Calls an external assembly language subroutine or a compiled routine written in another high-level language.

SYNTAX:
CALL <variable name>[(<argument list>)]

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name.

<argument list> contains the arguments passed to the external subroutine. <argument list> may contain only variables.

NOTES:
The CALL statement transfers program flow to an external subroutine. (See also "USR function" later in this section.) CALL passes unsegmented addresses.

The CALL statement generates the same calling sequence used by the Microsoft BASIC compiler (see "Assembly Language Subroutines" in Section 2).

EXAMPLE:

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
 .
 .
 .
```

CALLS STATEMENT

PURPOSE:
Calls an external assembly language subroutine or a compiled routine written in another high-level language.

SYNTAX:
CALLS <variable name>[(<argument list>)]

NOTES:
The CALLS statement works just like CALL, except that the segmented addresses of all arguments are passed. (CALL passes unsegmented addresses.) Because all FORTRAN parameters are call-by-refererice segmented addresses, CALLS should be used when accessing routines written with FORTRAN calling conventions.

To locate the routine being called, CALLS uses the segment address defined by the most recently executed DEF SEG statement.

CDBL FUNCTION

PURPOSE:
Converts X to a double precision number.

SYNTAX:
Y = CDBL(X)

NOTES:
X can be either an integer or a single precision number.

EXAMPLE:

```
10 A=454.67
20 PRINT A; CDBL (A)
```

will yield:

```
 454.67  454.6700134277344
```

CHAIN STATEMENT

PURPOSE:
Calls a program and pass variables to it from the current program.

SYNTAX:
CHAIN [MERGE]<filespec>[,[<line number exp>]
[,ALL][,DELETE <range>]]

See the examples below for descriptions of the syntax options.

NOTES:
<filespec> is the file specification of the program called.

The COMMON statement may be used to pass variables (see "COMMON Statement" later in this section).

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

Before running the CHAINed program, CHAIN performs a RESTORE to reset the pointer to DATA statements. READ, therefore, does not continue where it left off in the new program.

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.

If the ALL option is used and <line number exp> is not, a comma must hold the place of <line number expr>.

For example,

```
CHAIN "NEXTPROG",,ALL
```

is correct;

```
CHAIN "NEXTPROG",ALL
```

is incorrect. In the latter case, GW-BASIC assumes that ALL is a variable name and evaluates it as a line number expression.

continued on next page

The MERGE option allows a subroutine to be brought into the GW-BASIC program as an overlay. That is, the current program and the called program are merged (see "MERGE Command" later in this section). The called program must be an ASCII file if it is to be merged.

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

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

The CHAIN statement with the MERGE option leaves the files open and preserves the current OPTION BASE setting. A CHAINed program may have an OPTION BASE statement if no arrays are passed.

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 DEFFN statements containing shared variables must be restated in the CHAINed program.

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.

EXAMPLES:
Example 1:

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING COM-
MON 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 "PROG1",90
90 END
```

Example 3:

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING THE
MERGE, ALL, AND DELETE OPTIONS.
20 REM SAVE THIS MODULE ON THE DISK AS "MAINPRG".
30 A$ = "MAINPRG"
40 CHAIN MERGE "OVRLAY1",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, DELETE 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
```

CHDIR STATEMENT

PURPOSE:
To change the current MS-DOS directory.

SYNTAX:
CHDIR <pathname>

<pathname> is a string expression not exceeding 128 characters identifying the new directory which is to be the current directory.

NOTES:
CHDIR works exactly like the DOS command CHDIR.

EXAMPLES:
The following CHDIR statements make SALES the current directory on drive A, and INVENTORY the current directory on drive B:

```
CHDIR "SALES" ' SALES is now the current directory on drive A:

CHDIR "B:INVENTORY" ' INVENTORY is now the current directory on
drive B:
```

Subsequent usage of [A:] <filespec> will refer to files in the directory SALES. Usage of B:<filespec> will refer to files in the directory B:\INVENTORY.

Now that the current directory is SALES, MKDIR can be used to create subdirectories JOHN and MARY.

```
MKDIR "JOHN" 'Create subdirectory \SALES\JOHN
MKDIR "MARY" 'Create subdirectory \SALES\MARY
```

An alternate method to create the JOHN directory under the SALES directory is as follows:

```
MKDIR "\SALES\JOHN"
```

Then use CHDIR to make JOHN the current directory:

```
CHDIR "JOHN"
```

or:

```
CHDIR "\SALES\JOHN"
```

CHR$ FUNCTION

PURPOSE:
Returns a string corresponding to the ASCII character number entered as an argument. (ASCII codes are listed in Appendix C, "ASCII Character Codes".)

SYNTAX:
X$ = CHR$(I)

NOTES:
CHR$ is commonly used to send a special character. For instance, the BELL character (CHR$(7)) could be sent as a preface to an error message; also, a form feed (CHR$(12)) could be sent to clear the screen and return the cursor to the home position.

EXAMPLE:
The function:

```
PRINT CHR$(66)
```

will yield:

```
B
```

See "ASC function" earlier in this section for details on ASCII-to-numeric conversion.

CINT FUNCTION

PURPOSE:
Converts X to an integer by rounding the fractional portion.

SYNTAX:
Y = CINT(X)

NOTES:
If X is not in the range −32768 to 32767, an "Overflow" error occurs.

See the CDBL and CSNG functions in this section for details on converting numbers to the double precision and single precision data types, respectively. See also the FIX and INT functions, both of which return integers.

EXAMPLE:
The function:

```
PRINT CINT(45.67)
```

will yield:

```
 46
```

CIRCLE STATEMENT

PURPOSE:
Draws an ellipse with the center and radius specified.

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

<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 radius of the circle in pixels.

<color> is the numeric symbol for the color desired (see "COLOR Statement" later in this section). The default color is the foreground color.

<start> and <end> are the start and end angles in radians with the range -2π to 2π. These angles allow the user to specify where an 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 from adding 2 π. The start angle may be less than the end angle.

<aspect> is the aspect ratio; that is, the ratio of the x radius to the y radius. When default ratios are specified for the corresponding screen mode, a circle is drawn. The default aspect ratios are 5/6 for 320 $\times$ 200 resolution screen mode and 5/12 for 640 $\times$ 200 resolution screen mode. These default aspect ratios produce circles on display screens with a standard aspect ratio of 4/3.

If the aspect ratio is less than one, the radius given is the x radius. If it is greater than one, the y radius is given.

The CF option will cause the ellipse to be filled, just as the BF option causes the box to be filled in the LINE statement. (See the LINE statement later in this section.)

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

Clipping occurs when a circle includes points outside the screen or viewport limits. These points do not appear.

continued on next page

Coordinates can be shown in one of two ways: as absolutes, as in the syntax shown above; or relatively, using the STEP option to reference a point relative to the most recent point used. The syntax of the STEP option replaces (<xcenter>,<ycenter>) with:

STEP (<xoffset>,<yoffset>)

For example, if the most recent point referenced was (0,0), STEP (10,5) would reference a point at offset 10 from x and offset 5 from y.

EXAMPLE:
Assume that the last point plotted was 100,50. Then:

```
CIRCLE (200,200),50
```

and:

```
CIRCLE STEP (100,150),50
```

will both draw a circle at 200,200 with radius 50. The first example uses absolute notation; the second uses relative notation.

CLEAR STATEMENT

PURPOSE:
Sets all numeric variables to zero and all string variables to null, and closes all open files; also, optionally, sets the end of memory and the amount of stack space. The CLEAR statement also cancels the definition of all animation objects visible on the screen.

SYNTAX:
CLEAR [,[<expression1>][,<expression2>]]

NOTES:
The CLEAR statement performs the following actions:

- Closes all files.
- Clears all COMMON variables.
- Resets numeric variables and arrays to zero.
- Resets the stack and string space.
- Resets all string variables and arrays to null.
- Releases all disk buffers.
- Resets all DEF FN and DEF/SNG/DBL/STR statements.

<expression1> is a memory location that, if specified, sets the highest location available for use by GW-BASIC.

<expression2> sets aside stack space for GW-BASIC. The default is 768 bytes or one-eighth of the available memory, whichever is smaller.

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.

The CLEAR statement also affects visible objects defined with the DEF OBJECT statement. Any object which has been located on the screen with an OBJECT statement and made visible with an ACTIVATE statement is stopped and removed from the screen by the CLEAR statement. In other words, the CLEAR statement performs an implicit DEACTIVATE statement.

EXAMPLES:

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

CLIP FUNCTION

PURPOSE:
Returns the clipping status of an object.

SYNTAX:
X = CLIP(−1)
X = CLIP(0)
X = CLIP(<object number>)

<object number> is the number of a specific object to be tested for possible clipping at the boundary of the current window.

NOTES:
The CLIP function enables a program to test for the clipping of an object at the boundary of the current window. The operation of the CLIP function is similar to that of the ARRIVAL and COLLISION functions.

The CLIP function always returns 0 if the CLIP ON statement has not been executed.

When an object exceeds the boundary of the current window, GW-BASIC sets an internal clip flag for that object. The CLIP function tests the clip flag of one or more objects depending on which form of the function is used.

The CLIP(−1) form of the CLIP function latches the current state of the clip flags for all objects which have been clipped since the last time the clip flags were latched. If any of the clip flags for active objects are set, then the CLIP(−1) function returns −1, makes a copy of the affected clip flags, and clears the clip flags so that they can be set again. If no object has exceeded the boundary of the current window since the last time the clip status was latched, the CLIP(−1) function returns 0.

After using the CLIP(−1) function, the CLIP(0) function can be used to return the lowest numeric object which has the copy of its clip flag set. After returning this number, GW-BASIC clears the copy of the clip flag. The CLIP(0) form of the CLIP function returns 0 if the copies of all clip flags set by the CLIP(−1) function have been cleared by a corresponding number of CLIP(0) functions.

The third form of the CLIP function, CLIP(<object number>), is also used after using the CLIP(−1) function to make copies of all currently set clip flags. This form of the function returns the object number if the copy of that object's clip flag is set (it also clears the copy of the flag). Otherwise, the CLIP (<object number>) function returns a value of 0.

The ON CLIP statement automatically latches the clip flags and makes copies for testing with the CLIP(0) and CLIP(<object number>) functions before performing a GOSUB. Thus, the CLIP(−1) function is not required before testing the copies of the CLIP flags within such a subroutine.

Also see the CLIP statement for more information on testing for the clipping of objects.

EXAMPLES:
The following program uses line 100 to latch the clip flags for all objects. Line 110 sets a FOR-NEXT loop to test the copies of the clip flags of all 16 possible objects. Line 120 tests the copy of the clip flag for object K. Line 130 causes the program to test the next object's clip flag if the clip flag for object K was not set since the last time line 100 was executed. Line 140 transfers control to the subroutine associated with object K. The K in line 140 could be replaced with N because they will both contain the same number. Line 160 terminates the FOR-NEXT loop.

```
100 IF CLIP (−1) = 0 THEN 170
110 FOR K = 1 TO 16
120 N = CLIP(K)
130 IF N = 0 THEN 160
140 ON K GOSUB 1000,2000
150 GOSUB 3000
160 NEXT K
170 'Remainder of main program
.
.
.
990 END
1000 'Subroutine for clipping of object 1
.
.
.
1990 RETURN
2000 'Subroutine for clipping of object 2
.
.
.
2990 RETURN
3000 'Subroutine for clipping of all objects
.
.
.
3990 RETURN
```

CLIP STATEMENT

PURPOSE:
The CLIP ON statement enables clip testing. Clipping occurs when an object exceeds the boundary of the current window.

The CLIP OFF statement disables clipping testing.

The CLIP STOP statement suspends clipping testing until the next CLIP ON statement is executed.

SYNTAX:
CLIP ON
CLIP OFF
CLIP STOP

NOTES:
The CLIP ON statement enables clip testing by an ON CLIP statement (see "ON CLIP Statement" later in this section) as well as polling of the clipping status using the CLIP function. While the CLIP ON statement is in effect, and if a non-zero line number has been specified in the ON CLIP statement, GW-BASIC checks after the execution of every statement to see if any of the specified clipping events has occurred.

The CLIP OFF statement disables the testing for clipping by an ON CLIP statement. Clipping that occurs after a CLIP OFF statement has been executed is not discovered.

The CLIP STOP statement suspends clip testing. Clipping that occurs after this statement is remembered and trapped after the next CLIP ON statement is executed.

EXAMPLES:
The statement:

```
10 CLIP ON
```

enables clip testing as specified in one or more ON CLIP statements. The statement:

```
20 CLIP OFF
```

disables clip testing. The statement:

```
30 CLIP STOP
```

suspends clip testing. Clipping events that occur after this CLIP STOP statement are trapped after the next CLIP ON statement is executed.

CLOSE STATEMENT

PURPOSE:
Concludes I/O to a file. The CLOSE statement complements the OPEN statement.

SYNTAX:
CLOSE [[#]<file number>[,[#]<file number>...]]

NOTES:
<file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files.

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 for a sequential output file writes the final buffer of output.

The END statement and the NEW command also close all disk files automatically. (STOP does not close disk files.)

EXAMPLES:

```
CLOSE #1,#2
CLOSE 1,2
```

CLS STATEMENT

PURPOSE:
Erases the contents of the entire current screen.

SYNTAX:
CLS

NOTES:
The screen may also be cleared with the control sequence CTRL-L.

The function key display on line 25 remains visible if it is on the screen before execution of the CLS statement.

EXAMPLE:

```
10 CLS 'Clears the screen
```

COLLISION FUNCTION

PURPOSE:
Returns object collision status.

SYNTAX:
X = COLLISION(−1)
X = COLLISION(0)
X = COLLISION(<object number>)
X = COLLISION(<object number 1>,<object number 2>)

<object number> is the number of a specific object to be tested for a possible collision with another object.

NOTES:
The COLLISION function enables a program to test for the collision of one object with another. The operation of the COLLISION function is similar to that of the ARRIVAL and CLIP functions.

When an object collides with another, GW-BASIC sets an internal collision flag for that object. The COLLISION function tests the collision flag of one or more objects depending on which form of the function is used.

The COLLISION(−1) form of the COLLISION function latches the current state of the collision flags for all objects which have collided since the last time the collision flags were latched. If any of the collision flags for active objects are set, then the COLLISION(−1) function returns −1, makes a copy of the affected collision flags, and clears the collision flags so that they can be set again. If no object has collided since the last time the collision status was latched, the COLLISION(−1) function returns 0.

After using the COLLISION(−1) function, the COLLISION(0) function can be used to return the lowest numeric object which has the copy of its collision flag set. After returning this number, GW-BASIC clears the copy of the collision flag. The COLLISION(0) form of the COLLISION function returns 0 if the copies of all COLLISION flags set by the COLLISION(−1) function have been cleared by a corresponding number of COLLISION(0) functions.

After using the COLLISION (−1) function to make copies of all currently set COLLISION flags, the third form of the COLLISION function, COLLISION (<object number>) can be used. This form of the function returns the value of the lowest-numbered object which collided with <object number> (it also clears the flag for this collision pair). This form of the function returns 0 if no object collided with <object number>.

continued on next page

The fourth form of the COLLISION function, COLLISION(<object number>,<object number>), is also used after using the COLLISION(−1) function to make copies of all currently set COLLISION flags. This form of the function returns the second object number if the copy of that object pair's COLLISION flag is set (it also clears the copy of the flag). Otherwise, the COLLISION(<object number>,<object number>) function returns a value of 0.

If the COLLISION ON statement is in effect, the ON COLLISION statement automatically latches the collision flags and makes copies for testing with the COLLISION(0) and COLLISION(<object number>) functions. In this case, the COLLISION(−1) function is not required before testing the copies of the COLLISION flags.

See the COLLISION statement for more information on testing for object collisions.

EXAMPLES:
The following program uses line 100 to latch the COLLISION flags for all objects and to transfer control to the subroutine at line 1000. Line 1000 sets N equal to the number of the lowest-numbered object involved in a collision since the last time the collision status was latched. Line 1000 also clears the copy of the collision flag for object N. Line 1010 returns control to the main program if the copies of the collision flags for all objects are cleared. Line 1040 prints the numbers of the objects which have collided. After line 1030 returns the program to line 1000, the subroutine prints the number of the next object which has collided.

```
100 ON COLLISION GOSUB 1000
110 'Remainder of main program
.
.
.
990 END
1000 N = COLLISION(0)
1010 IF N = 0 THEN RETURN
1020 M = COLLISION(N)
1030 IF M = 0 THEN 1000
1040 PRINT "Objects" N "and" M "have collided"
1050 GOTO 1020
```

If objects 2 and 15 and objects 10 and 5 have collided since the last time the collision status was latched, this program would print the following messages:

```
Objects 2 and 15 have collided
Objects 5 and 10 have collided
```

The statement:

```
10 X = COLLISION(3)
```

sets X to the number of the lowest-numbered object which collided with object number 3. This example sets X to zero if no collisions with object 3 have occurred. The statement:

```
20 X = COLLISION(0)
```

sets X to the number of the lowest-numbered object involved in a collision since the COLLISION function was last executed. If no collisions have occurred, X is set to 0. The statement:

```
30 X = COLLISION(-1)
```

latches the current collision information. This statement sets X to −1 if at least one collision has occurred. Otherwise this statement sets X to 0. The statement:

```
40 X = COLLISION(3,4)
```

sets X to 4 if a collision between objects 3 and 4 has occurred and sets X to 0 if no collision has occurred.

COLLISION STATEMENT

PURPOSE:
The COLLISION ON statement enables object collision testing.

The COLLISION OFF statement disables the testing for object collisions.

The COLLISION STOP statement suspends the testing for object collisions.

SYNTAX:
COLLISION ON
COLLISION OFF
COLLISION STOP

NOTES:
The COLLISION ON statement enables collision testing by an ON COLLISION statement (see "ON COLLISION Statement" later in this section) as well as polling of the collision status using the COLLISION function.. While the COLLISION ON statement is in effect, and if a non-zero line number has been specified in the ON COLLISION statement, GW-BASIC checks after the execution of every statement to see if any specified collisions have occurred.

The COLLISION OFF statement disables the testing for collisions. Collisions which occur after a COLLISION OFF statement is executed are not discovered.

The COLLISION STOP statement suspends the testing for collisions. Collisions are remembered and trapped after the next COLLISION ON statement is executed.

EXAMPLES:
The statement:

```
10 COLLISION ON
```

enables collision testing. The statement:

```
20 COLLISION OFF
```

disables collision testing. The statement:

```
30 COLLISION STOP
```

suspends collision testing. Collision events which occur after a COLLISION STOP statement are trapped after the next COLLISION ON statement is executed.

COLOR STATEMENT (Text)

PURPOSE:
Selects from the color palette the colors for the foreground text, background, and screen border.

SYNTAX:
COLOR [<foreground>][,[<background>][,<border>]]

<foreground> is the color for text. <background> is the color for the background behind the text. <border> is the color of the border at the edge of the display area.

NOTES:
The numbers used to specify the <foreground>, <background>, and <border> colors refer to locations in the color palette of the Mindset Personal Computer. A given color index value may not always produce the same color because the colors in the palette can be reassigned using the PALETTE and PALETTE USING statements. Table 6-1 describes the default colors in the color palette.

Table 6-1: Default Palette Colors

Value	Color	Value	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	Off White	15	White

The value of the <foreground> parameter for the text mode must be in the range 0 to 31. Adding 16 to the color values causes characters to blink.

The range of the <background> parameter must be in the range 0 to 7. The range of the <border> parameter must be in the range 0 to 15.

You can specify a maximum of two different colors for text mode.

continued on next page

EXAMPLE:
The statement:

```
10COLOR  0,10,10
```

selects black characters on a light green background and border (provided that the color palette has not been altered by either the PALETTE or PALETTE USING statements).

COLOR STATEMENT (Graphics)

PURPOSE:
Selects one of 15 colors from the palette as the background color and one of two color groups as the foreground colors. See the PALETTE statement later in this section for a discussion of the colors in the palette.

SYNTAX:
COLOR [<background>][,[<color group>]]

<background> is the color index for the background. <color group> selects one of two foreground color groups.

Table 6-2 describes the default colors for each color group associated with the color values you select in graphic output statements such as CIRCLE, LINE, and DRAW.

Table 6-2: Graphics Color Palettes

Color Value	Color Group 0	Color Group 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

NOTES:
The value for the <background> parameter may range from 0 to 15. This value specifies an index into the palette to obtain the background color. The background color is also the color produced by using a color value of 0 in graphics statements. If the <background> parameter is omitted, the COLOR statement assumes the previous value for the background color.

The palette of the Mindset Personal Computer may be reprogrammed with different colors with the PALETTE and PALETTE USING statements. If the color palette has not been reprogrammed, the colors will be those shown in Table 6-1 under the description of the COLOR statement for text mode.

continued on next page

The value of the <color group> parameter is either even or odd. Even values for this parameter select color group 0 and odd values select color group 1.

The number of colors available depends on the current screen mode. Modes 2, 3, and 7 can use only colors 0 and 1 from the selected graphics palette. Modes 1, 5, and 6 can use colors 0, 1, 2, and 3. Mode 4 can specify any of the 16 colors from the palette.

EXAMPLE:
The program:

```
10 SCREEN 5
20 COLOR 10,1
```

places the system in screen mode 5 for 640 by 200 resolution, selects light green for the background and selects palette 1 so that color values 0, 1, 2, and 3 represent light green, cyan, magenta, and white, respectively.

COM STATEMENT

PURPOSE:
Enables or disables event trapping of communications activity on the specified channel.

SYNTAX:
COM(n) ON
COM(n) OFF
COM(n) STOP

(n) is the number of the communications channel. The range for (n) is 1 to 3.

NOTES:
The COM(n) ON statement enables communications event trapping by an ON COM statement (see "ON COM Statement" later in this section). While trapping is enabled, if a non-zero line number is specified in the ON COM statement, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, the ON COM statement is executed.

COM(n) OFF disables communications event trapping. If an event takes place, it is not remembered.

COM(n) STOP disables communications event trapping. However, if an event occurs, it is remembered and ON COM will be executed as soon as trapping is enabled.

EXAMPLE:

```
10 COM(1) ON
```

Enables error trapping of communications activity on channel 1.

COMMON STATEMENT

PURPOSE:
Passes variables to a CHAINed program.

SYNTAX:
COMMON <list of variables>

NOTES:
The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though they should normally 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.

Some versions of BASIC allow the number of dimensions in the array to be included in the COMMON statement. GW-BASIC will accept that syntax, but will ignore the numeric expression itself. For example, in GW-BASIC the following statements are both valid and are considered equivalent:

```
COMMON A()
COMMON A(3)
```

The number in parentheses is the number of dimensions in the array, not the dimensions themselves. For example, the variable A(3) in this example might correspond to a DIM statement of DIM A(5,8,4).

EXAMPLE:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
.
.
.
```

CONT COMMAND

PURPOSE:
Continues program execution after a CTRL-C has been typed or a STOP or END statement has been executed.

SYNTAX:
CONT

NOTES:
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:
See "STOP Statement" later in this section.

COS FUNCTION

PURPOSE:
Returns the cosine of X, where X is in radians.

SYNTAX:
Y = COS(X)

NOTES:
The calculation of COS(X) is performed in single precision.

EXAMPLE:
The function:

```
10 X=2*COS(.4)
20 PRINT X
```

will yield:

```
1.842122
```

CSNG FUNCTION

PURPOSE:
Converts X to a single precision floating-point number.

SYNTAX:
Y = CSNG(X)

NOTES:
See the CINT and CDBL functions earlier in this section for information about converting numbers to the integer and double precision data types, respectively.

