GW-BASIC Interpreter

XEROX®

610P72856

TABLE OF CONTENTS

TABLE OF CONTENTS

5. REFERENCE

INTRODUCTION

TABLE OF CONTENTS

1. INTRODUCTION

GW-BASIC is the most extensive implementation of BASIC available for personal computers. It meets the requirements of the ANSI standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, GW-BASIC provides sophisticated string handling and structured programming features that are especially suited for applications development. The GW-BASIC language has improved graphics possibilities, and gives users what they want from a BASIC-ease of use plus the features that makes a personal computer perform like a microcomputer.

GW-BASIC major features

GW-BASIC combined with MS-DOS provides a powerful and friendly environment for the BASIC programmer.

Some of the main features of GW-BASIC are as follows:

- Redirection of Standard Input (INPUT, LINE INPUT) and Standard Output (PRINT).
- Character device support allowing GW-BASIC to initialize and communicate with a peripheral device.
- Multiple directories for disk organization and directory management support (MKDIR/ CHDIR/RMDIR).
- Improved disk I/O facilities for handling large files.

- Screen Editor enhancements including text window support.

- SHELL allowing COMMAND or child processes to run without having to leave the GW-BASIC environment.

- Improved Graphics: Line Clipping VIEW, WINDOW, etc.

- User-defined keyboard trapping, error trapping and additional event trapping.

- Precise error reporting with system functions ERDEV and ERDEV$.

- Double precision transcendentals. Optional with the /D: switch in the GWBASIC command.

- CALL statements with parameter passing.

- Chaining with common variables: chaining is used to allow programs larger than the available memory. Application programs can be menu-driven to allow for maximum user friendliness.

- Optional declaration statements: variable names can be listed in a declaration statement explicitly specifying the variable to be a string, real, or integer variable.

System requirements

GW-BASIC, under the MS-DOS operating system, can be run using the minimum system configuration.

A minimum of one disk drive is required.

Manual contents

The manual is made up of five sections. They are

INTRODUCTION

GETTING STARTED describes the steps for getting starting with GW-BASIC.

TUTORIAL is for the beginner or new user.

GENERAL INFORMATION section is a **must** read section. It contains some very important information.

REFERENCE section contains all the information on GW-BASIC statements, commands, and functions. They are grouped together by task. At the beginning of the Reference section is an alphabetical listing with a short description and page location of every GW-BASIC statement, command, and function.

Notes:

GETTING STARTED

TABLE OF CONTENTS

1. GETTING STARTED

The following are the steps to get started with GW-BASIC:

1. Be sure your original GW-BASIC disk is write-protected.

2. Make a backup copy of the GW-BASIC disk.

   If you have hard disk, you may want to make a directory to contain GW-BASIC.

3. Store the original GW-BASIC disk in a safe place.

Loading GW-BASIC

MS-DOS users

With the MS-DOS system prompt displayed, type GWBASIC and press the Return key.

Continue on next page.

ScreenMate users

Highlight the GWBASIC.EXE file and then run it.

Continue on next page.

When loaded, GW-BASIC responds with the following:

```
GW-BASIC 2.x
(C) Copyright Microsoft 1983,84

XEROX -- GW-BASIC Rev. 1.x
Copyright (C) by Olivetti, 1984 - All rights Reserved

xxxxxx Bytes free
Ok
_
```

The "Ok" is the GW-BASIC prompt.

The "_" is the cursor.

The system is now waiting for input from you.

If you are a beginner, you may want to go through the TUTORIAL section.

(Note: The GWBASIC command may be entered with several options to optimize memory occupation, redirecting standard input or output, etc. *See the GWBASIC command in Chapter 19 of the Reference section.*)

Exiting GW-BASIC

To exit from GW-BASIC and return to MS-DOS or ScreenMate, type SYSTEM and press the Return key.

This closes all data files before returning to MS-DOS.

TUTORIAL

TABLE OF CONTENTS

1. INTRODUCTION

This section is a self-paced guide to learning GW-BASIC. The training in this section is written on the assumption that you have

- a knowledge of high school algebra
- an understanding of MS-DOS based personal computers
- know little or nothing about BASIC programming

Upon completion of the training, you will be able to create and execute programs which utilize the following GW-BASIC statements and system commands.

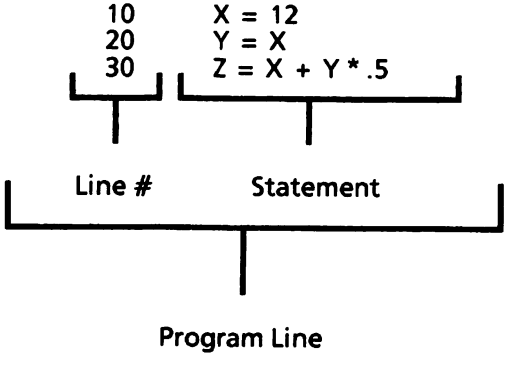
AUTO	KILL
DIM	LIST/LLIST
EDIT	LOAD
FILES	NEW
FOR/NEXT	PRINT/LPRINT
GOSUB/RETURN	READ/DATA
GOTO	RUN
IF	SAVE
INPUT	STOP

Rules for writing statements

Before you start any hands-on exercises on the system, you need to know a few rules of writing statements.

Every program in GW-BASIC is made up of program lines containing statements and **each program line must have a line number.**

Example:



In the example above, there are three program lines each beginning with a line number (10, 20, and 30). Also, there are three variables (X, Y, and Z).

Variables

Variables are names used to represent either numeric or string values that are used in a program.

Both letters and numbers may be used in a variable name, however **the first character must be a letter.**

The value of a variable may be assigned by the programmer (such as X and Y in the example on the previous page) or it may be assigned as the result of calculations in the program (such as Z in the example on the previous page).

String variable names are written with a dollar sign ($) as the last character. For example:

```
A$ = "SALES REPORT"
```

In the above statement, the dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string. A string consists of alphanumeric (both alphabetic and numeric characters) text enclosed in quotation marks.

Variables can be equal to:

- a constant — `10 X = 12`
- another variable — `20 Y = X`
- an expression — `30 Z = X + Y * .5`

A **constant** is the actual value assigned to a variable that GW-BASIC uses during execution. It can be either a numeric or string value.

Examples:

numeric constant	10 X = **12**
string constant	20 D$ = **"DATE"**

An **expression** is a grouping of variables and/or constants and the operators (arithmetic operators are discussed below) to produce a single value.

Examples:

```
10 Y = PI * R ^ 2
20 GROSS = HOURS * RATE
30 NET = GROSS - (TAX + DED)
```

Only variables are valid on the left side of an equal sign. A constant or expression can never be placed on the left side of the equal sign.

Arithmetic operators

Special characters called arithmetic operators perform certain arithmetic operations. They are as follows, in order of priority:

Priority	Operator	Operations
First	^	Exponentiation
Second	*, /	Multiplication/Division
Third	+, -	Addition/Subtraction

When the system examines your GW-BASIC statements, it will proceed with the calculations based on priority (e.g., it will multiply before it adds).

Example:

```
X = 6 * 2 ^ 4
```

This statement is calculated as follows:

- First, 2 is raised to the 4th power or 16. (Exponentiation has a higher priority than multiplication, so this operation is performed first.)
- Next, 6 is multiplied by 16 giving 96.
- The value of the variable X is 96.

If arithmetic operators of the same priority are in the same equation, then the system will work from left to right on these operators.

Example:

```
D = A + B - C
```

This statement is calculated as follows:

- First, A is added to B.
- Then C is subtracted from the sum of A + B giving the value of D.

Look at one more example to be sure you understand how the priority works in calculating equations.

Example:

```
Y = 10 + 6/3 * 4
```

This statement is calculated as follows:

- Since divide and multiply are the same priority, the operation will be from left to right. First, 6 is divided by 3 giving 2.
- Second, 2 is multipied by 4 giving 8.
- Third, 10 is added to 8 which results in 18. (Addition is last since it has the lowest priority.)

How to change priority of arithmetic operators

Priority of arithmetic operators can cause problems. For example, you may have calculations that require addition or subtraction before multiplication or division. This can be accomplished by putting parentheses around the calculations you want done first.

Here are a few rules for the use of parentheses.

- Parentheses force the innermost calculations to be accomplished first.
- The number of **left** parentheses **must** equal the number of **right** parentheses.
- Extra parentheses have no effect on calculations. (So if they help you, do not hesitate to use them.)

The following are some examples of algebraic formulas and the way they would look as GW-BASIC statements.

Formula	Statement
$A + \frac{B}{C}$	E = A + B/C

In the example below, the addition needs to take place before the division is performed. So B + C is put in parentheses in the statement.

$\frac{A}{B + C}$	W = A/(B + C)

In the next two examples, can you see why the parentheses are used.

$\frac{A \bullet B}{C \bullet D}$	X = (A*B)/(C*D)
$\frac{D - B}{6 \bullet A}$	F = (D-B)/(6*A)

In the example below, two sets of parentheses are needed (one set is within another set). First, B needs to be added to 1, then A is divided by the sum of B + 1. The result is then squared. **Remember, when using more than one set of parentheses together, the innermost set is calculated first.**

$\left(\frac{A}{B + 1}\right)^2$	Y = (A/(B + 1))^2

Summary

- Every program is made up of program lines containing statements and each program line **must begin with a line number**.
- Variables are names used to represent either numeric values or strings that are used in a program.
- Variable names can consist of letters and numbers, but the first character of the variable **must be a letter**.
- Only variables are valid on the left side of the equal sign. A constant or expression can **never** be placed on the left side of the equal sign.
- Arithmetic operations are performed by special characters called "arithmetic operators".
- The system performs calculations based on priority (e.g., it will multiply before it adds).
- If arithmetic operators of the same priority are in the same equation, then the system will work from left to right on these operators.
- By using parentheses in the calculations, you can change the priority of arithmetic operators.

- The rules of using parentheses are:
  - Parentheses force the innermost calculations to be accomplished first.
  - The number of **left** parentheses **must** equal the number of **right** parentheses.
  - Extra parentheses have no effect on calculations. (So if they help you, do not hesitate to use them.)

2. THE BASICS

Now that you are familiar with how to write statements, you are ready to start hands-on practice. You will begin with a short program that finds the value of the variable D.

Entering a new program

Be sure you have GW-BASIC loaded.

Using the NEW Command

Before entering a new program, the NEW command should be given. The NEW command deletes the current program in memory, if any, and clears all variables. This means that if you have finished working on a program and wanted to enter a new program, you would use this command. If you did not use the NEW command, the old program would still be in memory and your new program would be entered with it.

- Type the command NEW and press the Return key.

  Nothing noticeable happens, but memory has been cleared. The **Ok** prompt and the **cursor** are displayed on the screen.

Entering the Line Numbers

You can enter the line numbers in two ways.

- Type it in.
- Use the AUTO command to automatically enter the line numbers.

The AUTO command generates a line number automatically after every carriage return. It will number the lines 10, 20, 30, etc.

▸ Type the command AUTO and press the Return key.

The line number **10** and the cursor is displayed on the screen (see below). You are now ready to enter the program.

```
Ok
AUTO
10 _
```

Using the REM Statement

It is good practice to use REM (remark) statements in your programs. This is a nonexecutable statement which is used for documenting your programs. REM statements are very helpful when you or someone else must make program changes or corrections at a later date.

(Note: If you make a typo while entering a line, just use the Backspace key (←) to delete characters and retype. If you discover a mistake after you have pressed the Return key, just continue with the exercise and in the next section "EDITING THE PROGRAM" you will learn different ways to correct your program.)

At line number 10,

- Type the following REM ***FINDS THE VALUE OF *** and press the Return key.

  This statement tells what the program does.

Enter program main body

You are ready to enter the main body of the program. At line number 20,

- Type the following A = 20 and press the Return key.

At line number 30,

- Type the following B = 15 and press the Return key.

At line number 40,

- Type the following D = A + B and press the Return key.

Stop entering program

To exit the AUTO command, at line number 50,

- HOLD down the CTRL key while you press the C key.
- Release both keys.

  The system returns to command level.

You have entered your first program into internal memory. That is, it is in the CPU's (control processing unit) memory but not stored as a file on disk. You will be told later how to save the program on disk.

Editing the program

It is nearly impossible to write a "perfect" program the first time. Most programs will require editing for typing errors, program statement errors, or updating to meet future needs. In this section, you will learn some of the basic editing features of GW-BASIC.

(Note: As you make these changes, you will not see any changes in the original program you just typed or on the screen. Later on, you will be given instructions on how to display your edited program.)

So far, your program looks similar to this on the screen:

```
Ok
NEW
Ok
AUTO
10 REM ***FINDS THE VALUE OF D***
20 A = 20
30 B = 15
40 D = A + B
50
Ok
_
```

Adding a line

If you needed to add a line between the line numbers 20 and 30, you could use a line number between 21 and 29, inclusive. This means that between the line numbers 20 and 30, you could add up to nine program lines. For this exercise, you will use line number 25.

- Type the following `25 X = 40` and press the Return key.

  Line 25 is added to the program in memory.

You'll add another line between line numbers 30 and 40.

▸ Type the following 35 Y = (A + B) * D and press the Return key.

Line 35 is added to the program in memory.

Deleting a line

Next, delete a line from the program. To do this, type the line number that you want deleted and press the Return key.

▸ Type the line number 40 and press the Return key.

Line 40 is deleted from the program in memory.

Editing a specific line

There are two ways to correct a line.

- Retype the program line starting with the line number to be corrected.
- Use the EDIT command.

Using the first method, change program line number 10 as follows:

▸ Type the following 10 REM ***FINDS THE VALUE OF Y *** and press the Return key.

This line replaces the old line 10 in the program in memory.

There will be times when you will not want to retype the entire line. At these times, you would use the EDIT command. Only one line at a time can be edited.

To enter the EDIT mode,

- Type EDIT 35 and press the Return key.

  The program line to be edited is displayed on the screen, as shown below, with the cursor in position waiting for instructions. You are now in EDIT mode.

  35 Y = (A + B) * D

For this exercise, you want to change "D" to "X".

- Press the right arrow key (→) until the cursor is under the "D", as shown below.

  35 Y = (A + B) * D

You can now change the letter "D" to "X".

- Type the letter X.

  The letter "D" is changed to "X".

To exit the EDIT mode,

- Press the Return key.

  The system returns to command level.

Displaying the program

You have made several changes to your program and the screen should look similar to the following:

```
Ok
NEW
OK
AUTO
10 REM ***FINDS THE VALUE OF D***
20 A = 20
30 B = 15
40 D = A + B
50
Ok
25 X = 40
35 Y = (A + B) * D
40
10 REM ***FINDS THE VALUE OF Y***
EDIT 35
35 Y = (A + B) * X
_
```

Using the LIST or LLIST Command

How does the program in memory look after these changes have been made to it? The command to display the entire edited program on the screen is LIST. Or, if you wanted the program printed on the printer, the command would be LLIST (when using LLIST, be sure your printer is turned on).

▶ Type LIST and press the Return key.

The following is displayed on the screen.

```
LIST
10 REM ***FINDS THE VALUE OF Y***
20 A = 20
25 X = 40
30 B = 15
35 Y = (A + B) * X
Ok
_
```

Displaying/printing the results

How do you get the results of your program to display on the screen or print out on the printer? The PRINT (displays result on screen) or LPRINT (prints result on printer) statement is used in the program.

Examples:

To display on screen

```
10 X = 21
20 PRINT X
```

To print on printer

```
10 X = 21
20 LPRINT X
```

To practice displaying the result on the screen, use the program you just typed and edited. You'll need to add a PRINT statement at the end of the program.

- Type 40 PRINT Y and press the Return key.

  Line 40 is now added to the program in memory. This tells the system you want the value of Y displayed on the screen.

Executing the program

The RUN command executes the program currently in memory.

To run the program you just entered in memory,

- Type RUN and press the Return key.

  The value of Y is displayed on the screen as shown below.

```
RUN
 1400
Ok
_
```

Saving the program

The SAVE command saves the program in memory to disk. You'll need to type a file name to save the program under. File names must be enclosed in quotation marks when using the SAVE command. To save the program you have entered in memory,

▸ Type SAVE "FIRST" and press the Return key.

The program is saved as a file on the disk in the default drive under the file name of FIRST.

Some information about saving a program

- Each file (program) must have a distinct name so that you will be able to recall it from the disk. The file name is made of up to eight alphanumeric characters. Alpha-numeric means alphabetic and/or numeric characters can be used.

  MS-DOS will automatically generate the file type of ".BAS" for the program source code (your program).

- Saving a program does not clear it from memory. You use the NEW command to do that.

- Once the program is a file on the disk and you recall it at a later date but make no corrections to it, you do not have to resave it. It is permanently stored on the disk.

- If you make corrections to the file (program) and want to resave it, and if you resave it with the same file name, the original version is erased because the new version is written on top of the old version. If you want to keep the original version, just save the modified version under a different file name.

Display the files stored on a disk

When you have created several files on a disk, there may be times when you will want to recall a file but cannot remember its filename. Unless you keep them recorded on paper, how else are you to know what files are stored on what disk? To help you remember, there is a command called FILES. Look at the directory of your disk.

▶ Type FILES and press the Return key.

The directory of the disk is displayed on the screen (similar to the one shown below). The ".BAS" following the file name FIRST is a MS-DOS generated file type for program source code.

```
Ok
FILES
d:\path


(List of files)


Ok
_
```

Load a file (program) into memory

If you have saved a program and would like to recall it at a later date, use the LOAD command to load the file (program) from the disk to internal memory. This enables you to work with that file (program). To practice loading a file (program), use the one you just created and saved.

When using the LOAD command, you do not have to use the NEW command to clear memory. The LOAD command closes all open files and clears memory.

- Type LOAD "FIRST" and press the Return key.

  The file (program) named FIRST is loaded into internal memory.

  When using the LOAD command, the file name to be loaded must be enclosed in quotation marks.

How do you know that the program named FIRST has been loaded into memory?

- Type LIST and press the Return key.

  The file (program) named FIRST is displayed on the screen.

Delete a file (program) from disk

To delete a file (program) from the disk, you will use the KILL command. You'll create a file named OLD.BAS and then delete it from the disk.

The FIRST.BAS program is in internal memory and stored as a file on disk.

To create a file that you can delete, save the program in internal memory to a different file name on disk.

- Type SAVE "OLD" and press the Return key.

Check the files on the disk to see that OLD.BAS has been stored on the disk as a file.

- Type FILES and press the Return key.

  The system displays the files stored on the disk. Notice that the file OLD.BAS has been added.

You are now ready to delete the file OLD.BAS from the disk.

- Type KILL "OLD.BAS" and press the Return key.

  The file OLD.BAS has been deleted from the disk.

Check the directory of the disk to see if the file has been deleted.

- Type FILES and press the Return key.

  The file OLD.BAS is not listed on the disk.

Some information about deleting files

- The file name that you want to delete **must** be enclosed in quotation marks.

- Since you can have several files with the same name but different file types (extensions), you must include the one to three-character extension (if any) in the the KILL command (e.g., KILL"filename.ext"). For example, if the command KILL "PAYROLL" was given without including the extension, and the files PAYROLL.BAS and PAYROLL.DAT existed on the disk, the system would not know which file to delete. The system would display the "File not found" error message.

Exit GW-BASIC

If you decide not to continue with the next section until a later date, you can exit GW-BASIC. But before you exit GW-BASIC, it is recommended to always do a RESET command.

- Type RESET and press the Return key.

  All disk files are closed and the directory information is written to the disk.

To return to the MS-DOS operating system, enter the following command.

- Type SYSTEM and press the Return key.

  The MS-DOS system prompt or ScreenMate is displayed on the screen. You are now back in the MS-DOS operating system.

More about files

When using the commands SAVE, FILES, LOAD and KILL, you can reference files on other disk drives.

For example, if you wanted to save a program on the disk in Drive B, you would give the command

```
SAVE "B:filename"
```

where B could be replaced with any disk drive A-H.

For more information about these commands, see the GW-BASIC Reference section in this manual.

Summary

- The NEW command deletes the current program in memory and clears all variables. Use this command before entering a new program.
- The AUTO command generates a line number automatically after every carriage return.
- To exit the AUTO command, hold down the CTRL key while you press the C key.
- The REM statement is a nonexecutable statement used to document the program.
- To add a line to an existing program, use a line number between two existing line numbers where the new line is to be added and then type the new line.
- To delete a line from a program, type the line number and press the Return key.
- There are two ways to edit a line. One way is to retype the entire line. The other way is to go into EDIT mode and make the changes.
- After editing a program, use the LIST command to display the entire program on the screen or use the LLIST command to print the entire program on the printer. This way you can see how the edited program looks.

- To get the results of your program to display on the screen or print out on the printer, use the PRINT statement or LPRINT statement, respectively, in your program.

- The RUN command executes the program currently in memory.

- The SAVE command saves a program on disk.

- The FILES command displays the files stored on a disk.

- The LOAD command loads a file (program) from disk into internal memory.

- The KILL command deletes a file from the disk.

- The SYSTEM command returns the system to the MS-DOS operating system.

- When using the commands SAVE, FILES, LOAD, and KILL, you can reference files on other drives by putting the drive designator (A-H) with the file name enclosed in quotes (e.g., SAVE "B:filename").

3. INPUT AND OUTPUT DATA

This section describes how to input data needed in the program and how to output data from the program.

Input data

There are two methods of inputting data to the program.

- Use READ/DATA statements in the program.
- Use INPUT statement(s) in the program.

READ/DATA statements

If you had a programming problem that required six values, you could assign the values to the variables as follows:

```
10 U = 5
20 V = 3
30 W = 7
40 X = 10
50 Y = 35
60 Z = 1.5
```

You could also shorten this program by using the READ/DATA statements as shown below.

```
10 READ U, V, W, X, Y, Z
20 DATA 5, 3, 7, 10, 35, 1.5
```

The READ statement lists the variables (separated by commas), and the DATA statement lists the values (also separated by commas) for each variable in the READ statement. When the system executes statement 10, the variable U will have the value of 5, variable V will have the value of 3, variable W will have the value of 7, and so on.

Important information

READ statement

- 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.

DATA statement

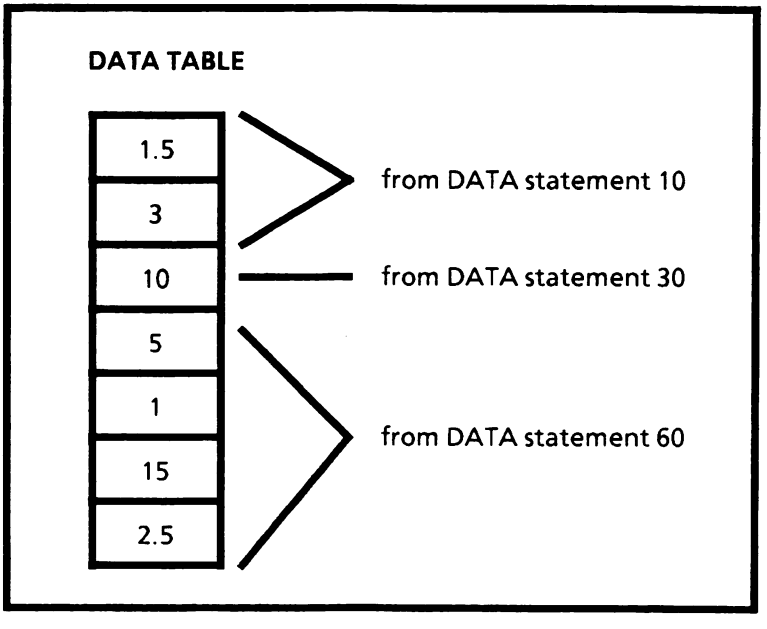
- DATA statements are nonexecutable and may be placed anywhere in the program.
- They may contain as many values (separated by commas) as will fit on a line.
- Any number of DATA statements may be used in a program.

Detailed explanation

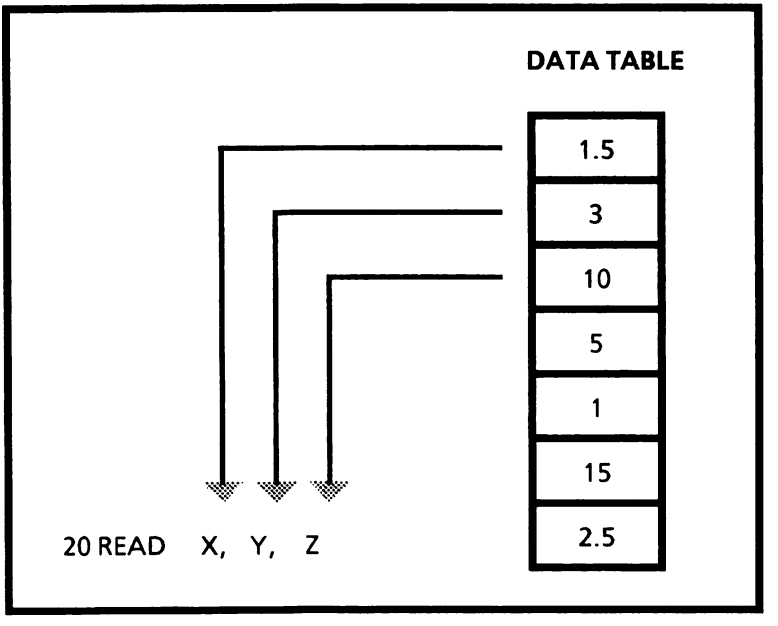
To better understand how the READ/DATA statements work, look at the following program:

```
10 DATA 1.5, 3
20 READ X, Y, Z
30 DATA 10
40 READ W
50 READ X, Y
60 DATA 5, 1, 15, 2.5
70 READ T
```

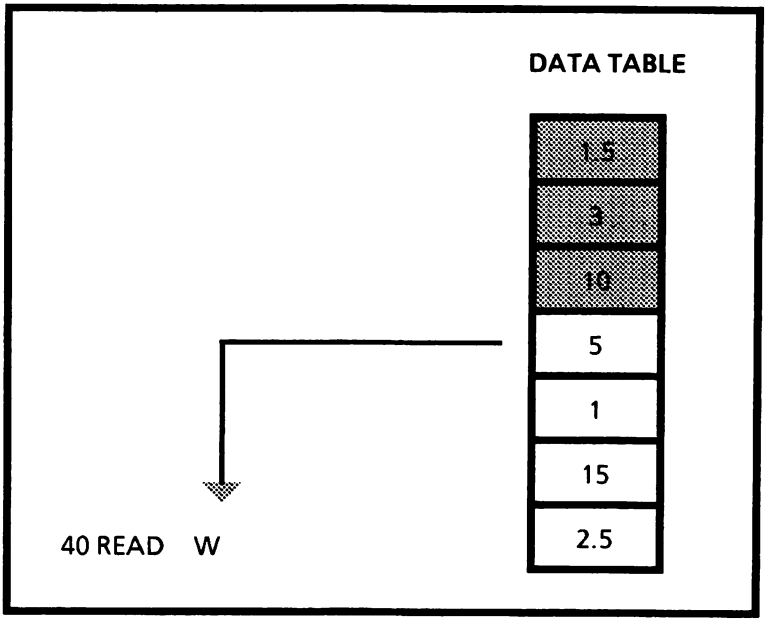
When the RUN command is given, the system takes all the DATA statements and sets up a table as illustrated below.