EXAMPLE:
The function:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
```

will yield:

```
 975.3421  975.3421
```

CSRLIN VARIABLE

PURPOSE:
Stores the current line position of the cursor in a numeric variable.

NOTES:
To return the current column position of the cursor, use the POS function.

EXAMPLE:

```
10 Y = CSRLIN 'Record current line.
20 X = POS(0) 'Record current column.
30 LOCATE 24,1 :PRINT "HELLO"
'Print HELLO on last line.
40 LOCATE Y, X 'Restore position to old line and column.
```

CVI, CVS, and **CVD** FUNCTIONS

PURPOSE:
Converts string values to numeric values.

SYNTAX:
Y = CVI(<2-byte string>)
Y = CVS(<4-byte string>)
Y = CVD(<8-byte string>)

NOTES:
Numeric values that are read in from a random disk 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 also "MKI$, MKS$, and MKD$ Functions" later in this section.

EXAMPLE:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

DATA STATEMENT

PURPOSE:
Stores the numeric and string constants that are accessed by the program's READ statement(s).

SYNTAX:
DATA <list of constants>

NOTES:
DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants (separated by commas) as will fit on a logical line. 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.

<list of constants> may contain either numeric or string constants. Numeric constants may be in any format; that is, fixed-point, floating-point, or integer constants. (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" later in this section.

DATE$ STATEMENT

PURPOSE:
Sets the current date. This statement complements the DATE$ variable, which retrieves the current date.

SYNTAX:
DATE$ = <string expression>

<string expression> returns a string in one of the following forms:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

EXAMPLE:

```
10 DATE$ = "11-08-1983"
```

The current date is set at November 8, 1983.

DATE$ VARIABLE

PURPOSE:
Retrieves the current date. (To set the date, use the DATE$ statement, described previously.)

NOTES:
The DATE$ variable contains a ten-character string in the form mm-dd-yyyy, where mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).

EXAMPLE:

```
10 PRINT DATE$
```

This statement prints the date, calculated from the date set with the DATE$ statement.

DEF FN STATEMENT

PURPOSE:
Defines and names a function written by the user.

SYNTAX:
DEF FN<name>[(<parameter list>)] = <function definition>

NOTES:
<name> must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<parameter list> consists of those 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.

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.

<function definition> is an expression that performs the operation of the function. It is limited to one logical line. 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 variable 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 DEF FN 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 encountered 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.

DEFINT/SNG/DBL/STR STATEMENTS

PURPOSE:
Declares variable types as integer, single precision, double precision, or string.

SYNTAX:
DEF<type> <range(s) of letters>

where <type> is INT, SNG, DBL, or STR.

NOTES:
Any variable names beginning with the letter(s) specified in <range(s) of letters> will be considered the type of variable specified in the <type> portion of the statement. However, a type declaration character always takes precedence over a DEFtype statement. (See "Variable Names and Declaration Characters" in Section 2.)

If no type declaration statements are encountered, GW-BASIC assumes that all variables without declaration characters are single precision variables.

EXAMPLES:
In the statement:

```
10 DEFDBL L-P
```

all variables beginning with the letters L, M, N, O, and P will be double precision variables. In the statement:

```
10 DEFSTR A
```

all variables beginning with the letter A will be string variables. In the statement:

```
10 DEFINT I-N,W-Z
```

all variables beginning with the letters I, J, K, L, M, N, W, X, Y, and Z will be integer variables.

DEF OBJECT STATEMENT

PURPOSE:
Defines an animation object as a group of arrays where each array contains a view of the object. The sequence of the arrays determines the order in which the views of that object appear.

SYNTAX:
DEF OBJECT <object number>[AS<array 1>[,<array 2>...]]

<object number> specifies the number for the object being defined.

<array 1>, <array 2> ... contain successive views of the object being defined.

NOTES:
Up to 8 views can be specified for an object. GW-BASIC displays the views in the order of their definition.

The <object number> of an object defines its priority. Objects are drawn in the order of their object numbers, with the lowest-numbered object drawn first. Object numbers must be in the range 1-16.

The DEF OBJECT statement sets all attributes for the specified object to their default values (see the description of the OBJECT statement later in this section). This statement returns an error if the specified object is already defined. An object can be undefined by using the DEF OBJECT statement without including any array names.

The DEF OBJECT statement automatically creates a working array for storing the screen image which is overlaid by the object. The size of this array is equal to that of the largest object view in the group defined by DEF OBJECT. The user cannot access this array, but should be aware of the additional memory that it requires.

EXAMPLES:
The statement:

```
10 DEF OBJECT 3 AS MAN1,MAN2,MAN3
```

defines the MAN1, MAN2, and MAN3 arrays as the different views of object 3. The statement also sets up an array to preserve the screen image overlaid by object 3. The statement:

```
10 DEF OBJECT 3
```

cancels the definition of object 3 as an animated object. Arrays MAN1, MAN2, and MAN3 remain unaltered by this statement.

DEF SEG STATEMENT

PURPOSE:
Assigns the current segment address that will be referenced by a subsequent BLOAD, BSAVE, CALL, CALLS, or POKE statement or by a USR or PEEK function.

SYNTAX:
DEF SEG [=<address>]

where <address> is a numeric expression returning an unsigned integer in the range 0 to 65535.

NOTES:
The address specified is saved for use as the segment required by BLOAD, BSAVE, CALL, CALLS, POKE, USR, and PEEK.

Entry of any value outside the <address> range 0 through 65535 will result in an "Illegal function call" error, and the previous value will be retained.

If the <address> option is omitted, the segment to be used is set to the GW-BASIC data segment (DS). This data segment value is the initial default value.

If the <address> option is given, it should be based on a 16-byte boundary. GW-BASIC does not check the validity of the specified address.

Note that DEF and SEG must be separated by a space. Otherwise, GW-BASIC will interpret the statement DEFSEG=100 to mean "assign the value 100 to the variable DEFSEG".

EXAMPLE:

```
10 DEF SEG = &HB800 'Set segment to B800 Hex
20 DEF SEG 'Restore segment to GW-BASIC data segment
```

DEF USR STATEMENT

PURPOSE:
Specifies the starting address of an assembly language subroutine.

SYNTAX:
DEF USR[<digit>] = <integer expression>

NOTES:
<digit> may be 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.

The value of <integer expression> is the starting address of the USR routine.

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:

```
.
.
.
200 DEF USR0 = 24000
210 X = USR0(Y^2/2.89)
.
.
.
```

DELETE COMMAND

PURPOSE:
Deletes program lines.

SYNTAX:
DELETE [<line number>][– <line number>]

NOTES:
GW-BASIC always returns to the command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

EXAMPLES:
The command:

```
DELETE 40
```

deletes line 40 only. The command:

```
DELETE 40 – 100
```

deletes lines 40 through 100, inclusive. The command:

```
DELETE  – 40
```

deletes all lines up to and including line 40. The command:

```
DELETE 40 –
```

deletes all lines from 40 to the end of the program.

DIM OBJECT STATEMENT

PURPOSE:
Dimensions an array large enough to contain the display data for a graphic object.

SYNTAX:
DIM OBJECT <array name>(<x size>,<y size>)

<array name> is the name of the numeric array to contain the display data for the object. <x size> and <y size> are the horizontal and vertical object dimensions, respectively. The object dimensions are specified in logical units defined by the current window.

NOTES:
The DIM OBJECT statement produces a single-dimensioned array to contain a view of an object of the specified size based on the screen mode at the time the statement was executed.

Each array produced by the DIM OBJECT statement is large enough to contain the required header information (as with the arrays used in the GET# and PUT# statements) and one copy of the object view. The DIM OBJECT statement automatically determines the number of elements in the array according to the array type (integer or floating-point array).

EXAMPLE:
The statement:

```
10 DIM OBJECT MAN(10,20)
```

creates an array named MAN which can contain an object 20 logical units high and 10 logical units wide. Unless the DEFINT statement has defined M as an integer, the MAN array defaults to the floating-point type.

DIM STATEMENT

PURPOSE:
Specifies the maximum values for array variable subscripts and allocates storage accordingly.

SYNTAX:
DIM <list of subscripted variables>

NOTES:
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 greater than the specified maximum is used, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (described later in this section).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

In theory, the maximum number of dimensions allowed in a DIM statement is 255. In reality, however, that number is impossible, because the statement and variable names and punctuation also count as spaces on the line, and the line itself has a limit of 255 characters. The number of dimensions is further limited by the amount of available memory.

EXAMPLE:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

DRAW STATEMENT

PURPOSE:
Draws a line as indicated by the subcommands described below.

SYNTAX:
DRAW <string expression>

<string expression> contains one or more of the subcommands described below.

NOTES:
The DRAW statement contains 17 subcommands. These subcommands combine many of the capabilities of the other GW-BASIC graphics statements into an "object definition language" called Graphics Macro Language (GML). An object definition language is a language that defines a complete set of characteristics of a particular object. In this case, the characteristics are motion (up, down, left, right), color, angle, and scale factor.

Each of the following GML commands begins movement from the "current graphics position". This position is usually the coordinate of the last graphics point plotted with another GML command, the LINE statement, or the PSET statement. The current position defaults to the center of the screen when a program is run.

The following GML commands move one unit if no argument is supplied.

Command	**Explanation**
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 down and right
G [<n>]	Move diagonally down and left
H [<n>]	Move diagonally up and left
M <x,y>	Move absolute or relative. If x is preceded by a plus (+) or minus (−) sign, 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.

continued on next page

The following prefix commands may precede any of the above movement commands:

Command	Explanation
B	Move but do not plot points.
N	Move but return to original position when done.
A <n>	Set angle n. n may range from 0 to 3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° are scaled so they will appear the same size as with 0 or 180° on a screen with the standard aspect ratio of 4/3.
TA <n>	Set turn angle n. <n> may range from −360° to 360°. If <n> is positive, rotation is counterclockwise. If <n> is negative, rotation is clockwise.
C <n>	Set color attribute n. n may range from 0 to 3 in screen modes 1, 5, and 6; 0 to 1 in modes 2, 3 and 7; and 0 to 15 in mode 4.
S <n>	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 expression>	Execute substring. This powerful command allows you to execute a second substring from a string, much like a GOSUB statement. You can have one string execute another, which executes a third, and so on.
P <paint color>, <border color>	Set object color to <paint color> and the border color to <border color>. <paint color> and <border color> may range from 0 to 3 in screen modes 1, 5, and 6; 0 to 1 in modes 2, 3 and 7; and 0 to 15 in mode 4.

"Tile" painting is not supported in DRAW.

Numeric arguments can be constants like "123" or "=<variable>", where <variable> is the name of a variable.

EXAMPLES:

```
10 U$ = "U30;" : D$ = "D30;" : L$ = "L40;" R$ = "R40;"
20 BOX$ = U$ + R$ + D$ + L$
30 DRAW "XBOX$;"
40 REM DRAW "XU$;XR$;XD$;XL$;" would have drawn the same box

10 FOR D = 0 TO 360 'Draw some spokes
20 DRAW "TA = D;NU50"
30 NEXT D

10 DRAW "U50R50D50L50" 'Draw a box
20 DRAW "BE10" 'Move up and right into box
30 DRAW "P1,3" 'Paint interior
```

EDIT COMMAND

PURPOSE:
Enters edit mode at the specified line.

SYNTAX:
EDIT <line number>

NOTES:
In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, GW-BASIC types the number of the line to be edited and waits for you to use the editor to make any necessary changes.

END STATEMENT

PURPOSE:
Terminates program execution, closes all files, and returns to command level.

SYNTAX:
END

NOTES:
END statements may be placed anywhere in the program to terminate execution. An END statement at the end of a program is optional. GW-BASIC always returns to command level after an END is executed.

Unlike the STOP statement, END does not cause a "Break in line nnnnn" message to be printed.

EXAMPLE:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

ENVIRON STATEMENT

PURPOSE:
Modifies parameters in BASIC's environment string table. Parameter modification may be to change the "PATH" parameter for a Child process (see the "ENVIRON$" function and the "SHELL" statement in this section; also see "The MS-DOS 2.0 Utilities - PATH command" in the Mindset MS-DOS Manual). This statement can also be used to pass parameters to a Child process by inventing a new environment parameter.

SYNTAX:
ENVIRON <parm>

<parm> is a valid string expression containing the new environment string parameter.

NOTES:
The following rules apply to the ENVIRON statement:

1. <parm> must be in the form <parm_id> = <text> where:

   a. <parm_id> is the name of the parameter, such as "PATH".

   b. <parm_id> must be separated from <text> by an equal sign or a blank, such as "PATH = " or "PATH ". ENVIRON takes everything left of the first blank or equal sign as the <parm_id>. The first character after <parm_id> which is not an equal sign or a blank is taken as <text>.

   c. <text> is the new parameter text. If <text> is a null string, or consists only of ";" (a single semicolon, such as "PATH = ;"), the parameter (including <parm_id> =) is removed from the environment string table and the table is compressed.

2. If <parm_id> does not exist, then <parm> is added to the end of the environment string table.

3. If <parm_id> already exists, it is deleted, the environment string table is compressed, and new <parm> is added to the end of the table.

EXAMPLES:
Unless changed by the MS-DOS command PATH, the BASIC environment string table is empty. The following statement will create a default "PATH" to the root directory on Disk A:

```
PATH = A:\
```

A new parameter may be added as follows:

```
ENVIRON "CAT=DOG"
```

The environment string table now contains:

```
PATH=A:\;CAT=DOG
```

The PATH may be changed to a new value by the following statement:

```
ENVIRON "PATH=A:\SALES;A:\ACCOUNTING"
```

The environment string table now contains:

```
CAT=DOG;PATH=A:\SALES;A:\ACCOUNTING
```

The PATH parameter may be appended to the environment string table by using the ENVIRON$ function with the ENVIRON statement:

```
ENVIRON "PATH=" + ENVIRON$("PATH") + ";B:\DOG"
```

The environment path parameters are now:

```
CAT=DOG;PATH=A:\SALES;A:\ACCOUNTING;B:\DOG
```

Finally, the parameter CAT may be deleted this way:

```
ENVIRON CAT=;
```

The environment string table now contains:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\DOG
```

ENVIRON$ FUNCTION

PURPOSE:
Retrieves the specified environment string from BASIC's environment string table.

SYNTAX:
X$ = ENVIRON$ (<parm>)

or:

X$ = ENVIRON$ (<nth parm>)

<parm> is a valid string expression containing the parameter to search for.

<nth parm> is an integer expression returning a value in the range 1 to 255.

NOTES:
The following rules apply to the ENVIRON$ function:

1. If a string argument is used, ENVIRON$ returns a string containing the text following <parm> = from the environment string table.

2. If <parm> = is not found, or if no text follows <parm> =, then a null string is returned.

3. If a numeric argument is used, ENVIRON$ returns a string containing the <nth parm> from the environment string table, including the <parm> = text.

4. If there is no <nth parm>, a null string is returned.

EXAMPLES:
Unless changed by the DOS command PATH, the initial BASIC environment string is empty (see "The ENVIRON Statement" earlier in this section). The ENVIRON statement example would cause the environment string table to contain:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\DOG
```

In this case, the statement:

```
PRINT ENVIRON$("PATH")
```

prints the string:

```
A:\SALES;A:\ACCOUNTING;B:\DOG
```

Also, the statement:

```
PRINT ENVIRON$(1)
```

prints the string:

```
PATH=A:\SALES;A:\ACCOUNTING;B:\DOG
```

The following program saves the BASIC environment string table in an array so it may be modified for a Child process. After the execution of the Child process is complete, the environment is restored.

```
10 DIM ENV.TBL$(10) 'Assume no more than 10 parms
20 N.PARMS = 1 'init number of parms
30 WHILE LEN(ENVIRON$(N.PARMS)) > 0
40 ENV.TBL$(N.PARMS) = ENVIRON$(N.PARMS)
50 N.PARMS = N.PARMS + 1
60 WEND
70 N.PARMS = N.PARMS-1 'adjust to correct number
80 'Now store new environment
90 ENVIRON "MYCHILD.PARM1 = SORT BY NAME"
100 ENVIRON "MYCHILD.PARM2 = LIST BY NAME"
          .
          .
          .
1000 SHELL "MYCHILD" 'Runs "MYCHILD.EXE"
1010 FOR I = 1 TO N.PARMS
1020 ENVIRON ENV.TBL$(I) 'Restore parms
1030 NEXT I
          .
          .
          .
```

EOF FUNCTION

PURPOSE:
Tests for the end-of-file condition.

SYNTAX:
X = EOF(<file number>)

NOTES:
Returns −1 (true) if the end of a sequential file has been reached. To avoid "Input past end" errors, use EOF to test for end-of-file while inputting.

EXAMPLE:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT#1,M(C)
50 C=C+1:GOTO 30
 .
 .
 .
```

ERASE STATEMENT

PURPOSE:
Eliminates arrays from memory.

SYNTAX:
ERASE <list of array variables>

NOTES:
After an array is erased, either a new DIM statement must be used to re-create the array, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to revise the dimension of an array without first erasing it, a "Duplicate definition" error occurs.

Make sure that you do not erase any arrays which are part of a current DEF OBJECT statement.

EXAMPLE:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

ERDEV and **ERDEV$** VARIABLES

PURPOSE:
When a device error routine is entered, the variable ERDEV contains the error code of the device error. Similarly, if the device causing the error is a character device, the variable ERDEV$ contains the name of the device.

The handler for interrupt X'24' sets ERDEV (and ERDEV$ if the device causing the error is a character device).

ERDEV will contain the INT 24 error code in the lower 8 bits, and the upper 8 bits will contain the "word attribute bits" (b15-b13) from the device header block.

If the error was on a character device, ERDEV$ will contain the 8-byte character device name. If the error was not on a character device, ERDEV$ will contain the two-character block device name (A:, B:, C:, etc.).

NOTES:
These variables are read-only variables which you cannot alter with a BASIC assignment statement.

EXAMPLE:
User-installed device driver "MYLPT2" caused a "Printer out of Paper" error via INT 24.

ERDEV contains the error number 9 in the lower 8 bits and the device header word attributes in the upper 8 bits.

ERDEV$ contains "MYLPT2"

ERR and **ERL** VARIABLES

PURPOSE:
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 65535 = ERL 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. GW-BASIC error codes are listed in Appendix A, "Error Codes and Error Messages".

NOTES:
These variables are read-only variables that you cannot alter with a BASIC assignment statement.

ERROR STATEMENT

PURPOSE:
Simulates the occurrence of a BASIC error, or enables the user to define error codes.

SYNTAX:
ERROR <integer expression>

NOTES:
ERROR can be used as a statement (part of a program source line) or as a command (in direct mode).

The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

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. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, GW-BASIC responds with the "Unprintable error" error message. Execution of an ERROR statement for which there is no error-handling routine causes an error message to be printed and execution to halt.

EXAMPLES:
The following example:

```
10 S=10
20 T=5
30 ERROR S+T
40 END
```

will yield:

```
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15
String too long
Ok
```

The following example illustrates a user-defined error code:

```
.
.
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
.
.
.
```

EXP FUNCTION

PURPOSE:
Returns e (base of natural logarithms) to the power of X. X must be <= 88.02969.

SYNTAX:
Y = EXP(X)

NOTES:
If X is greater than 88.02969, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXAMPLE:
The function:

```
10 X=5
20 PRINT EXP(X-1)
```

will yield:

```
54.59815
```

FIELD STATEMENT

PURPOSE:
Allocates space for variables in a random file buffer.

SYNTAX:
FIELD [#]<file number>,<field width> AS <string variable>...

NOTES:
Before a GET statement or a PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

<file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>.

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 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note: Do not use a fielded variable name in an INPUT or LET statement. After 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 from the random file buffer to string space.

EXAMPLES:
In Example 1, the statement:

```
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 also the descriptions of the GET, LSET, and RSET statements, elsewhere in this section.)

continued on next page

In Example 2, the following statements:

```
10 OPEN "R",#1,"A:PHONELST",35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAME$,10 AS PHONENBR$
25 GET #1
30 TOTAL = CVI(RECNBR$)
35 FOR I = 2 TO TOTAL
40 GET #1, I
45 PRINT NAME$, PHONENBR$
50 NEXT I
```

illustrate a multiply defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. For the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

In Example 3, the following statements:

```
10 FOR LOOP% = 0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS
A$(LOOP%)
30 NEXT LOOP%
```

show the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0),16 AS A$(1),...,16 AS A$(6),16 AS A$(7)
```

In Example 4, the following statements:

```
10 DIM SIZE% (4): REM ARRAY OF FIELD SIZES
20 FOR LOOP% = 0 TO 4:READ SIZE%
(LOOP%): NEXT LOOP%
30 DATA 9,10,12,21,41
.
.
.
120 DIM A$(4): REM ARRAY OF FIELDED VARIABLES
130 OFFSET% = 0
140 FOR LOOP% = 0 TO 4
150 FIELD #1,OFFSET% AS OFFSET$,SIZE%(LOOP%)
AS A$(LOOP%)
160 OFFSET% = OFFSET% + SIZE%(LOOP%)
170 NEXT LOOP%
```

create a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4) AS A$(4)
```

FILES STATEMENT

PURPOSE:
Prints the names of files residing on the specified disk or RAM cartridge.

SYNTAX:
FILES [<filespec>]

<filespec> includes a filename and an optional device designation.

NOTES:
If <filespec> is omitted, all the files on the currently selected drive will be listed. <filespec> is a string formula which may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks.

FILES also allows a pathname in place of <filespec>. The directory for the specified path is displayed. If an explicit path is not given, the current directory is assumed. Sub-directory names are displayed with "<DIR>" following the directory name.

If you start GW-BASIC directly from the cartridge, rather than from MS-DOS, the FILES statement operates only with RAM cartridge directories and pathnames are not valid.

RAM cartridges, like disks, must be formatted before data can be stored on them. Previously formatted RAM cartridges can be formatted to delete all data on the cartridge.

A special <filespec> is used with the FILES statement to format RAM cartridges. The following statement formats a RAM cartridge in cartridge port 1:

```
FILES "CART1:/FORMAT"
```

Use the <filespec> string "CART2:/FORMAT" to format a RAM cartridge in cartridge port 2. This special <filespec> can also be used with the KILL and NAME statements to format a RAM cartridge.

EXAMPLES:

```
FILES
```

shows all files on currently logged disk.

```
FILES "*.BAS"
```

shows all files with extension .BAS.

```
FILES "B:*.*" or FILES "B:"
```

shows all files on drive B.

```
FILES "TEST?.BAS"
```

shows all five-letter files whose names start with "TEST" and end with the .BAS extension.

```
FILES "CART1:*.*"
```

shows all files on the RAM cartridge in cartridge port 1.

```
FILES "\SALES"
```

displays the directory entry SALES. If SALES is a valid directory, this statement displays the list of files and subdirectories in SALES.