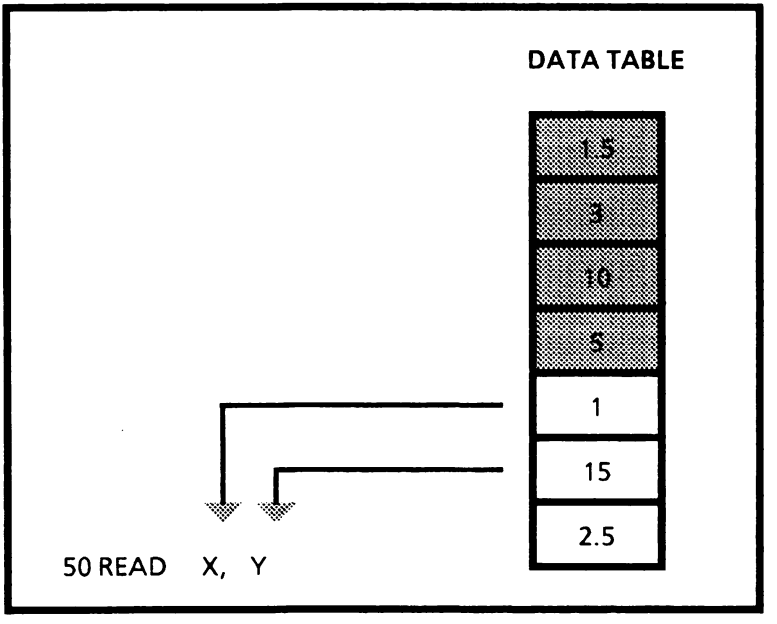
The first READ statement takes data from the table and places the data in variables X, Y, and Z, respectively.



The first three numbers are used up, so the next READ statement will start with the fourth number in the table.

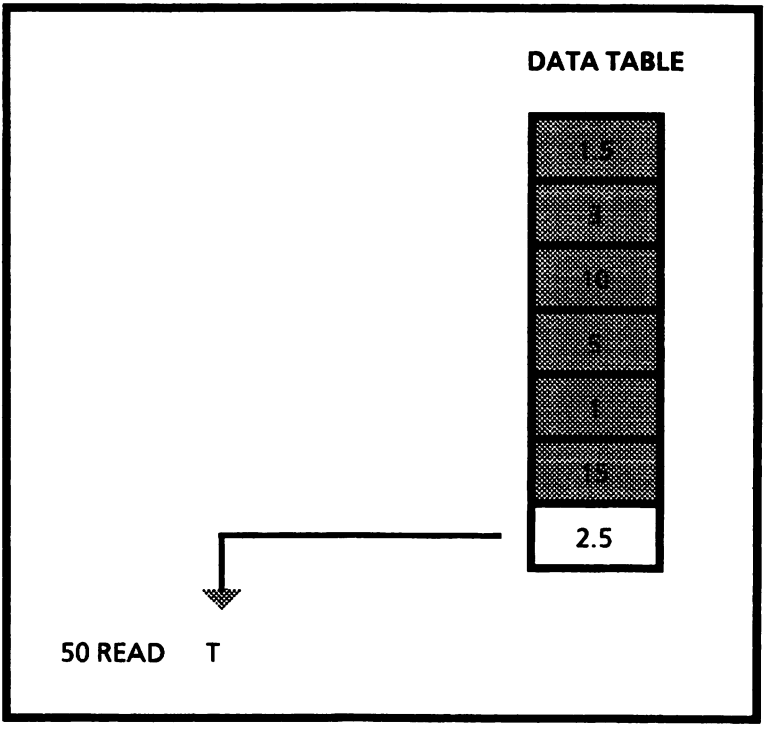


Notice that in the third READ statement, the variable names X and Y are used again, but the data in the variables are different because the READ statement has assigned new values to them. (That's why they are called variables.)



Before the above READ statement, X and Y had the values of 1.5 and 3, respectively. After the above READ statement, the values of X and Y have changed to 1 and 15, respectively.

The final READ statement takes the last data item from the data table.



After these READ and DATA statements are executed, the variables will have the values of:

X = 1
Y = 15
Z = 10
W = 5
T = 2.5

If you are having trouble understanding READ and DATA statements, review this section again.

Exercise

Practice using the READ/DATA statements using the program entitled FIRST that you created earlier. You must have GW-BASIC loaded in your system as well as the file (program) FIRST loaded in internal memory.

Load the file (program) FIRST into internal memory.

- Type LOAD "FIRST" and press the Return key.

  The program FIRST is loaded into internal memory.

Check to be sure the program FIRST is loaded into internal memory.

- Type LIST and press the Return key.

The FIRST program should look similar to the following:

```
Ok
LOAD "FIRST"
Ok
LIST
10 REM ***FINDS THE VALUE OF Y***
20 A = 20
25 X = 40
30 B = 15
35 Y = (A + B) * X
40 PRINT Y
Ok
_
```

First, you need to put in the READ statement,

- Type 20 READ A, X, B and press the Return key.

  This program line replaces line 20.

Next, enter the DATA statement.

- Type 30 DATA 20, 40, 15 and press the Return key.

  This program line replaces line 30.

You do not need program line 25 any more. So delete it.

- Type 25 and press the Return key.

  Program line 25 is deleted.

Now that you have made these changes, display the modified program on the screen.

- Type LIST and press the Return key.

The following should be displayed on the screen:

```
Ok
LIST
10 REM ***FINDS THE VALUE OF Y***
20 READ A, X, B
30 DATA 20, 40, 15
35 Y = (A + B) * X
40 PRINT Y
Ok
_
```

Before you run the program, you need to change the PRINT statement in line 40. The way the program above is written will only display the value 1400 on the screen. You may not know what it is. To program a message that will tell you the value of Y.

- Type 40 PRINT "THE VALUE OF Y IS" Y and press the Return key.

The text in quotes above is called a "string" and it will display on the screen before the value of Y is displayed. When you have a PRINT or LPRINT statement followed by text in quotes, that text will be displayed/printed exactly as it appears between the quotation marks.

You can now execute the program.

▶ Type RUN and press the Return key.

The following is displayed on the screen.

```
RUN
THE VALUE OF Y IS 1400
Ok

_
```

If you would like to save this program, you can use the SAVE command to do so. Remember, if you save it using the name FIRST, the revised program will be saved over the original. If you save the revised program with a different filename, you will have the original and the revised versions stored on disk.

INPUT statement

Another way of entering data into a program is the INPUT statement. This statement allows data to be entered via the keyboard. When an INPUT statement is encountered, program execution pauses and a question mark is displayed to indicate the program is waiting for data to be entered. The data that is entered is assigned to the variable(s) in the INPUT statement.

This is how it works using a simple program.

- Type NEW and press the Return key.

  Internal memory is cleared.

With internal memory cleared, you are ready to enter the program.

- Type 10 INPUT X and press the Return key.
- Type 20 Y = X ^ 2 and press the Return key.

  (*Note: Use the SHIFT + 6 keys to get the ^ symbol.*)
- Type 30 PRINT X "SQUARED IS" Y and press the Return key.

The program is in internal memory. You are now ready to execute it.

- Type RUN and press the Return key.

  The following is displayed:

```
RUN
?_
```

▸ Type the number 5 and press the Return key.

The following is displayed:

```
 5 SQUARED IS 25
Ok
_
```

If you want to save this program, use the SAVE command.

There can be more than one variable listed in the INPUT statement, but they must be separated by commas as demonstrated by the INPUT B, C statement in the example program on the next page. When entering more than one data item, you must separate them with commas The number of data items supplied must be the same as the number of variables in the list.

A prompt string may also be used to let you know what type of data to enter. The prompt string is enclosed in quotation marks and is followed by a semicolon and the variable name (e.g., INPUT "WHAT IS THE RADIUS";R). When the program is executed, the prompt string is displayed before the question mark.

Before entering the new program, clear internal memory.

▸ Type NEW and press the Return key.

Since there are quite a few lines to this program, use the AUTO command to automatically number the lines.

▸ Type AUTO and press the Return key.

The system automatically starts line numbering with the number 10.

Enter the following program. Type each of the lines below. Be sure to end each line with a Return.

```
REM ***Area of circle and value of D***
PI = 3.14
INPUT "WHAT IS THE RADIUS"; R
INPUT B, C
A = PI * R ^ 2
D = B + C
PRINT "THE AREA OF THE CIRCLE IS"; A
PRINT "THE VALUE OF D IS"; D
```

The program is in internal memory. To exit the AUTO command,

- HOLD DOWN the CTRL key while you press the C key. RELEASE both keys.

Before running the program, check it for typos. Use the LIST command to display the program on the screen. If you find any error(s), just retype the program line.

After you have checked the program for errors, you can run it.

- Type RUN and press the Return key.

  The system prompts you

  WHAT IS THE RADIUS? _

- Type 7.4 and press the Return key.

  The system prompts you

  ? _

- Type 20, 56 and press the Return key. Be sure to include the comma.

The following will display on the screen:

```
THE AREA OF THE CIRCLE IS 171.9464
THE VALUE OF D IS 76
Ok

—
```

Responding to INPUT with too many or too few items, or with the wrong type of data (string instead of numeric, etc.) causes the message ?Redo from start to be displayed on the screen. No assignment of input values is made until an acceptable response is given.

If you do not feel like you understand the INPUT statement and how it works, review this section.

Output data

This is a more detailed section on the way output data can be displayed on the screen using the PRINT statement or printed on the printer using the LPRINT statement.

Enter the following program and execute it. The commas, semicolons, and quotation marks are very important to the PRINT/LPRINT statement as will be explained later. Remember, if you make a mistake when entering a program line before you press the Return key, use the Backspace key to correct it.

- Type NEW and press the Return key.

  Internal memory is cleared.

- Type AUTO and press the Return key.

  System automatically starts line numbering with the number 10.

Type each line below. Be sure to end each line with a Return.

```
READ X, Y, Z
DATA 10.5, 5000, 25
PRINT
PRINT
PRINT "Example 1", X, Y, Z
PRINT
PRINT "Example 2"; X; Y; Z
PRINT
PRINT "Example 3",
PRINT X,
PRINT Y
PRINT
PRINT "Example 4";
PRINT X; Y;
PRINT Z
PRINT
PRINT "Example 5", X + Y + Z
```

To exit the AUTO command,

- HOLD DOWN the CTRL key while you press the C key. RELEASE both keys.

Before running the program, use the LIST command to display the program on the screen. **CHECK** to be sure the program lines are exactly as shown below. If not, just retype the program line(s) that is incorrect.

```
10 READ X, Y, Z
20 DATA 10.5, 5000, 25
30 PRINT
40 PRINT
50 PRINT "Example 1", X, Y, Z
60 PRINT
70 PRINT "Example 2"; X; Y; Z
80 PRINT
90 PRINT "Example 3",
100 PRINT X,
110 PRINT Y
120 PRINT
130 PRINT "Example 4";
140 PRINT X; Y;
150 PRINT Z
160 PRINT
170 PRINT "Example 5", X + Y + Z
```

Now run the program to see how the different PRINT statements display on the screen.

▶ Type RUN and press the Return key.

The data is displayed on the screen similar to the following:

```
RUN

Example 1          10.5        5000        25
Example 2  10.5  5000  25
Example 3          10.5        5000
Example 4  10.5  5000  25
Example 5          5035.5
Ok
_
```

To further understand how each of the PRINT statements work and why, examine the statements below:

```
10 READ X, Y, Z
20 DATA 10.5, 5000, 25
```

Assigns value to the variables.

```
30 PRINT
40 PRINT
```

Displays two blank lines.

```
50 PRINT "Example 1", X, Y, Z
```

GW-BASIC divides the line into print columns of 14 spaces each (5 columns on the screen and 9 columns on the printer; this may vary depending on the printer). Separating the variables or expressions with a **comma** in the PRINT or LPRINT statements will cause values to be displayed or printed in columns across the screen or paper. If there are not enough columns in one line on the screen or printer, some of the data will be displayed or printed on the next line.

```
60 PRINT
```

Displays a blank line.

```
70 PRINT "Example 2"; X; Y; Z
```

Separating variables or expressions with a **semicolon** in the PRINT or LPRINT statements will cause values to be displayed or printed immediately after the last value. (Typing one or more spaces between the variables or expressions has the same effect as typing a semicolon.)

You will notice after running the program that there are a few spaces separating the numbers. This is because there are a few rules the system must follow when printing numbers. They are:

-- Printed numbers are always followed by a space.

-- Positive numbers are preceded by a space.

-- Negative numbers are preceded by a minus sign.

```
80 PRINT
```

Displays a blank line.

```
90 PRINT "Example 3",
100 PRINT X,
110 PRINT Y
120 PRINT
130 PRINT "Example 4";
140 PRINT X; Y;
150 PRINT Z
```

When a comma or semicolon terminates the list of variables or expressions in a PRINT or LPRINT statement, the next PRINT or LPRINT statement begins displaying or printing on the same line, spacing accordingly.

```
160 PRINT
```

Displays a blank line.

```
170 PRINT "Example 5", X + Y + Z
```

Calculations can also be performed in the PRINT or LPRINT statement and the result displayed or printed.

Review this section again if you do not fully understand the PRINT or LPRINT statement. Also, do some practicing on your own.

Summary

- There are two methods for inputting data into a program. One way is using the READ/DATA statements and the other way is using the INPUT statement.

- Some important facts about READ/DATA statements are:

  READ Statement

  -- Must always be used in conjunction with a DATA statement.

  -- Assigns variables to DATA statement values on a one-to-one basis.

  DATA Statement

  -- Nonexecutable and may be placed anywhere in the program.

  -- May contain as many values as will fit on a line (separated by commas).

  -- Any number of DATA statements may be used in a program.

- Some important facts about the INPUT statement are:

  -- A question mark is displayed to indicate the program is waiting for data to be entered.

  -- You can use a prompt string before the question mark to let you know what type of data to enter.

  -- The data that is entered is assigned to variable(s) in the INPUT statement.

- There can be more than one variable listed in the INPUT statement, but these variables must be separated by commas.

- When entering more than one data item to the INPUT prompt, the data items are also separated by commas.

- The number of data items you enter at the INPUT prompt must be the same as the number of variables in the INPUT statement..

- Responding to INPUT with too many or too few items or with the wrong type of data (string instead of numeric, etc.) causes the message ?Redo from start to be displayed on the screen. No assignment of input values is made until an acceptable response is given.

- Some important facts about the PRINT/LPRINT statement are:

  - Just using a PRINT/LPRINT statement on a program line by itself will cause a blank line to be displayed/printed.

  - GW-BASIC divides the line into print columns of 14 spaces each. Separating the variables or expressions with a **comma** in the PRINT/LPRINT statement will cause values to be displayed/printed in columns across the screen/paper.

  - Separating variables or expressions with a **semicolon** in the PRINT /LPRINT statement will cause values to be displayed/printed immediately after the last value. (Typing one or more spaces between the variables or expressions has the same effect as typing a semicolon.)

-- Here are a few rules that apply when displaying/printing numbers:

  - o Printed numbers are always followed by a space.
  - o Positive numbers are preceded by a space.
  - o Negative numbers are preceded by a minus sign.

-- When a comma or a semicolon terminates the list of variables or expressions in a PRINT or LPRINT statement, the next PRINT or LPRINT statement begins displaying or printing on the same line, spacing accordingly.

-- Calculations can also be performed in the PRINT or LPRINT statement and the results displayed or printed.

*** NOTICE ***

By now, you should be familiar with entering and executing programs, so there will be no more step-by-step instructions. Enter the example programs given to see how they run and modify them with your own ideas. This will help you be creative in your programming.

Just as a reminder, here are the steps for creating and saving a program:

- Clear internal memory using the NEW command.
- Use the AUTO command to automatically number lines (optional).
- Type in the program. Be sure to correct typos.
- LIST out the program, check for errors, and compare it to the example program for accuracy. If an error is found, just retype the program line.
- Execute the program using the RUN command.
- If you want to save the program, use the SAVE command. Remember to put the file name in quotes (e.g., SAVE "filename").

The subjects covered in the last part of this tutorial may be more complicated to understand. If there is a section that you do not understand, study the example program given, then put yourself in the computer's place and "walk" through the program by putting on paper what the program tells you to.

Notes:

4. BRANCHING

There are two types of branching. One is unconditional and the other is conditional. Both are discussed in this section.

Unconditional branching

The GOTO statement is used to branch unconditionally out of the normal program sequence to a specified line number and continue execution of the program from that line number. Whenever the system encounters the GOTO instruction, it will branch to the program line specified and continue.

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
RUN
R = 5   AREA = 78.5
R = 7   AREA = 153.86
R = 12  AREA = 452.16
Out of DATA in 10
Ok
```

When no more data exists, the system will terminate the program with an **Out of DATA in 10** message. In this particular case, the **Out of DATA in 10** message is not an error but a valid termination of the program.

The following is an explanation of how the program is executed (program flow).

Program flow

```
10 READ R
```

The system assigns a value to the variable R from the values listed in the DATA statement in program line 60.

```
20 PRINT "R = "; R,
```

The text "R = " and the value of R are displayed on the screen.

```
30 A = 3.14 * R ^ 2
```

The value of A is calculated.

```
40 PRINT "AREA = "; A
```

The text "AREA =" and the value of A are displayed on the screen.

```
50 GOTO 10
```

Go to program line 10 and repeat program lines 10, 20, 30, 40 and 50 until all the values listed in the DATA statement are exhausted.

```
60 DATA 5, 7, 12
```

These values will be assigned to R.

Conditional branching

GW-BASIC has the ability to compare relative values of two numbers using the IF statement and Relational Operators. It can also branch if the result of the comparison is true. The following is the list of Relational Operators and their function.

Relational Operator	Function
X>Y	is X greater than Y?
X<Y	is X less than Y?
X=Y	is X equal to Y?
X>=Y	is X greater than or equal to Y?
X<=Y	is X less than or equal to Y?
X<>Y	is X not equal to Y?

Examine the example program below.

```
10 A = 10 : B = 15
20 IF A > B GOTO 50
30 PRINT B
40 END
50 PRINT A
RUN
 15
Ok
```

Notice there are two statements on program line 10. More than one statement can be placed on a program line, but each statement must be separated by a **colon**.

Also, notice program line 40 contains the END statement. This tells the system to end the program and return to the command level. By examining the program, can you tell why the END statement is located here?

If the END statement was not located at program line 40, the system would have also displayed the value of A. You do not have to have an END statement at the end of a program. The system automatically knows there are no more program lines to execute.

Program flow

```
10 A = 10 : B = 15
```

The variables A and B are assigned the values of 10 and 15, respectively.

```
20 IF A > B GOTO 50
```

A conditional statement is encountered which asks the question "Is the value of A greater than B?".

- If the conditional statement is true (A is greater than B), then
  - Go to program line 50.
  - Print the value of A. (program line 50)
  - Return to command level.
- If the conditional statement is false (in this case, it is), then
  - Continue with the next program line (line 30).
  - Print the value of B. (program line 30)
  - Return to command level. (program line 40)

Summary

- The **GOTO** statement is called the "unconditional branch".
- The **IF** statement is called the "conditional branch" because after the IF statement is a relational operator by which a comparison is made. If the comparison is true, the system branches. If it is false, the system goes to the next program line and continues executing the program.
- Use the **colon ":"** to separate two statements on the same program line.
- Use the **END** statement to end the program and return to command level.

5. LOOPING

The FOR . . . NEXT statements allow a series of instructions to be performed in a loop a given number of times. When you set up a loop in your program, it starts with a FOR statement and ends with a NEXT statement. All instruction and calculation statements that are to be repeated, in order, should be placed between these two statements.

Example:

```
10 FOR I = 1 TO 10
20 PRINT I;
30 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

Program flow

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

The FOR statement starts the loop.

The variable I is used as a counter so the system will know how many times the loop has been executed.

The first value (1) is the initial (starting) value of the counter. The second value (10) is the final value of the counter. So in this example, the loop begins with 1 and is performed ten times.

20 PRINT I;

Displays the value of I on the screen. The semicolon causes the changing value of I to be displayed on the same line.

30 NEXT

When the NEXT statement is encountered, the counter (variable I) is incremented by 1. A check is performed to see if the value of the counter is now greater than the final value (10). If it is not greater, it branches back to the program line after the FOR statement and the process is repeated. If it is greater, the system returns to command level.

The counter variable in the NEXT statement may be omitted as shown in the above example. Or, you can include the counter variable in the NEXT statement (e.g., 30 NEXT I).

Setting the initial and/or final values

On the previous pages, the initial and final values were set in the FOR statement (e.g., FOR I = 1 TO 10). You can also use variables and/or expressions to set the initial and/or final value(s).

Using variables

Variables can be used instead of numbers in the FOR-NEXT loop providing that the variables have been assigned values previously in the program. Examine the following example programs.

Example:

```
10 X = 2:Y = 12
20 FOR Z = X TO Y
30 PRINT Z;
40 NEXT Z
RUN
 2 3 4 5 6 7 8 9 10 11 12
Ok
```

In the above example, program line 10 assigns value to the variables X and Y which are used as the initial and final values of the FOR-NEXT loop.

The variables of the FOR-NEXT loop can also be assigned values using the INPUT or READ/DATA statements.

Examples:

```
10 REM ** Using READ/DATA statements
20 READ X, Y
30 FOR Z = X TO Y
40 PRINT Z;
50 NEXT Z
60 DATA 2, 12
RUN
 2 3 4 5 6 7 8 9 10 11 12
Ok
```

Examples: (cont'd)

```
10 REM ** Using INPUT Statement
20 INPUT "Enter the initial value"; X
30 INPUT "Enter the final value"; Y
40 FOR Z = X TO Y
50 PRINT Z;
60 NEXT Z
RUN
Enter the initial value? 2
Enter the final value? 12
 2 3 4 5 6 7 8 9 10 11 12
Ok
```

Using expressions

An expression can be used to set the initial and/or final value of the FOR-NEXT loop. The system calculates the expression(s) before the loop is executed the first time and does not recalculate the expression(s) again.

Example:

```
10 Y = 2
20 FOR X = Y + 2 TO Y * 6
30 PRINT X;
40 NEXT X
RUN
 4 5 6 7 8 9 10 11 12
Ok
```

In program line 20, Y is added to 2 to make the initial value of the loop 4. Also, Y is multiplied by 6 to make the final value of the loop 12.

Using the STEP statement with FOR

You can have a STEP statement with the FOR statement. This is used when you want to increment by more than 1 or decrement the value of the counter. Examine the programs below.

Examples:

```
10 FOR I = 2 TO 10 STEP 2
20 PRINT I;
30 NEXT I
40 PRINT:PRINT
50 PRINT "I = "; I
60 PRINT "LOOP TERMINATED"
RUN
 2 4 6 8 10

I = 12
LOOP TERMINATED
Ok
```

In the above program, the STEP 2 tells the system to increase the value of I by 2 every time it executes the FOR-NEXT loop until I is greater than 10.

Take the above program and modify it so that the counter is decremented by 2.

```
10 FOR I = 10 TO 2 STEP -2
20 PRINT I;
30 NEXT I
40 PRINT:PRINT
50 PRINT "I = "; I
60 PRINT "LOOP TERMINATED"
RUN
 10 8 6 4 2

I = 0
LOOP TERMINATED
Ok
```

Notice that when decrementing, the initial value is larger than the final value of the FOR-NEXT loop.

The STEP -2 decrements the value of the counter by 2 going from a large value to a smaller one.

Also, the counter can be stepped using fractional increments.

Example:

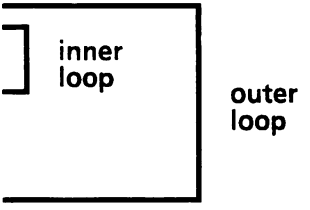
```
10 FOR C = 10 TO 13 STEP .5
20 PRINT C;
30 NEXT
RUN
 10  10.5  11  11.5  12  12.5  13
Ok
```

The STEP .5 increments the value of the counter by .5 each time through the loop until the final value is greater than 13.

Nested FOR ... NEXT 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 for its counter. The NEXT statement for the inside loop must appear before that for the outside loop. The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement.

```
10 FOR X = 1 TO 3
20 FOR Y = 1 TO 2
30 PRINT "Y = " Y
40 NEXT
50 PRINT
60 PRINT "X = " X
70 PRINT
80 NEXT
RUN
```




```
Y = 1          —— inner loop executed
Y = 2

X = 1          —— outer loop initial value set

Y = 1          —— inner loop executed
Y = 2

X = 2          —— outer loop incremented by 1

Y = 1          —— inner loop executed
Y = 2

X = 3          —— outer loop executed,
                  final value reached
Ok
```

After the RUN command has been given, you can see how the program on the previous page is executed with nested loops. The outer loop is set to the initial value and the inner loop is executed until the final value is reached. Then the outer loop is incremented and the inner loop again executes until the final value. This continues until the outer loop reaches the final value.

When you have a program with two or more successive NEXT statements, you can put them on one NEXT statement line as shown below.

```
5 rem **nested loops**
10 FOR X = 1 TO 2
20 PRINT "OUTSIDE"
30 FOR Y = 1 TO 2
40 PRINT "    INSIDE"
50 NEXT Y, X
RUN
OUTSIDE
    INSIDE
    INSIDE
OUTSIDE
    INSIDE
    INSIDE
Ok
```

In program line 50, the inside loop counter (Y) is listed first, then the outside loop counter (X) is listed.