```
FILES "\SALES\MARY\"
```

displays all files in MARY's directory.

FIX FUNCTION

PURPOSE:
Returns the truncated integer part of X.

SYNTAX:
Y = FIX(X)

NOTES:
FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The difference between FIX and INT is that FIX does not return the next lower number for negative X.

EXAMPLES:
The function:

```
PRINT FIX(58.75)
```

will yield:

```
58
```

The function:

```
PRINT FIX(-58.75)
```

will yield:

```
-58
```

FOR...NEXT STATEMENT

PURPOSE:
Enables a series of instructions to be performed in a loop a given number of times.

SYNTAX:
FOR <variable> = X TO Y [STEP Z]

.
.
.

NEXT [<variable>][,<variable>...]

X, Y, and Z are numeric expressions.

NOTES:
<variable> is used as a counter. The first numeric expression (X) is the initial value of the counter. The second numeric expression (Y) is the final value of the counter.

FOR...NEXT Loops

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is adjusted by the amount (Z) 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, GW-BASIC loops back to the statement after the FOR statement and the process is repeated. If the value of the counter is greater, execution continues with the statement following the NEXT statement.

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 decreased at each pass through the loop. The loop is executed until the counter is less than the final value.

The counter must be an integer or single precision numeric variable. Using a double precision numeric constant results in a "Type mismatch" error.

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.

continued on next page

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 the NEXT statement 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.

EXAMPLES:
In 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
```

will yield:

```
1 20
3 30
5 40
7 50
9 60
```

In Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

the loop does not execute because the initial value of the loop exceeds the final value.

In Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
```

will yield:

```
1 2 3 4 5 6 7 8 9 10
```

The loop executes ten times. The final value for the loop variable is always set before the initial value is set.

FRE FUNCTION

PURPOSE:
Returns the number of free bytes. With a numeric argument, FRE returns the number of bytes in memory that are not being used by GW-BASIC. Arguments to FRE are dummy arguments.

SYNTAX:
X = FRE(0)
X = FRE("")

NOTES:
FRE("") forces a garbage collection before returning the number of free bytes. Garbage collection may take as long as 1 to 1-1/2 minutes.

GW-BASIC does not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically results in shorter delays for each garbage collection.

EXAMPLE:
```
PRINT FRE(0)
```

prints the number of bytes free for BASIC statements.

GET STATEMENT (FILES)

PURPOSE:
Reads a record from a random disk file into a random buffer.

SYNTAX:
GET [#]<file number>[,<record number>]

NOTES:
<file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215. The smallest possible record number is 1.

The GET and PUT statements allow fixed-length input and output for GW-BASIC communications (COM) files. However, because of the low performance associated with telephone line communications, GET and PUT statements are not recommended for telephone communication.

After a GET statement has been executed, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

EXAMPLE:

```
GET #1,75
```

will place record number 75 from the file OPENed as #1 into the random buffer.

GET STATEMENT (GRAPHICS)

PURPOSE:
Used with the PUT statement to transfer graphic images (objects) to and from the screen and to retrieve graphic images for animation.

SYNTAX:
GET (X1,Y1) – (X2,Y2),<array name>

(X1,Y1) – (X2,Y2) is a rectangle which contains the object on the display screen. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ,B option.

<array name> is the name assigned to the array that will hold the image. The array can be any type except a string array. The array dimension must be large enough to hold the entire image. Unless the array is type integer, however, the contents of the array after a GET will be meaningless when interpreted directly.

NOTES:
The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. Later, a corresponding PUT statement can transfer the image stored in the array back onto the screen. Additionally, these objects can be used for animation.

The number of words required for an array depends on the number of bits per pixel in the current screen mode and on the size of the rectangle containing the image. If you use the DIM OBJECT statement described previously in this section to create an array for an object, the array will be sized properly for the GET statement. Otherwise, use these formulas to calculate the number of words required for an object array:

Modes 2, 3, and 7:	INT((X + 15)/16) * Y + 2
Modes 1, 5, and 6:	INT((X * 2 + 15)/16) * Y + 2
Mode 4:	INT((X * 4 + 15)/16) * Y + 2

In these formulas, X is the number of horizontal pixels and Y is the number of vertical pixels in the rectangle defined by the coordinates (X1,Y1) – (X2,Y2). The additional two words in the formulas hold the X and Y dimensions of the rectangle containing the object.

The actual value used in the DIM statement for an array depends on the type of the array. Integer arrays hold 1 word per element, single precision arrays hold 2 words per element, and double precision arrays hold 4 words per element.

EXAMPLE:

```
10 DIM FIGURE1 (44)
20 GET (10,10) – (30,30), FIGURE1
```

will record the image contained in the rectangle bounded by diagonal corners (10,10) and (30,30) and store it in the array named FIGURE1. (Note that each dimension of this rectangle is 21 pixels, not 20.) To determine the dimension required for the FIGURE1 array, this example assumes that the screen mode is 3 and that the array type is integer.

GOSUB...RETURN STATEMENTS

PURPOSE:
Branches to and returns from a subroutine.

SYNTAX:
GOSUB <line number>
.
.
.
RETURN [<line number>]

NOTES:
<line number> in the GOSUB statement is the first line of the subroutine.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

Simple RETURN statement(s) in a subroutine cause GW-BASIC to branch back to the statement following the most recent GOSUB statement. If logic dictates a return at different points in the subroutine, a subroutine may contain more than one RETURN statement.

The <line number> option may be included in the RETURN statement to return to a specific line number from the subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result.

Subroutines may appear anywhere in the program; however, each subroutine should be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

EXAMPLE:
The statements:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
```

will yield:

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

GOTO STATEMENT

PURPOSE:
Branches unconditionally out of the normal program sequence to a specified line number.

SYNTAX:
GOTO <line number>

NOTES:
If <line number> is an executable statement, that statement and those following are executed. If <line number> is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

EXAMPLE:

```
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
```

will yield:

```
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
Out of data in 10
```

HEX$ FUNCTION

PURPOSE:
Returns a string that represents the hexadecimal value of the decimal argument.

SYNTAX:
Y$ = HEX$(X)

NOTES:
X is rounded to an integer before HEX$(X) is evaluated.

EXAMPLE:
The function:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS    " A$ "    HEXADECIMAL"
```

will yield:

```
? 32
32 DECIMAL IS 20 HEXADECIMAL
```

For details on octal conversion, see the OCT$ function, later in this section.

IF...THEN[...ELSE]/IF...GOTO STATEMENTS

PURPOSE:
Makes a decision regarding program flow based on the result returned by an expression.

SYNTAX 1:
IF <expression> THEN { <statement(s)>|<line number>}
[ELSE { <statement(s)>|<line number>}]

SYNTAX 2:
IF <expression> GOTO <line number>
[ELSE { <statement(s)>|<line number>}]

NOTES:
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.

Nested 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 direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Floating-Point Computations

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.

EXAMPLES:
In Example 1:

```
200 IF I THEN GET#1,I
```

this statement GETs record number I from file number 1 if I is not zero.

In Example 2:

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
.
.
.
```

a test determines if I is greater than 10 and less than 20. If I is within this range, DB is calculated and execution branches to line 300. If I is not within this range, execution continues with line 110.

In Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

this statement causes printed output to go either to the screen or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

INKEY$ FUNCTION

PURPOSE:
Returns a string containing a character read from the keyboard.

SYNTAX:
X$ = INKEY$

NOTES:
The INKEY$ function does not echo characters to the screen. All characters are passed through to the program except for the BREAK key, CTRL-BREAK, and CTRL-C, which terminate the program.

The INKEY$ function returns the next character from the keyboard buffer. The string returned by INKEY$ is 0, 1, or 2 characters long. A null string is returned if there are no characters waiting to be read from the keyboard buffer.

INKEY$ returns a string containing a single character for keys with ASCII character codes (see Appendix C for a list of the ASCII character codes). A two-character string is returned for keys with extended codes. The first character is code 000, and the second character is the extended code for that key. (See Appendix C for a list of extended character codes.)

EXAMPLE:
This routine uses a FOR-NEXT loop to limit the amount of time allowed to answer a prompt:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$ = ""
1020 FOR I% = 1 TO TIMELIMIT%
1030 A$ = INKEY$ : IF LEN(A$) = 0 THEN 1060
1040 IF ASC(A$) = 13 THEN TIMEOUT% = 0 : RETURN
1050 RESPONSE$ = RESPONSE$ + A$
1060 NEXT I%
1070 TIMEOUT% = 1 :  RETURN
```

INP FUNCTION

PURPOSE:
Returns the byte read from port I. I must be in the range 0 to 65535.

SYNTAX:
X = INP(I)

NOTES:
INP is the complementary function to the OUT statement.

EXAMPLE:

```
100 A=INP(54321)
```

In 80186 assembly language, this function is equivalent to:

```
MOV DX,54321
IN AL,DX
```

INPUT STATEMENT

PURPOSE:
Allows input from the keyboard during program execution.

SYNTAX:
INPUT[;] [<"prompt string">{;|,}]<list of variables>

NOTES:
When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate that 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 keyboard.

To suppress the question mark, a comma may be used instead of a semicolon after the prompt string. For example, the statement INPUT "ENTER BIRTHDATE",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/linefeed sequence.

The data entered in response to the statement is assigned to the variable name(s) given in <variable list>. The number of data items supplied must be the same as the number of variables 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, and so on) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

EXAMPLES:
The statement:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
```

will yield:

```
? 5 (The 5 was typed in by the user in response to the question mark.)
5 SQUARED IS 25
```

The statement:

```
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
```

will yield:

```
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.9464

WHAT IS THE RADIUS?
```

INPUT# STATEMENT

PURPOSE:
Reads data items from a sequential device or file and assigns them to program variables.

SYNTAX:
INPUT#<file number>,<variable list>

NOTES:
<file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as 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, GW-BASIC ignores leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed is assumed to be the start of a number. The number terminates on a space, carriage return, linefeed, or comma.

If GW-BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and linefeeds. The first character encountered that is not a space, carriage return, or linefeed 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 return, or linefeed (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:

```
INPUT#2,A,B,C
```

INPUT$ FUNCTION

PURPOSE:
Returns a string of I characters, read from file number J. If the file number is not specified, the characters will be read from the screen.

SYNTAX:
X$ = INPUT$(I[,[#]J])

NOTES:
If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except CTRL-C, which is used to interrupt the execution of the INPUT$ function.

EXAMPLES:
Example 1:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 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
.
.
.
```

INSTR FUNCTION

PURPOSE:
Searches for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search.

SYNTAX:
J = INSTR([I,]X$,Y$)

NOTES:
I must be in the range 1 to 255. If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

EXAMPLE:
The function:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
```

will yield:

```
2 6
```

INT FUNCTION

PURPOSE:
Returns the largest integer <=X.

SYNTAX:
Y = INT(X)

EXAMPLES:
The function:

```
PRINT INT(99.89)
```

will yield:

```
99
```

The function:

```
PRINT INT(-12.11)
```

will yield:

```
-13
```

See the CINT and FIX functions, earlier in this section, which also return integer values.

IOCTL STATEMENT

PURPOSE:
Enables a BASIC program to send a control data string to a user-supplied character device driver any time after the driver has been OPENed.

SYNTAX:
IOCTL [#]<file number>, <string>

<file number> is the file number open to the device driver.

<string> is a valid string expression containing the control data as interpreted by the user-supplied driver.

NOTES:
IOCTL commands are generally 2 to 3 characters, optionally followed by an alphanumeric argument. An IOCTL command string may be up to 255 bytes long, with commands within the string separated by semicolons (';'). An example of an IOCTL command string is: "OO;SW132;GW".

EXAMPLES:
If a user had installed a device driver to replace LPT1, and that driver was able to set page length (the number of lines to print on a page before issuing a form feed character), then an IOCTL command to set or change the page length might have the form:

```
PLn
```

Where n is the new page length.

An IOCTL statement to open the new LPT1 driver and set the page length would then be:

```
OPEN "LPT1:" FOR OUTPUT AS #1
IOCTL #1,"PL66"
```

This statement opens LPT1 with an initial page length of 66 lines. The statement:

```
OPEN "\DEV\LPT1" FOR OUTPUT AS #1
IOCTL #1,"PL56"
```

would also open LPT1, but with an initial page length of 56 lines.

Of course, other user-definable IOCTL commands are possible, but the user must supply the device driver to interpret the commands.

IOCTL$ FUNCTION

PURPOSE:
Enables a BASIC program to read a control data string from a character device driver that is OPEN.

SYNTAX:
X$ = IOCTL$([#]<file number>)

<file number> is the file number open to the device.

NOTES:
The IOCTL$ function is generally used to get acknowledgement that an IOCTL statement succeeded or failed. It is also used to get device information after an IOCTL command is issued using the IOCTL statement.

EXAMPLE:

```
OPEN "\DEV\FOO" AS #1
IOCTL #1,"RAW" 'Tell device that data is "raw"
IF IOCTL$(1) = "0" THEN CLOSE 1
```

In this example, if character driver 'FOO' returns false from the raw data mode IOCTL request, the program closes the file because raw data is required by the program. This example assumes that a device driver FOO, programmed to accept the required control strings, has been installed.

KEY STATEMENT

PURPOSE:
Assigns softkey values to function keys and display the values.

SYNTAX:
KEY n,X$
KEY n,CHR$(<shift>) + CHR$(<scan code>)
KEY LIST
KEY ON
KEY OFF

n is the number of the key definition, as follows:

n	Key	n	Key
1	F1	11	Cursor up
2	F2	12	Cursor left
3	F3	13	Cursor right
4	F4	14	Cursor down
5	F5	15	User-defined
6	F6	16	User-defined
7	F7	17	User-defined
8	F8	18	User-defined
9	F9	19	User-defined
10	F10	20	User-defined

X$ is the text assigned to the specified key.

NOTES:
The KEY statement allows function keys to be designated for special "softkey" functions. Each of the ten 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 values (the names of keys are boxed):

Key	Value	Key	Value
F1	LIST [SPACE BAR]	F6	,"LPT1:" [RETURN]
F2	RUN [RETURN]	F7	TRON [RETURN]
F3	LOAD"	F8	TROFF [RETURN]
F4	SAVE"	F9	KEY [SPACE BAR]
F5	CONT [RETURN]	F10	SCREEN 0,0,0, [RETURN]

After softkeys have been designated, they can be displayed with the KEY ON and KEY LIST statements.

KEY ON causes the softkey values to be displayed on the twenty-fifth line of the screen. When the screen width is 80, ten softkeys are displayed. When the screen width is 40 characters, the first five of the ten softkeys are displayed; pressing CTRL-T causes the second five softkeys to be displayed. In either screen width, only the first 6 characters of each key are displayed. ON is the default state for the softkey display.

KEY OFF erases the softkey display from the twenty-fifth line, making that line available for program use. It does not disable the function keys.

KEY LIST displays all ten softkey values on the screen, with all 15 characters of each key displayed.

If the function key number is not in the range 1-20, 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.

GW-BASIC processes softkeys in the following order:

1. The line printer echo key is processed first. Defining the Print Screen key as a user-defined key trap will not prevent characters from being echoed to the line printer if depressed.

2. Function keys and the cursor direction keys are examined next. Defining a function key or cursor direction key as a user-defined key trap will have no effect, because they are considered pre-defined.

3. Finally, the user-defined keys are examined.

4. Any key that is trapped is not passed on. That is, the key is not read by GW-BASIC.

Key trapping applies to any key, including CTRL-C, CTRL-BREAK, or BREAK. This feature makes it possible to prevent BASIC Application users from accidentally halting a program with CTRL-BREAK.

continued on next page

EXAMPLES:

```
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.
80 KEY 1, "" 'Disables softkey 1
```

Assignments like these might be used to speed data entry.

The following routine initializes the first five 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 SOFTKEYS$(I)
50 KEY I,SOFTKEY$(I)
60 NEXT I
70 KEY ON 'Displays new softkeys
```

KEY(n) STATEMENT

PURPOSE:
Enables or disables event trapping of softkey or cursor direction key activity for the specified function key.

SYNTAX:
KEY(n) ON
KEY(n) OFF
KEY(n) STOP

(n) is the number of a function key or cursor direction key. (See "KEY Statement" for information on assigning softkey values to function keys.) The cursor direction keys are numbered sequentially after the function keys in the following order: up, left, right, down.

NOTES:
Note that the KEY statement assigns softkey and cursor direction values to function keys and displays the values. Do not confuse KEY ON and KEY OFF, which display and erase these values, with the event-trapping statements described here.

The KEY(n) ON statement enables softkey or cursor direction key event trapping by an ON KEY statement (see "ON KEY Statement" for more information). While trapping is enabled, if a non-zero line number is specified in the ON KEY statement, GW-BASIC checks between every statement to see if a softkey or cursor direction key has been used. If it has, the ON KEY statement is executed.

KEY(n) OFF disables the event trap. If an event takes place, it is not remembered.

KEY(n) STOP disables the event trap, but if an event occurs, it is remembered and an ON KEY statement will be executed as soon as trapping is enabled.

continued on next page

EXAMPLE:

```
10 KEY 4,"SCREEN 0,0" ' assigns softkey 4
20 KEY(4) ON 'enables event trapping
 .
 .
 .
70 ON KEY(4) GOSUB 200
 .
 .
 .
```

(key 4 pressed)

```
 .
 .
 .
200 'Subroutine for screen
```

KILL STATEMENT

PURPOSE:
Deletes a file from disk or a RAM cartridge.

SYNTAX:
KILL <filespec>

NOTES:
KILL is used for all types of disk files: program files, random data files, and sequential data files. The filespec may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at the position of the asterisk.

KILL also allows a pathname in place of <filespec>. KILL checks to see if the file is open; and, if so, will give a "File already open" error.

KILL, like OPEN, cannot distinguish a file in another directory from a file you may have open. It is possible to get an unexpected "File already open" error under these circumstances. (See "OPEN Statement" later in this section.)

KILL can delete disk files only if GW-BASIC was started from MS-DOS. Otherwise KILL can delete only BASIC files contained on RAM cartridges.

KILL may be used to format a cartridge by using <filespec> of the form "CARTn:/FORMAT" where n = 1 or 2. (See the FILES statement for more information on formatting RAM cartridges.)

Warning: Be extremely careful when using wild cards with this command.

EXAMPLES:

```
200 KILL "DATA1?.DAT"
```

Deletes all files whose six-letter names begin with DATA1 and have the extension .DAT.

The position taken by the question mark can contain any valid filename character.

```
210 KILL "DATA1.*"
```

deletes all files named DATA1, regardless of the file extension.

continued on next page

```
130 KILL "CART2:ABC.123"
```

deletes the file ABC.123 from the RAM cartridge in cartridge port 2.

LEFT$ FUNCTION

PURPOSE:
Returns a string comprising the leftmost I characters of X$.

SYNTAX:
Y$ = LEFT$(X$,I)

NOTES:
I must be in the range 0 to 255. If I is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) will be returned. If I = 0, the null string (length zero) is returned.

EXAMPLE:
The function:

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,3)
30 PRINT B$
```

will yield:

```
BAS
```

Also, see the MID$ and RIGHT$ functions elsewhere in this section.

LEN FUNCTION

PURPOSE:
Returns the number of characters in X$. Nonprinting characters and blanks are also counted.

SYNTAX:
Y = LEN(X$)

EXAMPLE:
The function:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
```

will yield:

```
16
```

LET STATEMENT

PURPOSE:
Assigns the value of an expression to a variable.

SYNTAX:
[LET]<variable> = <expression>

NOTES:
Notice that the word LET is optional; in other words, the equal sign is sufficient for assigning an expression to a variable name.

EXAMPLES:

```
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
  .
  .
  .
```

LINE STATEMENT

PURPOSE:
Draws a line or box on the screen.

SYNTAX:
LINE [(X1,Y1)] – (X2,Y2) [,[<color>][,B[F]][,<style>]]

(X1,Y1) is the coordinate for the starting point of the line.

(X2,Y2) is the ending point for the line.

<color> is the number of the color in which the line should be drawn. (See "COLOR Statement" earlier in this section for more information.) If the ,B or ,BF option is used, the box is drawn in this color.

,B draws a box in the foreground, with the points (X1,Y1) and (X2,Y2) as opposite corners.

,BF draws a filled box in the foreground.

<style> is a 16-bit integer mask used when placing pixels on the screen.

NOTES:
When out-of-range coordinates are given, line clipping occurs. Points plotted outside the screen or viewport limits do not appear.

The <style> argument provides dashed lines. Each time LINE stores a point on the screen, it will use the current circulating bit in <style>. If that bit is 0, storage does not occur. If the bit is a 1, then a normal storage occurs. After each point, the next bit position in <style> is selected.

Since a 0 bit in <style> does not clear out the old contents, the user may wish to draw a background line before a 'styled' line to force a known background.

<style> operates only for normal lines and boxes; it has no effect on filled boxes.

The coordinate form STEP (xoffset,yoffset) can be used in place of an absolute coordinate. For example, assume that the most recent point referenced was (0,0). The statement LINE STEP (10,5) would specify a point at offset 10 from X and offset 5 from Y.

If the STEP option is used for the second coordinate on a LINE statement, it is relative to the first coordinate in the statement. The only other way to establish a new "most recent point" is to initialize the screen with the CLS and SCREEN statements.

EXAMPLES:
The following examples assume a screen 320 pixels wide by 200 pixels high. The statement:

```
10 LINE -(X2,Y2)
```

draws a line from the last point to X2,Y2 in the foreground color. This is the simplest form of the LINE statement. The statement:

```
20 LINE (0,0)-(319,199)
```

draws a diagonal line across the screen (downward, from left to right). The statement:

```
30 LINE (0,100)-(319,100)
```

draws a line from left to right across the screen. The statement:

```
40 LINE (10,10)-(20,20),2
```

draws a line in color 2. The statements:

```
10 CLS
20 LINE -(RND*319,RND*199),RND*4
30 GO TO 20
```

draw lines forever, using random attributes. The statements:

```
10 FOR X=0 to 319
20 LINE (X,0)-(X,199),X AND 1
30 NEXT
```

draw an alternating line-on, line-off pattern. The statement:

```
10 LINE (0,0)-(100,100),,B
```

draws a box in the foreground (note that the color is not included). The statement:

```
20 LINE STEP (0,0)-STEP (200,200),2,BF
```

continued on next page

draws a filled box in the foreground in color 2. Coordinates are given as offsets. The statement:

```
10 LINE (0,0) – (160,100),3,,&HFF00
```

draws a dashed line from the upper left corner of the screen to the center.

LINE INPUT STATEMENT

PURPOSE:
Reads the input of an entire line (up to 254 characters) from the keyboard to a string variable, without the use of delimiters.

SYNTAX:
LINE INPUT[;][<"prompt string">;] <string variable>

NOTES:
<"prompt string"> is a string literal that is printed on the screen before input is accepted. A question mark is not printed unless it is part of <"prompt string">. All input from the end of <"prompt string"> to the carriage return is assigned to <string variable>. If a linefeed/carriage return sequence (in this order only) is encountered, both characters are echoed; however, the carriage return is ignored, the linefeed 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/linefeed sequence on the screen.

A LINE INPUT statement may be aborted by typing CTRL-C. In that case, GW-BASIC will return to command level. Typing CONT resumes execution at the LINE INPUT.

EXAMPLE:
See "LINE INPUT# Statement".

LINE INPUT# STATEMENT

PURPOSE:
Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

SYNTAX:
LINE INPUT <file number>,<string variable>

NOTES:
<file number> is the number under which the file was OPENed.
<string variable> is the variable name to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. (If a linefeed/carriage return sequence is encountered, it is preserved in the same way as in the LINE INPUT statement.) It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program. (See "SAVE Command" later in this section for more information.)

EXAMPLE:
The routine:

```
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
```

will yield:

```
CUSTOMER INFORMATION? LINDA JONES 234,4 MEMPHIS
LINDA JONES 234,4 MEMPHIS
```

LIST COMMAND

PURPOSE:
Lists all or part of the program currently in memory on the screen.

SYNTAX 1:
LIST [<line number>]

SYNTAX 2:
LIST [<line number>][–[<line number>]][,<device>]

<line number> is in the range 0 to 65529.

<device> is a string expression returning a device designation, such as "LPT1".

NOTES:
GW-BASIC always returns to command level after a LIST is executed.

Syntax 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing CTRL-C.) If <line number> is included, only the specified line will be listed.

Syntax 2

This syntax allows the following options:

1. If the first <line number> is followed by a hyphen and a second <line number> is not specified, the first line number and all higher-numbered lines are listed.

2. If only the second <line number> (preceded by a hyphen) is specified, all lines from the beginning of the program through that line are listed.

3. If both <line number(s)> are specified, the entire range is listed.

4. If the <device> is omitted, the listing is shown on the screen.

continued on next page

EXAMPLES:
The following statements illustrate Syntax 1:

```
LIST
```

lists the program currently in memory.

```
LIST 500
```

lists line 500.

The following statements illustrate Syntax 2:

```
LIST 150-
```

lists all lines from 150 to the end.

```
LIST -1000
```

lists all lines from the lowest number through 1000.

```
LIST 150-1000
```

lists lines 150 through 1000, inclusive.

```
LIST 150-1000,"LPT1"
```

lists lines 150 through 1000 on the line printer.

LLIST COMMAND

PURPOSE:
Lists all or part of the program currently in memory at the line printer.

SYNTAX:
LLIST [<line number>[– [<line number>]]]

NOTES:
LLIST assumes a 132-character-wide printer.

GW-BASIC always returns to command level after an LLIST command is executed.

The options for LLIST are the same as for LIST, except that the <device> is always LPT1:. Therefore, a <device> specification is illegal in the LLIST command.

EXAMPLE:
See the examples for "LIST Statement".

LOAD COMMAND

PURPOSE:
Loads a file from disk or RAM cartridge into memory.

SYNTAX:
LOAD <filespec>[,R]

NOTES:
The <filespec> must include the filename that was used when the file was saved. If an extension is not specified, MS-DOS supplies .BAS as the extension.

The device specifications for cartridge port 1 and cartridge port 2 are CART1: and CART2:, respectively.

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before loading the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

EXAMPLES:
The statement:

```
LOAD "STRTRK",R
```

loads and runs the program STRTRK. The statement:

```
LOAD "B:MYPROG"
```

loads the program MYPROG from the disk in drive B, but does not run the program. The statement:

```
LOAD "CART1:TEST"
```

loads the program TEST from the RAM cartridge in cartridge port 1.

LOC FUNCTION

PURPOSE:
With random disk files, LOC returns the actual record position within the file number of the last record read or written.

With sequential files, LOC returns the current byte position in the file, divided by 128.

When a file is opened for APPEND or OUTPUT, LOC returns the length of the file divided by 128.

SYNTAX:
X = LOC(<file number>)

where <file number > is the number under which the file was OPENed.

NOTES:
When a file is opened for sequential input, GW-BASIC reads the first sector of the file; consequently, LOC will return a 1 even before any input from the file occurs.

For a communications file, LOC(X) is used to determine if there are any characters in the input queue waiting to be read. If there are more than 255 characters in the queue, LOC(X) returns 255. Because strings are limited to 255 characters, this practical limit alleviates the need to test for string size before reading data into the string.

EXAMPLE:

```
200 IF LOC(1)>50 THEN STOP
```

LOCATE STATEMENT

PURPOSE:
Moves the cursor to the specified position. Optional parameters turn the blinking cursor on and off and define the vertical start and stop lines.

SYNTAX:
LOCATE [<row>][,[<col>][,[<cursor>][,[<start>][,<stop>]]]]

<row> is a line number on the screen. <row> should be a numeric expression returning an unsigned integer from 1 to 25. While the system is displaying the softkey values on the 25th line, 24 is the maximum value for <row>.

<col> is the column number on the screen. It should be a numeric expression returning an unsigned integer. The range will vary according to the number of columns per line (1 to 40 or 1 to 80).

<cursor> is a Boolean value indicating whether the cursor should be visible (<cursor> = 1) or invisible (<cursor> = 0).

<start> is the cursor starting line on the screen. It should be a numeric expression returning an unsigned integer.

<stop> is the cursor stop line (vertical) on the screen. It should be a numeric expression returning an unsigned integer.

NOTES:
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 the <start> and <stop> lines are the raster lines that are lit on the screen. A wider range between the start and stop lines will produce a taller cursor, such as one that occupies an entire character block.

If the <start> line is given but the <stop> line is omitted, <stop> assumes the same value as <start>.

EXAMPLES:
The statement:

```
10 LOCATE 1,1
```

moves cursor to the upper left corner of the screen. The statement:

```
20 LOCATE ,,1
```

makes the cursor visible; its position remains unchanged. The statement:

```
30 LOCATE ,,,7
```

leaves the position and cursor visibility unchanged. The cursor is set to display at the bottom of the character starting and ending on raster line 7. The statement:

```
40 LOCATE 5,1,1,0,7
```

moves the cursor to line 5, column 1, and turns the cursor on. The cursor will cover the entire character cell, starting at scan line 0 and ending on scan line 7.

LOF FUNCTION

PURPOSE:
Returns the length of the file in bytes.

SYNTAX:
X = LOF(<file number>)

<file number> is the number under which the file was OPENed.

EXAMPLE:
In the following example:

```
110 IF REC*RECSIZ>LOF(1) THEN PRINT "INVALID ENTRY"
```

the variables REC and RECSIZ contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

LOG FUNCTION

PURPOSE:
Returns the natural logarithm of X. X must be greater than zero.

SYNTAX:
Y = LOG(X)

EXAMPLE:
The function:

```
PRINT LOG(45/7)
```

will yield:

```
1.860752
```

LPOS FUNCTION

PURPOSE:
Returns the current position of the line printer's print head within the line printer buffer.

SYNTAX:
X = LPOS(I)

where I is the number assigned to the line printer.

NOTES:
LPOS does not necessarily give the physical position of the print head.

EXAMPLE:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT AND LPRINT USING STATEMENTS

PURPOSE:
Prints data at the line printer.

SYNTAX:
LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

NOTES:
Same as PRINT and PRINT USING, except that the output goes to the line printer.

LPRINT assumes a 132-character-wide printer.

EXAMPLES:
See PRINT and PRINT USING later in this section.

LSET AND RSET STATEMENTS

PURPOSE:
Moves data from memory to a random file buffer (in preparation for a PUT statement) and left-justifies or right-justifies the data in the buffer.

SYNTAX:
LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

NOTES:
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$ and MKS$ functions later in this section.

EXAMPLES:

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = DESC$
```

Note: LSET or RSET may also be used with a nonfielded 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.

MERGE COMMAND

PURPOSE:
Merges a specified disk file into the program currently in memory.

SYNTAX:
MERGE <filespec>

NOTES:
The <filespec> must include the filename used when the file was saved. The file must have been saved in ASCII format. If it was not, a "Bad file mode" error occurs.

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (The MERGE process may be thought of as "inserting" the program lines on disk into the program in memory.)

GW-BASIC always returns to command level after executing a MERGE command.

EXAMPLE:
The command:

```
MERGE "NUMBRS"
```

inserts, by sequential line number, all lines in the program NUMBRS into the program currently in memory.

MID$ STATEMENT

PURPOSE:
Replaces a portion of one string with another string.

SYNTAX:
MID$(<string exp1>,I[,J]) = <string exp2>

where I and J are integer expressions and <string exp1> and <string exp2> are string expressions.

NOTES:
The characters in <string exp1>, beginning at position I, are replaced by the characters in <string exp2>.

The optional J refers to the number of characters from <string exp2> that will be used in the replacement. If J is omitted, all of <string exp2> is used. However, whether J is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

MID$ is also a function that returns a substring of a given string.

EXAMPLE:
The routine:

```
10 A$ = "KANSAS CITY, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
```

will yield:

```
KANSAS CITY, KS
```

MID$ FUNCTION

PURPOSE:
Returns a string J characters long from X$, beginning with the Ith character.

SYNTAX:
Y$ = MID$(X$,I[,J])

NOTES:
I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I is greater than the number of characters in X$ (LEN(X$)), MID$ returns a null string.

Also see the LEFT$ and RIGHT$ functions elsewhere in this section.

EXAMPLE:
The function:

```
10 A$ = "GOOD    "
20 B$ = "MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
```

will yield:

```
GOOD EVENING
```

MKDIR STATEMENT

PURPOSE:
Creates a new directory in the MS-DOS file system.

SYNTAX:
MKDIR <pathname>

<pathname> is a string expression, not exceeding 128 characters, identifying the new directory to be created.

NOTES:
MKDIR works exactly like the DOS command MKDIR.

Possible errors:

"Bad file name"
"Path/File Access error"

EXAMPLES:
The following statements will create the subdirectories SALES and ACCOUNTING.

```
MKDIR "SALES" ' Create subdirectory SALES in root of current drive
MKDIR "ACCOUNTING" ' Create subdirectory ACCOUNTING in root of
current drive
MKDIR "B:INVENTORY" 'Create subdirectory INVENTORY in root on
drive B
```

The following statement is also valid:

```
MKDIR "\SALES"
```

MKI$, MKS$, and **MKD$** FUNCTIONS

PURPOSE:
Converts numeric values to string values.

SYNTAX:
X$ = MKI$(<integer expression>)
X$ = MKS$(<single precision expression>)
X$ = MKD$(<double precision expression>)

NOTES:
Any numeric value placed in a random file buffer with 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, and CVD Functions" earlier in this section.

EXAMPLE:

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$=MKS$(AMT)
120 LSET N$=A$
130 PUT #1
.
.
.
```

NAME STATEMENT

PURPOSE:
Changes the name of a disk or a RAM cartridge file.

SYNTAX:
NAME <old filename> AS <new filename>

NOTES:
<old filename> must exist and <new filename> must not exist; otherwise, an error will result. Also, <old filename> must be closed before rename. The same open file check is used as in OPEN and KILL, described elsewhere in this section.

A file may not be renamed with a new drive designation. If a new drive designation is attempted, a "Rename across disks" error will be generated. After a NAME Statement, the file exists on the same disk, in the same area of disk space, with the new name.

Pathnames are not allowed. That is, only files in the current directory may be renamed. Giving a path will result in a "Bad filename" error.

NAME may be used to format a cartridge by a new filename of the form "CARTn:/FORMAT", where n = 1 or 2. (See the FILES statement for more information on formatting RAM cartridges.)

EXAMPLE:
In the example:

```
NAME "ACCTS" AS "LEDGER"
```

the file that was formerly named ACCTS will now be named LEDGER.

NEW COMMAND

PURPOSE:
Deletes the program currently in memory, clears all variables and cancels the definitions of all animation objects.

SYNTAX:
NEW

NOTES:
NEW is entered in direct mode to clear memory before entering a new program. GW-BASIC always returns to command level after a NEW is executed.

NEW closes all files and turns tracing off.

EXAMPLE:

```
NEW
```

OBJECT FUNCTION

PURPOSE:
Returns the current value of a specific attribute for a given object.

SYNTAX:
X = OBJECT (<object number>,<attribute number>)

<object number> is the number of the object, and <attribute number> is the number of the attribute which the OBJECT function returns for that object.

NOTES:
See the description of the DEF OBJECT statement for information on object numbers, and of the OBJECT statement for information on attributes.

The OBJECT function returns parameter values used by the OBJECT statement. In some cases, GW-BASIC rounds the parameter values given in the OBJECT statement to the nearest usable value. For example, specifying a pixel location of 1.3 in an OBJECT statement results in an actual location of 1. The OBJECT function would then return a value of 1 for the attribute parameter.

EXAMPLE:
The function:

```
10 X = OBJECT (1,5)
```

sets variable X equal to the value of attribute 5, the period of view attribute, for object number 1.

OBJECT STATEMENT

PURPOSE:
Specifies one or more attributes of the specified object.

SYNTAX:
OBJECT <object number>,<attribute number> = <value> [,<attribute number> = <value>,...]

<object number> is the number of the object as specified in a previous DEF OBJECT statement. <attribute number> is the number of an attribute being set. <value> is the new value for the attribute.

NOTES:
The attributes for an object are as follows:

Attribute number	Attributes	Default
1	Current X position	0
2	Current Y position	0
3	Terminal X position	0
4	Terminal Y position	0
5	Period of view	0 (infinite)
6	X offset	0
7	Y offset	0
8	Current view	1
9	Transparency	0 (transparent)
10	Speed	0

The values of all attributes (except transparency) can be integer or real numbers. GW-BASIC converts the values to the closest usable value. For example, specifying a pixel location of 1.3 rounds to a pixel location of 1 internally. The OBJECT function returns the rounded values.

The position attributes are given in terms of logical units which define the resolution of the current window. When the current window includes the entire screen and the resolution is 320 X 200 then a horizontal logical unit is 1/320 of the screen width and a vertical logical unit is 1/200 of the screen height.

The current X and Y position attributes select the location for the object within the current window. If the speed attribute is greater than zero, the current X and Y position attributes select the origin for the object and the terminal X and Y position attributes select the destination for the object.

continued on next page

The period of view attribute is given in seconds; GW-BASIC converts this value internally to the nearest 1/60th second. This attribute determines how long GW-BASIC displays each view of the object before switching to the next view. GW-BASIC displays the views in the order they are listed in the DEF OBJECT statement. Setting the period of view to 0 causes GW-BASIC to display continuously the view selected by the current view attribute rather than switching views.

The X and Y offset values specify the point of the location indicated by the current X and Y position attributes. The X and Y offset attributes are relative to the center of the rectangle formed by the <x size> and <y size> attributes of the DIM OBJECT statement. The offset is the distance from location (0,0) at the center of the rectangle to the reference point of the object.

For example, the reference point for an arrow would be at the tip of the arrowhead. If the arrow pointed down and right and was 10 logical units in length and height, then the X and Y offsets should be 5 and 5 units, respectively, to place the tip of the arrowhead at the location specified by the current X and Y position attributes.

The current view is the number of one of the arrays in the array list included with the DEF OBJECT statement. The first array in the list is view number 1, the second array is view number 2, and so on. If the period of view attribute is zero, the current view is the only view shown. Otherwise, the current view is the first view shown before the other views are shown in sequence. After displaying the last view, the cycle begins again with the first view.

The transparency parameter refers to the background within the rectangle containing an object. The default value of 0 causes the background to be transparent. In other words, only those pixels in the rectangle which have a non-zero value will be displayed. Setting the transparency parameter to 1 causes the background in the rectangle to be displayed.

The speed attribute is a real value given in logical units per second. The default speed of 0 causes the object to be stationary. Setting the speed attribute to a value greater than 0 causes the object to move from the current X and Y position to the terminal X and Y position at the specified number of logical units per second.

See also the OBJECT function and the DIM OBJECT, DEF OBJECT, START/STOP OBJECT, ARRIVAL, CLIP, and COLLISION statements described elsewhere in this section.

EXAMPLE:
The statement:

```
10 OBJECT 1,1=30,2=20,3=50,4=40,10=5
```

places object number 1 at coordinates (30,20) in the current window, specifies the destination for the object at (50,40), and sets the speed to 5 logical units per second. The ACTIVATE 1 statement will then cause the object to move from (30,20) to (50,40) at a speed of 5 logical units per second.

OCT$ FUNCTION

PURPOSE:
Returns a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

SYNTAX:
Y$ = OCT$(X)

EXAMPLE:
The function:

```
PRINT OCT$(24)
```

will yield:

```
30
```

For details on hexadecimal conversion, see the HEX$ function earlier in this section.

ON ARRIVAL STATEMENT

PURPOSE:
Specifies the first line number of a subroutine to be performed when an object arrives at the destination indicated in an OBJECT statement.

SYNTAX:
ON ARRIVAL [(<object number>)] GOSUB <line number>

<object number> is the number of an object being monitored for arrival.

<line number> is the number of the first line of a subroutine to be performed when the specified arrival event occurs.

NOTES:
A <line number> of 0 disables the arrival event testing specified in the ON ARRIVAL statement.

Multiple ON ARRIVAL statements can be active simultaneously to trap the arrivals of different objects.

When the ON ARRIVAL statement does not include an object number, the subroutine is executed when any object arrives at its destination.

When the ON ARRIVAL statement includes an object number, the subroutine is executed when the specified object arrives at its destination.

Subroutines specified in ON ARRIVAL statements are invoked in the same order as the ON ARRIVAL statements were executed.

One use of the ON ARRIVAL statement is to automatically latch the arrival status of the objects on the screen. When any object specified in an ON ARRIVAL statement arrives at its destination, GW-BASIC makes a copy of the internal arrival flags of all objects which have reached their destination since the last time the status was latched. In this respect, ON ARRIVAL is the same as the ARRIVAL(−1) function. (See the ARRIVAL function in this section.)

Because the ON ARRIVAL (<object number>) form of the ON ARRIVAL statement executes its associated subroutine only when the specified object reaches its destination, it is usually unnecessary to use the ARRIVAL (0) function within the subroutine. However, you can use the ARRIVAL(0) function to test whether other objects have arrived at their destination.

continued on next page

The ON ARRIVAL statement will be executed only if an ARRIVAL ON statement has been executed (see "ARRIVAL Statement") to enable the testing of arrival events. If ARRIVAL ON is in effect, and if the <line number> in the ON ARRIVAL statement is not zero, GW-BASIC checks after the execution of every statement to see if the specified arrival event has occurred. If it has, GW-BASIC performs a GOSUB to the specified line.

If an ARRIVAL OFF statement has been executed, the arrival event is not discovered and the GOSUB is not performed (see "ARRIVAL Statement").

If an ARRIVAL STOP statement has been executed and an object specified in an ON ARRIVAL statement reaches its destination, the GOSUB is not performed immediately, but will be performed as soon as an ARRIVAL ON statement is executed (see "ARRIVAL Statement").

When the ON ARRIVAL subroutine is executed, an automatic ARRIVAL STOP is performed until all outstanding arrival events have been handled, so that recursive traps cannot take place. The RETURN from the final trapping subroutine automatically performs an ARRIVAL ON statement, unless the subroutine executes an explicit ARRIVAL OFF statement.

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the ON ARRIVAL subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active, and errors such as "FOR without NEXT" may result.

EXAMPLES:
The statement:

```
10 ON ARRIVAL (2) GOSUB 200
```

executes the event-handling subroutine at line 200 when object 2 arrives at its destination. The statement:

```
20 ON ARRIVAL GOSUB 300
```

executes the event-handling subroutine at line 300 whenever any object arrives at its destination.

ON CLIP STATEMENT

PURPOSE:
Specifies the first line number of a subroutine to be performed when an object exceeds (is clipped at) the border of the current window.

SYNTAX:
ON CLIP [(<object number>)] GOSUB <line number>

<object number> is the number of an object being monitored for possible clipping.

<line number> is the number of the first line of a subroutine to be performed when clipping occurs.

NOTES:
A <line number> of 0 disables the trapping of the clipping event specified in the ON CLIP statement.

Multiple ON CLIP statements can be active simultaneously to trap the clipping of different objects.

When the ON CLIP statement does not include an object number, the subroutine is executed when any object exceeds the boundary of the current window.

When the ON CLIP statement includes an object number, the subroutine is executed when the specified object exceeds the boundary of the current window.

ON CLIP subroutines are invoked in the same order as the ON CLIP statements were executed.

One use of the ON CLIP statement is to latch the clipping status of the objects on the screen. When any object specified in an ON CLIP statement exceeds the boundary of the current viewport, GW-BASIC makes a copy of the internal clip flags of all objects which have exceeded the boundary of the current viewport since the last time the status was latched. In this respect, ON CLIP is the same as the CLIP(−1) function. (See the CLIP function in this section.)

Because the ON CLIP (<object number>) form of the ON CLIP statement executes its associated subroutine only when the specified object exceeds the current viewport, it is usually unnecessary to use the CLIP(0) function within the subroutine. However, you can use the CLIP(0) function to test whether other objects have been clipped.

continued on next page

The ON CLIP statement will only be executed if a CLIP ON statement has been executed (see "CLIP Statement") to enable the testing of clipping events. If CLIP ON is in effect, and if the <line number> in the ON CLIP statement is not 0, GW-BASIC checks after the execution of every statement to see if the specified clipping event has occurred. If it has, GW-BASIC performs a GOSUB to the specified line.

If a CLIP OFF statement has been executed, the clipping event is not discovered and the GOSUB is not performed (see "CLIP Statement").

If a CLIP STOP statement has been executed and an object specified in an ON CLIP statement is clipped, the GOSUB is not performed immediately, but will be performed as soon as a CLIP ON statement is executed (see "CLIP Statement").

When the ON CLIP subroutine is performed, an automatic CLIP STOP is performed until all outstanding clip events have been handled, so that recursive traps cannot take place. The RETURN from the final trapping subroutine automatically performs a CLIP ON statement unless the subroutine executes an explicit CLIP OFF statement.

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the ON CLIP subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap remain active, and errors such as "FOR without NEXT" may result.

EXAMPLES:
The statement:

```
10 ON CLIP (2) GOSUB 200
```

executes the event-handling subroutine at line 200 whenever object 2 exceeds the boundary of the current window. The statement:

```
20 ON CLIP GOSUB 300
```

executes the event-handling subroutine at line 300 whenever any object exceeds the boundary of the current window.

ON COLLISION STATEMENT

PURPOSE:
To specify the first line number of a subroutine to be performed when an object collides with another object.

SYNTAX:
ON COLLISION [(<object number 1>[,<object number 2>])]
GOSUB <line number>

<object number 1> and <object number 2> are the numbers of objects being monitored for possible collisions.

<line number> is the number of the first line of a subroutine to be performed when a collision occurs.

NOTES:
A <line number> of 0 disables the collision event testing specified in the ON COLLISION statement.

Multiple ON COLLISION statements can be active simultaneously to trap the collisions of different objects.

When the ON COLLISION statement does not include object numbers, the subroutine is executed when any two objects collide.

When the ON COLLISION statement includes only the first object number, the subroutine is executed when the specified object collides with any other object.

ON COLLISION subroutines are invoked in the same order as the ON COLLISION statements were executed.

A collision is detected only if two objects overlap when they are actually being drawn on the screen. Fast-moving objects may move several pixels at each update, and may actually cross paths without overlapping on the screen.

A collision of two objects can cause more than one event to be trapped. Multiple events can also occur if more than two objects collide "simultaneously" (overlap during the same screen update).

One use of the ON COLLISION statement is to latch the collision status of the objects on the screen. When an object specified in an ON COLLISION statement collides with another object, GW-BASIC makes a copy of the internal collision flags of all objects which have collided with

continued on next page

another object since the last time the status was latched. In this respect, ON COLLISION is the same as the COLLISION(−1) function. (See the COLLISION function in this section.)

Because the ON COLLISION (<object number>) form of the ON COLLISION statement executes its associated subroutine only when the specified object collides with another object, it is usually unnecessary to use the COLLISION(0) function within the subroutine. However, you can use the COLLISION(0) function to test whether other objects have collided.

The ON COLLISION statement will be executed only if a COLLISION ON statement has been executed (see "COLLISION Statement" earlier in this section) to enable collision testing. If COLLISION ON is in effect, and if the <line number> in the ON COLLISION statement is not 0, GW-BASIC checks after the execution of every statement to see if the specified collision event has occurred. If it has, GW-BASIC performs a GOSUB to the specified line.

If a COLLISION OFF statement has been executed, the collision event is not discovered and the GOSUB is not performed (see "COLLISION Statement").

If a COLLISION STOP statement has been executed and a specified collision occurs, the GOSUB is not performed immediately, but will be performed as soon as a COLLISION ON statement is executed (see "COLLISION Statement").

When an ON COLLISION subroutine is performed, an automatic COLLISION STOP is performed until all outstanding collision events have been handled, so that recursive traps cannot take place. The RETURN from the final trapping subroutine automatically performs a COLLISION ON statement unless the subroutine executes an explicit COLLISION OFF statement.

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the ON COLLISION subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap remain active, and errors such as "FOR without NEXT" may result.

EXAMPLES:
The statement:

```
10 ON COLLISION (2,3) GOSUB 200
```

executes the collision event-handling subroutine at line 200 whenever objects 2 and 3 collide. The statement:

```
20 ON COLLISION (2) GOSUB 300
```

executes the collision event-handling subroutine at line 300 whenever object 2 collides with another object. The statement:

```
30 ON COLLISION GOSUB 400
```

executes the collision event-handling subroutine at line 400 whenever any two objects collide.

ON COM STATEMENT

PURPOSE:
Specifies the first line number of a subroutine to be performed when activity occurs on a communications channel.

SYNTAX:
ON COM(n) GOSUB <line number>

<line number> is the number of the first line of a subroutine to be performed when activity occurs on the specified communications channel.

(n) is the number of the communications channel.

NOTES:
A <line number> value of zero disables the communications event trap.

The ON COM statement will only be executed if a COM(n) ON statement has been executed (see "COM Statement" earlier in this section) to enable event trapping. If event trapping is enabled, and if the <line number> in the ON COM statement is not zero, GW-BASIC checks between statements to see if communications activity has occurred on the specified channel. If communications activity has occurred, a GOSUB will be performed to the specified line.

If a COM OFF statement has been executed for the communications channel (see "COM Statement"), the GOSUB is not performed and the event is not discovered.

If a COM STOP statement has been executed for the communications channel (see "COM Statement"), the GOSUB is not performed, but will be performed as soon as a COM ON statement is executed if activity has occurred on the channel.

When an event trap occurs (in other words, when the GOSUB is performed), an automatic COM STOP is executed so recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a COM ON statement unless an explicit COM OFF was performed inside the subroutine.

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GW-BASIC is not executing a program. Also, event trapping is automatically disabled when an error trap occurs.

EXAMPLE:
The statement:

```
10 ON COM(1) GOSUB 1000
```

will cause GW-BASIC to transfer control to line 1000 when the system receives a character from the first RS-232-C module.

ON ERROR GOTO STATEMENT

PURPOSE:
Enables error handling and specifies the first line of the error handling routine.

SYNTAX:
ON ERROR GOTO <line number>

NOTES:
After error handling has been enabled, all errors detected, including direct mode errors (for example, syntax errors), will cause a jump to the specified error handling routine. If <line number> does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0 statement. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes GW-BASIC to stop and print the error message for the error that caused the trap. All error handling routines should execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

EXAMPLE:

```
10 ON ERROR GOTO 1000
```

ON...GOSUB AND ON...GOTO STATEMENTS

PURPOSE:
Branches to one of several specified line numbers, depending on the value returned when an expression is evaluated.

SYNTAX:
ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

NOTES:
The value of <expression> determines which line number in the list will be used as the destination for branching. For example, if the value of <expression> is 3, the third line number in the list will be the destination of the branch. (If the value is a noninteger, 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), GW-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
```

ON KEY STATEMENT

PURPOSE:
Specifies the first line number of a subroutine to be performed when a specified key is pressed.