Summary

- FOR-NEXT statements allow a series of instructions to be performed in a loop a given number of times.

- When you set up a loop in your program, it starts with a FOR statement and ends with a NEXT statement. All instruction and calculation statements that are to be repeated should be placed between these two statements.

- FOR-NEXT statements can be used in a multiple statement program line.

- Variables can be used to set the initial and/or final values of the FOR-NEXT loop providing that the variables have been assigned values previously in the program by the programmer or by using either the INPUT or READ/DATA statements.

- An expression can be used to set the initial and/or final value(s) of the FOR-NEXT loop.

- A STEP can be used with the FOR statement, when you want to increment by more than 1 or you want to decrement.

- FOR-NEXT loops may be nested; that is, a FOR-NEXT loop may be placed within the context of another FOR-NEXT loop.

Notes:

6. ARRAYS

An array is a group of related data items which are stored in memory and can be accessed by a subscripted variable.

The subscripted variable consists of an array variable name followed by a subscript enclosed in parentheses. For example, X(5) is a subscripted variable; X5 is not.

The subscripted variable (as with the simple variable) names a memory location inside the system where data is to be stored.

Arrays are helpful when you want to work with a large quantity of related data. It is easier to work with one variable that is subscripted than to have many variables (e.g., A, B, C, D, E, etc.) and trying to remember what each of these variables represent.

This section will describe the one-dimensional array and the two-dimensional array.

One-dimensional array

A one-dimensional array consists of a variable name with one subscript, i.e., X(5).

Think of it as follows:

where the value of X(0) is 25, X(1) is 10, X(2) is 13, and so on.

Variables can be used for subscripts. For example, the subscripted variable, X(A), has the variable A for a subscript.

If A=5 then X(A) would be X(5), the value of X(5)=33.

If A=2 then X(A) would be X(2), the value of X(2)=13.

You can also use expressions as subscripts. For example, the subscripted variable X(Y+3) when Y=1. After the system calculates the subscript, the subscript would be four. So the subscripted variable would be X(4) and the value X(4)=24.

Assigning values to an array

How are values assigned to an array? The INPUT or READ/DATA statements with a FOR-NEXT loop can be used to accomplish this.

Using the INPUT statement

In the example below, the INPUT statement is used to fill the example array given on page 3-74.

Example:

```
10 FOR Y = 0 TO 5
20 PRINT "Enter value for X("; Y; ")";
30 INPUT X(Y)
40 NEXT X
RUN
Enter value for X( 0 )? 25
Enter value for X( 1 )? 10
Enter value for X( 2 )? 13
Enter value for X( 3 )? 51
Enter value for X( 4 )? 24
Enter value for X( 5 )? 33
Ok
```

Notice that the counter Y in the FOR statement is also used for the subscript in X(Y), causing the subscript to be incremented by one each time through the loop (e.g., first time through Y=0 so X(Y) is X(0), second time through Y=1 so X(Y) is X(1), and so on).

Using this program, the system asks for input six times. After the six values have been entered, the first value 25 is assigned to X(0), the second value 10 is assigned to X(1), the third value 13 is assigned to X(2), and so on.

Using READ/DATA statements

In the example below, READ/DATA statements are used to fill the example array with data.

Example:

```
5 rem ***using READ/DATA statements
10 FOR Y = 0 TO 5
20 READ X(Y)
30 PRINT "X("; Y; ") = "; X(Y)
40 NEXT
50 DATA 25, 10, 13, 51, 24, 33
RUN
X( 0 ) = 25
X( 1 ) = 10
X( 2 ) = 13
X( 3 ) = 51
X( 4 ) = 24
X( 5 ) = 33
Ok
```

This program is the same as the one on the previous page except that instead of inputting data via the keyboard you are reading the data from the program.

Direct input

A third way to fill an array with data is to store a constant value directly into the array.

Example:

```
10 rem ***set individual subscripts***
20 X(0) = 25 : X(1) = 10
30 X(2) = 13 : X(3) = 51
40 X(4) = 24 : X(5) = 33
50 rem ***display array values***
60 FOR I = 0 TO 5
70 PRINT "X("; I; ") = "; X(I)
80 NEXT
RUN
X( 0 ) = 25
X( 1 ) = 10
X( 2 ) = 13
X( 3 ) = 51
X( 4 ) = 24
X( 5 ) = 33
Ok
```

Using the DIM statement

The DIM statement is used to specify the maximum value for the subscript of an array variable and allocate storage accordingly.

If an array is used without a DIM statement, the maximum value of its subscript is assumed to be 10. If you attempt to enter an array variable with the subscript of 11, you will get a Subscript out of range error message.

If you wish to have an array with a subscript greater than 10, you need to use the DIM statement.

Example:

```
10 REM ***DIM Statement***
20 DIM A(20)
30 FOR X = 1 TO 15
40 A(X) = X
50 PRINT A(X);
60 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Ok
```

In the above program, the A(X) array is dimensioned to have a subscript of 0 to 20. You can have up to 20 as your subscript but no greater. If you try to use 21 as a subscript, you will get the error message Subscript out of range.

An array can be dimensioned only once during the run of a program. The system will give an error message and stop executing the program if it comes to a DIM statement for the same array a second time. However, you can use more than one DIM statement in a program if you are dimensioning different arrays.

Example:

Not allowed

```
10 DIM X(20)
20 FOR I = 1 TO 15:X(I) = I:PRINT X(I);:NEXT
30 DIM X(30)
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Duplicate Definition in 30
Ok
```

Allowed

```
10 DIM X(20)
20 FOR I = 1 TO 15:X(I) = I:PRINT X(I);:NEXT
30 PRINT
40 DIM Y(30)
50 FOR I = 1 TO 10:Y(I) = I:PRINT Y(I);:NEXT
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 1 2 3 4 5 6 7 8 9 10
Ok
```

More Information on the DIM Statement

(*Note: The two example programs below are not working examples. They only show the format of how to use the statement.*)

Variables and expressions can also be used to dimension arrays.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B)
30 DIM Y$(A)
40 DIM Z(C + 11)
```

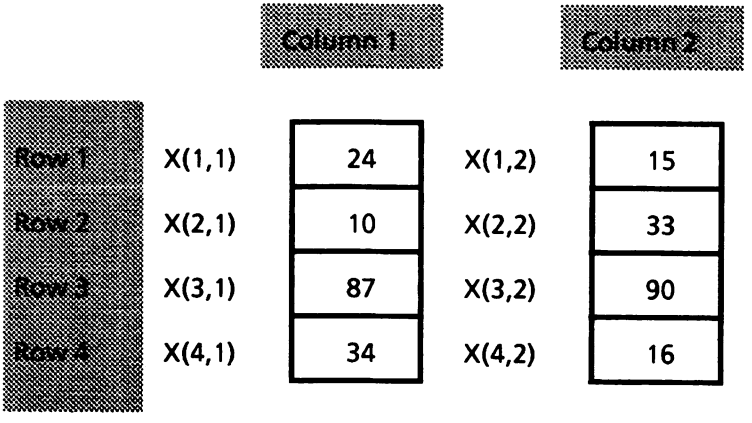
All the DIM statements can be on one line separated by commas.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B), DIM Y$(A), DIM Z(C + 11)
```

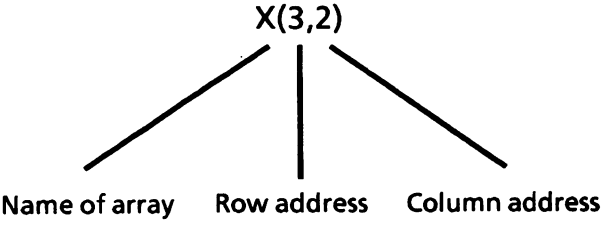
Two-dimensional array

The two-dimensional array is similar to the one-dimensional array except that is has two subscripts instead of one (e.g., X(1,4)). Think of it as an array made up of rows and columns.



where the value of X(1,1) is 24, X(3,2) is 90, X(3,1) is 87, and so on.

To get a data item out of the array, you must address it. This is done by telling the system which row and column to look for.



As with one-dimensional arrays, variables can be used for subscripts. For example, the subscripted variable, X(A,B), has the variables A and B for subscripts.

If A=2 and B=2 then X(A,B) would be X(2,2), the value of X(2,2)=33.

If A=4 and B=2 then X(A,B) would be X(4,2), the value of X(4,2)=16.

You can also use expressions as subscripts. For example, the subscripted variable X(Y-2,Y-1) when Y=3. After the system calculates the subscripts, the subscripted variable would be X(1,2) and the value of X(1,2)=15.

Assigning values to an array

The INPUT or READ/DATA statements with a FOR-NEXT loop can be used to assign values to an array.

Using the INPUT statement

In the example below, the INPUT statement is used to fill the example array given on page 3-82.

Example:

```
10 FOR R = 1 TO 4
20 FOR C = 1 TO 2
30 PRINT "Enter value for X("; R; ",";C;")";
40 INPUT X(R,C)
50 NEXT C, R
RUN
Enter value for X( 1 , 1 )? 24
Enter value for X( 1 , 2 )? 15
Enter value for X( 2 , 1 )? 10
Enter value for X( 2 , 2 )? 33
Enter value for X( 3 , 1 )? 87
Enter value for X( 3 , 2 )? 90
Enter value for X( 4 , 1 )? 34
Enter value for X( 4 , 2 )? 16
Ok
```

Notice that the counters R and C in the FOR statements are also used for the subscripts in X(R,C) causing the subscripts to be incremented by one each time through the loop (e.g., data is assigned to the array as follows: X(1,1), X(1,2), X(2,1), X(2,2), etc.).

The system asks for input eight times. After the values have been entered, value 24 is assigned to X(1,1), value 15 is assigned to X(1,2), value 10 is assigned to X(2,1), and so on.

Using READ/DATA statements

In the example below, READ/DATA statements are used to fill the example array with data.

Example:

```
10 FOR R = 1 TO 4
20 FOR C = 1 TO 2
30 READ X(R,C)
40 PRINT "X("; R; ","; C; ") = "; X(R,C),
50 NEXT C
60 PRINT
70 NEXT R
80 DATA 24, 15, 10, 33, 87, 90, 34, 16
RUN
X( 1 , 1 ) = 24          X( 1 , 2 ) = 15
X( 2 , 1 ) = 10          X( 2 , 2 ) = 33
X( 3 , 1 ) = 87          X( 3 , 2 ) = 90
X( 4 , 1 ) = 34          X( 4 , 2 ) = 16
Ok
```

This program is the same as the one on the previous page except that instead of inputting data via the keyboard you are reading the data from the program.

Direct input

A third way to fill an array with data is to store a constant value directly into the array.

Example:

```
10 rem ***set individual subscripts***
20 X(1,1) = 24 : X(1,2) = 15
30 X(2,1) = 10 : X(2,2) = 33
40 X(3,1) = 87 : X(3,2) = 90
50 X(4,1) = 34 : X(4,2) = 16
60 rem ***display array values***
70 FOR R = 1 TO 4
80 PRINT
90 FOR C = 1 TO 2
100 PRINT "X("; R; ","; C; ") = "; X(R,C),
110 NEXT C, R
RUN

X( 1 , 1 ) = 24          X( 1 , 2 ) = 15
X( 2 , 1 ) = 10          X( 2 , 2 ) = 33
X( 3 , 1 ) = 87          X( 3 , 2 ) = 90
X( 4 , 1 ) = 34          X( 4 , 2 ) = 16
Ok
```

Using the DIM statement

The purpose of the DIM statement is to specify the maximum values for array variable subscripts and allocate storage accordingly. As with the one-dimensional array, if the array subscripts is greater than 10, the two-dimensional array subscripts have to be dimensioned.

Example:

```
10 REM ***DIM Statement***
20 DIM A(20,15)
30 FOR X = 1 TO 20
40 FOR Y = 1 TO 15
50 A(X,Y) = X
60 NEXT Y, X
```

An array can be dimensioned only once during the run of a program. The system will give an error message and stop executing the program if it comes to a DIM statement for the same array a second time. However, you can use more than one DIM statement in a program if you are dimensioning different arrays.

More Information on the DIM Statement

(*Note: The two example programs below are not working examples. They only show the format of how to use the statement.*)

Variables and expressions can also be used to dimension arrays.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B,A)
30 DIM Y$(A,C)
40 DIM Z(C + 11,A-2)
```

All the DIM statements can be on one line separated by commas.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B,A), DIM Y$(A,C), DIM Z(C + 11,A-2)
```

Summary

- A one-dimensional array consists of an array variable name followed by a subscript enclosed in parentheses (referrred to as a subscripted variable).
- A two-dimensional array consists of an array variable name followed by two subscripts separated by commas and enclosed in parentheses (referred to as a subscripted variable).
- Variables and expressions can be used as subscripts.
- The INPUT and READ/DATA statements within Nested FOR-NEXT loops are used to assign values to an array.
- The DIM statement is used to specify the maximum value for an array variable subscript and allocate storage accordingly.
- An array can be dimensioned only once during the run of a program.
- Variables and expressions can also be used to dimension an array.

Notes:

7. SUBROUTINES

A subroutine is a group of statements within a program that perform a particular function.

The statement that tells the system to go to a subroutine is the GOSUB statement. It is followed by a line number that corresponds to the first program line in the subroutine.

The last statement in a subroutine is the RETURN statement. It automatically causes the system to return to the main program, that is, to the line number immediately following the GOSUB statement that originally called the subroutine. A subroutine may contain more than one RETURN statement should logic dictate a return at different points in the subroutine. **Never exit a subroutine with a GOTO statement.**

A subroutine may be called any number of times during a program.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. The use of REM statements to tell what each subroutines does is very helpful.

To prevent inadvertent entry into a subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. In the program on the next page, the END statement is used.

Subroutines are very helpful when you need to do the same thing more than once in a program. They save coding time in that you only enter the lines once.

Example:

```
10 A = 25 : B = 30
20 GOSUB 100
30 PRINT C
40 END
100 REM **multiplication subroutine**
110 C = A * B
120 RETURN
RUN
 750
Ok
```

Program flow

```
10 A = 25 : B = 30
```

Values are assigned to the variables A and B.

```
20 GOSUB 100
```

A subroutine is called. The program must go to line 100 to continue.

```
100 REM **multiplication subroutine**
```

Subroutine description.

```
110 C = A * B
```

The value of C is calculated.

```
120 RETURN
```

This tells the system to return to the program line immediately after the GOSUB statement that called this subroutine. Go to program line 30 and continue.

```
30 PRINT C
```

The value of C is displayed on the screen.

```
40 END
```

The program is ended and the system returns to command level.

Nested subroutines

A subroutine may be called from within another subroutine. This is called a nested subroutine.

Example:

```
10 A = 25 : B = 30
20 GOSUB 100
30 PRINT C, D
40 END
100 REM **multiplication subroutine**
110 C = A * B
120 GOSUB 200
130 RETURN
200 REM **addition subroutine**
210 D = A + B
220 RETURN
RUN
 750          55
Ok
```

Program flow

```
10 A = 25 : B = 30
```

Values are assigned to the variables A and B.

```
20 GOSUB 100
```

A subroutine is called. The program must go to line 100 to continue.

```
100 REM **multiplication subroutine**
```

Subroutine description.

```
110 C = A * B
```

The value of C is calculated.

```
120 GOSUB 200
```

A second subroutine is called. The program must go to line 200 to continue.

```
200 REM **addition subroutine**
```

Subroutine description.

```
210 D = A + B
```

The value of D is calculated.

```
220 RETURN
```

This tells the system to return to the program line immediately following the GOSUB statement that called this subroutine. Go to program line 130 and continue.

```
130 RETURN
```

The program must go to the line immediately following the GOSUB statement that called this subroutine; that is, go to program line 30 and continue.

```
30 PRINT C, D
```

Displays the value of C and D on the screen.

```
40 END
```

The program is ended and the system returns to command level.

Summary

- A subroutine is a group of statements within a program that perform a particular function and may be accessed from any point within the program.

- A subroutine
  - -- may be called any number of times during a program
  - -- starts with a GOSUB statement and ends with a RETURN statement
  - -- may contain more than one RETURN statement depending on logic flow
  - -- may appear anywhere in the program
  - -- may be called from within another subroutine

- Never exit a subroutine with a GOTO statement.

- To prevent inadvertent entry into the subroutine, you may precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Notes:

WHERE TO GO FROM HERE

In this tutorial, you have covered only the bare basics of GW-BASIC. Now you can go on to the Reference Guide section of this manual to learn more detailed information on the commands and statements that you have encountered in this tutorial.

Don't be afraid of your personal computer and GW-BASIC. The best way to learn is to "play" with it. Start with a small program and keep modifying and adding to it until you have a large program.

Also, there are a lot of good books written about learning the BASIC language that you can purchase from bookstores or computer stores.

Junior colleges in most areas are offering introductory courses in computer languages.

Notes:

GENERAL INFORMATION

TABLE OF CONTENTS

1. MODES OF OPERATION

After GW-BASIC is loaded into memory and displays the **Ok** prompt, it is ready for instructions from you. This is known as being at *command level.*

At this point, the GW-BASIC Interpreter may be used in either of two modes: the direct mode or the program mode.

Direct mode

In the direct mode, GW-BASIC statements and commands are executed as they are entered. They are not preceded by line numbers. After each direct statement followed by a carriage return, the screen will display the "Ok" prompt. 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 the GW-BASIC Interpreter as a "calculator" for quick computations that do not require a complete program.

Examples:

```
Ok
PRINT 32 * 3 + 10
 106
Ok
A = 15.21
Ok
```

(Assigns the constant 15.21 to the variable A. You can use A in successive computations to represent this value.)

Program mode

The program mode is used for entering programs. To let GW-BASIC know that you are entering a program line, you will begin with a line number. The program line is stored in memory. The program currently stored in memory can be executed by entering the RUN command.

Example:

```
Ok
10 A = 32
20 B = 3
30 C = 10
40 PRINT A * B + C
RUN
 106
Ok
```

Program Lines and Line Numbers

GW-BASIC program lines may contain a maximum of 255 characters and have the following format (square brackets indicate optional input):

nnnnn statement [:statement...] [' comment]

A GW-BASIC program line always begins with a line number (nnnnn, an unsigned integer in the range 1 to 65529), and ends with a carriage return. A program line is stored in memory as soon as you enter the carriage return. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references when branching and editing.

If a program line contains more than 255 characters, the extra characters are truncated when the carriage return is pressed. Even though the extra characters are displayed on the screen, they are not processed by GW-BASIC.

A *statement* is either executable or non-executable. Executable statements are program instructions that tell BASIC what to do next while running a program. For example, PRINT X is an executable statement. Non-executable statements, such as DATA or REM, do not cause any program action when GW-BASIC sees them.

Statements in a program line may be entered in either upper or lowercase, or a combination of both. GW-BASIC will convert everything to uppercase, except for remarks, DATA statements, and strings enclosed in quotation marks.

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

Example:

```
Ok
100 A = 32:B = 3:C = 10
200 PRINT A*B + C
RUN
 106
Ok
```

At the end of a GW-BASIC line (before the carriage return), you may enter a comment string preceded by a single quotation mark (').

A comment string preceded either by the keyword REM or by a single quotation mark may also be written just after the line number.

It is possible to extend a logical program line over more than one physical screen line by using the line feed (CTRL CR, CR is the carriage return key). Line feed lets you continue typing a logical program line on the next physical screen line without entering a carriage return. The program line is not processed until a carriage return is entered. When there is more than one statement in a program line, using a line feed can make the program line more readable. See example on the next page.

Examples:

Using the line feed

```
10 FOR I = 1 TO 10:
PRINT I;:
NEXT I
```

Not using the line feed

```
10 FOR I = 1 TO 10:PRINT I;:NEXT I
```

Also, you may type up to 255 characters on a logical program line without issuing either a line feed or a carriage return. The text is wrapped and continues on the next physical screen line.

2. CHARACTER SET

GW-BASIC recognizes the following sets of characters:

- alphabetic characters (upper and lowercase letters of the alphabet)
- numeric characters (the digits 0 through 9)
- special characters (see below and next page)

Character	Name
	blank
=	equal sign or assignment symbol
+	plus sign
-	minus sign
*	asterisk or multiplication symbol or "and"
/	slash or division symbol
\	backslash or integer division symbol
^	up arrow or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent sign
#	number or pound sign
$	dollar sign
!	exclamation point
[	left bracket
]	right bracket
.	period or decimal point
,	comma
;	semicolon
:	colon

'	single quotation mark (apostrophe)
"	double quotation mark
&	ampersand
?	question mark or symbol that can be used for PRINT
<	less than
>	greater than
@	at-sign
_	underscore

See "Appendix A - ASCII Character Codes" for a complete list of the characters that can be printed or displayed by GW-BASIC.

3. CONSTANTS, VARIABLES AND EXPRESSIONS

This chapter illustrates the principal elements that may be entered in a GW-BASIC line. They are:

Constants

Variables

Expressions

Constants

Constants are the values GW-BASIC uses during program 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"
"$25,000.000"
"Number of Employees"

Numeric constants

Numeric constants are positive or negative numbers. A plus sign (+) is optional with a positive number. GW-BASIC numeric constants cannot contain commas. There are five types of numeric constants:

Integer

Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

Fixed-Point

Positive or negative real numbers, i.e., numbers that contain decimal points.

Floating-Point

Positive or negative numbers represented 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 (single precision) or D (double precision) and an optionally signed integer (the exponent). The letter E (or D) means "times ten to the power of". The range for floating-point constants is 10E-38 to 10E+38. Study the examples below.

Examples:

50E-5

where **50** is the mantissa, **E** means single precision, and **-5** is the exponent. This floating point constant could be read as "50 times ten to the -5 power". Examine the program below.

```
Ok
10 PRINT 50E-5
20 PRINT 50 * 10 ^ -5
RUN
 .0005
 .0005
Ok
```

Another way to find the equivalent of a floating point constant is to look at the exponent.

If the exponent is negative, start with the decimal point and move it to the **left**, adding zeroes, if necessary. For example:

235.988E-4 is equivalent to .0235988
235E-4 is equivalent to .0235

If the exponent is positive, start with the decimal point and move it to the **right**, adding zeroes, if necessary. For example:

235.988E4 is equivalent to 2359880
235E4 is equivalent to 2350000

Hex

Hexadecimal numbers begin with the prefix &H and can have up to four digits. Hexadecimal digits are 0-9 and A-F. For example:

```
&H76
&H32F
&HFFAA
```

Octal

Octal numbers begin with the prefix &O or just & and can have up to six digits. Octal digits are 0-7. For example:

```
&O347
&1234
```

Single and double precision form for numeric constants

Numeric constants may be either single 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 numeric constants are stored with 16 digits of precision and printed with up to 16 digits of precision.

A single precision constant is any numeric constant that has:

- seven or fewer digits, or
- exponential form using E, or
- a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

- eight or more digits, or
- exponential form using D, or
- a trailing number sign (#)

Examples:

Single Precision	Double Precision
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Variables

Variables are names used to represent values that are used in a GW-BASIC program. There are two types of variables: numeric and string.

The value of a variable may be assigned explicitly by you, or it may be assigned as the result of calculations in the program. When assigning data to a variable, the types must match. For example, you cannot assign a character string to a numeric variable.

Before a numeric variable is assigned a value, its value is assumed to be zero. String variables are assumed to be null---contain no characters and have a length of zero.

Variable names

GW-BASIC variable names may be any length. However, only the first 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. **The first character must be a letter.** Special type declaration characters are allowed as the last character of the name.

A variable name may not be a reserved word, but embedded reserved words are allowed. A reserved word is a word that has special meaning to GW-BASIC. Reserved words include all GW-BASIC commands, statements, function names and operator names. (*See Appendix D for a complete list of GW-BASIC reserved words.*) For example, you could not have a variable name of READ but you could have a variable name of BREAD.

If a variable begins with FN, it is assumed to be a call to a user-defined function.

Declare variable types

Variables may represent either a numeric value or a string. Also, if numeric, what precision it is.

String Variable Declaration

String variable names are 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 Declaration

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.

Integer variables produce the faster and most compact object code. For example, the following program executes approximately 30 times faster when the loop control variable "I" is replaced with "I%".

```
10 FOR I = 1 TO 10
20       A(I) = 0
30 NEXT I
```

See the examples on the next page of GW-BASIC variable names.

Examples:

PI#	declares a double precision value
PAY!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
X	represents a single precision value

There is a second method by which variable types may be declared. The GW-BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types of certain variable names. These statements are described in detail in Chapter 19.

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-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. *For more detailed information on arrays, see Chapter 2 in the Reference section of this manual.*

Memory requirements

Variables Type	Bytes
Integer	2
Single Precision	4
Double Precision	8
Array Type	
Integer	2 per element
Single Precision	4 per element
Double Precision	8 per element

Strings:
3 bytes overhead plus the present contents of the string

Type conversion

When necessary, GW-BASIC will convert a numeric constant from one type to another. The following rules and examples should be observed:

1. If a numeric constant of one type is assigned to a numeric variable of a different type, the number will be stored as the type declared in the variable name. For example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
Ok
```

If a string variable is assigned to a numeric value or vice versa, a Type mismatch error occurs.

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

```
10 D# = 6#/7
20 PRINT D#
RUN
.8571428571428571
Ok
```

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

```
10 D = 6#/7
20 PRINT D
RUN
.8571429
Ok
```

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

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

```
10 C% = 55.88
20 PRINT C%
RUN
 56
Ok
```

4. 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. For example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04    2.039999961853027
Ok
```

5. Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs. A full description of Logical Operators follows later in the chapter.

Expressions and operators

An expression may be a string or numeric constant, or 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. The GW-BASIC operators may be divided into five categories:

Arithmetic
Relational
Logical
Functional
String

Arithmetic operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating-Point Division	X*Y X/Y
\	Integer Division	X\Y
MOD	Modulus Arithmetic	X MOD Y
+,-	Addition, Subtraction	X+Y X-Y

If arithmetic operators of the same priority are in the same equation, then the system will work from left to right on these operators.

How To Change Priority of Arithmetic Operators

Priority of arithmetic operators can cause problems. For example, if you needed to add or subtract before you multiply or divide, how could this be done?

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Within the parentheses, the usual order of operations is maintained.

Here are a few rules that apply to the use of parentheses.

- Parentheses force the innermost calculations to be accomplished first.
- The number of **left** parentheses **must** equal the number of **right** parentheses.
- Extra pair of parentheses have no effect. (So if they help you, do not hesitate to use them.)

On the next page are some examples of algebraic formulas and the way they would look as GW-BASIC statements.

Formula	Statement
$A + \frac{B}{C}$	10 E = A + B/C

In the example below, the addition needs to take place before the division is performed. So B+C is put in parentheses in the statement.

$\frac{A}{B+C}$	15 W = A/(B + C)

In the next two examples, can you see why the parentheses are used.

$\frac{A \cdot B}{C \cdot D}$	45 X = (A*B)/(C*D)
$\frac{D-B}{6 \cdot A}$	35 F = (D-B)/(6*A)

Integer Division and Modulus Arithmetic

Two additional operators are available in GW-BASIC: integer division and modulus arithmetic.

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

Examples:

```
Ok
PRINT 10\4
 2
Ok
PRINT 25.68\6.99
 3
Ok
```

Integer division follows multiplication and floating-point division in order of precedence.

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

Examples:

```
PRINT 10.4 MOD 4
 2              (10\4 = 2 with a remainder of 2)
Ok
PRINT 25.68 MOD 6.99
 5              (26\7 = 3 with a remainder of 5)
Ok
```

Division By Zero and Overflow

If, during the evaluation of an expression, a division by zero is encountered, the "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 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. For example:

```
10 A = 5:B = 10
20 C = A/0:D = A*B:PRINT C,D
RUN
Division by zero
 1.701412E+38          50
Ok
```

If overflow occurs, the Overflow error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues. For example:

```
10 A = 1.701411834000001D+38:B = 8
20 C = 10:D = 15
30 E = B/A:F = C + D:PRINT E,F
RUN
Overflow
 4.701978E-38       25
Ok
```

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 IF statements in Chapter 5 of the Reference section of this manual).

Operator	Relation Tested	Expression
=	Equality	X=Y
<> (or ><)	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<= (or =<)	Less than or equal to	X<=Y
>= (or=>)	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable.)

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

Examples:

```
50 If X + Y < (T-1)/Z then print "true"
```

The above expression is true (-1) if the value of X plus Y is less than the value of T-1 divided by Z.

```
200 IF SIN(X) < 0 GOTO 1000

500 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 returns a bitwise 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 the following table. The operators are listed in order of precedence.

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 subsequent decision.

Examples:

```
20 IF D<200 AND F>4 THEN 80

100 IF I = 10 OR K = 0 THEN 50

60 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 in bit-by-bit, i.e., 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 — the result is 16, as

```
63 = binary  111111
16 = binary  010000
             ------
             010000
```

15 AND 14 = 14 — the result is 14, as

```
15 = binary  1111
14 = binary  1110
             ----
             1110
```

-1 AND 8 = 8 — the result is 8, as

```
-1 = binary  1111111111111111
8 = binary   0000000000001000
             ----------------
             0000000000001000
```

4 OR 2 = 6 — the result is 6, as

```
4 = binary   100
2 = binary    10
             ---
             110
```

10 OR 10 = 10 — the result is 10, as

```
10 = binary  1010
10 = binary  1010
             ----
             1010
```

-1 OR -2 = -1 — the result is -1, as

```
-1 = binary  1111111111111111
-2 = binary  1111111111111110
             ----------------
             1111111111111111
```

NOT X = -(x + 1) — The two's complement of any integer is the bit ocmplement plus one.

For example:

1111100110011000	original value (negative)

Invering all the bits:

0000011001100111	inverted value

Adding 1

0000011001101000	absolute value (inverted + 1)

So the value of the given pattern is: -1640

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 the Referecne section of the manual.

GW-BASIC also allows "user-defined" functions that are written by the programmer. They are discussed in Chapter 27 of the Reference section.

String operators

Strings may be concatenated using the plus sign (+).

Example:

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

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

=	equal	< >	not equal
<	less than	< =	less than or equal to
>	greater than	> =	greater than or equal to

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes (*see Appendix D for a complete list of 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.

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.

See examples on next page.

Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH"<"SMYTHE"
B$ < "9/12/84"
        where B$ = "8/12/84"
```

All of the above are true statements.

Notes:

4. DISK FILE HANDLING

Wherever a file name is required in a disk command or statement, use a name that conforms to the file naming conventions for your operating system. The MS-DOS operating system will append a default extension .BAS to the file name given in a SAVE, RUN, MERGE or LOAD command.

Device independent input/output

GW-BASIC provides device-independent input/output that permits flexible approaches to data processing. Using device independent I/O means that the syntax for access is the same for any device.

The following statements, commands, and functions support device-independent I/O (see individual descriptions in the Reference section).

BLOAD
BSAVE
CHAIN
CLOSE
EOF
GET
INPUT#
INPUT$
LINE INPUT#
LIST
LOAD
LOC
LOF
MERGE
OPEN
POS
PRINT#
PRINT# USING
PUT
RUN
SAVE
WIDTH
WRITE#

How MS-DOS keeps track of your files

A file is a collection of records. The names of files are kept in directories on disk. These directories also contain information on the size of the files, their location of the disk, and the dates that they were created and updated. The directory you are working in is called your current "working" directory.

An additional system area is called the File Allocation Table. It keeps track of the location of your files on the disk. It also allocates the free space on your disk so that you can create new files.

These two system areas, the directories and the File Allocation Table, enable MS-DOS to recognize and organize the files on your disks. The File Allocation Table is created onto a new disk when you format it and one empty directory is created, known as the "root" directory.

Naming files

File specifications follow MS-DOS naming conventions. The *filespec* is a string expression with the following format;

[*device*:]*filename*

All filespecs may begin with a device specification such as A: or B: or COM1: or LPT1:. If no device is specified, the current drive is assumed.

A file name can comprise:

-- one to eight characters. For example, NEWFILE.

-- one to eight characters, followed by a period (.) and a one to three character file name extension. For example, NEWFILE.DAT.

A file name may be made up of any of the following characters:

A-Z	0-9	$	&	#	~
%	'	(	)	-	_
@	^	{	}	!	

Alphabetic characters within the file name can be entered in upper or lower case, but MS-DOS will translate lower case letters into upper case.

GW-BASIC supplies the extension .BAS if no extension is given, but NAME and KILL do not follow this rule: they do not supply any extension.

File specification for communications devices is slightly different. The file name is replaced with a list of options specifying such things as line speed. *Refer to Chapter 4 of the Reference section for details.*

Remember that if you use a string constant for the filespec, you must enclose it in quotation marks. The only exception to this rule is the GWBASIC command, where a file specifier is string constant not included in quotation marks.

Examples:

```
LOAD "B:ARSENAL.RED"

GWBASIC PAYROLL
```

Naming devices

The device name is a string of four characters or less followed by a colon (:), and may be one of the following:

A: first disk drive	(any access mode)
B: second disk drive	(any access mode)
C: first hard disk drive	(any access mode)
D: second hard disk drive	(any access mode)
KYBD: keyboard	(Input only)
SCRN: screen	(Output only)
LPT1: first printer	(Output or random)
LPT2: second printer	(Output or random)
LPT2: third printer	(Output or random)
COM1: RS232 Com 1	(Input and Output)
COM2: RS232 Com 2	(Input and Output)
COM3: RS232 Com 3	(Input and Output)
COM4: RS232 Com 4	(Input and Output)

5. CONTROL CHARACTERS

You can generate control characters by holding down the CTRL or ALT key while pressing another key. GW-BASIC recognizes the following control characters:

CTRL BREAK

Can be used for different purposes:

-- To interrupt the program at the following GW-BASIC instruction and return to GW-BASIC comand level.

-- To cancel automatic line numbering mode while entering a program.

-- To return to command level, without saving any changes that you made to the current line.

CTRL G

Sounds the bell

CTRL NUM LOCK

Causes the system to pause so as to temporarily halt printing or program listing. The pause continues until you press any key (except SHIFT, CTRL or ALT).

CTRL T

Scrolls the function key display horizontally across the screen (on the 25th screen line), when the width is 40. When the width is 80, it toggles the function key display ON and OFF.

ALT CTRL DEL

Perform a System Reset by holding down the **CTRL** and **ALT** keys, and then pressing the **DEL** key.

CTRL PRTSC

All text sent to the screen is also sent to the system printer. You can stop printing by repeating the key sequence.

If you press **PRTSC** while holding down **SHIFT**, MS-DOS will make a single printed copy of the entire display screen.

To print both text and graphics, you have to use the MS-DOS GRAPHICS command before entering GW-BASIC.

CTRL L

Outputs a formfeed character. It has the same function as the CLS statement, i.e., it clears the screen or the current graphics viewport (if a viewport has been defined).

CTRL Z

Sets an end of file condition.

Other control characters are described in Chapter 12 of the Reference section.

Direct Entry of GW-BASIC Keywords

A GW-BASIC Keyword is entered by holding down the **ALT** key while pressing one of the alphabetic keys (A-Z). Keywords associated with each letter are listed below:

A-AUTO	N-NEXT
B-BLOAD	O-OPEN
C-COLOR	P-PRINT
D-DELETE	Q-****
E-ELSE	R-RUN
F-FOR	S-SCREEN
G-GOTO	T-THEN
H-HEX$	U-USING
I-INPUT	V-VAL
J-****	W-WIDTH
K-KEY	X-XOR
L-LOCATE	Y-IIII
M-MERGE	X-****

**** = unused keys

Notes:

6. SYNTAX CONVENTIONS

1. Uppercase letters and words, and the symbols listed below, should be typed in the actual line exactly as shown.

(	)
,	;
:	;;
=	/
\	#
$	-
<	>

In the statement:

WRITE#*filenum, list-of-expressions*

WRITE# and the comma (,) after *filenum* should be typed as shown.

2. Lowercase italics letters and words represent "variable information" (or "parameters") that you must provide.

In the statement:

KILL *filespec*

filespec should be replaced by a specified value; for example, "MYFILE.DAT".

3. The symbols listed below are used to define the syntax of a line, but should not be typed in the actual line:

|	vertical stroke ("or" sign), to indicate alternatives
{ }	braces, to indicate a choice
[ ]	brackets, to indicate optional
...	ellipsis, to indicate repetition
-	hypen, to join multiple-name parameters.

In some statements or commands (e.g., LIST, LLIST, etc.), the hyphen is used as an operator to separate parameters. In this case, bold face is used to distinguish hyphens that are used for this purpose from hyphens used to join multiple-name parameters.

REFERENCE

Notes:

TABLE OF CONTENTS

1. ALPHABETICAL LISTING

This chapter is an alphabetical listing of all the GW-BASIC commands, statements and functions with a short description of each and the page number where they are described. Some commands, statements and functions can be used to do more than one task, they will have more than one page number listed.

CLEAR command, 5-311
Clears all numeric variables to zero, all string variables to null, and closes all open files. Options sets the highest memory location available for use by GW-BASIC and the amount of stack space.

CLOSE statement, 5-116, 5-143
Terminates I/O to a file (5-143) or device (5-116).

CLS statement, 5-380
Erases all or part of the screen.

COLOR (medium resolution mode) statement, 5-231
Defines the palette background and foreground colors. In addition, the default grapics foreground and background colors, and the text foreground color can be defined.

COLOR (high resolution mode) statement, 5-234
Defines the (default) graphics foreground, the graphics background and the text foreground colors.

COLOR (super resolution mode) statement, 5-236
Defines the (default) graphics foreground, the graphics background and the text foreground colors.

COLOR (text mode) statement, 5-382
Sets the text foreground and background colors.

COM(*n*) statement, 5-66
COM(n) ON enables, COM(n) OFF disables, and COM(n) STOP suspends event trapping of communications activity on the specified channel.

COMMON statement, 5-95
Defines a common area which is not erased by the CHAINed program, and allows you to pass variables from one program to another.

CONT command, 5-422
Resumes program execution after a CTRL BREAK has been typed or a STOP or END statement has been executed.

COS function, 5-370
Returns the cosine of the argument.

CSNG function, 5-106
Converts any numeric argument to a single precision number.

CSRLIN function, 5-385
Returns the current line (row) position of the cursor.

CVI, CVS, CVD functions, 5-144
Convert string values to numeric values.

DATA statement, 5-290
Creates an "internal" file, i.e., a sequence of data belonging to the program. Each data item will then be assigned to a program variable by a READ statement.

DATE$ function, 5-313
Retrieves the current date.

DATE$ statement, 5-313
Sets the current date.

DEF FN statement, 5-449
Defines and names a user-written function.

DEF SEG statement, 5-46
Assigns the current "segment" address.

DEF USR statement, 5-47
Enables access to a machine language subroutine by specifying the starting address.

DEFINT/SNG/DBL/STR statement, 5-315
Declares the variable type in accordance with the letter(s) specified.

DELETE command, 5-199
Erases program lines.
DIM statement, 5-21
Specifies the array name(s), the number of dimensions and the subscript(s) upper bound per dimension.
DRAW statement, 5-238
Draws an object as defined by a sequence of single character commands. (graphics mode only)

EDIT command, 5-200
Lets you change a specified program line.
END statement, 5-420
Terminates program execution, closes all open data files, and returns to command level.
ENVIRON statement, 5-316
Allows modification of parameters in GW-BASIC's Environment String Table.
ENVIRON$ function, 5-318
Allows you to retrieve the specified Environment String from GW-BASIC's Environment String Table.
EOF function, 5-61 (com file), 5-145 (data file)
Indicates that the end of a file has been reached.
ERASE statement, 5-26
Releases space and variable names previously reserved for arrays.
ERDEV function, 5-117, 5-202
Holds the actual value of a device error.
ERDEV$ function, 5-117, 5-202
Holds the name of the device causing the error if it was a character device.
ERR function, 5-203
Returns the error code.
ERL function, 5-203
Returns the number of the line which contains the error.
ERROR statement, 5-205
Simulates the occurrence of a GW-BASIC error, or generates a user-defined error.
EXP function, 5-371
Returns e (base of natural logarithms) to the power of the argument.

FIELD statement, 5-146
Allocates space for variables in a random file buffer.
FILES command, 5-175
Displays the names of files residing on the specified directory.
FIX function, 5-372
Returns the truncated integer part of the argument.
FOR...NEXT statements, 5-304
Allows a series of statements to be performed in a loop a given number of times.
FRE function, 5-320
Returns the number of bytes in memory not being used by GW-BASIC.

GET (COM files), 5-62
Reads a specified number of bytes into the communications buffer.
GET (files), 5-149
Reads a record from a random disk file into a random buffer.

GET (graphics), 5-243
Reads points from a screen area.

GOSUB...RETURN statements, 5-74
GOSUB transfers control to a GW-BASIC subroutine by branching to the specified line. RETURN transfers control to the statement following the most recent GOSUB (or ON...GOSUB) executed.

GOTO statement, 5-76
Transfers control to a specified program line.

GWBASIC command, 5-321
Initializes GW-BASIC and the operating environment (GWBASIC is an MS-DOS command, not a GW-BASIC command).

HEX$ function, 5-107
Returns a string which represents the hexadecimal value of the decimal argument.

IF..GOTO..ELSE or IF..THEN..ELSE statements, 5-77
Makes a decision regarding program flow based on the result of a specified condition.

INKEY$ function, 5-295
Returns a one- or two-character string read from the keyboard or a null string if no character is pending at the keyboard.

INP function, 5-118
Returns the byte read from a port.

INPUT statement, 5-297
Allows input from the keyboard during program execution.

INPUT# statement, 5-151
Reads data items from a sequential disk file and assigns them to program variables.

INPUT$ function, 5-63, 5-153, 5-299
Returns a string of characters from the keyboard (5-299) or file (COM files 5-63; data files (5-153).

INSTR function, 5-440
Searches for the first occurrence of a given substring in a string and returns the position at which the match is found.

INT function, 5-373
Returns the largest integer that is equal to or less than the argument.

IOCTL statement, 5-119
Sends a "Control Data" string to a Character Device Driver anytime after the Driver has been OPENed.

IOCTL$ function, 5-121
Returns a "Control Data" string from a Character Device Driver that is OPEN.

KEY statement, 5-328
Sets a function key to automatically type any sequence of characters. Other options allow you to enable or disable the function key display from the 25th line, or to list the function key values.

KEY(*n*) statement, 5-83
KEY(n) ON enables, KEY(n) OFF disables, and KEY(n) STOP suspends event trapping of the specified key.

KILL command, 5-173
Deletes a disk file.

RESET command, 5-172
Closes all open data files on all drives.

RESTORE statement, 5-294
Permits DATA statements to be re-read either from the beginning of the internal file or from a specified line.

RESUME statement 5-209
Continues program execution after an error trapping routine has been performed.

RIGHT$ function, 5-445
Returns a substring from a specified string, extracting its rightmost characters.

RMDIR command, 5-354
Removes a directory from a specified disk.

RND function, 5-338
Returns a random number between 0 and 1.

RUN command, 5-178, 5-432
Runs the current program or loads a program from disk and runs it.

SAVE command, 5-186, 5-435
Saves the current program on disk and gives it a name.

SCREEN function, 5-402
Returns the ASCII code (0-255) or the color number for the character at the specified screen location.

SCREEN statement, 5-267, 5-403
Sets the specifications for the display screen.

SGN function, 5-375
Returns 1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

SHELL command, 5-283
Loads and executes another program (.EXE, .COM. or .BAT).

SIN function, 5-376
Calculates the sine of the argument.

SOUND statement, 5-365
Produces sound via a speaker.

SPACE$ function, 5-446
Returns a string of a specified number of spaces.

SPC function, 5-408
Skips n spaces in a PRINT, LPRINT, or PRINT# statement.

SQR function, 5-377
Returns the square root of a positive numeric expression.

STOP statement, 5-421
Terminates program execution then returns to command level.

STR$ function, 5-109
Returns the string representation of the value of a specified numeric expression.

STRING$ function, 5-447
Returns a string of specified length whose characters all have the same ASCII code or equal the first character of a given string.

SWAP statement, 5-339
Exchanges the values of two variables.

SYSTEM command, 5-423
Closes all open data files and returns control to MS-DOS.

TAB function, 5-409
Tabs the cursor or the print head to a specified position, in PRINT, LPRINT, or PRINT# statements.

TAN function, 5-378
Returns the tangent of the argument.

TIME$ function, 5-340
Retrieves the current time.

TIME$ statement, 5-340
Sets the current time.

TIMER function, 5-342
Returns a single precision number indicating the seconds that have elapsed since midnight or system reset.

TIMER statement, 5-333
TIMER ON enables, TIMER OFF disables, and TIMER STOP suspends real time event trapping.

TRON command, 5-114
Causes the line number of each statement executed to be displayed.

TROFF command, 5-114
Stops the line number displaying initiated by TRON.

USR function, 5-39
Calls a machine language subroutine.

VAL function, 5-110
Converts the string representation of a number to its numeric value.

VARPTR function, 5-50, 5-169
VARPTR(*variable*) (page 5-50) returns the address of *variable*. VARPTR(*filenum*) (page 5-169) returns the starting address of the disk I/O buffer (for sequential files) or the starting address of the FIELD buffer (for random files).

VARPTR$ function, 5-343
Returns a character form of the memory address of the variable.

VIEW statement 5-272
Defines subsets of the screen called "viewports", into these, window contents will be mapped. (graphics mode only)

VIEW PRINT statement, 5-410
Sets the boundary of the text window.

WAIT statement, 5-126
Suspends program execution while monitoring the status of a machine input port.

WHILE...WEND statements, 5-307
Loop through a series of statements as long as a given condition remains true.

WIDTH statement, 5-127, 5-276, 5-411
Sets the line width in characters.

WINDOW statement, 5-279
Defines the logical dimensions of the current viewport. (graphics mode only)

WRITE statement, 5-416
Writes data to the screen.

WRITE# statement, 5-170
Writes data to a sequential file.

Notes:

2. ARRAYS

This chapter describes arrays and the statements that are used with arrays.

The subsections are

Definition of an array
One-dimension array
Two-dimension array
DIM statement
ERASE statement
OPTION BASE statement

Definition of an array

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-dimension array, T(1,4) would reference a value in a two-dimension array, and so on.

The maximum number of dimensions for an array is 255.

The maximum number of elements per dimension is 32,767. Both these values are also limited by the memory size of your system.

Memory Requirements

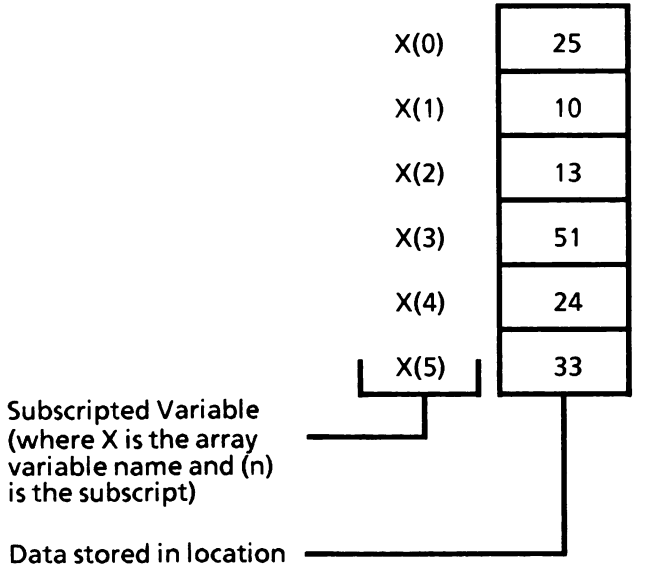
The number of bytes required by an array is listed below.

Array Type	Bytes
Integer	2 per element
Single Precision	4 per element
Double Precision	8 per element

One-dimension array

A one-dimension array is a group of related data items which are stored in memory and can be accessed by a subscripted variable. The subscripted variable consists of an array variable name followed by a subscript enclosed in parentheses. For example, X(5) is a subscripted variable; X5 is not.

The subscripted variable (as with the simple variable) names a memory location inside the system where to store data. Think of it as follows:



where the value of X(0) is 25, X(1) is 10, X(2) is 13, and so on.

This is helpful when you want to work with a large quantity of related data and it is easier working with one variable that is subscripted than having to use many variables (e.g., A, B, C, D, E, etc.) and trying to remember what each of these variables represent.

Variables as Subscripts

Variables can also be used for subscripts. For example, the subscripted variable, X(A), has the variable A for a subscript.

If A=5 then X(A) would be X(5), the value of X(5)=33.

If A=2 then X(A) would be X(2), the value of X(2)=13.

Expressions as Subscripts

You can also use expressions as subscripts. For example, the subscripted variable X(Y+3) when Y=1. After the system calculates the subscript, the subscript would be equal to four. So the subscripted variable would be X(4) and the value of X(4)=24.

See the DIM, ERASE, and OPTION BASE statements in this chapter for information on dimensioning arrays, redimensioning of arrays, and changing the default base of the subscript.

Assign values to a one-dimension array

The INPUT or READ/DATA statements with a FOR-NEXT loop can be used to assign values to a one-dimension array.

Using the INPUT Statement

In the example program below, the INPUT statement is used to fill the example array with data.

Example:

```
10 FOR Y = 0 TO 5
20 PRINT "Enter value for X("; Y; ")";
30 INPUT X(Y)
40 NEXT X
RUN
Enter value for X( 0 )? 25
Enter value for X( 1 )? 10
Enter value for X( 2 )? 13
Enter value for X( 3 )? 51
Enter value for X( 4 )? 24
Enter value for X( 5 )? 33
Ok
```

(*Note: The* **bold** *text is what you would enter at the keyboard followed by a Return.*)

Notice that the counter Y in the FOR statement is also used for the subscript in X(Y) causing the subscript to be incremented by one each time through the loop (e.g., first time through Y=0 so X(Y) is X(0), second time through Y=1 so X(Y) is X(1), and so on).

Using this program, the system asks for input six times. After six values have been entered, the first value 25 is assigned to X(0), second value 10 is assigned to X(1), third value 13 is assigned to X(2), and so on.

Using the READ/DATA Statements

In the example below, READ/DATA statements are used to fill the example array with data.

Example:

```
5 rem ***using READ/DATA statements
10 FOR Y = 0 TO 5
20 READ X(Y)
30 PRINT "X("; Y; ") = "; X(Y)
40 NEXT
50 DATA 25, 10, 13, 51, 24, 33
RUN
X( 0 ) = 25
X( 1 ) = 10
X( 2 ) = 13
X( 3 ) = 51
X( 4 ) = 24
X( 5 ) = 33
Ok
```

This program is the same as the one on the previous page except that instead of inputting the data via the keyboard you are reading the data from the program.

Using Direct Input

A third way to fill an array with data is to store a constant value directly into the array.

Example:

```
10 rem ***set individual subscripts***
20 X(0) = 25 : X(1) = 10 : X(2) = 13
30 X(3) = 51 : X(4) = 24 : X(5) = 33
40 rem ***display array values***
50 FOR I = 0 TO 5
60 PRINT "X("; I; ") = "; X(I)
70 NEXT
RUN
X( 0 ) = 25
X( 1 ) = 10
X( 2 ) = 13
X( 3 ) = 51
X( 4 ) = 24
X( 5 ) = 33
Ok
```

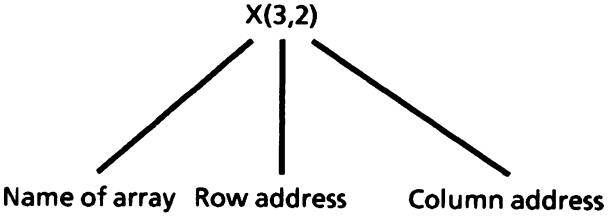
Two-dimension array

The two-dimension array is similar to the one-dimension array except that is has two subscripts instead of one (e.g., X(1,4)). Think of it as an array made up of rows and columns.

	Column 1		Column 2	
Row 1	X(1,1)	24	X(1,2)	15
Row 2	X(2,1)	10	X(2,2)	33
Row 3	X(3,1)	87	X(3,2)	90
Row 4	X(4,1)	34	X(4,2)	16

where the value of X(1,1) is 24, X(3,2) is 90, X(3,1) is 87, and so on.

To get a data item out of the array, you must address it. This is done by telling the system which row and column to look for.



Variables as Subscripts

As with one-dimension arrays, variables can also be used for subscripts. For example, the subscripted variable, X(A,B), has the variables A and B for a subscripts.

If A=2 and B=2 then X(A,B) would be X(2,2), the value of X(2,2)=33.

If A=4 and B=2 then X(A,B) would be X(4,2), the value of X(4,2)=16.

Expressions as Subscripts

You can also use expressions as subscripts. For example, the subscripted variable X(Y-2,Y-1) where Y=3. After the system calculates the subscripts, the subscripted variable would be X(1,2) and the value of X(1,2)=15.

See the DIM, ERASE and OPTION BASE statements in this chapter for information on dimensioning arrays, redimensioning of arrays and changing the default base of the subscript.

Assign values to a two-dimension array

The INPUT or READ/DATA statements with a FOR-NEXT loop can be used to assign values to a two-dimension array.

Using the INPUT Statement

In the example program below, the INPUT statement is used to fill the example array with data.

Example:

```
10 FOR R = 1 TO 4
20 FOR C = 1 TO 2
30 PRINT "Enter value for X("; R; ",";C;")";
40 INPUT X(R,C)
50 NEXT C, R
RUN
Enter value for X( 1 , 1 )? 24
Enter value for X( 1 , 2 )? 15
Enter value for X( 2 , 1 )? 10
Enter value for X( 2 , 2 )? 33
Enter value for X( 3 , 1 )? 87
Enter value for X( 3 , 2 )? 90
Enter value for X( 4 , 1 )? 34
Enter value for X( 4 , 2 )? 16
Ok
```

(*Note: The* **bold** *text is what you would enter at the keyboard followed by a Return.*)

Notice that the counters R and C in the FOR statements are also used for the subscripts in X(R,C) causing the subscripts to be incremented by one each time through the loop (e.g., data is assigned to the array as follows: X(1,1), X(1,2), X(2,1), X(2,2), etc.).

The system asks for input eight times. After the values have been entered, value 24 is assigned to X(1,1), value 15 is assigned to X(1,2), value 10 is assigned to X(2,1), and so on.

Using READ/DATA Statements

In the example below, READ/DATA statements are used to fill the example array with data.

Example:

```
10 FOR R = 1 TO 4
20 FOR C = 1 TO 2
30 READ X(R,C)
40 PRINT "X("; R; ","; C; ") = "; X(R,C),
50 NEXT C
60 PRINT
70 NEXT R
80 DATA 24, 15, 10, 33, 87, 90, 34, 16
RUN
X( 1 , 1 ) = 24          X( 1 , 2 ) = 15
X( 2 , 1 ) = 10          X( 2 , 2 ) = 33
X( 3 , 1 ) = 87          X( 3 , 2 ) = 90
X( 4 , 1 ) = 34          X( 4 , 2 ) = 16
Ok
```

This program is the same as the one on the previous page except that instead of inputting the data via the keyboard you are reading the data from the program.

Using Direct Input

A third way to fill an array with data is to store a constant value directly into the array.

Example:

```
10 X(1,1) = 24 : X(1,2) = 15
20 X(2,1) = 10 : X(2,2) = 33
30 X(3,1) = 87 : X(3,2) = 90
40 X(4,1) = 34 : X(4,2) = 16
50 FOR R = 1 TO 4
60 PRINT
70 FOR C = 1 TO 2
80 PRINT "X("; R; ","; C; ") = "; X(R,C),
90 NEXT C, R
RUN

X( 1 , 1 ) = 24          X( 1 , 2 ) = 15
X( 2 , 1 ) = 10          X( 2 , 2 ) = 33
X( 3 , 1 ) = 87          X( 3 , 2 ) = 90
X( 4 , 1 ) = 34          X( 4 , 2 ) = 16
Ok
```

DIM statement

The DIM statement specifies the array name, the number of dimensions and the subscript upper bound per dimension. The DIM statement may specify one or more arrays.

Syntax:

DIM *array(list-of-subscripts)*[, *array(list-of-subscripts)*] ...

where

array is a valid array name. Any legal variable name may be used.

list-of-subscripts is one or more numeric expressions which specify the array dimensions. Each subscript must be separated from the next by a comma. The number of subscripts specifies the number of dimensions, and the value of each specifies the subscript upper bound.

The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

If an array name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10.

If a subscript is used that is greater than the maximum specified, a Subscript out of range error occurs.

The DIM statement sets all the elements of the specified numerical arrays to an initial value of zero and elements of string arrays to null strings.

If you wish to have an array with a subscript greater than 10, you need to use the DIM statement.

Example:

```
10 REM ***DIM Statement***
20 DIM A(20)
30 FOR X = 1 TO 15
40 A(X) = X
50 PRINT A(X);
60 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Ok
```

In the above program, the A(X) array is dimensioned to be able to have a subscript of 0 to 20. You can have up to 20 as your subscript but no greater. If you try to use 21 as a subscript, you will get the error message Subscript out of range.

Theoretically, the maximum number of dimensions allowed in a DIM statement is 255 and the maximum number of elements per dimension is 32767. In reality, however, these numbers would be impossible, since the name and punctuation are also counted as spaces in the line, and the line itself has a limit of 255 characters.

If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, a Duplicate Definition in nnnnn error results. Therefore, it is good programming practice to put the required DIM statements at the beginning of a program, outside of any processing loops.

If no DIM is specified, the first reference to an array element in the program will create the array with the specified number of dimensions. For example, if a program statement refers to:

```
AR1(3,5,10)
```

then AR1 is created with three dimensions and a default upper bound of 10 for each dimension.

An array can be dimensioned only once during the run of a program. The system will give an error message and stop executing the program if it comes to a DIM statement for the same array a second time. However, you can use more than one DIM statement in a program if you are dimensioning different arrays.

Example:

Not allowed

```
10 DIM X(20)
20 FOR I = 1 TO 15:X(I) = I:PRINT X(I);:NEXT
30 DIM X(30)
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Duplicate Definition in 30
Ok
```

Allowed

```
10 DIM X(20)
20 FOR I = 1 TO 15:X(I) = I:PRINT X(I);:NEXT
30 PRINT
40 DIM Y(30)
50 FOR I = 1 TO 10:Y(I) = I:PRINT Y(I);:NEXT
RUN
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 1 2 3 4 5 6 7 8 9 10
Ok
```

If you try to redimension an array without first erasing it, a Duplicate Definition in nnnnn error occurs, as shown above. You must first use the ERASE statement to erase an array before redimensioning it. You will get the same error message if a DIM statement is preceded by an array reference .

If the DIM statement is jumped using the GOTO or GOSUB statement, the subscript upper bound per dimension is not set.

Example:

```
10 I = 1
20 GOTO 40
30 DIM A(50)
40 A(10) = 3
50 A(11) = 45
RUN
Subscript out of range in 50
Ok
```

The system displays:

```
Subscript out of range in 50
```

when statement 50 is executed, as statement 30 is jumped over and an upper bound of 10 is assumed by default.

More Information on the DIM Statement

(Note: The two example programs below are not working examples. They only show the format of how to use the statement.)

Variables and expressions can also be used to dimension arrays.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B)
30 DIM Y$(A)
40 DIM Z(C + 11)
```

All the DIM statements can be on one line separated by commas.

Example:

```
10 A = 25:B = 100:C = 10
20 DIM X(B), DIM Y$(A), DIM Z(C + 11)
```

Number of Elements Per Dimension

no DIM statement

OPTION BASE 0 set	11 elements (subscripts 0-10 are allowed in each dimension)
OPTION BASE 1 set	10 elements (subscripts 1-10 are allowed in each dimension)

with DIM statement

OPTION BASE 0 set	the number of elements in each dimension is calculated by adding 1 to each upper bound subscript
OPTION BASE 1 set	the number of elements in each dimension coincides with each upper bound subscript

To Define an Array

Set the subscript lower bound. Use the default OPTION BASE 0 or set OPTION BASE 1.

Use the DIM statement to

- Assign a name to the array
- Set the number of dimensions
- Set the subscript upper bounds per dimension

ERASE statement

The ERASE statement releases space and variable names previously reserved for arrays. The data is lost and the array(s) no longer exist.

Syntax:

ERASE *array* [, *array*] ...

where

array is the name of an array to be erased.

Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes.

If an attempt is made to redimension an array without first ERASEing it, a Duplicate Definition in nnnnn error occurs.

More than one array variable can be used in the ERASE statement separated by commas (e.g., ERASE X,Y).

It is not good programming practice to reuse an identifier. This may generate errors or reduce the program readability. You may, however, find it useful to redeclare an erased array; for example, when an array name is known by a subroutine and you want to pass arrays with different number of dimensions or subscript upper bounds to this subroutine.

Example:

```
10 DIM X(17)
    .
    .
    .
130 ERASE X
140 DIM X(12,50)
```

Upon execution of statement 130, array X is deleted and the corresponding memory space is made free. You may define another array (see statement 140) with the same name but different number of dimensions and upper bounds.

OPTION BASE statement

The OPTION BASE statement defines the minimum value for array subscripts.

Syntax:

OPTION BASE *n*

where

n is an integer expression and may be 1 or 0.

The default base is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

Example:

```
10 OPTION BASE 1
20 FOR X = 0 TO 5
30          A(X) = X
40          PRINT A(X)
50 NEXT
RUN
Subscript out of range in 30
Ok
```

In this example, the minimum value of the array subscript was set to 1. But in line 30, the first array subscript is 0. So, the Subscript out of range error message is displayed.

A CHAINed program may have an OPTION BASE statement if no arrays are passed. The CHAINed program will inherit the OPTION BASE value of the CHAINing program.

3. 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 allows you to call an assembly language subroutine to return a value in the same way you call GW-BASIC intrinsic functions. However, it is recommended that you use the CALL or CALLS statement for interfacing machine language programs with GW-BASIC. 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 it can be loaded. To do so, use the /M: option of the GWBASIC command (*see "GWBASIC command" in Chapter 19*). The /M: option 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.

Loading subroutines into memory

An assembly language subroutine can be loaded into memory in several ways, the most simple being to use the BLOAD command (*see the "BLOAD command" in this chapter*). Also, you could SHELL a program that exits, but stays resident, leaving the linked, relocated image in memory (*see Chapter 16*). As a third choice, you could execute a program that exits but stays resident, and then run GW-BASIC.

The following guidelines must be observed if you choose to BLOAD, or read and poke, an .EXE file into memory:

1. Make sure the subroutines do not contain any long references, address offsets that exceed 64K or that take the user out of the code segment. These long references require handling by the .EXE loader.

2. Skip over the first 512 bytes (the header) of the linker's output file .(EXE), then read in the rest of the file.

The following two sections illustrate two standard ways of loading assembly language subroutines.

Using the POKE Statement

Short, assembly language subroutines can be POKEd (*see the "POKE statement" in this chapter*) into memory as follows: after assembling the subroutine machine code, you should create DATA statements containing the value in hexadecimal of each byte of code (represented as &Hxx). The subroutine is then POKEd into the specified area of memory, byte-by-byte, in a loop.

The subroutine may then be called using the USR function or the CALL statement. If you use the USR function, then the subroutine entry address must be defined with a DEF USR statement. This defines the USR function call offset into the current segment. If you use the CALL statement, then the subroutine entry address is the value of the numeric variable entered just after CALL. This variable must contain the offset into the current segment. In both cases, the segment is defined by the DEF SEG statement.

Using the BLOAD Command

An assembly language subroutine can be BLOADed into memory as follows:

1. Firstly, create an .EXE file of the subroutine using the linker, then load GW-BASIC under DEBUG by entering:

   ```
   DEBUG GWBASIC.EXE
   ```

2. To determine the location of GW-BASIC in memory, display and record the values contained in registers CS, IP, SS, SP, DS, and ES, using the R command (for use in step 5).

3. Load the .EXE file into high memory using DEBUG, overlaying the transient section of COMMAND.COM.

4. To determine the subroutine memory location, display the registers using the R command. The CS and IP register values should be recorded for use in steps 6 and 7.

5. Reset the register values to their original values as recorded in step 2 using the R command. Breakpoints may optionally be included in the subroutine using the G command which branches to the GW-BASIC entry point.

6. Load your application program. Modify the DEF SEG and either the DEF USR statement or the CALL variable to correspond with the subroutine memory location as defined in step 4 (i.e., the CS register value for DEF SEG and the IP register value for the DEF USR or CALL variable).

7. The subroutine memory area should be BSAVEd in GW-BASIC direct mode, using both the CS and IP register values from step 4, and the assembler listing or LINK map code length.

8. Ensure that your application program contains a DEF SEG with the appropriate CS register value, followed by a BLOAD statement.

   BLOAD can place a subroutine in an alternate location if the subroutine is self-relocatable. Possible alternatives include an unused screen or file buffer, or a string variable area. (*Refer to the "BLOAD command" and "VARPTR function" in this chapter.*) In this case, remember also to modify the associated DEF SEG statement.

9. Finally, the updated application program should be saved.

Note: If GW-BASIC is run under DEBUG, DEBUG is loaded first, as a precaution against being overwritten. Any breakpoints, or the SYSTEM command, returns control to DEBUG.

Calling the subroutine from GW-BASIC

CALL Statement

The CALL statement is the recommended way of calling machine language programs with GW-BASIC. It is preferable to the USR function unless you are running programs that already contain USR functions.

The syntax of the CALL statement is:

CALL *numvar* [(*variable* [, *variable*]...)]

where

numvar contains the offset into the current segment that is the starting point in memory of the subroutine being called

*variable*s indicates a list of variables, separated by commas, that are to be passed to the subroutine as arguments.

The current segment is either the default, or that which has been defined by a DEF SEG statement.

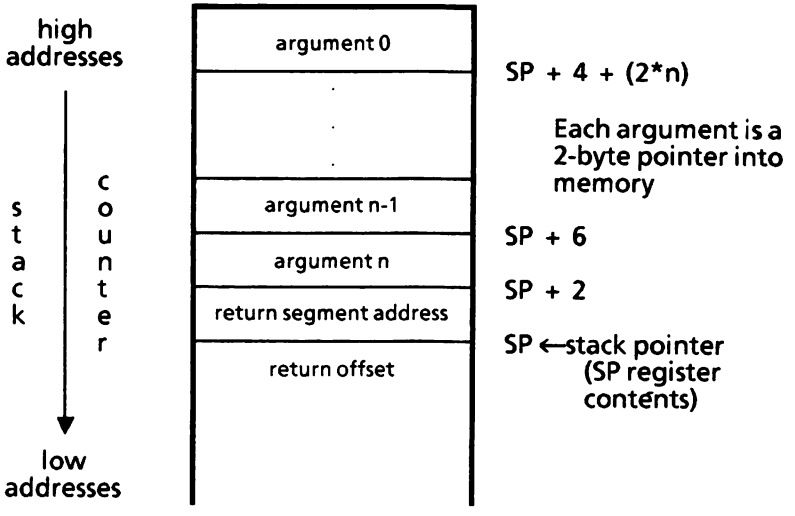
Invoking the CALL statement causes the following to occur:

1. For each variable specified in the statement, the two-byte offset of the variable's location within the GW-BASIC segment is pushed onto the stack.

2. The GW-BASIC return address code segment (CS), and offset (IP) are pushed onto the stack.

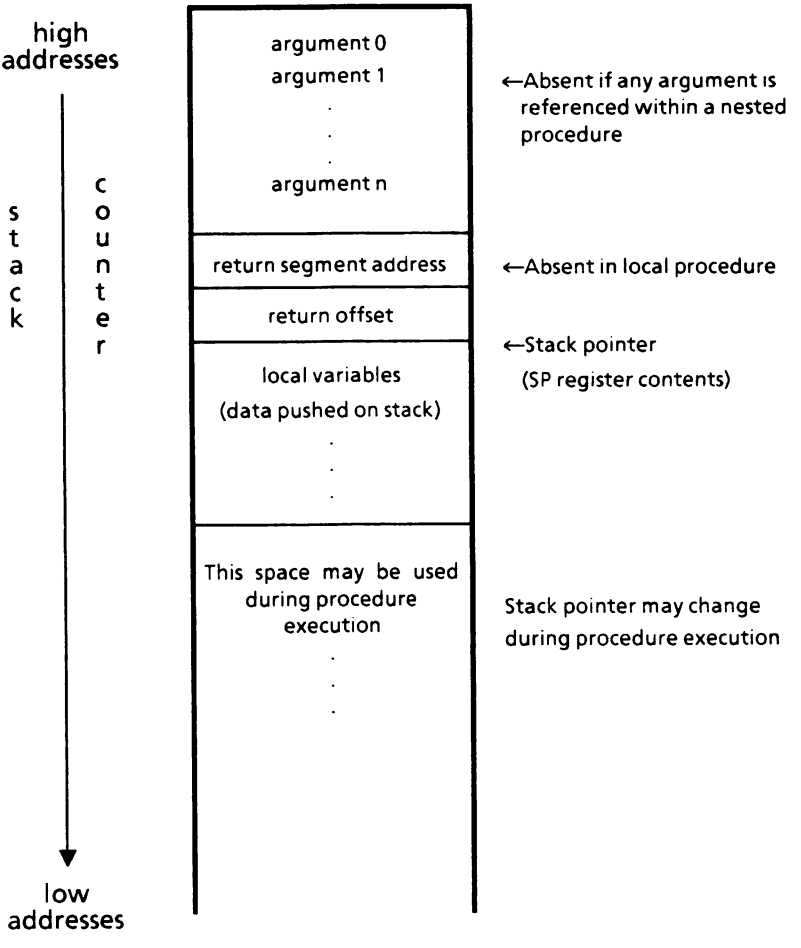
3. Control is transferred to the machine language routine using the segment address, which is given in the last DEF SEG statement and the offset given in *numvar*.

The following two diagrams illustrate the state of the stack at the time the CALL statement is executed, and the condition of the stack during execution of the called subroutine, respectively.



The above diagram illustrates the stack layout when the CALL statement is activated.

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



The above diagram illustrates the 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 enabled before exiting. The stack must be cleaned up on exit.

2. The called program must know the number and length of the arguments passed. The following routine shows an easy way to reference 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 or on the stack. Be careful not to modify the return segments and offset stored on the stack.

4. The called subroutine must clean up the stack. A preferred way to do this is to perform a RET *n* statement (where *n* is two times the number of arguments in the argument list) 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 3 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). Bytes 1 and 2, respectively, are the lower and upper 8 bytes of the string starting address in string space.

   Note: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add + "" to the string literal in the program. For example, use:

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

   This will force the string literal to be copied into 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.

See example on next page.

Example:

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

Line 100 sets the segment to 80000 Hex. The value of variable VAR is added into the address as the low word after the DEF SEG value is left shifted 4 bits, i.e., multiplied by 16. (This is a function of the microprocessor, not of GW-BASIC.) Here, VAR is set to &H7FA, so that the call to VAR will execute the subroutine at location 80000;7FA Hex (absolute address 8007FA Hex).

The following sequence in 8086 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              ;Restore pointer
RET     6               ;Restore stack, return
```

Note: The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction:

```
MOVS WORD
```

will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy 8 bytes if they were double precision.

CALLS Statement

The CALLS statement should be used to access subroutines that were written using MS-FORTRAN calling conventions. CALLS works just like CALL and has the same syntax, except that 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 pushed and then the offset is also 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 statement is four times the number of arguments.

USR Function

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

The syntax of the USR function is:

USR [*n*] (*argument*)

where

n is an integer from 0 to 9. It specifies which USR routine is being called. If *n* is omitted, USR0 is assumed.

argument is any numeric or string expression.

A DEF SEG statement must be executed prior to a USR function call to ensure 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 that was 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	Single precision floating-point number
8	Double precision floating-point number

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

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.
FAC-3 contains the lower 8 bits of the argument.

If the argument is a single precision floating-point number:

FAC-2 contains the middle 8 bits of mantissa.
FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating-point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register points to 3 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, respectively, are the lower and upper 8 bits of the string starting address in the GW-BASIC data segment. If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy the program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

GW-BASIC has extended the USR function interface to allow calls to MAKINT and FRCINT. This allows access to these routines without giving their absolute addresses. The address ES:BP is used as an indirect far pointer to the routines FRCINT and MAKINT.

To call FRCINT from a USR routine use CALL DWORD ES:[BP]. To call MAKINT from a USR routine use CALL DWORD ES:[BP + 4].

Example:

```
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.

BLOAD command

Loads a memory image file into memory.

Syntax:

BLOAD *"filespec"* [, *offset*]

where

"filespec" is a string expression which specifies the file to be loaded.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

offset is an integer expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement at which loading is to start.

The BLOAD and BSAVE statements allow you to load into memory, and save on a file, machine language routines. When these routines are resident in memory, they can be CALLed from your GW-BASIC program by a CALL statement.

The BLOAD and BSAVE statements also allow you to load and save any portion of memory, for instance, you can save and display screen images (specifying the screen buffer as the current segment by a DEF SEG statement).

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

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

Warning: BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. You must be careful not to load over GW-BASIC, or the operating system.

Examples:

```
10 'Load a machine language program into memory at 60:F000
20 DEF SEG  'Restore segment to GW-BASIC's DS
30 BLOAD "B:PROG1",&HF000 'Load PROG1 into the DS
```

```
10 'Load the screen buffer
20 DEF SEG = &HB800 'Point segment at screen buffer
30 BLOAD "FILE1",0 'Load FILE1 into screen buffer
```

Note the DEF SEG statement in 20 and the offset of 0 in 30; this guarantees that the correct address is used.

An example under BSAVE illustrates how FILE1 was saved.

BSAVE command

Saves sections of the main memory on the specified file.

Syntax:

BSAVE *"filespec"*, *offset*, *length*

where

"filespec" is a string expression which specifies the name of the file to be saved.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

offset is an integer expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG.

length is an integer expression in the range 1 to 65535, specifying the length of the memory image to be saved.

A memory image file is a byte-for-byte copy of what is in memory.

The BLOAD and BSAVE statements allow you to load into memory, and save on a file, machine language routines. When these routines are resident in memory, they can be CALLed from your GW-BASIC program by a CALL statement.

The BLOAD and BSAVE statements also allow you to load and save any portion of memory, for instance, you can save and display screen images (specifying the screen buffer as the current segment by a DEF SEG statement).

A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

Examples:

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

This example saves 256 bytes starting at 60000:F000 in the file PROG1.

```
10 'Save the screen buffer
20 DEF SEG = &HB800 'Point segment at screen buffer
30 BSAVE "FILE1",0,16384 ' Save screen buffer in FILE1
```

The DEF SEG statement must be used to set up the segment address to the screen buffer. The offset of 0 and the length 16384 specify that the entire 16K screen buffer is to be saved.

DEF SEG statement

Assigns the current "segment" of memory.

Syntax:

DEF SEG [= *address*]

where

address is a numeric expression returning an unsigned integer in the range 0 to 65535. The address specified identifies the segment address used by BLOAD, BSAVE, PEEK, POKE, DEF USR, and CALL.

If ***address*** is omitted, the segment is set to GW-BASIC's data segment. This is the initial default value.

If ***address*** is specified, the value is shifted left 4 bits (i.e., if address is in hexadecimal, a zero is appended) to form the current segment address.

Note: GW-BASIC does not check if the resultant segment is valid.

If you enter a value outside the specified range, an Illegal function call error results. Previous value will be retained.

If you do not separate DEF and SEG by at least one blank, GW-BASIC would interpret DEF SEG as the name of a variable. For instance:

```
100 DEFSEG = 150
```

would assign the value 150 to variable DEFSEG.

Example:

```
10 DEF SEG = &HB800 'set segment to screen buffer
100 DEF SEG 'Restore segment to GW-BASIC's DS
```

Note that in statement 10 the screen buffer is at absolute address B8000 hex, as the last hexadecimal digit is dropped on the DEF SEG statement.

DEF USR statement

Enables access to a machine language subroutine by specifying the starting address. The subroutine may be subsequently called by the associated USR function.

Syntax:

DEF USR [*n*] = *offset*

where

n may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If *n* is omitted, DEF USR0 is assumed.

offset is an integer expression from 0 to 65535. It specifies the starting address of the subroutine as an offset into the current segment which is defined by the last DEF SEG statement executed.

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. To obtain the starting address of a subroutine, GW-BASIC adds the value of *offset* to the current segment value.

Example:

```
100 DEF SEG = 0
      .
      .
      .
200 DEF USR0 = 24000
210 X = USR0(Y*2/2.89)
      .
      .
      .
      .
```

PEEK function

Returns the byte read from the specified memory location.

Syntax:

PEEK(*offset*)

where

offset is a numeric expression returning an integer in the range -32768 to 65535. It indicates the address of the memory location from which a byte will be returned. It is the *offset* from the current segment, which was defined by the last DEF SEG statement. For the interpretation of a negative value of *offset*, see the VARPTR function described in this chapter.

The returned value is an integer in the range 0 to 255.

If *offset* is outside the specified range, an Illegal function call error is returned.

PEEK is the complementary function of the POKE statement.

Example:

```
100 A = PEEK(&H5A00)
```

POKE statement

Writes a byte into a memory location.

Syntax:

POKE *offset, byte*

where

offset is a numeric expression returning an integer in the range 0 to 65535. It indicates the address of the memory location where the data is to be written. It is the *offset* from the current segment, which was defined by the last DEF SEG statement.

byte is the data byte. It must be in the range 0 to 255.

You can use POKE and PEEK for passing arguments and data to assembly language subroutines.

If either *offset* or *byte* is outside the specified range, an Illegal function call error is returned.

The complementary function to POKE is PEEK.

Warning: Use POKE carefully. If it is used incorrectly, it can cause GW-BASIC or MS-DOS to crash.

Example:

```
10 POKE &H5A00,&HFF
```

VARPTR function

Returns the address of the first byte of data identified with ***variable***.

Syntax:

VARPTR(***variable***)

where

variable is any numeric or string program variable.

The address returned will be an integer in the range -32768 to 32767. This integer value is the offset into GW-BASIC's Data Segment. If a negative address is returned, add it to 65536 to obtain the actual address.

The ***variable*** must have been defined prior to execution of VARPTR. Otherwise an Illegal function call error results. Variables are defined by executing any reference to the variable. Both numeric and string variables may be used. For string variables, the address of the first byte of the string description is returned. (*See Appendix F on how GW-BASIC allocates variables.*)

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. All simple variables should be assigned before calling VARPTR for an array, because the address of the arrays change whenever a new simple variable is assigned.

Example:

```
10 X = USR(VARPTR(Y))
```

4. ASYNCHRONOUS COMMUNICATIONS

This chapter describes how GW-BASIC may be used to support RS232 asynchronous communications with other computers and peripherals.

This chapter is intended for experienced programmers interested in setting up and using asynchronous communications.