SYNTAX:
ON KEY(n) GOSUB <line number>

(n) is the number of a function key, as defined by the KEY (n) statement.

<line number> is the number of the first line of a subroutine to be performed when the specified function or cursor direction key is pressed.

NOTES:
A <line number> of 0 disables the event trap.

The ON KEY statement will only be executed if a KEY(n) ON statement has already been executed (see "KEY(n) Statement" earlier in this section) to enable event trapping. If event trapping is enabled, and if the <line number> in the ON KEY statement is not 0, GW-BASIC checks between statements to see if the specified function or cursor direction key has been pressed. If it has, a GOSUB will be performed to the specified line.

If a KEY(n) OFF statement has been executed for the specified key, (see "KEY(n) Statement"), the keystroke is not discovered and the GOSUB is not performed.

If a KEY STOP statement has been executed for the specified key, (see "KEY(n) Statement"), the GOSUB is not performed, but will be remembered and performed as soon as a KEY(n) ON statement is executed.

When the ON KEY subroutine is performed, an automatic KEY(n) STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a KEY(n) ON statement unless an explicit KEY(n) OFF was performed inside the subroutine.

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GW-BASIC is not executing a program. Also, event trapping is automatically disabled when an error trap occurs.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

EXAMPLE:

```
10 KEY 4,"SCREEN 0,0" 'assigns softkey 4
20 KEY(4) ON 'enables event trapping
.
.
.
70 ON KEY(4) GOSUB 200
.
.
.
(key 4 pressed)
.
.
.
200 'Subroutine for screen
```

ON PLAY STATEMENT

PURPOSE:
Specifies the first line of a subroutine to be performed when the number of notes in the background music queue goes below a specified value. This statement enables continuous music during program execution.

SYNTAX:
ON PLAY(n) GOSUB <line number>

(n) is an integer expression. The play event subroutine is called whenever the size of the background music queue decreases from n to n−1.

<line number> is the statement line number of the play event trap subroutine.

NOTES:
The following rules apply to event trapping on the background music queue:

1. A play event trap is issued only when playing background music (in other words, PLAY "MB..."). Play event traps are not issued when running in Music Foreground (in other words, default case, or PLAY "MF...").

2. A play event trap is not issued if the background music queue is already empty when a PLAY ON is executed.

3. Choose conservative values for (n). An ON PLAY(32) statement will cause event traps so often that there will be little time to execute the rest of your program.

The ON PLAY statement uses only voice 1 to test the number of notes in the background music queue.

See also the PLAY statement for information on the PLAY ON, PLAY OFF, and PLAY STOP statements.

EXAMPLE:

```
ON PLAY (5) GOSUB 900
```

executes the subroutine at line 900 whenever the background music queue decreases from five notes to 4.

ON STRIG STATEMENT

PURPOSE:
Specifies the first line number of a subroutine to be performed when the joystick trigger or mouse switch is pressed.

SYNTAX:
ON STRIG(n) GOSUB <line number>

(n) is an even integer in the range 0 to 6 as follows:

n	Joystick or Mouse	Switch
0	A	right
2	A	left
4	B	right
6	B	left

<line number> is the number of the first line of a subroutine to be performed when the joystick trigger or mouse switch is pressed.

NOTES:
A <line number> of 0 disables the event trap.

The ON STRIG statement will only be executed if a STRIG ON statement has already been executed (see "STRIG Statement/ Function" later in this section) to enable event trapping. If event trapping is enabled, and if the <line number> in the ON STRIG statement is not 0, GW-BASIC checks between statements to see if the joystick trigger or mouse switch has been pressed. If it has, a GOSUB will be performed to the specified line.

If a STRIG OFF statement has been executed (see "STRIG Statement") and a trigger or button is pressed, the event is not discovered and the GOSUB is not performed.

If a STRIG STOP statement has been executed (see "STRIG Statement"), the GOSUB is not performed, but will be performed as soon as a STRIG ON statement is executed.

When an event trap occurs (in other words, when the GOSUB is performed), an automatic STRIG STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a STRIG ON statement unless an explicit STRIG OFF was performed inside the subroutine.

continued on next page

The RETURN <line number> form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GW-BASIC is not executing a program. Also, event trapping is automatically disabled when an error trap occurs.

EXAMPLE:
The statement:

```
10 ON STRIG(0) GOSUB 1000
```

will cause GW-BASIC to transfer control to line 1000 when the system detects that the button has been pressed on the first joystick.

ON TIMER, TIMER ON, TIMER OFF, TIMER STOP STATEMENTS

PURPOSE:
Causes periodic event traps.

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

(n) is a numeric expression in the range 1 through 86400 (1 second through 24 hours). Values outside this range result in an "Illegal function call" error.

<line number> is the statement line number of the TIMER event trap subroutine.

TIMER ON	Enables TIMER event trapping.
TIMER OFF	Disables TIMER event trapping.
TIMER STOP	Suspends TIMER event trapping.

NOTES:
These statements provide an interval timer for applications programs.

The TIMER ON statement enables event trapping on the TIMER. After TIMER ON is executed, GW-BASIC transfers control every (n) seconds to the subroutine specified by the ON TIMER statement.

TIMER OFF disables TIMER event trapping and TIMER STOP suspends TIMER event trapping. If the specified time elapses after a TIMER STOP statement is executed, then GW-BASIC executes the ON TIMER subroutine immediately after the next TIMER ON statement.

EXAMPLE:
Display the time of day on line 1 every minute.

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
                .
                .
                .
10000 OLD_ROW=CSRLIN 'Save current Row
10010 OLD_COL=POS(0) 'Save current Column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLD_ROW,OLD—COL 'Restore Row and Column
10040 RETURN
```

OPEN STATEMENT

PURPOSE:
Allows I/O to a file or device.

SYNTAX:
OPEN <mode>,[#]<file number>,<filespec>[,<record length>]

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

<mode> is a string expression whose first character is one of the following:

Character	Meaning
O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.
A	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend (append) the file.

If <mode> is omitted, the default random access mode is assumed.

<file number> is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filespec> is a string expression containing a name that conforms to the rules for disk filenames of the operating system.

OPEN also allows <pathname> in place of <filespec>. For example, if MARY is your current directory, then:

```
OPEN "REPORT" ...
OPEN "\SALES\MARY\REPORT" ...
OPEN "..\MARY\REPORT" ...
OPEN "..\..\MARY\REPORT" ...
```

all refer to the same file.

Because you can refer to the same file in a subdirectory by using different paths, it is nearly impossible for BASIC to know from looking at the path that the file is the same. For this reason, BASIC will not let you OPEN the file for OUTPUT or APPEND if it is already OPEN, even if the path is different.

<record length> is an integer expression that, if included, sets the record length for random files. Do not use this option with sequential files.

The <record length> cannot exceed the maximum set with /S: at start-up. If the <record length> option is not used, the default length is 128 bytes.

NOTES:
A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

A RAM cartridge file can be OPENed for sequential access only.

EXAMPLES:

```
10 OPEN "I",2,"INVEN"

10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

OPEN COM STATEMENT

PURPOSE:
Opens and initializes a communications channel for input/output.

SYNTAX:
OPEN "COMn: [<speed>][,[<parity>][,[<data>][,[<stop>][,RS][,CS[m]][,DS[m]][,CD[m]][,BIN][,ASC][,LF]]]]" AS [#]<device number>

COMn: is the name of the device to be opened. The value of n must be 1, 2, or 3.

<speed> is the baud rate, in bits per second, of the device to be opened. The value <speed> must be 110, 300, 600, 1200, 2400, 4800, or 9600.

<parity> designates the parity of the device to be opened. Valid entries are: N (none), E (even), or O (odd).

<data> designates the number of bits per byte. Valid entries are 7 or 8.

<stop> designates the stop bit. The value of this parameter must be 1 or 2.

RS suppresses the RTS (Request To Send) line signal.

CS[m] controls the CTS (Clear To Send) line signal.

DS[m] controls the DSR (Data Set Ready) line signal.

CD[m] controls the CD (Carrier Detect) line signal.

LF specifies that a linefeed is to be sent after a carriage return. See "NOTES" for further discussion of LF.

BIN opens the device in binary mode. BIN is selected by default unless ASC is specified. See "NOTES" for further discussion of BIN.

ASC opens the device in ASCII mode. See "NOTES" for further discussion of ASC.

<device number> is the number of the device to be opened.

NOTES:
The OPEN COM statement must be executed before a device can be used for RS-232-C communication.

Any syntax errors in the OPEN COM statement will result in a "Bad file name" error. The incorrect parameter will not be shown.

A "Device timeout" error will occur if Data Set Ready (DSR) is not detected.

The <speed>, <parity>, <data>, and <stop> options must be listed in the order shown in the above syntax. The remaining options may be listed in any order, but they must be listed after the <speed>, <parity>, <data>, and <stop> options.

The m argument for the CS[m], DS[m], and CD[m] options specifies an interval in milliseconds. After executing an OPEN COM statement, GW-BASIC checks the specified line signal for this amount of time before returning a "Device timeout" error. The value of m can range from 0 to 65535. The status of these line signals is not checked if m is set to 0 or if m is omitted.

LF allows communication files to be printed on a serial line printer. When LF is specified, a linefeed character (0AH) is automatically sent after each carriage return character (0CH), including 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 expanded to spaces, a carriage return is not forced at the end-of-line, and CTRL-Z is not interpreted as the end-of-file and CTRL-Z is not sent when the channel is closed.

In ASC mode, tabs are expanded, carriage returns are forced at the end-of-line, CTRL-Z is treated as the end-of-file, XON/XOFF protocol is supported (if enabled), and CTRL-Z is sent when the channel is closed.

EXAMPLE:
The statement:

```
10 OPEN "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 and output will be in the binary mode. Other lines in the program may now access channel 1 as device number 2.

OPTION BASE STATEMENT

PURPOSE:
Declares the minimum value for array subscripts.

SYNTAX:
OPTION BASE n

n is 1 or 0.

NOTES:
The default base is 0. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is 1.

The OPTION BASE statement must be coded before you define or use any arrays.

A CHAINed program may have an OPTION BASE statement only if no arrays are passed. The CHAINed program inherits the OPTION BASE value of the CHAINing program.

EXAMPLE:

```
10 OPTION BASE 1
```

OUT STATEMENT

PURPOSE:
Sends a byte to a machine output port.

SYNTAX:
OUT I,J

I is the port number. It must be an integer expression in the range 0 to 65535.

J is the data to be transmitted. It must be an integer expression in the range 0 to 255.

EXAMPLE:
The BASIC statement:

```
100 OUT 12345,255
```

is equivalent to the 8086 assembly language routine:

```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

PAINT STATEMENT

PURPOSE:
Fills a graphics figure with the color specified.

SYNTAX:
PAINT (<xstart>,<ystart>)[,<paint color>[,<border color>]
[,<background>]]

<xstart> and <ystart> are the coordinates where painting is to begin. Painting should always start on a non-border point.

<paint color> is the number of the color to be placed in the filled area of the figure (see "COLOR Statement" earlier in this section). If the <paint color> is not specified, the foreground color will be used.

<border color> identifies the border color of the figure to be filled. When the border color is encountered, painting of the current line and direction will stop. If the <border color> is not specified, the <paint color> will be used.

<background> is used in paint tiling (see "NOTES").

NOTES:
Painting is complete when a line is painted without changing the color of any pixel; in other words, the entire line is equal to the paint color.

The PAINT 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.

PAINT supports "tiling". Like LINE, PAINT looks at a "tiling" mask each time a point is put down on the screen. If <paint color> is a string formula, then "tiling" is performed as follows.

The tile mask is always 8 bits wide and may be from 1 to 64 bytes long. Each byte in the tile string masks 8 bits along the X axis when putting down points. Each byte of the tile string is rotated as required to align along the Y axis such that tile_byte_mask = Y MOD tile_length.

Rotating each byte of the tile string is done so that the tile pattern is replicated uniformly over the entire screen (as if the statement PAINT (0,0)... were used).

Tiling effects change as a function of the screen mode. We will use screen mode 1 as an example. Because there are 2 bits per pixel in mode 1 (see the sample screen below), each byte of the tile pattern only describes 4 pixels. In this case, every 2 bits of the tile byte describe one of the four possible colors associated with each of the 4 pixels to be put down.

```
          x increases -->
          Bit of tile byte
X,Y   8  7  6  5  4  3  2  1
0,0   x  x  x  x  x  x  x  x   Tile byte 1
0,1   x  x  x  x  x  x  x  x   Tile byte 2
0,2   x  x  x  x  x  x  x  x   Tile byte 3
      .
      .
      .
0,63  x  x  x  x  x  x  x  x   Tile byte 64 (maximum allowed)
```

In modes with a resolution of 1 bit per pixel (modes 2, 3, and 7) (see the sample screen below), the screen can be painted with 'x's by the following statement:

```
PAINT
(320,100),CHR$(&H81)+CHR$(&H42)+CHR$(&H24)+CHR$
(&H18)+CHR$(&H18)+CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

This statement appears on the screen as:

```
          x increases -->

X,Y   Bit of tile byte
0,0   x           x    CHR$(&H81)   Tile byte 1
0,1    x         x     CHR$(&H42)   Tile byte 2
0,2     x       x      CHR$(&H24)   Tile byte 3
0,3      xx            CHR$(&H18)   Tile byte 4
0,4      xx            CHR$(&H18)   Tile byte 5
0,5     x   x          CHR$(&H24)   Tile byte 6
0,6    x     x         CHR$(&H42)   Tile byte 7
0,7   x       x        CHR$(&H81)   Tile byte 8
```

<background> is a string formula returning one character. When omitted, the default is CHR$(0).

When supplied, <background> specifies the "background tile" pattern or color byte to skip when checking for boundary termination.

continued on next page

Occasionally, you may want to tile-paint over an already painted area that is the same color as two consecutive lines in the tile pattern. Normally, paint quits when it encounters two consecutive lines of the same color as the point being set (the point is surrounded). It would not be possible to draw alternating blue and red lines on a red background without the <background>. Paint would stop as soon as the first red pixel was drawn. Specifying red [CHR$(&HAA)] as the background_attribute allows the red line to be drawn over the red background.

You cannot specify more than two consecutive bytes in the tile string that match the <background>. Specifying more than two will result in an "Illegal function call" error.

EXAMPLE:
The statement:

```
10 PAINT (5,15),2,0
```

paints the figure at coordinates 5,15 with color 2 and border color 0.

PALETTE STATEMENT

PURPOSE:
Changes one of the colors in the palette.

SYNTAX:
PALETTE [<palette number>, <color number>]

<palette number> is the number of the color being replaced by the color referred to by <color number>. The range for <palette number> is 0 to 15 and the range for <color number> is −1 to 15.

NOTES:
The color palette is a table consisting of 16 integer (two-byte) values. When you first power-up your Mindset Personal Computer, the system uses default values for <palette number> and <color number>. The system also uses the default values when you use the PALETTE statement without specifying the palette number or color number. Table 6-3 describes the default color palette. See the PALETTE USING statement for an explanation of the hexadecimal color values.

Table 6-3: Default Color Palette Definition

Palette Index	Color Name	Hex Value	Palette Index	Color Name	Hex Value
00	Black	0000	08	Dark gray	8092
01	Blue	11C8	09	Light blue	91D8
02	Green	2020	10	Light green	A03B
03	Cyan	3160	11	Light cyan	B1FA
04	Red	4004	12	Light red	C007
05	Magenta	5144	13	Light magenta	D1CE
06	Brown	6024	14	Yellow	E037
07	Light gray	7124	15	White	F1FF

GW-BASIC maintains two color palettes: a reference palette and a user's palette. The availability of a reference palette enables a program to rearrange palette colors in any order. The reference palette remains constant, thus providing the default 16 colors listed in Table 6-3 (regardless of changes you make to your palette).

The <color number> index selects a color from the reference palette. The <palette number> index selects a color in the user's palette to be replaced with the color selected by the <color number> index. GW-BASIC uses the colors in the user's palette for the screen display.

continued on next page

A value of −1 for the <color number> parameter indicates that the existing color (at the <palette number> location in the user's palette) is not to be modified. Negative numbers other than −1 are illegal for the <color number> parameter.

EXAMPLE:
The statement:

```
PALETTE 0,2
```

changes color 0 in the user's palette to color 2 from the reference palette.

PALETTE USING STATEMENT

PURPOSE:
Specifies new colors by assigning a set of array values to the user's palette.

SYNTAX:
PALETTE USING <array name>(<array index>)

<array name> is the name of the array containing new information for the user's color palette. <array index> specifies the element in the array to use as the first position in the user's palette.

NOTES:
The color palette is a table consisting of 16 integer (two-byte) values. When you first power-up your Mindset Personal Computer, the system uses default values for the color palette and color numbers. See Table 6-3 under the description of PALETTE for the default color palette arrangement.

GW-BASIC maintains two color palettes: a reference palette and a user's palette. The reference palette remains constant, thus providing the default 16 colors listed in Table 6-3 (regardless of changes you make to your palette).

The PALETTE USING statement transfers color values from the array to the user's palette. The <array index> parameter determines which element in the array becomes the first element in the user's palette.

The PALETTE USING statement continues transferring values from the array to the user's palette until all 16 color values are filled. GW-BASIC checks to make sure that the array contains enough elements to completely fill the palette.

If a <color number> of −1 was assigned to a position with the PALETTE statement, the color in that position will not be changed by a subsequent PALETTE USING statement. Also, a value of −1 in the array causes the corresponding color in the user's palette to remain unchanged.

The colors in the palette are initially the same for operation with either an RGB color monitor or a television. Use an array to rearrange the user's palette using the colors from the reference palette. This array should contain numbers in the range 0 to 15 in the upper 4 bits to specify the RGB colors for the user's palette. The number of colors for an RGB monitor is fixed at 16, but the user can specify up to 512 different colors for operation with a television. The lower 9 bits in each palette definition

continued on next page

word select the color for operation with a television. A complete application program should set the colors for both RGB monitors and televisions to insure compatibility with all systems.

The diagram below shows the format for the 16-bit word that defines each color in the user's palette. See Table 6-3 in the description of the PALETTE statement for a list of the default hexadecimal color values.