The GW-BASIC statements used for communications are also described in this chapter.

The subsections are

Opening communications file
Communication I/O
An exercise in communication I/O
EOF function
GET (COM files) statement
INPUT$ function
LOC function
LOF function
ON COM(*n*) GOSUB and COM(*n*) statements
OPEN COM statement
PUT (COM files) statement

Opening communications files

The OPEN COMmunications statement allocates a buffer for input and output in a similar manner as the OPEN statement for disk files. *Refer to the OPEN COM statement in this chapter for a full description.*

Communication I/O

Since the communication port is opened as a file, all Input/Output statements that are valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files. They are:

```
INPUT #
LINE INPUT #
INPUT$
```

COM sequential output statements are the same as those for disk, and are:

```
PRINT #
PRINT # USING
WRITE #
```

For details of coding syntax and usage of these statements, refer to the alphabetized listing of the commands, statements and functions in Chapter 1 .

The GET and PUT statements are only slightly different for COM files. (*See the GET (COM files) and PUT (COM files) statements described in this section.*)

Communication I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates above 2400 bps, it may be necessary to suspend character transmission from the host computer long enough to catch up. This can be done by sending XOFF (CHR$(19)) to the host and XON (CHR$(17)) when ready to resume.

GW-BASIC provides three functions which help in determining when an overrun condition is imminent. These are:

LOC(*f*) Returns the number of characters in the input buffer waiting to be read. The input buffer can hold more than 255 characters determined by the /C: switch option in the GWBASIC command line (*see the "GWBASIC command" in Chapter 19*). If there are more than 255 characters in the buffer, LOC(*f*) returns 255. Since a string is limited to 255 characters, this practical limit means that you do not have to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, LOC(*f*) returns the actual count.

LOF(*f*) Returns the amount of free space in the input buffer. That is, *size*-LOC(*f*), where *size* is the size of the communications buffer as set by the /C: option. LOF may be used to detect when the input buffer is reaching its maximum capacity.

EOF(*f*) If true (-1), indicates that the input buffer is empty. Returns false (0) if any characters are waiting to be read.

Possible Errors

Communication Buffer Overflow
If a read is attempted after the input buffer is full (i.e., LOF(*f*) returns 0).

Device I/O Error
If any of the following line conditions are detected on reception: Overrun Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.

Device Fault
If Data Set Ready (DSR) is lost during I/O.

The INPUT$ Function for COM Files

The INPUT$ function is preferable to the INPUT# and LINE INPUT# statements when reading COM files, since all ASCII characters may be significant in communications. INPUT# is least desirable because input stops when a comma (,) or CR (carriage return) is received and LINE INPUT# terminates when a CR is received.

INPUT allows all characters read to be assigned to a string. Remember from the coding rules that INPUT$ (*n,f*) will return *n* characters from the *#f* file. The following statements are therefore the most efficient for reading a COM file:

```
10 WHILE NOT EOF(1)
20        A$ = INPUT (LOC(1),#1)

          Process data returned in A$

60 WEND
```

The above statements return the characters in the buffer into A$ and process them, provided there are characters in the buffer. If there are more than 255 characters, only 255 will be returned at a time to prevent String Overflow. If this is the case, EOF(1) is false and input continues until the input buffer is empty. The sequence of events is therefore simple, concise, and fast.

An exercise in communication I/O

The following program enables your personal computer to be used as a conventional terminal. Besides Full Duplex communication with a host, the TTY program allows data to be "Down-loaded" to a file. Conversely, a file may be "Up-loaded" (transmitted) to another machine.

In addition to demonstrating the elements of Asynchronous communication, this program should be useful in transferring GW-BASIC programs and data to and from your system.

```
10 REM
20 REM *** RS232 TEST PROGRAM ***
30 REM
40 SCREEN 0,0
50 KEY OFF:CLS:CLOSE
60 DEFINT A-Z
70 FALSE = 0:TRUE = NOT FALSE
80 MENU = 5
90 XOFF = CHR$(19):XON$ = CHR$(17)
100 ON COM(1) GOSUB 730
110 COMFIL$ = "COM1:1200,E,7"
120 OPEN COMFIL$ AS1
130 REM
140 REM *** TALK MODE ***
150 REM
160 CLS
170 LOCATE 25,1:PRINT "RS232 test program
running in TALK MODE";
180 PAUSE = FALSE
190 LOCATE 1,1
200 A$ = INKEY$:IF A$ = "" THEN 220
210 IF ASC(A$) = MENU THEN 290 ELSE
PRINT#1,A$;
220 IF EOF(1) THEN 200
230 IF LOC(1)>50 THEN PAUSE = TRUE:
PRINT#1,XOFF$;
240 A$ = INPUT$(LOC(1),1)
250 PRINT A$;:IF LOC(1)>0 THEN 230
260 IF PAUSE THEN PAUSE = FALSE:
PRINT#1,XON$;
270 GOTO 200
```

```
280 REM
290 REM *** COMMAND MODE ***
300 REM
310 CLS
320 LOCATE 25,1:PRINT "RS232 test program
running in COMMAND MODE";
330 LOCATE 1,1
340 INPUT "FILE ";DSKFIL$
350 LOCATE 1,1:PRINT STRING (80," "):LOCATE 1,1
360 INPUT "(T)ransmit or (R)eceive ";TXRX$
370 IF RXRX$ = "T" THEN OPEN DSKFIL$ FOR INPUT
AS 3:GOTO 580
380 IF TXRX$ = "R" THEN 410
390 GOTO 350
400 REM
410 REM *** FILE RECEIVE MODE ***
420 REM
430 LOCATE 25,32:PRINT "FILE RECEIVE MODE ";
440 OPEN DSKFIL$ FOR OUTPUT AS 3
450 IF EOF(1) THEN GOSUB 520
460 IF LOC(1) > 50 THEN PAUSE = TRUE:
PRINT#1,XOFF$;
470 A$ = INPUT$(LOC(1),1)
480 PRINT#3,A$;
490 IF LOC(1) > 0 THEN 460
500 IF PAUSE THEN PAUSE = FALSE:
PRINT#1,XON$;
510 GOTO 450
520 FOR I = 1 TO 5000
530 IF NOT EOF(1) THEN RETURN
540 NEXT I
550 CLOSE#3
560 RETURN 140
570 REM
580 REM *** FILE TRANSMIT MODE ***
590 REM
600 LOCATE 25,32 : PRINT "FILE TRANSMIT MODE
";
610 COM(1) ON
620 XFLAG = 1
630 WHILE NOT EOF(3)
640 A$ = INPUT$(1,3)
650 WHILE XFLAG = 0:WEND
660 PRINT#1,(A$);
670 WEND
680 COM(1) OFF
690 PRINT#1,CHR$(26);
700 CLOSE 3
710 GOTO 140
```

```
720 REM
730 REM ***XON/XOFF RECEIVING ROUTINE***
740 REM
750 IF EOF(1) THEN RETURN
760 B$ = INPUT$(LOC(1),1)
770 IF LEN(B$) = 2 THEN 790
780 IF B$ = XOFF$ THEN 810
790 XFLAG = 1
800 RETURN
810 XFLAG = 0
820 RETURN
```

Notes on the TTY programming example

Line No.	Comments
10-90	Define the screen attributes and initialize program variables.
100	Specifies the line number of the first statement of the COM trap routine associated with channel number 1.
110-120	Open and initialize communications channel 1 with a speed of 1200 bps, parity even, and 7 data bits.
170	Displays message indicating the operation mode (Talk Mode) on the 25th screen line.
180-260	Send characters entered from keyboard to channel number 1, and display characters received from the channel. Statement 210 transfers control to statement 290 (where Command Mode is entered), if the user enters CTRL + E.
320	Displays a message indicating the new mode (Command Mode) on the 25th screen line.

Line No.	Comments
340	Asks the user to enter the name of the file to transmit or receive, depending on the character (T or R) entered upon execution of statement 360.
370	If the user enters T, opens the specified file for INPUT and branches to statement 580.
380	If the user enters R, branches to statement 410 (where Receive Mode is entered).
410-560	The program is in Receive Mode, as displayed by statement 430 on the 25th screen line. Statement 440 opens the specified file for OUTPUT. Statement 450 checks if characters are pending on the receive buffer. If no characters are pending, control is transferred to statement 520, otherwise to the following statement (line 460).
520-540	A FOR...NEXT loop is activated to wait until characters arrive in the receive buffer. If no character arrives within the specified number of iterations, the Receive Mode is exited. Control is then transferred to statement 550 where the file is closed, and then to statement 140 returning in "TALK MODE". If characters arrive, control is transferred to statement 460.
460	Checks if the number of characters in the receive buffer is greater than 50. If the number is greater than 50, it sends an XOFF character to the channel to stop transmission.
470-480	If the number of characters in the receive buffer is less than or equal to 50, characters are read from the receive buffer and written to the file.

Line No.	Comments
490-510	Check if there are still characters in the receive buffer. If yes, control is transferred to statement 460. If no, an XON character is sent (if an XOFF was sent before) and control is transferred to statement 450.
570-710	The program is in Transmit Mode, as displayed by statement 600 on the 25th screen line ("File Transmit Mode"). The file has already been opened for input at statement 370. Statement 610 enables COM trapping. Statements 630 to 670 form a WHILE...WEND loop to read and transmit the file (statement 640 reads one character at a time and statement 660 sends it to the communications channel). The character transmission is suspended if an XOFF character is received (see statement 650). The character transmission is resumed if an XON character is received. Statements 680 to 710 disable COM trapping, send an EOF character, close the file and return to "Talk Mode".
730-820	Form the COM trap routine. Statement 750 checks if characters are pending in the receive buffer. If no character is pending, a RETURN is executed. If two characters are pending, the transmission of characters is enabled (statement 790) and the routine is exited (RETURN). Two characters in the receive buffer means that both an XON and an XOFF have been received. If only one character is pending, the transmission of characters is disabled (if this character is XOFF) or enabled (if this character is XON).

EOF function

Tests for the end-of-file condition.

Syntax:

EOF(*filenum*)

where

filenum is the file number specified in the OPEN statement.

When EOF is used with a communications device, the definition of the end-of-file condition is dependent on the mode (ASCII or binary) in which the device was OPENed.

In binary mode, EOF is true when the input queue is empty (LOC(n) = 0). It becomes false when the input queue is not empty.

In ASCII mode, EOF is false until a CTRL Z is received, and from then on it will remain true until the device is closed.

GET (COM files) statement

Reads a specified number of bytes into the communications buffer.

Syntax:

GET [#] *filenum* , *length*

where

filenum is an integer expression returning a valid file number.

length is an integer expression returning the number of bytes to be transferred into the communications buffer. *length* cannot be greater than the value specified by the LEN clause in the OPEN COM statement.

Example:

```
100 GET # 2, 80
```

INPUT$ function

Returns a string of characters read from a file.

Syntax:

INPUT$ (*length, [#]filenum)*

where

length is an integer expression specifying the number of characters to be read from a file.

filenum is the file number specifying the file to be read.

The INPUT$ function is preferred over INPUT# and LINE INPUT# statements, when reading COM files, since all ASCII characters may be significant in communications. INPUT# is least desirable because input stops when a comma (,) or carriage return is encountered and LINE INPUT# terminates when a carriage return is encountered.

Example:

```
10 WHILE NOT EOF(1)
20 A$ = INPUT$(LOC(1),#1)
     .
     .    Process data returned in A$
     .
60 WEND
```

The above sequence of statements is read: "While there is something in the input queue, return the number of characters in the queue and store them in A$. If there are more than 255 characters, only 255 will be returned at a time to prevent String Overflow. Further, if this is the case, EOF(1) is false and input continues until the input queue is empty."

LOC function

Returns the current position in the file.

Syntax:

LOC(*filenum*)

where

filenum is the number under which the file was OPENed.

For communications files, LOC is used to determine if there are any characters in the input queue waiting to be read. The input queue can hold more than 255 characters determined by the /C: switch option in the GWBASIC command line (*see the "GWBASIC command" in Chapter 19*).

If there are more than 255 characters in the queue, LOC returns 255. Since strings are limited to 255 characters, this practical limit removes the need to test for string size before reading data into them.

If fewer than 255 characters remain in the queue, the value returned by LOC depends on whether the device was opened in ASCII or binary mode. In either mode, LOC will return the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. Any attempt to read the end-of-file will result in an Input past end error.

Example:

```
100 IF LOC(2) > 100 THEN STOP
```

LOF function

Returns the length of the named file in bytes.

Syntax:

LOF(*filenum*)

where

filenum is the number under which the file was OPENed.

For communications files, LOF may be used to check if the input buffer is getting full as it returns the amount of free space in the input buffer. That is:

buffer-size - LOC(*filenum*)

where *buffer-size* is the size of the communications buffer. It defaults to 256 bytes, but may be changed with the /C: option in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

ON COM(n) GOSUB COM(n) statements

These statements are used for conditional branching. Specifies the first line number of a subroutine to be executed as soon as characters arrive in the communications buffer. This is also known as "event trapping".

Syntax:

ON COM(*n*) GOSUB *linenum*

COM(*n*) ON | OFF | STOP

where

n is an integer expression that specifies the number of the communications channel. It may be 1, 2, 3, or 4.

linenum is the line number of the subroutine that is to be performed when the characters arrive in the communications buffer. A line number of 0 disables the communications event trap.

The COM(*n*) statement enables or disables trapping of communications activity on the specified channel.

The ON COM(*n*) GOSUB statement specifies the first program line of a subroutine to be performed when characters arrive in the communications buffer.

To enable the ON COM(*n*) GOSUB statement, a COM(*n*) ON statement must first be executed. While trapping is enabled, and if a non-zero *linenum* is specified in the ON COM(*n*) GOSUB statement, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, the ON COM(*n*) GOSUB statement is executed and the corresponding subroutine activated.

The COM(*n*) OFF statement disables the trapping routine. If a COM(*n*) OFF statement is executed and an event takes place, the GOSUB is not performed and the event is not remembered.

The COM(*n*) STOP statement suspends the trap. If an event occurs, it is remembered. If a COM(*n*) STOP statement is executed and an event takes place, the GOSUB is not performed but will be performed as soon as a COM(*n*) ON statement is executed.

When a trap occurs (i.e., the GOSUB is performed), an automatic COM(*n*) STOP statement is executed so that recursive traps cannot take place. The RETURN from the trap subroutine will automatically perform a COM(*n*) ON statement unless an explicit COM(*n*) OFF statement was performed inside the subroutine.

The RETURN *linenum* 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, and event trapping is automatically disabled when an error trap occurs resulting from an ON ERROR statement.

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 since, at high baud rates, the overhead of trapping and reading for each individual character may cause the COM interrupt buffer to overflow.

Example:

```
100 ON COM(2) GOSUB 1000
110 COM(2) ON
      .
      .
      .
1000 REM COM activity
      .
      .
      .
1050 RETURN 200
```

OPEN COM statement

Opens and initializes a communications channel for input/output.

Syntax:

OPEN "COM*n* : [*speed*][, [*parity*][, [*data*][, [*stop*] [,RS][,CS [t]][,DS[t]][,BIN][,ASC][,LF]]]] " [FOR *mode*] AS [#] *file#* [LEN = *record-length*]

where

n is 1, 2, 3, or 4. It specifies the number of a legal communications device.

speed is an integer constant which sets the baud rate in bits per second of the device to be opened. Valid values are: 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800 or 9600. The default value is 300 bps.

parity designates the parity of the device to be opened. Valid entries are:

E (even) - default value
M (mark)
N (none)
O (odd)
S (space)

data designates the number of data bits per byte. Valid entries are: 5, 6, 7 (default), or 8.

stop designates the stop bit. Valid entries are: 1, 1.5, or 2. If omitted, then 75 and 110 bps transmit two stop bits, all others transmit one stop bit.

RS suppresses RTS (Request To Send).

CS[*t*] controls CTS (Clear To Send).

DS[*t*] controls DSR (Data Set Ready).

CD[*t*] controls CD (Carrier Detect).

BIN opens the device in binary mode. BIN is selected by default, unless ASC is specified.

ASC opens the file in ASCII mode.

LF specifies that a linefeed is to be sent after a carriage return.

mode is one of the following string expressions:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.

If the ***mode*** expression is omitted, it is assumed to be random input/output. Random cannot, however, be explicitly chosen as ***mode***.

filenum is the number of the file to be OPENed.

record-length is the length of the records written to or read from a communications buffer. This value cannot be greater than the value fixed by the /C: switch in the GWBASIC command line (*see "**GWBASIC command" in Chapter 19*). The default ***record-length*** for the receive buffer is 2 bytes. The length of the transmit buffer is 128 bytes.

The OPEN COM statement must be executed before a device can be used for RS232 communications.

A COM device may be OPENed to only one file number at a time.

Any syntax errors in the OPEN COM statement will result in a Bad file name error.

The *speed*, *parity*, *data*, and *stop* options must be listed in the order shown in the syntax. The remaining options may be listed in any order, but they must be listed after the *speed*, *parity*, *data*, and *stop* options.

The CS, DS, and CD options allow you to specify a time (*t*) to wait for the signal before returning a Device Timeout error. This time is expressed in milliseconds, ranging from 0 to 65535. Default values are: CD = 0, CS = 1000, DS = 1000.

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 (0DH). This includes the carriage return sent as a result of the width setting. Note that INPUT# and LINE INPUT#, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed.

The LF option is superseded by the BIN option.

In the BIN mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and CTRL Z is not treated as end-of-file. When the channel is closed, CTRL Z will not be sent over the RS232 line. The BIN option supersedes the LF option.

In ASC mode, tabs are expanded, carriage returns are forced at the end-of-line, CTRL Z is treated as end-of-file, and XON/XOFF protocol is enabled. When the channel is closed, CTRL Z will be sent over the RS232 line.

Example:

```
10 OPEN "COM1:9600,N,8,1,BIN" AS # 2
```

Opens communications channel 1 in random mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in binary mode. Other lines in the program may now access channel 1 as file number 2.

PUT (COM files) statement

Writes a specified number of bytes to a communications file.

Syntax:

PUT [#] *filenum*[, *length*]

where

filenum is an integer expression returning a valid file number.

length is an integer expression returning the number of bytes to be transferred out of the communications buffer. ***length*** cannot exceed the value specified by the LEN clause in the OPEN COM statement.

Example:

```
100 PUT #2,80
```

5. BRANCHING

This chapter describes the different ways to branch to other segments of a program.

There are two types of branching:

Unconditional

Conditional

The following statements are used for unconditional branching:

GOSUB ... RETURN

GOTO

These statements are used for conditional branching:

IF ... GOTO [... ELSE]

IF ... THEN [... ELSE]

ON ... GOSUB

ON ... GOTO

ON KEY(n) GOSUB

GOSUB ... RETURN statements

GOSUB unconditionally transfers control to a GW-BASIC subroutine by branching to the specified line number. RETURN transfers control to the statement following the most recent GOSUB (or ON...GOSUB) executed, or to a specified line number.

Syntax:

```
GOSUB linenum1
  .
  .
  .
RETURN [linenum2]
```

where

linenum1 is the first line number of the subroutine.

linenum2 is any line of your program different from *linenum1* and from the line number of the GOSUB statement.

A subroutine may be called any number of times in a program, and it may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause GW-BASIC to branch back to the statement following the most recent GOSUB or ON...GOSUB statement executed. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

The *linenum2* 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, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Never exit a subroutine with a GOTO statement.

If either *linenum1* or *linenum2* does not exist in the program, an Undefined line number error is returned.

Example:

```
10 A = 25:B = 30
20 GOSUB 100
30 PRINT C
40 END
100 REM multiplication subroutine
110 C = A*B
120 RETURN
RUN
 750
Ok
```

GOTO statement

The GOTO statement unconditionally branches out of the normal program sequence to a specified program line number.

Syntax:

GOTO *linenum*

where

linenum is the number of a line in the program.

If *linenum* is the line number of an executable statement, that statement and those following are executed.

If it is the line number of a nonexecutable statement, execution begins at the first executable statement encountered after *linenum*.

If the specified *linenum* does not exist in the program, an Undefined line number error is returned.

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
RUN
R = 5         AREA = 78.5
R = 7         AREA = 153.86
R = 12        AREA = 452.16
Out of DATA in 10
Ok
```

IF ... GOTO [... ELSE]
IF ... THEN [... ELSE] statements

These statements are used for conditional branching. They make a decision regarding program flow based on the result of a specified condition.

Syntax 1:

IF *condition* GOTO *linenum* [ELSE *statement(s)* or *linenum*]

Syntax 2:

IF *condition* THEN *statement(s)* or *linenum*

where

condition may be a numeric, relational, or logical expression. GW-BASIC determines whether the condition is true or false by testing the result of the expression for non-zero and zero, respectively. A non-zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non-zero or zero by merely specifying the name of the variable as *condition*.

statements are one or more statements. Each statement must be separated from the preceding one by a colon (:).

linenum is a line number of the program in memory.

If the result of *condition* is true (not zero), the GOTO or THEN clause is executed. GOTO is always followed by a line number. THEN may be followed by either a line number for branching or one or more statements to be executed.

If the result of *condition* is false (zero), the GOTO or THEN clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

A comma is allowed before the THEN.

If an IF...THEN statement is followed by a line number in the direct mode, an Undefined line error results unless a statement with the specified line number had previously been entered in the program mode.

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:

```
100 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.

See examples on next page.

Examples:

This statement GETs record number I from a random file, if I is not zero.

```
200 IF I THEN GET#1,I
```

In the example below, a test determines if I is less than 20 and greater than 10. If I is in this range, DB is calculated and the result displayed on the screen then execution branches to line 40. If I is not in this range, execution continues with line 30.

```
10 I = 15
20 IF (I<20) * (I>10) THEN DB =
    1979-1:PRINT DB:GOTO 40
30 PRINT "OUT OF RANGE"
40 END
RUN
 1978
Ok
```

Note the asterisk (*) in line 20. In this case, the asterisk does not mean to multiply, it means "and". The word "and" could have been used instead of the asterisk.

In the following example, statement 30 causes output to go either to the screen or the printer depending on the value of the variable (IOFLAG). In this case, IOFLAG is equal to one so the output is displayed on the screen. If statement 20 was changed to 20 IOFLAG=0, then output would go to the printer.

```
10 A$ = "HELLO"
20 IOFLAG = 1
30 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
RUN
HELLO
Ok
```

Nesting of IF statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line.

Examples:

```
300 IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS" ELSE PRINT "EQUAL"
```

The above example 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.

```
100 IF A = B THEN IF B = C THEN PRINT "A = C" ELSE
PRINT "A<>C"
```

will not print "A<>C" when A<>B.

ON ... GOSUB
ON...GOTO statements

These statements are used for conditional branching. Branches to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Syntax:

ON *numexp* GOSUB *linenum* [, *linenum*] ...

ON *numexp* GOTO *linenum* [, *linenum*] ...

where

numexp is a numeric expression (from 0 to 255) which determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. If *numexp* is not an integer, it will be rounded up to an integer.

linenum is the line number to which the branch will be made.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of *numexp* is either 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 *numexp* is negative or greater than 255, an Illegal function call error occurs.

Example:

```
100 L = 5
110 ON L-1 GOTO 200, 300, 400, 500
120 END
200 PRINT "L = 1"
210 END
300 PRINT "L = 2"
310 END
400 PRINT "L = 3"
410 END
500 PRINT "L = 4"
510 END
RUN
L = 4
Ok
```

In the example, L is equal to 5. After the expression L-1 is calculated, L is equal to four. So in this case, program flow would go to line number 500 since it is the fourth number in the ON...GOTO statement.

ON KEY(n) GOSUB
KEY(n) statements

These statements are used for conditional branching. Specifies the first line number of a subroutine to be executed when a specified key is pressed. This is also known as "event trapping".

Syntax:

ON KEY(*n*) GOSUB *linenum*

KEY(*n*) ON | OFF | STOP

where

n is an integer in the range 1 to 20 and indicates the key to be trapped.

1-10	Function keys F1 to F10
11	Cursor up key
12	Cursor left key
13	Cursor right key
14	Cursor down key
15-20	Keys defined in the form KEY *n*, CHR$(*shift*) + CHR$(*scan code*) (*See "KEY statement" in Chapter 19*)

linenum is the line number of the subroutine that is to be performed when the specified function or cursor direction key is pressed. A line number of 0 disables the event trap.

(*NOTE: Do not confuse KEY ON and KEY OFF, which displays/erases the value of the functions keys at the bottom of the screen, with the event trapping statements described in this section.*)

The KEY(*n*) statement enables or disables trapping of a specified key during the execution of a program.

The ON KEY(*n*) GOSUB statement specifies the first program line of a subroutine to be performed when the specified key is pressed.

To enable the ON KEY(*n*) GOSUB statement, a KEY(*n*) ON statement must first be executed. While trapping is enabled, and if a non-zero *linenum* is specified in the ON KEY(*n*) GOSUB statement, GW-BASIC checks between every statement to see if the specified key has been pressed. If it has, the ON KEY(*n*) GOSUB statement is executed and the corresponding subroutine activated. The text that would normally be associated with the specified key is not displayed.

The KEY(*n*) OFF statement disables the trapping routine. If a KEY(*n*) OFF statement is executed for the specified key, the GOSUB is not performed and the event is not remembered.

The KEY(*n*) STOP statement disables the trap; but if the specified key is pressed, it is remembered. If a KEY(*n*) STOP statement is executed for a specified key, the GOSUB is not performed but will be performed as soon as a KEY(*n*) ON statement is executed.

When a trap occurs (i.e., the GOSUB is performed), an automatic KEY(*n*) STOP statement is executed so that recursive traps cannot take place. The RETURN from the trap subroutine will automatically perform a KEY(*n*) ON statement unless an explicit KEY(*n*) OFF statement was performed inside the subroutine.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use INPUT$ or INKEY$ 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.

The RETURN *linenum* 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 anyother 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, and event trapping is automatically disabled when an error trap occurs resulting from an ON ERROR statement.

Example:

```
10 KEY(1) ON
20 KEY(2) ON
30 KEY(3) ON
40 ON KEY(1) GOSUB 1000
50 ON KEY(2) GOSUB 2000
60 ON KEY(3) GOSUB 3000
70 CLS
80 LOCATE 10,30
90 PRINT "F1 - TIME"
100 LOCATE 12,30
110 PRINT "F2 - DATE"
120 LOCATE 14,30
130 PRINT "F3 - EXIT"
140 LOCATE 18,10
150 PRINT "PRESS FUNCTION KEY DESIRED"
160 FOR X = 1 TO 40000:NEXT:CLS:END
1000 REM ***F1 PRESSED***
1010 CLS
1020 LOCATE 12,20
1030 PRINT "THE TIME IS ";TIME$
1040 FOR Y = 1 TO 1000:NEXT Y
1050 RETURN 70
2000 REM ***F2 PRESSED***
2010 CLS
2020 LOCATE 12,20
2030 PRINT "THE DATE IS ";DATE$
2040 FOR Y = 1 TO 1000:NEXT Y
2050 RETURN 70
3000 REM ***F3 PRESSED***
3010 CLS
3020 END
```

(See next page for execution of this program.)

When the program on the previous page is executed, the screen is cleared and the following is displayed.

F1 - TIME

F2 - DATE

F3 - EXIT

PRESS FUNCTION KEY DESIRED

If F1 is pressed, the screen will clear, display the time for a few seconds and then return to the above screen.

If F2 is pressed, the screen will clear, display the date for a few seconds and then return to the above screen.

If F3 is pressed, the screen will clear, exit the program, and system returns to command level.

The above screen will only be displayed for about a minute then the program is exited. If you want to stay in the program longer, you can change the loop in program line 160 to more than 40000. Also, if you want the time and/or date to be displayed longer, you can change the loop in program lines 1040 and 2040,respectively, to a larger value.

See more examples on the next page.

```
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
      .
      .
250 RETURN
```

In the above example, the programmer has overridden the normal function associated with function key 4, and replaced it with "SCREEN 0,0", which will be displayed whenever that key is pressed. The value may be reassigned and it will resume its standard function when the system is rebooted.

```
100 KEY 15, CHR$(&H04) + CHR$(83)
105 REM **Key 15 now is CTRL DEL **
110 KEY(15) ON
      .
      .
1000 PRINT "If you want to stop processing for a
break"
1010 print "press the CTRL key and DEL at the
same time".
1030 ON KEY (15) GOSUB 3000.
      .
      The user presses CTRL DEL
      .
3000 REM ** Suspend processing loop.
3010 CLOSE #1
3020 RESET
3030 CLS
3035 PRINT "Enter CONT to continue."
3040 STOP
3050 OPEN "A", #1, "ACCOUNTS.DAT"
3060 RETURN
```

In the above example, the programmer has enabled the CTRL DEL key to enter a subroutine which closes the files and stops program execution until the operator is ready to continue.

Notes:

6. CHAINING PROGRAMS

This chapter describes the statements used when chaining programs together.

They are:

CHAIN

COMMON

MERGE

CHAIN statement

CHAIN transfers control and passes variables to another program.

Syntax:

CHAIN [MERGE] *"filespec"* [, [*linenum*] [, [ALL] [,DELETE *range*]]]

where

"filespec" is a string expression which specifies the name of the called program file.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

linenum 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 it is omitted, execution begins at the first line. *linenum* is not affected by a RENUM command.

range is the range of line numbers to be deleted, if the DELETE option is used. *range* line numbers are affected by the RENUM command.

Before running a CHAINed program, CHAIN carries out a RESTORE. This resets the pointer to the beginning of the internal data file.

If the MERGE option is used, a MERGE operation is performed with the current program and the CHAINed program. The CHAINed program must be an ASCII file. 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. MERGEing may be thought of as "inserting" the program lines on disk into the program in memory. The MERGE option leaves the files open, preserves the current OPTION BASE setting, and preserves variable types and user-defined functions, for use by the CHAINed program.

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.

If the MERGE option is omitted, the CHAINing program is lost (except common variables) before loading the CHAINed program. CHAIN does not preserve variable types or user functions. Thus, any DEFtype or DEF FN statements containing shared variables must be repeated in the CHAINed program.

If the ALL option is used, every variable in the current program is passed to the CHAINed program.

If the ALL option is unused and *linenum* is omitted, two commas must be inserted between the *"filespec"* and the ALL option. For example:

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

is correct, but:

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

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

If the ALL option is omitted, the current program must contain one or more COMMON statements to list the variables that are passed. (*See the COMMON statement in this chapter.*)

If the DELETE option is used, a section of the current program (specified by *range* of line numbers) will be deleted before loading the CHAINed program.

DELETE is often used with MERGE and line options, to load overlays. After an overlay is brought in, it is usually desirable to delete it so a new overlay may be brought in.

CHAIN is used in two different ways in the following examples.

Example 1:

In the first example, the two string arrays are dimensioned and declared as common variables. When PROG1 gets to line 90, it chains to PROG2, which loads the three elements of the B$ array. At line 100 of PROG2, control chains back to PROG1 starting at program line 100. This process can be observed through the descriptive text that prints as the programs execute.

```
Ok
NEW
10 rem this program demonstrates chaining using
COMMON to pass variables.
20 rem save this module on disk as "PROG1" using
the A option.
30 DIM A$(2),B$(3)
40 COMMON A$(),B$()
50 A$(1) = "Variables in common must be"
60 A$(2) = "assigned values before chaining."
70 B$(1) = "" : B$(2) = "" :B$(3) = ""
90 CHAIN "PROG2"
100 PRINT
110 PRINT B$(1)
120 PRINT B$(2)
130 PRINT B$(3)
140 PRINT
150 END
SAVE "PROG1",A
Ok
```

Example 1: - (continued)

```
Ok
NEW
Ok
10 rem the statement "DIM A$(2), B$(3)" 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
60 PRINT A$(1):PRINT A$(2)
70 B$(1) = "Note how the option of specifying a
starting line"
80 B$(2) = "number when chaining avoids"
90 B$(3) = "the dimension statement in PROG1."
100 CHAIN "PROG1",100
110 END
SAVE "PROG2",A
Ok
RUN "PROG1"

Variables in common must be
assigned values before chaining.

Note how the option of specifying a starting line
number when chaining avoids
the dimension statement in PROG1.

Ok
```

Example 2:

In the second example, the MERGE, ALL, and DELETE options are used. After A$ is loaded from the MAINPRG program, control chains to line 1010 of OVRLAY1. At line 1040 of OVRLAY1, it chains to line 1010 of OVRLAY2, keeping all variables and deleting all of OVRLAY1 program lines. Control then passes to OVRLAY2. This process can be observed through the descriptive text that prints as the programs execute.

```
Ok
NEW
Ok
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
SAVE "MAINPRG"
Ok
NEW
Ok
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
SAVE "OVRLAY1",A
Ok
NEW
Ok
1000 rem save this module on the disk as
"OVRLAY2" using the A option.
1010 PRINT A$; " HAS CHAINED TO ";B$;"."
1020 END
SAVE "OVRLAY2",A
Ok
RUN "MAINPRG"
MAINPRG HAS CHAINED TO OVRLAY1.
OVRLAY1 HAS CHAINED TO OVRLAY2.
Ok
```

COMMON statement

COMMON defines a common area which is not erased by the CHAINed program, and allows you to pass variables from one program to another.

Syntax:

COMMON *variable[, variable]...*

where

variable is the name of a numeric or string variable which is required to be passed to the CHAINed program. For array variables, place a set of parentheses "()" after the variable name.

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning.

Variables specified in COMMON statements are allocated in the common area starting from the beginning and in the order in which they appear in the program.

The CHAINed program need not specify, through the use of COMMON statements, the common variable specified by the CHAINing program. The CHAINed program will use these variables with the same names specified in the CHAINing program. Each type definition statement (DEFINT, DEFSNG, DEFDBL, DEFSTR) referring to common variables, must precede the associated COMMON statements and must be repeated in the CHAINed program.

Common variables must always be initialized within the CHAINing program. Common arrays must be explicitly described by DIM statements in the CHAINing program but not in the CHAINed program; otherwise, a Duplicate definition error occurs. The DIM statements must be written before the associated COMMON statements.

Example 1:

The example below shows that the CHAINed program need not specify, through the use of COMMON statements, the common variables specified by the CHAINing program.

In this example, the values of the variables A1, B1, C1, and D1$ in the program PG1 are passed to the CHAINed program PG2, which may display them (see program line 20).

```
10 REM PG1
20 COMMON A1, B1, C1, D1$
      .
      .
      .
80 CHAIN "PG2"
90 END

10 REM PG2
20 PRINT A1, B1, C1, D1$
      .
      .
      .
120 END
```

Example 2:

Each type definition statement (DEFINT, DEFSNG, DEFDBL, DEFSTR) referring to common variables, must precede the associated COMMON statement and must be repeated in the CHAINed program. Note the DEFDBL statements, both with PG1 and PG2.

```
10 REM PG1
20 DEFDBL C
30 COMMON A1, B1, C1, D1$
      .
      .
      .
90 CHAIN "PG2"
100 END

10 REM PG2
20 DEFDBL C
      .
      .
      .
130 END
```

Example 3:

It is not good programming practice to repeat the same variable name (in this case A$) either in different COMMON statements of the same program, or in the same COMMON statement. In any case, multiple definitions are equivalent to a single definition.

```
10 REM PROGRAM1
20 COMMON A$, B$, C$
30 COMMON A$, A1
    .
    .
    .
100 END
```

Example 4:

A COMMON statement can also specify array names. Such specifications are followed by a pair of parentheses.

Each use of common array must be explicitly described by a DIM statement in the CHAINing program (but not in the CHAINed one; otherwise, a Duplicate definition error occurs).

The DIM statement must be written before the associated COMMON statement.

```
10 REM PG1
20 DIM A1(15,20)
30 COMMON A1(),B1,C1
    .
    .
    .
100 CHAIN "PG2"
110 END

10 REM PG2
    .
    .
    .
50 PRINT A1(1,1)
    .
    .
    .
90 END
```

Example 5:

The COMMON statement is a declarative statement, thus it allocates a common area even if control of execution does not pass through it.

When executing program "MOD1", program "MOD2" is CHAINed: it displays both A and B variables, even if statement 50 of "MOD1" is jumped over.

```
10 REM MOD1
20 A = 1:B = 2
30 COMMON A
40 GOTO 60
50 COMMON B
60 CHAIN "MOD2"

10 REM MOD2
20 PRINT A;B
```

MERGE command

Merges the current program with a specified file previously saved in ASCII format.

Syntax:

MERGE *"filespec"*

where

"filespec" is a string expression which specifies the name of the called program file.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

The MERGE command allows you to include a specified program saved (in ASCII format) on a disk, with the program in memory.

MERGE is similar to LOAD, except that the program in memory is not erased before the disk program is loaded. Instead, the disk program is merged into the resident program. That is, program lines in the disk program will simply be inserted into the resident program in sequential order. If a line of the disk program and a line of the resident program have the same line number, the line of the disk program replaces that in memory.

Example:

```
MERGE "B:PAYROLL"
```

Notes:

7. CONVERSION FUNCTIONS

This chapter describes the functions that are used for conversion. For example, converting a given numeric expression to a double precision number.

The following functions are described:

ASC
CDBL
CHR$
CINT
CSNG
HEX$
OCT$
STR$
VAL

ASC function

ASC returns the ASCII decimal code for the first character of a given string.

Syntax:

ASC(*stringexp*)

where

stringexp can be a string of text enclosed with quote marks ("TEST") or a string variable (X$).

The ASC function returns the ASCII code (0-255) corresponding to the first character of the *stringexp*. *See Appendix A for a complete list of all ASCII codes.*

If *stringexp* is null, an Illegal function call error is returned.

See the CHR$ function in this chapter for ASCII-to-string conversion.

Examples:

The following example shows that the ASCII code for capital letter "T" is 84.

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
PRINT ASC("TEST")
 84
Ok
```

CDBL function

CDBL converts a given numeric expression to a double precision number.

Syntax:

CDBL(*numexp*)

where

numexp can be a number or a numeric variable.

Examples:

```
10 A = 454.67
20 PRINT A, CDBL(A)
RUN
 454.67          454.6700134277344
OK
PRINT CDBL(454.67)
 454.6700134277344
Ok
```

CHR$ function

CHR$ returns a one-character string whose ASCII decimal code is the value of the argument.

Syntax:

CHR$(*n*)

where

n is an integer expression which must be in the range of 0 to 255. It represents an ASCII code. If it is outside the specified range, an Illegal function call is returned.

CHR$ is normally used to send a special character to the screen or printer. For instance, the BEL (beep) character (CHR$(7)) could be sent as a preface to an error message, or a form feed character (CHR$(12)) could be sent to clear the screen and return the cursor to the home position.

PRINT CHR$(*n*) may also be used to display an ASCII character, where *n* is the ASCII code (*see Appendix A*).

See the ASC function in this chapter for ASCII-to-numeric conversion.

Examples:

```
Ok
PRINT CHR$(66)
B
Ok
100 PRINT CHR$(7)  'BEEP
150 PRINT CHR$(LINEFEED%)
200 IF CHR$(INP(IN.PORT%)) = "A" THEN GOSUB
1000
```

CINT function

CINT converts any numeric argument to an integer by rounding the fractional portion.

Syntax:

CINT(*numexp*)

where

numexp is a number or numeric variable.

If *numexp* is not in the range -32768 to 32767, an Overflow error occurs.

If the fractional portion of *numexp* is >=.5, the integer part is rounded up; otherwise, a truncation occurs.

See the CDBL and CSNG functions in this chapter for details on converting numbers to the double precision and single precision data types, respectively. See also the FIX and INT functions in Chapter 22, both of which return integers.

Examples:

```
10 P = 45.67
20 PRINT P, CINT(P)
RUN
 45.67          46
Ok
PRINT CINT (-3.71)
-4
Ok
```

CSNG function

CSNG converts any numeric argument to a single precision number.

Syntax:

CSNG(*numexp*)

where

numexp is a number or numeric variable.

See the CINT and CDBL functions in this chapter for converting numbers to the integer and double precision data types, respectively.

Examples:

```
10 A# = 975.3421115
20 PRINT A#, CSNG(A#)
RUN
 975.3421115        975.3421
Ok
PRINT CSNG(975.3421115)
 975.3421
Ok
```

HEX$ function

HEX$ returns a string which represents the hexadecimal value of the decimal argument.

Syntax:

HEX$(*numexp*)

where

numexp is a number or numeric variable.

numexp is rounded to an integer before HEX$ is evaluated.

If ***numexp*** is negative, the two's complement form is used.

See the OCT$ function on the next page for octal conversion.

Examples:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "decimal is " A$ " hexadecimal"
RUN
?32
 32 decimal is 20 hexadecimal
Ok
PRINT HEX$(32)
20
Ok
```

OCT$ function

OCT$ returns a string which represents the octal value of the decimal argument.

Syntax:

OCT$(*numexp*)

where

numexp is a number or numeric variable.

numexp is a numeric expression from -32768 to 65535, which is rounded to the nearest integer before OCT$ is evaluated.

If *numexp* is negative, the two's complement form is used.

See the HEX$ function on the previous page for hexadecimal conversion.

Examples:

```
10 INPUT X
20 A$ = OCT$(X)
30 PRINT X "decimal is " A$ " octal"
RUN
?24
 24 decimal is 30 octal
Ok
PRINT OCT$(24)
30
Ok
```

STR$ function

STR$ returns the string representation of the value of a specified numeric expression.

Syntax:

STR$(*numexp*)

where

numexp is a number or numeric variable.

See the VAL function on the next page.

Examples:

```
10 INPUT "ENTER A NUMBER";N
20 PRINT N, LEN(STR$(N))-1
RUN
ENTER A NUMBER? 6789
 6789          4
Ok
```

In the above example, to use the LEN function to find the total number of digits entered, N has to be converted.

```
10 A$ = STR$(70)
20 PRINT A$
Ok
RUN
70
Ok
```

In the above example, 70 (the argument of STR$) is a number, but the contents of A$ is a two character string whose value is 70.

```
10 A! = 1.3
20 A# = VAL(STR$(A!))
30 PRINT A#
RUN
 1.3
Ok
```

The conversion in line 20 causes the value of A! to be stored accurately in the double-precision variable A#.

VAL function

VAL converts the string representation of a number to its numeric value.

Syntax:

VAL(*stringexp*)

where

stringexp must be a numeric character(s) stored as a string.

The VAL function strips leading blanks, tabs, and linefeeds from the argument string.

The remaining string is converted to a number, if it is a valid numeric representation; otherwise, VAL returns 0 (zero). For example:

```
VAL("-3")
```

returns -3.

```
VAL("ABC")
```

returns 0.

See the STR$ function in this chapter for numeric-to-string conversion.

Example:

```
10 READ PERSON$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$) < 90000 THEN PRINT PERSON$
TAB(25) "OUT OF STATE" ELSE PRINT PERSON$
TAB(25) "CALIFORNIA"
30 GOTO 10
40 DATA BOB JONES, REDWOOD, CA, 90777
50 DATA LINDA SMITH, DENVER, CO, 60233
60 DATA BILL DOE, LONG BEACH, CA, 91811
70 DATA JOHN DOE, BEAUMONT, TX, 77507
RUN
BOB JONES          CALIFORNIA
LINDA SMITH        OUT OF STATE
BILL DOE           CALIFORNIA
JOHN DOE           OUT OF STATE
Out of DATA in 10
Ok
```

Notes:

8. DEBUGGING

This chapter describes the commands used for debugging a program.

TRON/TROFF commands

TRON (TRACE ON) causes the line number of each statement executed to be listed.

TROFF (TRACE OFF) stops the line number listing initiated by TRON.

Syntax:

```
TRON

TROFF
```

The TRON command executed in either direct or program mode is used as a debugging tool. With TRON in operation, each line number of the program is displayed on the screen as it is executed.

The numbers appear enclosed in square brackets.

The trace flag is disabled with the TROFF command or when a NEW command is executed.

Example:

```
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
TRON
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

9. DEVICES AND I/O PORT INFORMATION

This chapter describes the following statements and functions:

CLOSE
ERDEV
ERDEV$
INP
IOCTL
IOCTL$
OPEN
OUT
WAIT
WIDTH

CLOSE statement

Terminates I/O to a device.

Syntax:

CLOSE [[#]*devicenum*[,[#]*devicenum*]...]

where

devicenum is the number under which the device was opened.

A CLOSE with no arguments closes all open files and devices.

The association between a particular device and device number terminates upon execution of a CLOSE statement. The device may then be reopened using the same or a different device number; likewise, a device number may now be reused to open any device.

ERDEV and ERDEV$ functions

ERDEV is an integer function which contains the error code returned by the last device to declare an error.

ERDEV$ is a string function which contains the name of the device driver which generated the error.

Syntax:

ERDEV or ERDEV$

ERDEV is set by the Interrupt X'24' handler, when an error within MS-DOS is detected. 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:, D:, etc.)

For the sake of compatibility between different releases, it is advisable to perform error checking by using ERDEV rather than ERDEV$.

Example:

If a user installed device driver "MYLPT2" caused a Printer out of Paper error via INT 24, and the driver's error number for that problem was 9, ERDEV will contain the error number 9 in the lower 8 bits and the device header word attributes in the upper 8 bits, and ERDEV$ will contain "MYLPT2".

INP function

Returns the byte read from a port.

Syntax:

INP(*portnum*)

where

portnum is a valid port number in the range 0 through 65535.

INP is the complementary function to the OUT statement.

Examples:

```
100 A = INP(54321)
```

IOCTL function

Sends a "Control Data" string to a Character Device Driver anytime after the Driver has been OPENed.

Syntax:

IOCTL[#]*filenum, string*

where

filenum is the file number open to the Device Driver.

string is a string expression containing the Control Data.

IOCTL commands are generally two to three characters optionally followed by an alphanumeric argument. An IOCTL command string may be up to 255 bytes long.

The IOCTL statement works only if:

1. The device driver is installed.
2. The device driver processes IOCTL strings.
3. GW-BASIC performs an OPEN on a file on that device.

Most standard MS-DOS device drivers don't process IOCTL strings, and it is necessary for you to determine whether the specified driver can handle the command.

If a user has installed his own Driver to replace LPT1, and that Driver is able to set Page Length, then an IOCTL command to set or change the page length might be:

PLn

where n is the new page length.

Also see the IOCTL$ function on the next page.

Example:

Opening the new LPT1 driver and setting the Page Length to 66 lines would then be:

```
10 OPEN "LPT1:" FOR OUTPUT AS #1
20 IOCTL#1, "PL66"
```

Possible Errors

Bad file number
IOCTL to a Driver that is not OPEN.

Illegal function call
Device does not support IOCTL.

Device Fault
Error in Control Data.

IOCTL$ function

Returns a "Control Data" string from a Character Device Driver that is OPEN.

Syntax:

IOCTL$([#]*filenum*)

where *filenum* is the file number open to the device.

The IOCTL$ function is most frequently used to receive acknowledgement that an IOCTL statement succeeded or failed, or to obtain current status information.

IOCTL$ could be used to ask a communications device to return the current baud rate, information on the last error, logical line width, etc.

The IOCTL$ function works only if:

1. The device driver is installed.
2. The device driver processes IOCTL strings.
3. GW-BASIC performs an OPEN on a file on that device.

Example:

```
10 OPEN "FN1" AS #1
20 IOCTL #1, "RAW"
30 IF IOCTL$(1) = "0" THEN CLOSE 1
```

if the Character Driver "FN1" gives a "false" return from the Raw data mode IOCTL request, then close the file and stop processing.

Possible Errors

Bad file number
: IOCTL to a Driver that is not OPEN.

Illegal function call
: Device does not support IOCTL

OPEN statement

Allows I/O to a device.

Syntax:

OPEN *device* [FOR *mode*] AS [#] *devicenum*

where

device is a string expression which specifies the device to be opended.

mode is a literal string not enclosed in quotation marks. The valid modes are:

INPUT specifies sequential input mode.

OUTPUT specifies sequential output mode.

If the FOR *mode* clause is omitted, it specifies random I/O mode.

devicenum is an integer expression returning a number in the range 1 through 255. The number is used to associate an I/O buffer with a device. This association exists until a CLOSE or CLOSE *devicenum* statement is executed. The device is referred in any I/O statement by this number.

OPEN allocates a buffer for I/O to the device and determines the mode of access that will be used with the buffer. The *devicenum* parameter specifies the number which will be associated with the device as long as it is open and will be used by other I/O statements to refer to the device.

For each device, the following OPEN modes are allowed:

KYBD:	INPUT only
SCRN:	OUTPUT only
LPT1:	OUTPUT or random*
LPT2:	OUTPUT or random*
LPT3:	OUTPUT or random*
COM1:	INPUT or OUTPUT
COM2:	INPUT or OUTPUT
COM3:	INPUT or OUTPUT
COM4:	INPUT or OUTPUT

*GW-BASIC will not send a line feed after each carriage return, if a printer has been opened in random mode with a width of 255.

The GW-BASIC file I/O system allows you to take advantage of user installed devices.

Character devices are opened and used in the same manner as disk files. However, characters are not buffered by GW-BASIC as they are for disk files. The record length is set to one.

GW-BASIC only sends a CR (carriage return X'0D') as end of line. If the device requires a LF (line feed X'0A'), the driver must provide it. When writing device drivers, keep in mind that GW-BASIC users will want to read and write control information. Writing and reading of device control data is handled by the GW-BASIC IOCTL statement and IOCTL$(f) function.

See examples on next page.

Examples:

If you write and install a device called F01, then the OPEN statement might appear as:

```
10 OPEN "F01" FOR OUTPUT AS #1
```

To open the printer for output, you could use the line:

```
100 OPEN "LPT1:" FOR OUTPUT AS #1
```

which uses the GW-BASIC device driver, or as part of a pathname as in:

```
100 OPEN "DEV LPT1" FOR OUTPUT AS #1
```

which uses the MS-DOS device driver.

Possible Errors

Device not available

You have attempted to open a nonexistent device.

Device I/O error

Reception error. Usually caused by an incorrectly written device driver (user-installed).

OUT statement

Transmits a byte to an output port.

Syntax:

OUT *port,byte*

where

port is an integer expression in the range 0 to 65535 and represents a port number.

byte is an integer expression in the range 0 to 255 and represents the data to be transmitted.

OUT is the complementary statement to the INP function.

If ***port*** or ***byte*** is outside the specified range, an Illegal function call error is returned.

Example:

```
100 OUT 1234,255
```

WAIT statement

Suspends a program execution while monitoring the status of a machine input port.

Syntax:

WAIT *port,i[,j]*

where

port is an integer expression in the range 0 to 65535 and represents a port number.

i,j are integer expressions from 1 to 255.

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is 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 nonzero, execution continues with the next statement. If *j* is omitted, it is assumed to be zero.

Note: It is possible to enter an infinite loop with the WAIT statement, in which case, it will be necessary to manually restart the machine. To avoid this, WAIT must have the specified value at port during some point in the program execution.

Example:

```
100 WAIT 32,2
```

WIDTH statement

Sets the line width in characters. GW-BASIC adds a carriage return after outputting the specified number of characters.

Syntax:

WIDTH *device, size*

where

device is a string expression indicating the device that is to be used. Valid devices are: SCRN:, LPT1:, LPT2:, LPT3:, COM1:, COM2:, COM3:, or COM4:.

size is an integer expression in the range 0 to 255. It specifies the new width.

The default line width for the specified *device* is set to *size*. The line widths of currently open files are not modified.

Stores the new *size* without changing the current width, if the *device* is already open. A subsequent OPEN *device* FOR OUTPUT AS *#devicenum* will use the specified value for width initially.

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 functions, returns to zero after position 255. WIDTH 255 is the default for communications files.

If *size* is outside the above specified ranges, an Illegal function call error is returned. The previous value is retained.

See example on next page.

Example:

```
10 WIDTH "LPT1:",5
20 OPEN "LPT1:" FOR OUTPUT AS #1
30 PRINT #1, "1234567890"
40 PRINT #1, ""
50 WIDTH #1, 6
60 PRINT #1, "1234567890"
RUN
```

will yield on the printer:

```
12345
67890

123456
7890
```

10. DISK DATA FILES --- SEQUENTIAL AND RANDOM ACCESS

This chapter describes disk data files.

There are two types of disk data files that may be created and accessed by a GW-BASIC program. They are:

Sequential files

Random access files

In this chapter, two items will be covered.

First, how to create and access sequential files and random access files.

Second, the statements and functions pertaining to disk data files will be described.

Creating and accessing disk data files

Sequential files

Sequential files are easier to create than random access files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to or read from a sequential file is a series of ASCII characters which are stored and loaded, one item after another (sequentially).

The statements and functions used with sequential files are:

OPEN	PRINT#	INPUT$	LOC
CLOSE	PRINT# USING	INPUT#	LOF
EOF	WRITE #	LINE INPUT#	

Creating a sequential file

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.