```
Bit:      15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
Color:     I  R  G  B        B  B  B  G  G  G  R  R  R
Display:  RGB Monitor              Television
```

Three bits control each primary color for television operation. Each bit is worth twice that of the next lower bit for a given color. For example, bit 0 sets the value of the red component to 1, bit 1 adds 2 to the red value, and bit 2 adds 4. Therefore, the palette entry for the most intense red is 7. Similarly, the value for the most intense white requires all 9 bits to be set for a value of 511.

EXAMPLE:
The statement:

```
10 PALETTE USING P (0)
```

causes these changes in the user palette:

Index	Initial Palette	Array P	Updated Palette
00	&H4004	&H7124	&H7124
01	&HC007	&HC007	&HC007
02	&HA03B	&H2020	&H2020
03	&H7124	&HFFFF	&H7124
04	&H81FA	&H11C8	&H11C8
05	&HE037	&H5144	&H5144
.	.	.	.
.	.	.	.
.	.	.	.
15	&H3160	&D1CE	&HD1CE

Note that &HFFFF (the fourth element of array P) is equal to −1 and causes the corresponding element in the initial palette to remain unchanged.

PEEK FUNCTION

PURPOSE:
Returns the byte read from memory location (I).

SYNTAX:
X = PEEK(I)

NOTES:
The returned value is an integer in the range 0 to 255. I must be in the range 0 to 65535. I is the offset from the current segment, which was defined by the last DEF SEG statement. For the interpretation of a negative value of I, see "VARPTR Function" later in this section.

PEEK is the complementary function of the POKE statement.

EXAMPLE:

```
A=PEEK(&H5A00)
```

PLAY STATEMENT

PURPOSE:
Plays music as specified by <string expression>.

The PLAY ON statement enables event trapping on the background music queue. The PLAY OFF statement disables event trapping on the background music queue. The PLAY STOP statement suspends event trapping on this queue.

SYNTAX:
PLAY [<string1>][,<string2>[,<string3>[,<string4>]]]
PLAY ON
PLAY OFF
PLAY STOP

<string1> is a string expression which controls voice 1. This expression consists of one or more of the PLAY statement subcommands listed under "NOTES".

<string2>,<string3>, and <string4>, control the other 3 voices which may be used simultaneously with voice 1.

NOTES:
PLAY uses a concept similar to that in DRAW 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 subcommands used for <string expression> are:

Subcommand	Explanation
>	Increments one octave. This subcommand will not increment above octave 6.
<	Decrements one octave. This subcommand will not decrement below octave 0.
A-G [# | +, −]	Plays a note in the range A-G. The symbol # or + after the note specifies sharp; the symbol − specifies flat.

Subcommand	Explanation
L <n>	Sets the length of each note. L 4 is a quarter note, L 1 is a whole note, and so on. 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, A 16 is equivalent to L 16 A.
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. MF is the default setting.
MB	Music (PLAY statement) and SOUND are set 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. The maximum number of notes which may be played as background music queue is 32 including the commands which control the shape of the envelope for a voice.
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 (because in the 7 possible octaves there are 84 notes). n = 0 means a rest.
O <n>	Sets the current octave. n selects one of seven octaves, numbered 0 through 6.
P <n>	Specifies a pause value n, ranging from 1 through 64. This option corresponds to the length of each note, set with L <n>.

continued on next page

Subcommand	Explanation
T <n>	Sets the "tempo", or the number of L 4's in one second. The integer 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, and so on. Periods may appear after a pause (P). In this case, the pause length is scaled in the same way notes are scaled.
V <n>	Specifies the volume level for a voice using a value from 0 to 255. The loudest volume, 255, is the default. If the QS<n> parameter is used to control the envelope shape for each note, then the V<n> parameter specifies the initial volume for that note in the attack/decay cycle.
QS <n>	Defines the shape of the envelope for each note played by the specified voice. The range of <n> can be from 0 to 7. The default value of 0 specifies that no attack/decay processing takes place. A value of 7 specifies the maximum attack/decay processing. During the attack portion of the attack/decay cycle, the voice begins playing each note at the volume specified with the V<n> parameter. The volume increases to 255 at a rate proportional to the value of the QS<n> parameter. After playing the note for the time specified by the I<n> parameter, the volume of the note decays to 0 at a rate proportional to the value of the QS<n> parameter.
X <string>	Executes a substring.

The ON PLAY statement specifies a value and a subroutine line number. The PLAY ON statement enables event trapping to cause GW-BASIC to execute the subroutine whenever the number of notes in the background music queue goes below the specified value. The ON PLAY statement uses only voice 1 to test the number of notes in the background music queue.

PLAY OFF disables event trapping and PLAY STOP suspends event trapping. If the number of notes in the background music queue goes below the specified value after a PLAY STOP statement, the subroutine is executed immediately after the next PLAY ON statement.

EXAMPLES:

```
PLAY "XA$;"

PLAY "X" + VARPTR$(A$)
```

PLAY FUNCTION

PURPOSE:
Returns the number of notes currently in the background music queue.

SYNTAX:
I = PLAY(n)

<n> is a dummy argument and may be any value.

NOTES:
PLAY(n) will return 0 when in Music Foreground mode.

EXAMPLE:
The function:

```
I=PLAY(1)
```

places in I the number of notes currently in the background music queue.

PMAP FUNCTION

PURPOSE:
Maps logical (world) coordinates to physical coordinates or physical coordinates to world coordinates.

SYNTAX:
X = PMAP (I,n)

The parameters for PMAP are as follows:

n	I	Returned Value
0	world coordinate X	physical coordinate X
1	world coordinate Y	physical coordinate Y
2	physical coordinate X	world coordinate X
3	physical coordinate Y	world coordinate Y

NOTES:
The WINDOW statement defines the world coordinates and the VIEW statement defines the physical coordinates. PMAP provides a simple method of translating coordinate specifications from one system to the other.

EXAMPLE:
The statement:

```
10 X = PMAP (32,2)
```

translates the physical X coordinate of 32 to the corresponding world X coordinate. The correspondence is determined by the WINDOW and VIEW statements.

POINT FUNCTION

PURPOSE:
Reads the color value of a pixel from the screen. If the specified point is out of range, the value −1 is returned.

A second form of POINT returns the current graphics accumulator values.

SYNTAX 1:
I = POINT (<xcoordinate>,<ycoordinate>)

<xcoordinate> and <ycoordinate> are the coordinates of the pixel to be referred to.

SYNTAX 2:
I = POINT (n)

The parameter n returns a coordinate value as follows:

n	Returned Value
0	Current physical X coordinate.
1	Current physical Y coordinate.
2	Current world X coordinate if WINDOW is active. Otherwise, returns the same value as if n = 0.
3	Current world Y coordinate if WINDOW is active. Otherwise return the same value as if n = 1.

EXAMPLES:

```
10 SCREEN 1
20 FOR C=0 TO 3
30 PSET (10,10),C
40 IF POINT(10,10)<>C THEN PRINT "Broken!"
50 NEXT C

5 SCREEN 2
10 IF POINT(I,I)<>0 THEN PRESET (I,I)
ELSE PSET (I,I)
'invert current state of a point
20 PSET (I,I),1-POINT(I,I) 'another way to invert a point
```

POKE STATEMENT

PURPOSE:
Writes a byte into a memory location.

SYNTAX:
POKE I,J

I and J are integer expressions.

NOTES:
I and J are integer expressions. The expression I represents the address of the memory location and J is the data byte. I must be in the range −32768 to 65535. I is the offset from the current segment, which was set by the last DEF SEG statement. For the interpretation of the negative values of I, see "VARPTR Function" later in this section.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

Warning: Use POKE carefully. If it is used incorrectly, it can cause GW-BASIC to crash.

EXAMPLE:

```
10 POKE &H5A00,&HFF
```

POS FUNCTION

PURPOSE:
Returns the current horizontal (column) position of the cursor.

SYNTAX:
I = POS(X)

NOTES:
The leftmost position is 1. X is a dummy argument. To return the current line position of the cursor, use the CSRLIN function.

Also see "LPOS Function".

EXAMPLE:

```
IF POS(X)>60 THEN BEEP
```

PRESET STATEMENT

PURPOSE:
Draws a specified point on the screen. PRESET works exactly like PSET, except that if the <color> is not specified, the background color is selected.

SYNTAX:
PRESET (<xcoordinate>,<ycoordinate>)[,<color>]

<xcoordinate> and <ycoordinate> specify the absolute location of the pixel to be set.

<color> is the number assigned to the color to be used for the specified point.

NOTES:
If an out-of-range coordinate is given, clipping occurs. Points plotted outside the screen or viewport limits do not appear.

Coordinates can be shown as absolutes, as in the preceding syntax, or the STEP option can be used to refer to a point relative to the most recent point used. The syntax of the STEP option is:

STEP (<xoffset>,<xoffset>)

For example, if the most recent point referred to were (0,0), STEP (10,0) would refer to a point offset 10 from X and 0 from Y.

EXAMPLE:

```
5 REM DRAW A LINE FROM (0,0) TO (100,100)
10 FOR I=0 TO 100
20 PRESET (I,I),1
30 NEXT

35 REM NOW ERASE THAT LINE
40 FOR I=0 TO 100
50 PRESET STEP (-1,-1)
60 NEXT
```

This example draws a line from (0,0) to (100,100) and then erases that line by overwriting it with the background color.

PRINT STATEMENT

PURPOSE:
Displays data on the screen.

SYNTAX:
PRINT [<list of expressions>]

NOTES:
If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed on the screen. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.) Quotation marks cannot be used as part of the string because they are used as string delimiters.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. GW-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 exceeds the number of screen columns, GW-BASIC continues printing on the next physical line.

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 printed using the unscaled format. For example, 1E-7 is printed as .0000001 and 1E-8 is printed 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 printed using the unscaled format. For example, 1D-15 is printed as .0000000000000001 and 1D-16 is printed as 1D-16.

Substitutes for PRINT

A question mark may be used in place of the word PRINT in a PRINT statement.

EXAMPLES:
In Example 1, the routine:

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
```

will yield:

```
10          0          -25          3125
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

In Example 2, the routine:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
```

will yield:

```
? 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 the PRINT statements on lines 20 and 30 to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

In Example 3, the routine:

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
```

will yield:

```
5 10 10 20 15 30 20 40 25 50
```

continued on next page

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Each number is 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.

PRINT USING STATEMENT

PURPOSE:
Prints strings or numbers using a specified format.

SYNTAX:
PRINT USING <string exp>;<list of expressions>

NOTES:
<string exp> is a string literal (or variable) composed of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

<list of expressions> is composed of the string expressions or numeric expressions to be printed, separated by a comma or a semicolon.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field.

Character	**Meaning**
!	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.
&	Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field.

continued on next page

Character	Meaning

#	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. For example:
	
	`PRINT USING "##.##";.78`
	`0.78`
	
	`PRINT USING "###.##";987.654`
	`987.65`
	
	`PRINT USING "##.##  ";10.2,  5.3,  66.789,  .234`
	`10.20  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. For example:
	
	`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. For example:
	
	`PRINT USING "**#.# ";12.39,  -0.9,765.1`
	`*12.4 * -0.9 765.1`

Character	Meaning
$$	A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is for the dollar sign itself. The exponential format cannot be used with the $$. Negative numbers cannot be used unless the minus sign trails to the right. For example: PRINT USING "$$###.##"; 456.78 $456.78
**$$	The **$ at the beginning of a format string combines the effect 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 for the dollar sign. The exponential format cannot be used with **$. When negative numbers are printed, the minus sign will appear immediately to the left of the dollar sign. For example: PRINT USING "**$##.##";2.34 ***$2.34 A comma placed immediately 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 placed 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. For example: PRINT USING "####,.##";1234.5 1,234.50 PRINT USING "####.##,";1234.5 1234.50,
^^^^	Four carets may be placed after the digit-position characters to specify exponential format. The four carets 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.

continued on next page

Character	Meaning
	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. For example: 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. For example: PRINT USING "_!##.##_!";12.34 !12.34! The literal character itself may be an underscore by placing "_ _" (two underscores) 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. For example: 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.

EXAMPLES:

In Example 1, the statement

```
10 A$ = "LOOK":B$ = "OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\   \";A$;B$
50 PRINT USING "\     \";A$;B$;"!!"
```

will yield:

```
LO
LOOKOUT
LOOK OUT  !!
```

In Example 2, the statement

```
10 A$ = "LOOK":B$ = "OUT"
20 PRINT USING   "!";A$;
30 PRINT USING   "&";B$
```

will yield :

```
LOUT
```

PRINT # and **PRINT # USING** STATEMENTS

PURPOSE:
Writes data to a sequential file.

SYNTAX:
PRINT #<file number>,[USING <string exp>;]
<list of expressions>

NOTES:
<file number> is the number used when the file was OPENed for output. <string exp> consists of formatting characters, as described under the "PRINT USING Statement". The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT # does not compress data. 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, be careful to delimit the data so it will be input correctly.

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, any extra blanks 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 in 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 string 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 now 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 linefeeds, write them to the file surrounded by explicit quotation marks or the CHR$ equivalent, 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$. To separate these strings properly in the file, write double quotation marks to the file image using CHR$(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"
```

Finally, 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 file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

See also "WRITE # Statement" later in this section.

PSET STATEMENT

PURPOSE:
Draws a point on the screen at the specified point in the specified color.

SYNTAX:
PSET (<xcoordinate>,<ycoordinate>)[,<color>]

<xcoordinate> and <ycoordinate> are absolute coordinates which specify the point on the screen to be colored.

<color> is the number of the color to be used.

NOTES:
When GW-BASIC scans coordinate values, it allows them to extend 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. Therefore, the bottom left corner of the screen is 0,199 in screen modes 1 through 5 and 0,399 in screen modes 6 and 7.

PSET allows the <color> to be left off the command line. If <color> is omitted, the default is the foreground color.

Coordinates can be shown as offsets by using the STEP option to refer to a point relative to the most recent point used. The syntax of the STEP option is:

STEP (<xoffset>,<yoffset>)

<xoffset> and <yoffset> are relative coordinates which specify the point on the screen to be colored.

For example, if the most recent point specified were (0,0), PSET STEP (10,0) would refer to a point at offset 10 from X and offset 0 from Y.

EXAMPLE:

```
5 REM DRAW A SERIES OF POINTS FROM (0,0) TO (100,100)
10 FOR I=0 TO 100
20 PSET (I,I)
30 NEXT

35 REM NOW ERASE THOSE POINTS
40 FOR I=0 TO 100
50 PSET STEP (-1,-1),0
60 NEXT
```

This example draws a series of points from (0,0) to (100,100) and then erases them by overwriting it with the background color.

PUT STATEMENT (FILES)

PURPOSE:
Writes a record from a random buffer to a random access file.

SYNTAX:
PUT [#]<file number>[,<record number>]

NOTES:
<file number> is the number under which the file was OPENed. If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 16,777,215. The smallest record number is 1.

The GET and PUT statements allow fixed-length input and output for GW-BASIC COM files. However, because of the low performance associated with telephone line communications, you should normally not use GET and PUT for telephone communication.

Note: PRINT #, PRINT # USING, and WRITE # may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE #, GW-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.

PUT STATEMENT (GRAPHICS)

PURPOSE:
Used with the GET statement to transfer graphic images to and from the screen.

SYNTAX:
PUT (X1,Y1),<array name>[,<action verb>]

(X1,Y1) specifies the point where a stored image is to be displayed on the screen. The specified point is the coordinate of the top left corner of the image. If the image to be transferred is too large to fit on the screen, an "Illegal function call" error will result.

<array name> is the name of the array in which the object image was stored with a graphics GET statement.

<action verb> is one of the following: PSET, PRESET, AND, OR, or XOR.

PSET transfers the data onto the screen verbatim.

PRESET is the same as PSET, except that all points in the image are inverted.

AND is used to transfer points from the array only where points already exist from another image on the screen.

OR is used to superimpose the image onto an existing image.

XOR causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like that of the cursor. When an image is PUT against a complex background twice, the background is restored unchanged. Thus, a user can move an object around the screen without obliterating the background.

The default <action verb> is XOR.

NOTES:
The graphics version of the PUT statement is the complement of the graphics GET statement. It enables a program to transfer an object to the screen from an object array.

<action verb> enables the program to specify the interaction between the stored image and images already present on the screen. The <action verb> has different effects in different screen modes. The color changes caused by PRESET, AND, OR, and XOR depend on the number of bits per pixel in each mode.

continued on next page

EXAMPLE:
The statement:

```
10 PUT (10,10),FIGURE1, OR
```

places the object stored in the FIGURE1 array on the screen. The statement places the upper left corner of the rectangle containing the object at screen location (10,10). The <action verb>, OR, causes the image to be superimposed over any images which are already on the screen.

RANDOMIZE STATEMENT

PURPOSE:
Reseeds the random number generator.

SYNTAX:
RANDOMIZE [<expression>]

NOTES:
If <expression> is omitted, GW-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.

Use the statement RANDOMIZE TIMER to get a new random seed without user input.

EXAMPLES:
The routine:

```
10 RANDOMIZE
20 FOR I = 1 TO 5
30 PRINT RND;
40 NEXT I

RUN
```

will yield:

```
Random number seed (-32768 to 32767)? 6
  .4417627      .1085309      .182628       .9246312      .2432385
Ok
RUN
Random number seed (-32768 to 32767)? 7
  .84815        .279994       .496364       .4990483      .9731121
Ok
RUN
Random number seed (-32768 to 32767)? 6
  .4417627      .1085309      .182628       .9246312      .2432385
Ok
```

continued on next page

Note that the numbers your program produces may not be the same as the ones shown here.

READ STATEMENT

PURPOSE:
Reads values from a DATA statement and assigns them to variables. (See "DATA Statement" earlier in this section for more information.)

SYNTAX:
READ <list of variables>

NOTES:
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 variables, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

The READ statement can act upon DATA statements in one of two ways: either 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 message is printed. 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.

To reread DATA statements from the start, use the RESTORE statement (see "RESTORE Statement" later in this section).

EXAMPLES:
In Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

this program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

continued on next page

In Example 2, the routine:

```
10 PRINT "CITY", "STATE",   " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,",   COLORADO,   80211
40 PRINT C$,S$,Z
```

reads string and numeric data from the DATA statement in line 30 and yields:

```
CITY       STATE          ZIP
DENVER,  COLORADO    80211
```

REM STATEMENT

PURPOSE:
Allows explanatory remarks to be inserted in a program.

SYNTAX:
REM [<remark>]

NOTES:
REM statements are not executed but are output exactly as entered when the program is listed.

A GOTO or GOSUB statement may branch into REM statements. 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.

Do not use REM in a DATA statement, because it would be considered legal data.

EXAMPLES:

```
.
.
.
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
.
.
.
```

RENUM COMMAND

PURPOSE:
Renumbers program lines.

SYNTAX:
RENUM [[<new number>][,[<old number>][,<increment>]]]

NOTES:
<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.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL test statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

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.

EXAMPLES:

```
RENUM
```

Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.

```
RENUM 300, ,50
```

Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up, starting with line number 1000 and proceeding in increments of 20.

RESET COMMAND

PURPOSE:
Closes all files on all drives and RAM cartridges.

SYNTAX:
RESET

NOTES:
RESET closes all open files on all drives and RAM cartridges and writes the directory track to every disk with open files.

All files must be closed before a disk is removed from its drive.

EXAMPLE:

```
RESET
```

RESTORE STATEMENT

PURPOSE:
Allows DATA statements to be reread from a specified line.

SYNTAX:
RESTORE [<line number>]

NOTES:
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
50 PRINT A;B;C;D;E;F;
RUN
```

produces the following output:

```
57 68 79 57 68 79
```

RESUME STATEMENT

PURPOSE:
Continues program execution after an error recovery procedure has been performed.

SYNTAX:
RESUME [0]

RESUME NEXT

RESUME <line number>

NOTES:
Any one of the syntax forms shown above may be used, depending upon where execution is to resume:

Syntax	**Explanation**
RESUME [0]	Execution resumes at the statement that caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error handling 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 80
 .
 .
 .
```

RETURN STATEMENT

See "GOSUB...RETURN Statements".

RIGHT$ FUNCTION

PURPOSE:
Returns the rightmost I characters of string X$.

SYNTAX:
Y$ = RIGHT$(X$,I)

NOTES:
If I is greater than or equal to the number of characters in X$ (LEN(X$)), then RIGHT$(X$,I) returns X$. If I = 0, the null string (length zero) is returned.

Also see the LEFT$ and MID$ functions described earlier in this section.

EXAMPLE:
The function:

```
10 A$ = "DISK BASIC"
20 PRINT RIGHT$(A$,5)
```

will yield:

```
BASIC
```

RMDIR STATEMENT

PURPOSE:
Removes a directory from the MS-DOS file system.

SYNTAX:
RMDIR <pathname>

<pathname> is a string expression not exceeding 128 characters identifying the subdirectory to be removed from its parent directory.

NOTES:
RMDIR works exactly like the MS-DOS command RMDIR.

EXAMPLES:
Assume that the current directory is \SALES\JOHN:

```
RMDIR "\ACCOUNTING"  ' Delete subdirectory ACCOUNTING
from the SALES\JOHN directory
```

or:

```
RMDIR  "..\..\INVENTORY"
```

Possible Errors:

"Bad file name"
"Path/File Access error" - usually indicating that the directory is not specified properly.

The subdirectory to be deleted must be empty of all files except '.' and '..'; otherwise, a "Path not found" error is given.

RND FUNCTION

PURPOSE:
Returns a random number between 0 and 1.

SYNTAX:
Y = RND[(X)]

NOTES:
The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see "RANDOMIZE Statement" earlier in this section).

X <0 always restarts the same sequence for any given X. X > 0 or X omitted generates the next random number in the sequence. X = 0 repeats the last number generated.

EXAMPLE:
The function:

```
10 FOR  I=1  TO  5
20 PRINT  INT(RND*100);
30 NEXT
```

could yield:

```
24 30 31 51 5
```

RUN COMMAND

PURPOSE:
Executes the program currently in memory (see Syntax 1). Also loads a file from disk into memory and runs the file (see Syntax 2).

SYNTAX 1:
RUN [<line number>]

SYNTAX 2:
RUN <filespec>[,R]

NOTES:
Execution of the RUN command automatically cancels the definitions of all animation objects.

For Syntax 1, if <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. GW-BASIC always returns to command level after a RUN statement is executed.

For Syntax 2, the <filespec> must include the filename used when the file was saved. (GW-BASIC appends a default filename extension of .BAS if one was not supplied in the SAVE command.)

This version of RUN 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.

EXAMPLES:

```
RUN

RUN "NEWFIL",R
```

SAVE COMMAND

PURPOSE:
Saves a program file onto a disk or RAM cartridge.

SYNTAX:
SAVE <filespec>[{,A |,P}]

NOTES:
<filespec> is a quoted string that conforms to your operating system's requirements for filenames. GW-BASIC appends a default filename extension of .BAS if one is not supplied in the SAVE command. If a filename already exists, the existing file will be written over.

The A option saves the file in ASCII format. If the A option is not specified, GW-BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command (described earlier in this section) requires an ASCII format file, and some operating system commands such as LIST may also require an ASCII format file.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

EXAMPLES:
The statement:

```
SAVE "COM2",A
```

saves the program COM2 in ASCII format. The statement:

```
SAVE "PROG",P
```

saves the program PROG as a protected file which cannot be listed or altered.

SCREEN STATEMENT

PURPOSE:
Selects text mode or one of the 7 graphics modes, enables or disables color operation, selects the active page for screen operations, and selects the page to be displayed on the screen.

SYNTAX:
SCREEN [<mode>] [,[<burst>][,[<active page>][,<visible page>]]]

<mode> is one of the 8 screen modes listed in Table 6-4.

<burst> enables or disables color operation.

<active page> selects the active page for text output operations.

<visible page> selects the page to be displayed on the screen.

NOTES:
The Mindset Personal Computer supports 8 screen modes. <mode> is a numeric expression which evaluates to an integer in the range 1 to 8. The characteristics of each screen mode are shown in Table 6-4.

Table 6-4: Screen Modes

Screen Mode	Graphics Resolution	Number of Colors	Screen Buffering	Vertical Interlacing
0	Text	2	single/double	no
1	320 x 200	4	single/double	no
2	640 x 200	2	single/double	no
3	320 x 200	2	single/double	no
4	320 x 200	16	single	no
5	640 x 200	4	single	no
6	320 x 400	4	single	yes
7	640 x 400	2	single	yes

When the SCREEN statement changes the screen mode, it erases the screen. If the user changes SCREEN values of parameters other than mode, the screen is not erased.

<burst> is an expression which evaluates to a Boolean value. A false value disables color operation, a true value enables color operation.

<active page> is an expression which evaluates to the number of the page (0 or 1) which is to receive text from PRINT and PRINT USING statements. <visible page> is an expression determining which page (0 or 1) is to be displayed on the screen.

EXAMPLES:
The statement:

```
10 SCREEN  0,0
```

selects black and white text mode. The statement:

```
10 SCREEN  ,,,1
```

selects page 1 for display on the screen. The statement:

```
10 SCREEN  4,1
```

selects graphics mode 4 for 16-color operation with a resolution of 320 x 200.

SCREEN FUNCTION

PURPOSE:
Reads a character or its color from a specified screen location.

SYNTAX:
Y = SCREEN(<row>,<column>[,X])

<row> is a valid numeric expression returning an unsigned integer in the range 1 to 24 (1 to 25 if KEY OFF has been executed).

<column> is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon the screen width.

X is a valid numeric expression returning a Boolean result.

NOTES:
The ordinate of the character at the specified coordinates is stored in the numeric variable. If the optional parameter X is given and is non-zero, the color attribute for the character is returned instead.

EXAMPLES:
In the function:

```
100 X = SCREEN (10,10)
```

if the character at (10,10) is A, the statement returns 65. The function:

```
100 X = SCREEN (1,1,1)
```

returns the color attribute of the character in the upper left corner of the screen.

SGN FUNCTION

PURPOSE:
Indicates the value of X, relative to zero:

If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns −1.

SYNTAX:
Y = SGN(X)

EXAMPLE:
The function:

```
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.

SHELL STATEMENT

PURPOSE:
Loads and executes another program (.EXE or .COM). When the program finishes, control returns to the BASIC program at the statement following the SHELL statement. A program executed under control of BASIC is referred to as a "Child process".

Child processes (or "children") are executed when SHELL loads and runs a copy of COMMAND with the "/C " switch. By using COMMAND in this way, parameters are correctly parsed into the default FCBs, and built-in MS-DOS commands such as DIR, PATH, and SORT may be executed.

SYNTAX:
SHELL [<command string>]

<command string> is a valid string expression containing the name of a program to run and (optionally) command arguments for that program.

NOTES:
The following rules apply to the SHELL statement:

1. The program name in <command string> may have any filename extension. If the file has no extension, COMMAND first looks for a file of that name with a .COM extension, then for an .EXE file, and, finally, for a .BAT file. If none is found, SHELL issues a "File not found" error.

2. Any text separated from the program name by at least one blank will be processed by COMMAND as one or more program parameters.

3. SHELL works just like the XENIX command 'SH'. That is, BASIC remains in memory while the Child process is running. When the child finishes, BASIC continues.

4. You cannot SHELL to another copy of BASIC. When the user attempts to run BASIC as a Child process, BASIC recognizes the situation before initialization and returns to the Parent copy of BASIC after issuing the message: "You cannot run BASIC as a child of BASIC". This restriction is provided as an implementation option for cases where it is necessary to protect the integrity of the BASIC Parent program.

5. SHELL with no <command string> will give you a new COMMAND shell. You may now do anything that COMMAND allows. When you are ready to return to BASIC, enter the DOS command EXIT.

6. SHELL is not recommended when running animation programs.

Child Processes

BASIC cannot totally protect itself from its children. When a SHELL statement is executed, many things may already be going on. For example, files may already be OPEN and devices may be in use. The following guidelines will help to prevent Child processes from harming the BASIC environment.

1. Hardware considerations:

   In general, the state of all hardware should be preserved during a SHELL command. However, BASIC users should refrain from using certain devices within Child processes executed using the BASIC SHELL command. Specific areas of concern are:

   a. Screen Device - Child processes can sometimes modify screen mode parameters, although useful information may be displayed by a Child process. This problem is further complicated by Screen Editor issues.

   b. Interrupt Vectors - Save and restore interrupt vectors the child intends to use. The Child process should perform this task.

   c. Other hardware - Many devices are placed in a specific state by BASIC. These devices may be used by the Child process without the user being aware of any limitations and cause unpredictable results.

2. File System considerations:

   A child that alters any file open in the BASIC parent may cause disastrous results.

3. Memory Management:

   Unless BASIC is started with the /M: switch, it will try to free any memory it is not using before it passes control through the SHELL statement to COMMAND.

   With the /M: switch, BASIC must assume that the user intends to load something in the top of BASIC's Memory Block. This assumption prevents BASIC from "compressing the workspace" before doing the SHELL. Consequently, use of the /M: switch may cause SHELL to fail on an "Out of memory" error.

   To avoid this problem, the user should load machine-language subroutines before BASIC is run. Such loading can be accomplished

continued on next page

by placing "Pocket Code" at the end of machine- language subroutines that allows them to exit to DOS and stay resident. For example:

```
CSEG        SEGMENT        CODE
.
;Machine language subroutine
.           RET                ;Last instruction
START::
            INT            27H ;Terminate, stay resident

CSEG        ENDS
            END            START
```

Be sure to "load" these subroutines by running them before running BASIC. The AUTOEXEC.BAT file is very useful for loading and running these subroutines.

A child should NEVER "terminate and stay resident". Doing so may not leave BASIC enough room to expand its workspace to the original size. If BASIC cannot restore the workspace, all files are closed, the error message "SHELL can't continue" is printed, and BASIC exits to MS-DOS.

EXAMPLES:

```
SHELL 'get a new COMMAND
A>DIR{user types DIR to see files}
A>EXIT{user types EXIT to return to BASIC}
```

Control returns to BASIC.

The user writes some data to be sorted, invokes the SHELL sort to sort it, then reads the sorted data to write a report:

```
10 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
.
. 'Write data to be sorted
.
1000 CLOSE 1
1010 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
.
. 'Process the sorted data
.
10 SHELL "DIR | SORT >FILES."
20 OPEN "FILES." FOR INPUT AS #1
.
. 'Examine, change, or delete file names
.
```

SIN FUNCTION

PURPOSE:
Returns the sine of X, where X is in radians.

SYNTAX:
Y = SIN(X)

NOTES:
COS(X) = SIN(X+3.14159/2).

EXAMPLE:
The function:

```
PRINT SIN(1.5)
```

will yield:

```
.9974951
```

See also "COS Function" earlier in this section.

SOUND STATEMENT

PURPOSE:
Generates a sound through the speaker.

SYNTAX:
SOUND <freq>,<duration>[,<volume>[,<voice>]]

<freq> is the desired frequency in hertz. <freq> must be a numeric expression returning an unsigned integer in the range 37 to 32767.

<duration> is the duration in clock ticks. Clock ticks occur 18.2 times per second. <duration> must be a numeric expression returning an unsigned integer in the range 0 to 65535.

<volume> uses a value from 0 to 255 to set the volume for the note. The default value of 255 produces the loudest sound.

<voice> selects one of four voices to play the note. The range of <voice> is 0 to 3. Voice 0 is the default.

NOTES:
If the duration is zero, any SOUND statement currently 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 produces random sounds from voice 0 at full volume.

SPACE$ FUNCTION

PURPOSE:
Returns a string of spaces of length X.

SYNTAX:
Y$ = SPACE$(X)

NOTES:
The expression X is rounded to an integer and must be in the range 0 to 255.

Also see "SPC Function" later in this section.

EXAMPLE:
The function:

```
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I
40 NEXT I
```

will yield:

```
1
  2
    3
      4
        5
```

SPC FUNCTION

PURPOSE:
Skips spaces in a PRINT statement. I is the number of spaces to be skipped.

SYNTAX:
PRINT...SPC(I)...
LPRINT...SPC(I)...

NOTES:
SPC may be used only with PRINT and LPRINT statements. I must be in the range 0 to 255. A semicolon (;) is assumed to follow the string of spaces printed by the SPC(I) command.

Also see "SPACE$ Function" earlier in this section.

EXAMPLE:
The function:

```
PRINT "OVER" SPC(15) "THERE"
```

will yield:

```
OVER                    THERE
```

SQR FUNCTION

PURPOSE:
Returns the square root of X.

SYNTAX:
Y = SQR(X)

NOTES:
X must be > = 0.

EXAMPLE:
The function:

```
10 FOR   X=10 TO 25 STEP 5
20 PRINT   X,   SQR(X)
30 NEXT
```

will yield:

```
10          3.162278
15          3.872984
20          4.472136
25          5
```

STICK FUNCTION

PURPOSE:
Returns the X and Y coordinates of the two joysticks.

SYNTAX:
X = STICK(I)

(I) is a numeric expression returning an unsigned integer in the range 0 to 7.

NOTES:
The values for I can be:

Value	Explanation
0	Returns the X direction switch position for joystick A. A value of 1 indicates right and −1 indicates left. Setting I = 0 also latches the X and Y values for both joysticks and the mouse for the STICK (I) function when I = 1 to 7.
1	Returns the Y direction switch position of joystick A. A value of 1 indicates up and −1 indicates down.
2	Returns the X direction switch position of joystick B. Indicator values are the same as for joystick A.
3	Returns the Y direction switch position of joystick B. Indicator values are the same as for joystick A.
4	Returns the change in the X coordinate for mouse A since the last STICK function call.
5	Returns the change in the Y coordinate for mouse A since the last STICK function call.
6	Returns the change in the X coordinate for mouse B since the last STICK function call.
7	Returns the change in the Y coordinate for mouse B since the last STICK function call.

The coordinate data for a mouse is not scaled in any way; it represents the number of pulses received from the mouse. The user must scale this data to represent screen coordinates, and the scaling factor will vary according to the size of each individual screen.

Note: Executing the STRIG statement, or a trigger event when trigger event trapping is enabled, resets any accumulated mouse data to 0.

EXAMPLES:
The following routine:

```
10 CLS
20 LOCATE  1,1
30 PRINT  "X=";STICK(0)
40 PRINT  "Y=";STICK(1)
50 GOTO 20
```

creates an endless loop to display the values of the direction switch positions for joystick A.

The following routine could be used to determine the current position of a mouse in screen mode 2, where the screen has an aspect ratio of 4/3:

```
10  XSCALE=4:YSCALE=3      'Set aspect ratio
20  XREMAINDER=0:  YREMAINDER=0      'Clear remainder
values
30  LOCATE 1,1      'Assume mouse is at upper left
40  X=O:Y=O      'Set X and Y as upper left
      .
      .
      .
200  Z=STICK(0)      'LATCH stick data
210  XREMAINDER=XREMAINDER+STICK(4)      'Add change in
X to accumulated change
220  YREMAINDER=YREMAINDER+STICK(5)      'Add change in
Y to accumulated change
230  X=X+XREMAINDER/XSCALE      'Scale, change, and update
X
240  Y=Y+YREMAINDER/YSCALE      'Scale, change, and update
Y
250  XREMAINDER=XREMAINDER MOD XSCALE      'Save any
excess X movement
260  YREMAINDER=YREMAINDER MOD YSCALE      'Save any
excess Y movement
```

STOP STATEMENT

PURPOSE:
Terminates program execution and returns to command level.

SYNTAX:
STOP

NOTES:
STOP statements may be used anywhere in a program to terminate execution. STOP is often used for debugging. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

The STOP statement does not close files.

GW-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

EXAMPLE:
The function:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
```

will yield:

```
? 1,2,3
BREAK IN 30

PRINT L
30.76923
CONT
115.9
```

STOP/START OBJECT STATEMENTS

PURPOSE:
STOP OBJECT halts the motion and freezes the current view of one or more objects on the screen. START OBJECT restarts one or more objects which have been halted with a STOP OBJECT statement.

SYNTAX:
START OBJECT [<object number>[,<object number>...]]
STOP OBJECT [<object number>[,<object number>...]]

<object number> is the number of an object being halted by the STOP OBJECT statement or being restarted by the START OBJECT statement. <object number> must be in the range 1 to 16.

NOTES:
The STOP OBJECT statement freezes the specified object(s) on the screen but does not erase them, as does the DEACTIVATE statement. The START OBJECT statement unfreezes the specified object(s) after they have been stopped by a STOP OBJECT statement.

If no object numbers are specified, STOP OBJECT freezes all objects and START OBJECT restarts all objects.

EXAMPLES:
The statement:

```
10 STOP OBJECT 1,2,3,4,5
```

freezes objects numbered 1 through 5. The statement:

```
20 START OBJECT 1
```

unfreezes only object number 1.

STR$ FUNCTION

PURPOSE:
Returns a string representation of the value of X.

SYNTAX:
Y$ = STR$(X)

NOTES:
Also see "VAL Function".

EXAMPLE:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

STRIG STATEMENT/FUNCTION

PURPOSE:
The STRIG ON statement enables event trapping of joystick or mouse switch activity.

The STRIG OFF statement disables event trapping of joystick or mouse switch activity.

The STRIG STOP statement disables event trapping of joystick or mouse switch activity; however, if the joystick or mouse switch is pressed, that event will be remembered and will be trapped as soon as event trapping is enabled.

The X=STRIG(I) function returns the status of a specified joystick trigger or mouse switch.

SYNTAX:
STRIG ON
STRIG OFF
STRIG STOP
X = STRIG(I)

X is a numeric variable for storing the result of the function.

I is a numeric expression returning an unsigned integer in the range 0 to 7, designating which trigger is to be checked.

NOTES:
The STRIG ON statement enables joystick/mouse event trapping by an ON STRIG statement (see "ON STRIG Statement" earlier in this section). While trapping is enabled, if a non-zero line number is specified in the ON STRIG statement, GW-BASIC checks between every statement to see if the joystick trigger or mouse switch has been pressed.

The STRIG OFF statement disables event trapping. If an event occurs (in other words, if the trigger or switch is pressed), it will not be remembered.

The STRIG STOP statement disables event trapping; however if a trigger or switch is pressed it will be remembered, and the event trap will take place as soon as trapping is enabled.

continued on next page

In the STRIG(I) function, the values for (I) can be:

Value	Explanation
0	Returns −1 if joystick A right trigger was pressed since the last STRIG(0) function call; returns 0 if not.
1	Returns −1 if joystick A right trigger is currently down, 0 if not.
2	Returns −1 if joystick A left trigger was pressed since the last STRIG(2) function call, 0 if not.
3	Returns −1 if joystick A left trigger is currently down, 0 if not.
4	Returns −1 if joystick B right trigger was pressed since the last STRIG(0) function call, 0 if not.
5	Returns −1 if joystick B right trigger is currently down, 0 if not.
6	Returns −1 if joystick B left trigger was pressed since the last STRIG(0) function call, 0 if not.
7	Returns −1 if joystick B left trigger is currently down, 0 if not.

When a trigger/switch event trap occurs, that occurrence of the event is destroyed. Therefore, the X=STRIG(I) function will always return a false value inside a subroutine unless the event has been repeated since the trap occurred. Consequently, if you wish to perform different procedures for various joysticks or switches, you must set up a different subroutine for each joystick or switch, rather than including all the procedures in a single subroutine.

EXAMPLE:
In the following routine:

```
10 IF STRIG(0) THEN BEEP
20 GOTO 10
```

an endless loop is created to beep whenever the trigger button on joystick 0 is pressed.

STRING$ FUNCTION

PURPOSE:
Returns a string of length I whose characters all have ASCII code J or the first character of X$.

SYNTAX:
Y$ = STRING$(I,J)
Y$ = STRING$(I,X$)

EXAMPLE:
The function:

```
10 X$ = STRING$(10,45)
20 PRINT   X$   "MONTHLY REPORT"   X$
```

will yield:

```
----------MONTHLY REPORT----------
```

SWAP STATEMENT

PURPOSE:
Exchanges the values of two variables.

SYNTAX:
SWAP <variable>,<variable>

NOTES:
Variables of any type (integer, single precision, double precision, or string) may be swapped, but the two variables must be of the same type or a "Type mismatch" error results.

If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will result.

EXAMPLE:
The routine:

```
10 A$ = " ONE " : B$ = " ALL ": C$ = "FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
```

will yield:

```
ONE FOR ALL
ALL FOR ONE
```

SYSTEM COMMAND

PURPOSE:
Closes all open files, disables animation, and returns control to the operating system.

SYNTAX:
SYSTEM

NOTES:
When a SYSTEM command is executed, a "warm start" is performed (in other words, all open files are closed, and the operating system is reloaded without deleting any existing programs or memory except GW-BASIC itself).

This command is available only when GW-BASIC is started from MS-DOS.

TAB FUNCTION

PURPOSE:
Moves the print position to I.

SYNTAX:
PRINT...TAB(I)...
LPRINT...TAB(I)...

NOTES:
If the current print position is already beyond space I, TAB goes to that position on the next line.

Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255.

TAB may only be used in PRINT and LPRINT statements.

EXAMPLE:
The function:

```
10 PRINT "NAME"  TAB(25)  "AMOUNT"  :  PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
```

will yield:

```
NAME                          AMOUNT

G. T. JONES                   $25.00
```

TAN FUNCTION

PURPOSE:
Returns the tangent of X. X should be given in radians.

SYNTAX:
Y = TAN(X)

NOTES:
If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXAMPLE:

```
10 Y=Q*TAN(X)/2
```

TIME$ STATEMENT

PURPOSE:
Sets the time. This statement complements the TIME$ variable, which retrieves the time.

SYNTAX:
TIME$ = <string expression>

<string expression> returns a string 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)

NOTES:
A 24-hour clock is used; 8:00 p.m., therefore, would be entered as 20:00:00.

EXAMPLE:
The statement:

```
10 TIME$ = "08:00:00"
```

sets the current time at 8:00 a.m.

TIME$ VARIABLE

PURPOSE:
Retrieves the current time. (To set the time, use the TIME$ statement, described earlier in this section.)

NOTES:
The TIME$ variable returns an eight-character string in the form hh:mm:ss, where hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m., therefore, would be shown as 20:00:00.

The time is set by a BASIC program with the TIME$ statement or by the operating system.

EXAMPLE:
At 1:00 a.m., the function:

```
10 PRINT TIME$
```

will yield:

```
01:00:00
```

TIMER VARIABLE

PURPOSE:
Returns a floating-point number representing the elapsed number of seconds either since midnight or since a system reset.

NOTES:
TIMER is a read-only variable. It cannot be set by a BASIC program.

The TIMER variable can automatically provide a random seed for the RANDOMIZE statement (see "RANDOMIZE Statement" earlier in this section).

EXAMPLE:
The program:

```
10 T=TIMER
20 INPUT "Enter a character",  A$
30 PRINT TIMER - T;"seconds."
```

prints the number of seconds from the start of program execution to the pressing of the RETURN key after a character is entered.

TRON/TROFF STATEMENTS/COMMANDS

PURPOSE:
Traces the execution of program statements.

SYNTAX:
TRON

TROFF

NOTES:
As an aid in debugging, the TRON statement (executed in either 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:
The routine:

```
TRON

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
RUN
```

will yield:

```
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]

TROFF
RUN
1 10 20
2 20 30
```

USR FUNCTION

PURPOSE:
Calls an assembly language subroutine.

SYNTAX:
X = USR[<digit>][(<argument>)]

<digit> specifies which USR routine is being called. See the DEF USR statement earlier in this section for rules governing <digit>. If <digit> is omitted, USR0 is assumed.

<argument> is the value passed to the subroutine. It may be any numeric or string expression.

NOTES:
If a segment other than the default segment (data segment DS) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

EXAMPLE:
In the following use of the USR function:

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5
130 Y = USR0(X)
140 PRINT Y
```

the type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

VAL FUNCTION

PURPOSE:
Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example, VAL(" −3") returns −3.

SYNTAX:
Y = VAL(X$)

EXAMPLE:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699
      THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815
      THEN PRINT NAME$ TAB(25)   "LONG BEACH"
.
.
.
```

See the STR$ function for details on numeric-to-string conversion.

VARPTR FUNCTION

PURPOSE:
Returns the address of the first byte of data identified with <variable name>.

For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>. (See Syntax 2.)

SYNTAX 1:
I = VARPTR(<variable name>)

SYNTAX 2:
I = VARPTR(#<file number>)

NOTES:
A value must be assigned to <variable name> before execution of VARPTR. Otherwise an "Illegal function call" error results. A variable of any type may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned (see "Assembly Language Subroutines" in Section 2 for a discussion of the string descriptor). The address returned will be an integer in the range 32767 to −32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to an assembly 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.

Note: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

EXAMPLE:

```
100 X=USR(VARPTR(Y))
```

VARPTR$ FUNCTION

PURPOSE:
Returns a character form of the memory address of the variable.

SYNTAX:
X$ = VARPTR$(<variable name>)

<variable name> is the name of a variable in the program.

NOTES:
VARPTR$ is primarily used with the DRAW and PLAY statements.

A value must be assigned to <variable name> before execution of VARPTR$. Otherwise, an "Illegal function call" error results. A variable of any type (numeric, string, or array) may be used.

VARPTR$ returns a three-byte string in the form:

byte 0 = type
byte 1 = low byte of address
byte 2 = high byte of address

Note, however, that the individual parts of the string are not considered characters.

The possible types are as follows:

Byte 0 Value	**Variable Type**
2	Integer
3	String
4	Single precision
8	Double precision

Note: Because array addresses change whenever a new variable is assigned, always assign all simple variables before calling VARPTR$ for an array element.

EXAMPLE:
The function:

```
10 PLAY "X" + VARPTR$(A$)
```

uses the subcommand X, plus the address of A$, as the string expression in the PLAY statement.

continued on next page

This statement is equivalent to:

```
10 PLAY "XA$;"
```

VIEW STATEMENT

PURPOSE:
Segments the screen into separate viewports for graphics operations.

SYNTAX:
VIEW [[SCREEN][(X1,Y1) – (X2,Y2)[,[<fill>][,[<border>]]]]]

(X1,Y1) are the screen coordinates of the upper left corner of the viewport and (X2,Y2) are the screen coordinates of the lower right corner. The X and Y coordinates define the rectangle within the screen that graphics will map into; they must be within the physical bounds of the screen.

Initially, RUN or VIEW with no arguments defines the entire screen as the viewport.

<fill> defines the color to fill the view area. If <fill> is omitted, the view area is not filled.

The <border> attribute draws a line surrounding the viewport if space for a border is available. If <border> is omitted, no border is drawn.

NOTES:
For the form VIEW (X1,Y1) – (X2,Y2): All points plotted are relative to the viewport. That is, X1 and Y1 are added to the X and Y coordinates before putting down the point on the screen.

For the form VIEW SCREEN (X1,Y1) – (X2,Y2): All coordinates are absolute and may be inside or outside the screen limits; however, only those coordinates within the VIEW limits will be plotted.

EXAMPLES:
The statement:

```
10 VIEW (10,10) – (200,100)
```

causes the point set down by the subsequent statement (PSET(0,0),3) to appear at the physical screen location 10,10.

The statement:

```
10 VIEW SCREEN (10,10) – (200,100)
```

prevents the point set down by the subsequent statement (PSET (0,0),3) from appearing, because 0,0 is outside the viewport. However, PSET (10,10),3 is within the viewport, and causes the point to appear in the upper left corner of the viewport.

VIEW PRINT STATEMENT

PURPOSE:
Sets the top and bottom boundaries of the text window.

SYNTAX:
VIEW PRINT [<top line> TO <bottom line>]

<top line> is the top boundary and <bottom line> is the bottom boundary of the text window. The range of these parameters is from 1 to 25.

NOTES:
Statements and functions which operate within the text window include CLS, LOCATE, and the SCREEN function. The Screen Editor will limit functions such as scroll and cursor movement to the text window.

If no parameters are specified, VIEW PRINT will initialize the text window to include the whole screen.

EXAMPLE:
The statement:

```
10 VIEW PRINT 3 to 13
```

sets the text window to include the 11 lines from line 3 to line 13.

WAIT STATEMENT

PURPOSE:
Suspends program execution while monitoring the status of a machine input port.

SYNTAX:
WAIT <port number>,I[,J]

I and J are integer expressions.

NOTES:
The WAIT statement causes program 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 J, and then AND'ed with I. If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is non-zero, program execution continues with the next statement. If J is omitted, its value is assumed to be zero.

Warning: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to restart the machine manually. To avoid this, WAIT must have the specified value at <port number> during some point in the program execution.

EXAMPLE:
The statement:

```
100 WAIT 32,2
```

causes program execution to be suspended until port 32 receives a binary value of 2.

WHILE...WEND STATEMENTS

PURPOSE:
Executes a series of statements in a loop as long as a given condition is true.

SYNTAX:
WHILE <expression>
.
.
[<loop statements>]
.
.
WEND

NOTES:
If <expression> is not zero (in other words, if the expression is true), <loop statements> are executed until the WEND statement is encountered. GW-BASIC then returns to the WHILE statement and checks <expression> again. 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.

Note: Do not direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

EXAMPLE:

```
100  'BUBBLE SORT ARRAY A$
105  FLIPS=1   'FORCE ONE PASS THRU LOOP
110  WHILE FLIPS
115  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
```

WIDTH STATEMENT

PURPOSE:
Sets the printed line width in number of characters for the screen or line printer.

SYNTAX 1:
WIDTH [LPRINT]<size>

SYNTAX 2:
WIDTH <file number>,<size>

SYNTAX 3:
WIDTH <device>,<size>

<size> is a numeric expression in the range 0 to 255. It specifies the width of the printed line.

If <size> is 255, the line width is "infinite"; that is, GW-BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

<file number> is a numeric expression in the range 1 to 15. It designates the number of the file that is open.

<device> is a string expression indicating the device that is to be used.

NOTES:
With Syntax 1, if LPRINT is included, the line width is set for the line printer width. If the LPRINT option is omitted, the line width is set for the screen display. The <size> value in this case must be 40 or 80. GW-BASIC clears the screen when it changes the width from 40 to 80 or from 80 to 40.

With Syntax 2, if the file is open to the line printer, the width is immediately changed to the specified size. This feature allows the width to be changed while the file is open.

With Syntax 3, a width assignment is stored but the current setting is not changed. A subsequent OPEN <device> FOR OUTPUT AS #<file number> will use the specified value for the width while the file is open.

continued on next page

EXAMPLE:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

WINDOW STATEMENT

PURPOSE:
WINDOW enables you to draw lines, graphs, or objects in a coordinate system not bounded by the physical limits of the screen. Such drawing is done by using arbitrary programmer-defined coordinates called "world coordinates".

A world coordinate is any valid floating-point number pair. GW-BASIC then converts world coordinate pairs into the appropriate physical coordinate pairs for subsequent display within the screen space. To make this transformation from world space to the physical space of the viewing surface (screen), the statement must indicate what portion of the unbounded (floating point) world coordinate space contains the information to be displayed.

This rectangular region in world coordinate space is called a "window".

SYNTAX:
WINDOW [[SCREEN](X1,Y1) – (X2,Y2)]

WINDOW defines the "window" transformation from X1,Y1 (upper left X,Y coordinates) to X2,Y2 (lower right X,Y coordinates). The X and Y coordinates may be any floating-point number and define the "world coordinate space" that graphics will map into the physical coordinate space defined by the VIEW statement.

Initially RUN or WINDOW with no arguments disables "window" transformation.

NOTES:
WINDOW inverts the Y coordinate on subsequent graphics statements. This inversion enables the screen to be viewed in true cartesian coordinates.

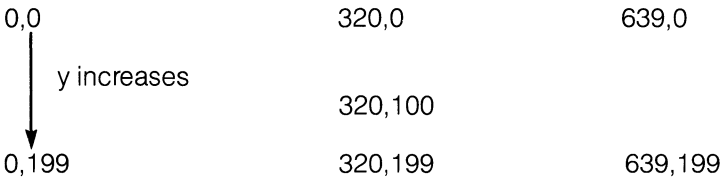
The WINDOW SCREEN variant does not invert the Y coordinate.

EXAMPLES:
The following program:

```
10 NEW
20 SCREEN 2
```

continued on next page

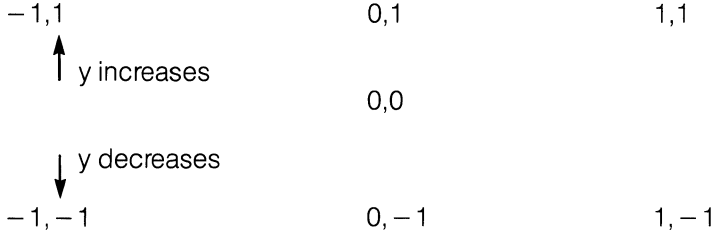
causes the screen to appear like this:



The following statement:

```
30 WINDOW (-1,-1)-(1,1)
```

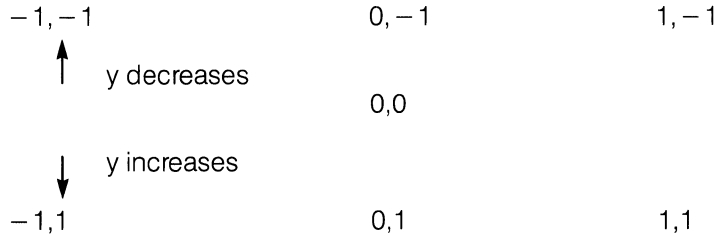
changes the screen so it appears like this:



However, the variant:

```
30 WINDOW SCREEN (-1,-1)-(1,1)
```

causes the screen to appear like this:



WRITE STATEMENT

PURPOSE:
Displays data on the screen.

SYNTAX:
WRITE [<list of expressions>]

NOTES:
If <list of expressions> is omitted, a blank line is displayed. If <list of expressions> is included, the values of the expressions are shown on the screen. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, GW-BASIC inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement.

EXAMPLE:
The routine:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
```

will yield:

```
80,   90,"THAT'S ALL"
```

WRITE # STATEMENT

PURPOSE:
Writes data to a sequential file.

SYNTAX:
WRITE # <file number>,<list of expressions>

NOTES:
<file number> is the number under which the file was OPENed in "O" mode (see "OPEN Statement" in this section).

<list of expressions> lists string or numeric expressions. They must be separated by commas.

The difference between WRITE # and PRINT # is that WRITE # inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted in the file after the last item in the list is written there.

EXAMPLE:
If 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$.

Appendix A

Error Codes and Error Messages

This appendix lists the error messages GW-BASIC displays if an error causes program execution to halt.

The ON ERROR GOTO statement can be used to trap the errors before GW-BASIC prints an error message. In this case, the ERR variable contains the number of the error and ERL contains the number of the line which caused the error.

GW-BASIC Error Messages

Number	Message and Meaning
1	NEXT without FOR A variable in a NEXT statement does not correspond to a variable in any previously executed, unmatched FOR statement.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, misspelled command or statement, incorrect punctuation). GW-BASIC automatically enters edit mode at the line that caused the error.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

Number	Message and Meaning

4 Out of data

A READ statement is executed when there are no DATA statements containing unread data remaining in the program.

5 Illegal function call

A parameter that is out of range is passed to a math or string function. A function call error may also occur as the result of:

a. A negative or unreasonably large subscript.

b. A negative or zero argument with LOG.

c. A negative argument to SQR.

d. A negative mantissa with a noninteger exponent.

e. A call to a USR function for which the starting address has not yet been given.

f. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

g. A negative record number used with GET or PUT.

6 Overflow

The result of a calculation is too large to be represented in the Microsoft GW-BASIC number format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs or variables, or contains expressions that are too complicated for a file buffer to be allocated.

8 Undefined line

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

Number	Message and Meaning
9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.
10	Duplicate definition Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.
11	Division by zero A division by zero error is encountered in an expression; or, the operation of exponentiation results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or else positive machine infinity is supplied as the result of the exponentiation. In either case, execution continues.
12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. GW-BASIC will allocate string space dynamically until memory is exhausted.
15	String too long An attempt is made to create a string more than 255 characters long.
16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.

Number	Message and Meaning
17	Can't continue An attempt is made to continue a program that: a. Has halted because of an error. b. Has been modified during a break in execution. c. Does not exist.
18	Undefined user function A USR function is called before the function definition (DEF statement) is given.
19	No RESUME An error-handling routine is entered but contains no RESUME statement.
20	RESUME without error A RESUME statement is encountered before an error- handling routine is entered.
21	Unprintable error An error condition exists but no message is available to describe it.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt has been made to enter a line that has too many characters.
24	Device timeout The requested device is not available at this time.
25	Device fault An incorrect device designation has been entered.

Number	Message and Meaning
26	FOR without NEXT A FOR statement was encountered without a matching NEXT.
27	Out of paper The printer device is out of paper.
28	Unprintable error An error condition exists, but no message is available to describe it.
29	WHILE without WEND A WHILE statement does not have a matching WEND.
30	WEND without WHILE A WEND statement was encountered without a matching WHILE.
31-49	Unprintable error An error condition exists, but no message is available to describe it.

Disk Error Messages

Number	Message and Meaning
50	Field overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal error An internal malfunction has occurred in GW-BASIC. Report to Mindset the conditions under which the message appeared.

Number	Message and Meaning
52	Bad file number A statement or command references a file with a file number that either is not OPEN or is out of the range of file numbers specified at initialization.
53	File not found A LOAD, KILL, NAME, or OPEN statement/command references a file that does not exist on the current disk.
54	Bad file mode An attempt is made to use PUT, GET, or LOF with a sequential file; to LOAD a random file; or to execute an OPEN statement with a file mode other than I, O, or R.
55	File already open An OPEN statement in sequential output mode is issued for a file that is already open; or a KILL statement is given for a file that is open.
56	Unprintable error An error condition exists, but no message is available to describe it.
57	Device I/O error An I/O error occurred during a disk I/O operation. It is a fatal error; in other words, the operating system cannot recover from the error.
58	File already exists The filename specified in a NAME statement is identical to a filename already in use on the disk.
59-60	Unprintable error An error condition exists, but no message is available to describe it.
61	Disk full All disk storage space is in use.

Number	Message and Meaning
62	Input past end An INPUT statement is executed either after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	Bad record number In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equal to zero.
64	Bad filename An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement (for example, a filename with too many characters).
65	Unprintable error An error condition exists, but no message is available to describe it.
66	Direct statement in file A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.
67	Too many files An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.
68	Device unavailable The device specified is not available at this time.
69	Communications buffer overflow Not enough space has been reserved for communications I/O.
70	Disk write protected The disk has a write-protect tab intact, or is a disk that cannot be written to.

Number	Message and Meaning
71	Disk not ready This error condition could be caused by a number of problems. The most likely is that the disk is not inserted properly.
72	Disk media error A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive may not be working properly.
74	Rename across disks An attempt was made to rename a file to a new name that was declared to be on a disk other than the disk specified for the old name. The renaming operation is not performed.
75	Path/file access error During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct path to filename connection. The operation is not completed.
76	Path not found During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to find the path specified. The operation is not completed.
**	You cannot run BASIC as a Child of BASIC No error number. During initialization, BASIC discovers that it is being run as a Child process. BASIC is not run and control returns to the Parent copy of BASIC.
**	Can't continue after SHELL No error number. Upon returning from a Child process, the SHELL statement discovers that there is not enough memory for BASIC to continue. BASIC closes any open files and exits to MS-DOS.

Appendix B

Derived Mathematical Functions

Functions that are not intrinsic to GW-BASIC may be calculated as follows.

Function	GW-BASIC Equivalent
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X) = −ATN (X/SQR(−X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(X/SQR(X*X−1)) +SGN(SGN(X)−1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(X/SQR(X*X−1)) +(SGN(X)−1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)−EXP(−X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X) = (EXP(X)−EXP(−X))/ (EXP(X)+EXP(−X))
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(−X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)−EXP(−X))
HYPERBOLIC COTANGENT	COTH(X) = (EXP(X)+EXP(−X))/ (EXP(X)−EXP(−X))
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X+SQR(X*X−1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1+X)/(1−X))/2

Function	GW-BASIC Equivalent
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(−X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X) *SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X+1)/(X−1))/2

Appendix C

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	030	1EH	RS
001	01H	SOH	031	1FH	US
002	02H	STX	032	20H	SPACE
003	03H	ETX	033	21H	!
004	04H	EOT	034	22H	"
005	05H	ENQ	035	23H	#
006	06H	ACK	036	24H	$
007	07H	BEL	037	25H	%
008	08H	BS	038	26H	&
009	09H	HT	039	27H	'
010	0AH	LF	040	28H	(
011	0BH	VT	041	29H	)
012	0CH	FF	042	2AH	*
013	0DH	CR	043	2BH	+
014	0EH	SO	044	2CH	,
015	0FH	SI	045	2DH	–
016	10H	DLE	046	2EH	.
017	11H	DC1	047	2FH	/
018	12H	DC2	048	30H	0
019	13H	DC3	049	31H	1
020	14H	DC4	050	32H	2
021	15H	NAK	051	33H	3
022	16H	SYN	052	34H	4
023	17H	ETB	053	35H	5
024	18H	CAN	054	36H	6
025	19H	EM	055	37H	7
026	1AH	SUB	056	38H	8
027	1BH	ESCAPE	057	39H	9
028	1CH	FS	058	3AH	:
029	1DH	GS	059	3BH	;

Dec = decimal, Hex = hexadecimal (H), CHR = character

Dec	Hex	CHR	Dec	Hex	CHR
060	3CH	<	094	5EH	^
061	3DH	=	095	5FH	_
062	3EH	>	096	60H	`
063	3FH	?	097	61H	a
064	40H	@	098	62H	b
065	41H	A	099	63H	c
066	42H	B	100	64H	d
067	43H	C	101	65H	e
068	44H	D	102	66H	f
069	45H	E	103	67H	g
070	46H	F	104	68H	h
071	47H	G	105	69H	i
072	48H	H	106	6AH	j
073	49H	I	107	6BH	k
074	4AH	J	108	6CH	l
075	4BH	K	109	6DH	m
076	4CH	L	110	6EH	n
077	4DH	M	111	6FH	o
078	4EH	N	112	70H	p
079	4FH	O	113	71H	q
080	50H	P	114	72H	r
081	51H	Q	115	73H	s
082	52H	R	116	74H	t
083	53H	S	117	75H	u
084	54H	T	118	76H	v
085	55H	U	119	77H	w
086	56H	V	120	78H	x
087	57H	W	121	79H	y
088	58H	X	122	7AH	z
089	59H	Y	123	7BH	{
090	5AH	Z	124	7CH	|
091	5BH	[	125	7DH	}
092	5CH	\	126	7EH	~
093	5DH	]	127	7FH	DEL

Extended Codes

The INKEY$ variable returns a two-character string for keys with extended (non-ASCII) codes. In this case, the first character is code 000 and the second code (the extended code) represents one of the keys listed below:

Extended Code	Key	Extended Code	Key
3	NUL character	66	F8
		67	F9
15	SHIFT-TAB	68	F10
16	ALT-Q	71	HOME
17	ALT-W	72	▲
18	ALT-E	73	PG UP
19	ALT-R	75	◀
20	ALT-T	77	▶
21	ALT-Y	79	END
22	ALT-U	80	▼
23	ALT-I	81	PG DN
24	ALT-O	82	INS
25	ALT-P	83	DEL
30	ALT-A	84	SHIFT-F1
31	ALT-S	85	SHIFT-F2
32	ALT-D	86	SHIFT-F3
33	ALT-F	87	SHIFT-F4
34	ALT-G	88	SHIFT-F5
35	ALT-H	89	SHIFT-F6
36	ALT-J	90	SHIFT-F7
37	ALT-K	91	SHIFT-F8
38	ALT-L	92	SHIFT-F9
44	ALT-Z	93	SHIFT-F10
45	ALT-X	94	CTRL-F1
46	ALT-C	95	CTRL-F2
47	ALT-V	96	CTRL-F3
48	ALT-B	97	CTRL-F4
49	ALT-N	98	CTRL-F5
50	ALT-M	99	CTRL-F6
59	F1	100	CTRL-F7
60	F2	101	CTRL-F8
61	F3	102	CTRL-F9
62	F4	103	CTRL-F10
63	F5	104	ALT-F1
64	F6	105	ALT-F2
65	F7	105	ALT-F3

continued on next page

Extended Code	Key	Extended Code	Key
106	ALT-F4	120	ALT-1
107	ALT-F5	121	ALT-2
108	ALT-F6	122	ALT-3
109	ALT-F7	123	ALT-4
110	ALT-F8	124	ALT-5
112	ALT-F9	125	ALT-6
113	ALT-F10	126	ALT-7
114	CTRL-PRT SC	127	ALT-8
115	CTRL-◀	128	ALT-9
116	CTRL-▶	129	ALT-0
117	CTRL-END	130	ALT--
118	CTRL-PG DN	131	ALT- =
119	CTRL-HOME	132	CTRL-PG UP

Note: The extended codes 59 through 68 are available only when the corresponding function keys are not defined as soft keys.

Appendix D

GW-BASIC Reserved Words

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

ABS
ACTIVATE
AND
ASC
ARRIVAL
ATN
AUTO
BEEP
BLOAD
BSAVE
CALL
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLIP
CLOSE
CLS
COLLISION
COLOR
COM
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DATE$
DEACTIVATE
DEFDBL
DEFINT
DEFSNG
DEFSTR
DEF FN
DEF OBJECT
DEF USR
DELETE
DIM
DRAW
EDIT
ELSE
END
ENVIRON
ENVIRON$
EOF
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
END
EXP
FIELD
FILES
FIX
FOR
FRE

GET
GOSUB
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
IOCTL
IOCTL$
KEY
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKDIR
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OBJECT
OCT$
ON
OPEN
OPEN COM
OPTION
OR
PAINT
PALETTE
PALETTE USING
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT USING
PRINT #
PRINT # USING
PSET
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RIGHT$
RMDIR
RND
RSET
RUN
SAVE
SCREEN
SGN
SHELL
SIN
SOUND
SPACE
SPC
SQR
START
STICK
STOP
STR$
STRIG
STRING$
SWAP
SYSTEM
TAB
TAN

THEN
TIME$
TIMER
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR$
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#
XOR

Index

MINDSET GW™-BASIC

Version 1.01

Update

Please add these pages to the binder containing your GW-BASIC Reference Manual. It's a good idea to review the changes and additions listed here before you begin using GW-BASIC, even if you're unfamiliar with GW-BASIC programming and may not understand all the terms and details discussed.

Each item is listed by topic and, with a few exceptions, by page reference to the GW-BASIC Reference Manual. You may wish to note some of these changes and additions on the appropriate pages of your manuals.

System requirements

For you to use this version of GW-BASIC, your Mindset Personal Computer System must include at least 256K bytes RAM. In addition, two disk drives will allow you to take full advantage of the capabilities and ease of use of this version of GW-BASIC.

NVRAM cartridges

NVRAM cartridges should not be used for file creation or storage with this version of GW-BASIC. Instead, you should use diskettes for file storage. Future versions of Mindset GW-BASIC will allow users to take full advantage of NVRAM cartridge capability.

SHIFT-PRT SCN and GW-BASIC

If you press SHIFT-PRT SCN to copy the text on your screen to a printer (see your Mindset Operation Guide, page 3-3) while using GW-BASIC, the last character in some lines may not be printed.

Sound and music functions

1-2

All the sound and music functions and statements in GW-BASIC, including BEEP (see page 6-13) and CHR$(7) (the BELL character--see page 6-23), produce sound only if your Mindset Personal Computer is connected to an external amplifier and speaker system through the AUDIO LEFT connector on the back of the System Unit.

Keyword entry with ALT-M

1-7

Pressing ALT-M enters the keyword MOTOR.

Animation and scrolling of the screen

3-1

During animation, scrolling of the screen should be avoided--it may result in incorrect display of the background.

Starting the diskette version of GW-BASIC

4-3

To start the diskette version of GW-BASIC after loading MS™-DOS, you must place your GW-BASIC diskette in the default drive of your computer system.

Static file space allocation switch--/I

4-5

GW-BASIC is configured so that the/I option is always selected. This affects the amount of memory allocated for file buffers and the number of default files that may be opened.

Control functions

5-4--5-5

Table 5-1: GW-BASIC Control Functions is incorrect. The following table lists the correct codes, keys, and functions.

Hex Code	Decimal Code	Key	Function
03	03	C	Break
05	05	E	Truncate line (clear text to end of logical line on screen and in memory)
07	07	G	Beep
08	08	H	Backspace, deleting characters passed over
09	09	I	Tab (8 spaces)
0A	10	J	Linefeed
0B	11	K	Move cursor to home position
0C	12	L	Clear window
0D	13	M	Carriage return (enter current logical line)
0E	14	N	Append to end of line
12	18	R	Toggle insert/typeover mode
14	20	T	Display function key contents
15	21	U	Clear logical line (on screen only)
17	23	W	Delete word
1A	26	Z	Clear to end of window (on screen only)
1C	28	\	Cursor right
1D	29	]	Cursor left
1E	30	^	Cursor up
1F	31	_	Cursor down (underscore)
7F	127	DEL	Delete character at cursor--functions without the CTRL key having to be pressed

Note also that the ESC key clears a logical line on screen only.

OPEN and line printer operation

6-160

You cannot send output to a line printer with the statement OPEN "LPTn"--a "Bad file mode" error message will result.

VIEW--fill and border parameters	When used with a fill parameter, the optional border parameter will not result in the drawing of a border.

6-247

WIDTH and line printer operation	SYNTAX 2, WIDTH <file number>, <size>, will not work with a line printer.

6-251

100463-001

Backing Up Your MINDSET GW™-BASIC Diskettes

Version 1.01

Although your GW-BASIC package includes two complete program diskettes, it's a good idea to make at least one backup copy of your own before you start using GW-BASIC. To prevent unauthorized use, your original program diskettes are protected in such a way that you must use one of them, rather than a copy, to load GW-BASIC into your computer. However, a backup copy will come in handy if, for example, you lose one of your originals and accidentally write over some of the files on the other.

Before you make your backup copy, you may also wish to add to at least one of your original program diskettes some DOS files that will enable you to boot up your computer system and run GW-BASIC using the same diskette--that is, without first having to load MS™-DOS separately.

Before you make a backup copy--

Space has been reserved on your GW-BASIC program diskettes for MS-DOS system and COMMAND.COM files. Added to a program diskette, these files enable you to boot up your computer system, then run GW-BASIC using only your GW-BASIC program diskette--that is, without first having to load MS-DOS separately.

To add these files to a GW-BASIC program diskette, first load MS-DOS into your system. Be sure that there's no write-protect tab on your GW-BASIC diskette. With your MS-DOS diskette in drive A and your GW-BASIC diskette in drive B, type SYS B: and press RETURN. (If you have only one disk drive, see Appendix A in your MS-DOS Reference Manual.) When the message "System transferred" appears, the system files have been added to your GW-BASIC diskette.

Next, type COPY COMMAND.COM B: and press RETURN. When the message "1 File(s) copied" appears, the COMMAND.COM file has been added to your GW-BASIC diskette.

(See your MS-DOS Reference Manual for details on the SYS command and the COMMAND.COM file.)

You may now use your GW-BASIC program diskette to boot up your system, then run GW-BASIC. If your system is turned off, place the GW-BASIC diskette in the default drive and turn on the system; if the system is already on, place the GW-BASIC diskette in the default drive and press ALT-RESET. Next, when prompted for an entry, type BASIC and press RETURN.

To make a backup copy--

Using MS-DOS, first format a blank double-sided, double-density diskette. Use the default options provided by the FORMAT command--that is, with

your MS-DOS diskette in drive A and your blank diskette in drive B, type FORMAT B: and press RETURN. (If you have only one disk drive, see Appendix A in your MS-DOS Reference Manual.)

Next, use the MS-DOS DISKCOPY command to copy your GW-BASIC program diskette onto your newly formatted diskette. Type DISKCOPY A: B: and press RETURN, then follow the prompts that appear on your screen.

(See your Introductory Guide to MS-DOS or MS-DOS Reference Manual for details on formatting and copying diskettes.)

It's a good idea to attach a write-protect tab to both your original program diskettes and your backup copy. Put one of the originals and the backup in a safe place, and use the other original each time you want to load GW-BASIC.

If your working GW-BASIC program diskette becomes damaged--

If you accidentally write over any files on your working GW-BASIC program diskette, use the MS-DOS DISKCOPY command to copy GW-BASIC back onto your working diskette either from your other original or from your backup copy.

100463-001