```
OPEN "O",#1,"EMPLOYEE"
```

2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)

```
PRINT #1, EMP$; DEPT$; HIREDATE$
```

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.

```
CLOSE #1
OPEN "I",#1,"EMPLOYEE"
```

4. Use the INPUT# statement to read data from the sequential file into the program.

```
INPUT#1,EMP$,DEPT$,HIREDATE$
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma (,) at the end of the format string serves to separate the items in the disk file.

If you want commas to appear in the file as delimiters between variables, the WRITE statement can be used. The statement

```
WRITE 1, A, B$
```

could be used to write these two variables to the file with a comma delimiting them.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data. For example,

```
100 IF LOC(1) > 50 THEN STOP
```

would end program execution if more than 50 sectors had been written to, or read from, file #1 since it was OPENed.

Program 1 is a short program that creates a sequential file, "EMPLOYEE", from information you input at the keyboard.

PROGRAM 1

CREATE A SEQUENTIAL DATA FILE

```
10 OPEN "O",#1,"EMPLOYEE"
20 INPUT "NAME";EMP$
25 IF EMP$ = "DONE" THEN END
30 INPUT "DEPARTMENT";DEPT$
40 INPUT "DATE HIRED";HIREDATE$
50 PRINT#1,EMP$;",";DEPT$;",";HIREDATE$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/78

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/72

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
Ok
```

Accessing a sequential file

Program 2 accesses the file "EMPLOYEE" that was created in Program 1 and displays the name of everyone hired in 1978.

PROGRAM 2

ACCESS A SEQUENTIAL FILE

```
10 OPEN "I",#1,"EMPLOYEE"
20 INPUT#1,EMP$,DEPT$,HIREDATE$
30 IF RIGHT$(HIREDATE$,2) = "78" THEN PRINT
EMP$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 reads, sequentially, every item in the file and prints the name of the employees hired in 1978. When all the data has been read, line 20 causes an Input past end error.

To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

Or, you could use the WHILE...WEND control structure, which also uses the EOF function. The revised program looks like this:

```
10 OPEN "I",#1,"EMPLOYEE"
15 WHILE NOT EOF(1)
20        INPUT #1,EMP$,DEPT$,HIREDATE$
30        IF  RIGHT$(HIREDATE$,2) = "78"  THEN
          PRINT EMP$
40 WEND
```

Adding data to a sequential file

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in the output "O" mode, you destroy its current contents.

Instead, use the append ("A") mode. If the file doesn't already exist, the OPEN statement will work exactly as it would if output ("O") mode had been specified.

Program 3 can be used to add data to the EMPLOYEE file created earlier.

PROGRAM 3

ADD DATA TO A SEQUENTIAL FILE

```
10 OPEN "A",#1,"EMPLOYEE"
20 INPUT "NAME";EMP$
30 IF EMP$ = "DONE" THEN 80
40 INPUT "DEPARTMENT";DEPT$
50 INPUT "DATE HIRED";HIREDATE$
60 PRINT #1,EMP$;",";DEPT$;",";HIREDATE$
70 PRINT:GOTO 20
80 CLOSE 1
```

Random access files

Creating and accessing random access files requires more program steps than sequential files, but there are advantages to using random access files. One advantage is that random access files require less room on the disk, because GW-BASIC stores them in a packed binary format. A sequential file is stored as a series of ASCII characters.

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk. It is not necessary to read through all the information on disk with random access files, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions used with random access files are:

OPEN	FIELD	LSET/RSET
PUT	CLOSE	GET
MKI$	CVI	LOC
MKS$	CVS	
MKD$	CVD	LOF

Creating a random access file

The following program steps are required to create a random access file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes, unless it was set to another value with the /I/S: switches when loading GW-BASIC (see "GWBASIC command" in Chapter 19).

```
OPEN "R",#1,"CUSTOMER",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random access file.

```
FIELD #1 20 AS CUSTNAME$, 4 AS
AMT$, 8 AS PHONE$
```

3. Use the LSET command to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ to make an integer value into a single precision value, and MKD$ to make an integer value into a double precision value.

```
LSET CUSTNAME$ = CUST$
LSET AMT$ = MKS$(AMT)
LSET PHONE$ = TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

The LOC function, with random access files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file#1 is greater than 50.

Program 4 writes information that is input at the keyboard to a random access file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Warning: Do not use a FIELDed string variable in an INPUT or LET (assignment) statement. This causes the pointer for that variable to point into string space instead of into the random access file buffer.

PROGRAM 4

CREATE A RANDOM ACCESS FILE

```
10 OPEN "R",#1,"CUSTOMER",32
20 FIELD #1, 20 AS CUSTNAME$, 4 AS AMT$, 8 AS
PHONE$
30 INPUT "ENTER 2-DIGIT CODE (99 TO
STOP)";CODE%
35 IF CODE% = 99 THEN CLOSE:END
40 INPUT "NAME";CUST$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET CUSTNAME$ = CUST$
80 LSET AMT$ = MKS$(AMT)
90 LSET PHONE$ = TEL$
100 PUT #1,CODE%
110 GOTO 30
RUN
ENTER 2-DIGIT CODE (99 TO STOP)? 10
NAME? JOHN DOE
AMOUNT? 250.00
PHONE? 344-1234

ENTER 2-DIGIT CODE (99 TO STOP)? 20
NAME? TOM SMITH
AMOUNT? 475.00
PHONE? 899-5643

ENTER 2-DIGIT CODE (99 TO STOP)? 99
OK
```

Accessing a random access file

The following program steps are required to access a random access file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"CUSTOMER",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1, 20 AS CUSTNAME$, 4 AS
AMT$, 8 AS PHONE$
```

(*Note: In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.*)

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values that are read in from a random access file buffer must be converted from strings back into numbers using the "convert" functions. CVI converts a 2-byte string to an integer, CVS converts a 4-byte string to a single precision number, and CVD converts an 8-byte string to a double precision number.

Program 5 accesses the random access file "CUSTOMER" that was created in program 4. By inputting the two-digit code at the keyboard, the information associated with that code is read from the file and displayed.

PROGRAM 5

ACCESS A RANDOM ACCESS FILE

```
10 OPEN "R",#1,"CUSTOMER",32
20 FIELD #1, 20 AS CUSTNAME$, 4 AS AMT$, 8 AS
PHONE$
30 INPUT "ENTER 2-DIGIT CODE (99 TO
STOP)";CODE%
35 IF CODE% = 99 THEN END
40 GET #1,CODE%
50 PRINT CUSTNAME$
60 PRINT USING "$$###.##";CVS(AMT$)
70 PRINT PHONE$:PRINT
80 GOTO 30
RUN
ENTER 2-DIGIT CODE (99 TO STOP)? 10
JOHN DOE
 $250.00
344-1234

ENTER 2-DIGIT CODE (99 TO STOP)? 99
Ok
```

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

PROGRAM 6

INVENTORY PROGRAM

```
120 OPEN "R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$, 2 AS R$, 4
    AS P$
130 CLS:PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE
    PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW
REORDER LEVEL"
220 PRINT:PRINT:INPUT "PLEASE ENTER
    FUNCTION";FUNCTION
225 IF (FUNCTION < 1) OR (FUNCTION > 6)
    THEN PRINT "BAD FUNCTION NUMBER":
    GOTO 130
230 ON FUNCTION GOSUB 900, 250, 390, 480,
    560, 680
240 GOTO 130
250 REM ***BUILD NEW ENTRY***
260 GOSUB 840
270 IF ASC(F$) <> 255 THEN INPUT
    "OVERWRITE";ADDR$: IF ADDR$ <> "Y"
    THEN RETURN
280 LSET F$ = CHR$(0)
```

```
290 INPUT "DESCRIPTION";DESCRIPT$
300 LSET D$ = DESCRIPT$
310 INPUT "QUANTITY IN STOCK";QTY%
320 LSET Q$ = MKI$(QTY%)
330 INPUT "REORDER LEVEL";REORDER%
340 LSET R$ = MKI$(REORDER%)
350 INPUT "UNIT PRICE";PRICE
360 LSET P$ = MKS$(PRICE)
370 PUT#1,PART%
380 RETURN
390 REM ***DISPLAY ENTRY***
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "NULL
      ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND
      #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL
      #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";
      CVS(P$)
465 PRINT:INPUT "PRESS RETURN KEY TO
      RETURN TO MENU ";RET$
470 RETURN
480 REM ***ADD TO STOCK***
490 GOSUB 840
500 IF ASC(F$) = 255 THEN PRINT "NULL
      ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";
      ADD%
520 QTY% = CVI(Q$) + ADD%
530 LSET Q$ = MKI$(QTY%)
540 PUT#1,PART%
550 RETURN
560 REM ***REMOVE FROM STOCK***
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "NULL
      ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";LESS%
610 QTY% = CVI(Q$)
620 IF (QTY%-LESS%)  0 THEN PRINT "ONLY";
      QTY%; " IN STOCK":GOTO 600
630 QTY% = QTY%-LESS%
640 IF QTY% < = CVI(R$) THEN PRINT
      "QUANTITY NOW";QTY%;" REORDER
      LEVEL"; CVI(R$)
650 LSET Q$ = MKI$(QTY%)
```

```
660 PUT#1,PART%
670 RETURN
680 REM ***DISPLAY ITEMS BELOW REORDER
    LEVEL***
690 FOR I = 1 TO 100
710       GET#1,I
720       IF CVI(Q$) CVI(R$) THEN PRINT D$;"
          QUANTITY"; CVI(Q$) TAB(50) "REORDER
          LEVEL";CVI(R$)
730 NEXT I
735 PRINT:INPUT "PRESS RETURN KEY TO
    RETURN TO MENU ";RET$
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART% < 1) OR (PART% > 100) THEN
    PRINT "BAD PART NUMBER":GOTO 840 ELSE
    GET#1,PART%:RETURN
890 END
900 REM ***INITIALIZE FILE***
910 INPUT "ARE YOU SURE";CONFIRM$:IF
    CONFIRM$<>"Y" THEN RETURN
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940  PUT#1,I
950 NEXT I
960 RETURN
```

CLOSE statement

For sequential and random access files.

Terminates I/O to a file or device.

Syntax:

CLOSE [[#]*filenum* [, [#]*filenum*]...]

where

filenum is the number under which the file was OPENed.

The CLOSE statement is complementary to the OPEN statement.

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 always closes all disk files automatically. STOP does not close disk files.

Examples:

The following statement causes the files with files number 1 and 2 to be closed.

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

The next statement causes all open files to be closed.

```
1500 CLOSE
```

CVI, CVS, CVD functions

For random access files only.

Converts string values to numeric values.

Syntax:

CVI(*2-byte string*)
CVS(*4-byte string*)
CVD(*8-byte string*)

Numeric values that are read in from a random file buffer 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.

Example:

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

EOF function

For sequential and random access files.

Tests for the end-of-file condition.

Syntax:

EOF (*filenum*)

where

filenum is the file number specified in the OPEN statement.

Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid Input past end errors.

When the EOF function is used with random access files, it returns "true" if the last executed GET statement was unable to read an entire record because of an attempt to read beyond the end.

Example:

```
5 DIM M(500)
10 OPEN "I",1,"PAYROLL"
20 C = 0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C = C + 1:GOTO 30
                .
                .
                .
```

This example reads data from the sequential file named "PAYROLL". Values are read into array M until the end of file is reached.

FIELD statement

For random access files only.

Allocates space for variables in a random file buffer.

Syntax:

FIELD [#]*filenum*, *width* AS *stringvar* [, *width* AS *stringvar*] ...

where

filenum is the file number specified in the OPEN statement.

width is the number of characters to be allocated to *stringvar*.

stringvar is a string variable name that will be used for random file access.

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

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.

Do not use a FIELDed variable name in an INPUT or LET (assignment) statement.

Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET (assignment) statement with that variable name is executed, the variable no longer refers to the random file record buffer, but to the variables stored in string space.

Note that a variable name, previously defined in a FIELD statement, may be inserted to the right of the equal sign in an assignment statement.

See the four examples below.

Example 1:

```
10 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.

Example 2:

```
10 OPEN "R",#1,"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
```

Illustrates a multiple 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. In the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

Example 3:

```
10 FOR LOOP% = 0 TO 7
20 FIELD #1, (LOOP% * 16) AS OFFSET$, 16 AS
A$(LOOP%)
30 NEXT LOOP%
```

Shows 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)
```

Example 4:

```
10 FIELD #1,255 AS TST$
```

Note that you must observe the maximum length restriction for various variables. For example in the FIELD statement above the maximum length of TST$ is 255.

GET statement

For random access files only.

Reads a record from a random disk file into a random buffer.

Syntax:

GET [#]*filenum*, [, *recordnum*]

where

filenum is the file number specified in the OPEN statement.

recordnum is the number of the record to be read, in the range 1 to 16,777,215.

If *recordnum* is omitted, the next record (after the last GET) is read into the buffer.

The largest possible record number is 16,777,215. This permits large files with short record lengths.

After a GET statement has been executed, you can either refer to FIELDed variables or use INPUT #, LINE INPUT # to read characters from the random file buffer. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

See example on next page.

Example:

```
10 OPEN "R",1,"EMPLOYEE",48
20 FIELD 1, 20 AS R1$, 20 AS R2$, 8 AS R3$
30 FOR L = 1 TO 4
40       GET 1,L
50       PRINT R1$, R2$, CVD(R3$)
60 NEXT
70 CLOSE 1
80 END
Ok
RUN
SUPER MANN          USA          11234621
ROBIN HOOD          England      23462101
```

This program retrieves information stored in the specified file. The data read into the buffer may be accessed by the program. This is done here by a PRINT statement (see statement 50). These data items were written to the file by the PUT statement.

INPUT # statement

For sequential files only.

Reads data items from a sequential disk file and assigns them to program variables.

Syntax:

INPUT *#filenum, variable* [, *variable*] ...

where

filenum is the number used when the file was OPENed for input.

variable is a numeric or string variable which will receive a data item from the file. The type of data in the file must match the type specified by the variable name.

With INPUT #, no question mark is displayed, 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, leading spaces, carriage returns and linefeeds are ignored. 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 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:

```
100 INPUT #1, X$, Y$, Z$
```

This example uses the INPUT # statement to read data from a sequential file into the program.

INPUT $ function

For sequential files only.

Returns a string of characters read from a file.

Syntax:

INPUT$(*length,#filenum*)

where

length is an integer expression specifying the number of characters to be read from a file.

filenum is the file number specifying the file to be read.

INPUT$ allows all characters read to be assigned to a string.

Example:

```
5 rem list the contents of a sequential file in
hexadecimal
10 OPEN "I",1,"EMPLOYEE"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

LINE INPUT# statement

For sequential files only.

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Syntax:

LINE INPUT#*filenum*, *stringvar*

where

filenum is the number under which the file was OPENed.

stringvar 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. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return. If a linefeed/carriage return sequence is encountered, it is preserved.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program.

See example on next page.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "Customer? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1,C$
70 PRINT C$
80 CLOSE 1
RUN
Customer? Linda Jones  234  Memphis
Linda Jones  234  Memphis
Ok
```

LOC function

For sequential and random access files.

Returns the current position in the file.

Syntax:

LOC(*filenum*)

where

filenum is the number under which the file was OPENed.

For 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 size of the file in (bytes/128).

For random access disk files, LOC returns the record number just read or written from a GET or PUT statement.

Example:

```
.
.
.
200 IF LOC(1) > 50 THEN STOP
.
.
.
```

LOF function

For sequential and random access files.

Returns the length of the named file in bytes.

Syntax:

LOF(*filenum*)

where

filenum is the number under which the file was OPENed.

For sequential and random access files, LOF returns the size of the file in bytes.

Note that, when a file is OPENed for APPEND or OUTPUT, LOF returns the size of the file divided by 128.

Examples:

In this example, the variables REC and RECSIZ contain the record number and recond length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

```
110 IF REC * RECSIZ > LOF(1) THEN PRINT
"INVALID ENTRY"
```

In the next example, the statements will get the last record of the file MYFILE (residing on the disk in Drive B) assuming that the file was created with a record length of 128 bytes.

```
10 OPEN "B:MYFILE" AS #2
20 GET #2, LOF(1)/128
```

LSET and RSET statements

For random access files only.

LSET stores a string value in a random buffer field left-justified.

RSET stores a string value in a random buffer field right-justified.

Syntax:

LSET *stringvar* = *stringexp*

RSET *stringvar* = *stringexp*

where

stringvar represents a fielded string variable (i.e., a string variable previoulsy used in a FIELD statement)

stringexp represents the string to be left or right-justified in a given field.

If *stringexp* requires fewer bytes than were FIELDed to *stringvar*, 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.

Example:

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = MKI$(COUNT%)
```

MKI$, MKS$, MKD$ functions

For random access files only.

Converts numeric values to string type values.

Syntax:

MKI$(*integer expression*)
MKS$(*single precision expression*)
MKD$(*double precision expression*)

Any numeric value that is 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.

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
.
.
.
```

OPEN statement

For sequential and random access files.

Allows I/O to a file.

Syntax:

OPEN *filespec* [FOR *mode1*] AS [#]*filenum* [LEN = *record-length*]

OPEN *mode2*, [#]*filenum*, *filespec* [, *record-length*]

where

filespec is a string expression which specifies the file to be opened. It may optionally include a drive name or path name. If drive name is omitted, the default drive is assumed. If path name is omitted, the current working directory is assumed.

mode1 is a literal string not enclosed in quotation marks. It determines the initial file pointer position and the action to be taken if the file does not exist. The valid modes and actions taken are:

INPUT specifies sequential input mode. Positions the pointer to the beginning of an existing file. A File not found error is given if the file does not exist.

OUTPUT specifies sequential output mode. Positions the pointer to the beginning of the file. If the files does not exist, one is created.

APPEND specifies sequential output after the last record on the file. Positions the pointer to the end of the file. If the file does not exist, one is created.

If the FOR *mode1* clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the Random I/O mode. That is. records may be read or written at will at any position within the file.

filenum is an integer expression returning a number in the range 1 through 255. The number is used to associate an I/O buffer with a disk file. This association exists until a CLOSE or CLOSE *filenum* statement is executed. The file is referred in any I/O statement by this number.

record-length is an integer expression from 1 to 32767. This value sets the record length to be used for random access files (*see the FIELD statement*). If omitted, the *record-length* defaults to 128 byte records. The specified *record-length* may not be greater than the value specified by the /S: switch on the GWBASIC command line (*see "GWBASIC command" in Chapter 19*). GW-BASIC will ignore this option if it is used to OPEN a sequential file.

mode2 is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

A specifies sequential output mode and sets the file pointer at the end of the file, and the record number as the last record of the file. A PRINT or WRITE statement will then extend (append) the file.

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 and determines the mode of access that will be used in the buffer.

The *filenum* parameter specifies the number which will be associated with the file as long as it is open and will be used by other I/O statements to refer to the file.

The maximum number of files that may be open simultaneously is set by the /F: switch in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*). This number falls within the range 1 to 15, and defaults to 3.

Rules

1. If you enter a value outside of the corresponding range, an Illegal function call error is returned, and the file will not be opened.

2. If the file is opened for INPUT, attempts to write to the file will result in a Bad File Mode error. If a file opened for input does not exist, a File not found error occurs.

3. When a disk file is opened for APPEND, the pointer position is initially at the end of the file and the record is set to the last record of the file. PRINT# or WRITE# will then extend the file.

4. If the file is opened for OUTPUT or APPEND, attempts to read the file will result in a Bad File Mode error.

5. If you open a file which does not exist for output, append or random access, you will create that file.

6. A file can be opened for sequential input or random access on more than one file number at a time. A file may NOT be opened for OUTPUT or APPEND on more than one file number at a time.

   Moreover, since it is possible to reference the same file in a subdirectory via different path names, it is impossible for GW-BASIC to know that it is the same file simply by looking at the path name. For this reason, GW-BASIC will not let you open the file for OUTPUT or APPEND if it is on the same disk, even if the path name is different.

Examples:

```
10 OPEN "I",2,"INVEN"

10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

Possible Errors

Bad file name

Bad file number

Bad file mode

Too many files (Too many files are open. *See the /F: switch in the "GWBASIC command" in Chapter 19.*)

File not found

File already open

Illegal function call (Usually caused by an excessive record length. *See the /S: switch in the "GWBASIC command" in Chapter 19.*)

PRINT# and PRINT# USING statements

For sequential files only.

Writes data sequentially to a disk file.

Syntax:

PRINT#*filenum*, [USING *format-string;*] *list of expressions*

where

filenum is the number used when the file was OPENed for output.

format-string is a string expression (usually a constant or variable) composed of formatting characters described in the PRINT USING statement. *See the alphabetized listing of the commands, statements and functions for location of the PRINT USING statement description.*

list of expressions is a list of the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the file, just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data, so that it will be input correctly from the disk file.

In the *list of expressions*, numeric expressions should be delimited by semicolons. For example,

```
50 PRINT#1,A;B;C;X;Y;Z
```

If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the disk file.

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the disk file, use explicit delimiters in the *list of expressions*.

For example, let A$="CAMERA" and B$="93604-1". The statement

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

would write CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

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

The image written to the file is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, write them to the disk file surrounded by explicit quotation marks, using CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

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

would write the following image to file:

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

and the statement

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

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$.

To separate these strings properly in the disk file, write double quotation marks to the file image using CHR$(34). The statement

```
500 PRINT#1, CHR$(34); A$; CHR$(34);
CHR$(34); B$; CHR$(34)
```

writes the following image to the file:

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

and the statement

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

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example,

```
700 PRINT#1,USING"$$###.##,";J;K;L
```

PUT statement

For random access files only.

Writes a record from a random buffer to a random access file.

Syntax:

PUT[#]*filenum*[,*recordnum*]

where

filenum is the number under which the file as OPENed.

recordnum specifies the number of the record in the file. It must be in the range 1 to 16,777,215.

LSET, RSET, 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.

See example on next page.

Example:

```
10 OPEN "R",1,"RAND",48
20 FIELD 1,20 AS R1$,20 AS R2$,8 AS R3$
30 FOR L = 1 TO 4
40       INPUT "Name";N$
45       IF N$ = "DONE" GOTO 120
50       INPUT "Address";M$
60       INPUT "Phone";P#
70       LSET R1$ = N$
80       LSET R2$ = M$
90       LSET R3$ = MKS$(P#)
100  PUT 1,L
110 NEXT L
120 CLOSE 1
130 END
Ok
RUN
Name? Super Man
Address? USA
Phone? 11234621
Name? Robin Hood
Address? England
Phone? 23462101
Name? DONE
Ok
```

Statement 10 opens the random file RAND, with a record length of 48 on the disk in Drive A. The file number is 1. Statement 20 divides the buffer into fields.

Statement 100 writes a record to file RAND, with the record number being set by the control variable of the FOR...NEXT loop.

VARPTR function

For sequential and random access files.

For sequential files, returns the starting address of the disk I/O buffer assigned to *filenum*.

For random access files, returns the address of the FIELD buffer assigned to *filenum*.

Syntax:

VARPTR(#*filenum*)

where

filenum is the number under which the file was OPENed.

The address returned will be an integer in the range -32768 to 32767. This integer value is the offset into GW-BASIC's Data Segment. If a negative address is returned, add it to 65536 to obtain the actual address.

WRITE# statement

For sequential files only.

Writes data to a sequential file.

Syntax:

WRITE#*filenum,list of expressions*

where

filenum is the number under which the file was OPENed in "O" mode.

list of expressions is a list of 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 you to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in the list is written to the file.

If A$="CAMERA" and B$="93604-1", the statement:

```
100 WRITE#1,A$,B$
```

writes the following image to file:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

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

would input "CAMERA" to A$ and "93604-1" to B$.

11. DISK FILES

This chapter describes how to

Close all open data files
Delete file(s) from disk
Display file names (directory)
Execute (run) a program file (.BAS)
Load a program file (.BAS) from disk into memory
Move a file from one directory to another
Rename a file
Save a program to disk

Close all open data files

The RESET command/statement closes all open data files on all drives.

Syntax:

```
RESET
```

RESET closes all open data files on all drives, and forces all blocks in memory to be written to disk. Thus, if the machine loses power, all files will be properly updated.

All files must be closed before a disk is removed from its drive.

Note that RESET performs the same action as CLOSE with no arguments, if all open data files are residing on disk.

Delete file(s) from disk

The KILL command/statement deletes a file from the disk.

Syntax:

KILL "filespec"

where

"filespec" is a string expression which specifies the file to be deleted. The file name must include the extension, if one exists.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

The file name may contain question marks (?) or asterisks (*) used as wildcards. **Be extremely careful when using wildcards with this command.**

If the file name has an extension on it (e.g., .BAS, .DAT, etc.), the extension has to be included with the KILL command (e.g., KILL "PAYROLL.DAT"). If it is not included, a File not found error will occur.

KILL can only be used to delete a file. You must use the RMDIR command (Chapter 20) to remove a directory.

KILL is used for all types of disk files: program files, random data files and sequential data files.

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 one you may have open. It is possible to get an unexpected File already open error under these circumstances.

Examples:

```
KILL "ACCOUNTS.BAS"
```

Deletes the program file entitled ACCOUNTS.BAS from the disk in the default drive.

```
KILL "B:PAYROLL.DAT"
```

Deletes the file entitled PAYROLL.DAT from the disk in Drive B.

```
200 KILL "DATA1.DAT"
```

This program line deletes a data file entitled DATA1.DAT from the disk in the default drive during execution of a program.

```
310 KILL "DATA1.*"
```

This program line deletes all files named DATA1, regardless of the extension from the disk in the default drive.

```
220 KILL "SALES\*.DAT"
```

This program line deletes all files with the extension .DAT in a subdirectory entitled SALES of the current "working" directory.

Display file names (directory)

The FILES command/statement displays the names of files and subdirectories residing on the specified disk/path name.

Syntax:

FILES [*"filespec"*]

where

"filespec" is a string expression specifying either a file name or a path name, and optionally a drive name. If the drive name is omitted, the MS-DOS default drive is assumed. If no path name is specified, the current "working" directory for the specified drive is assumed.

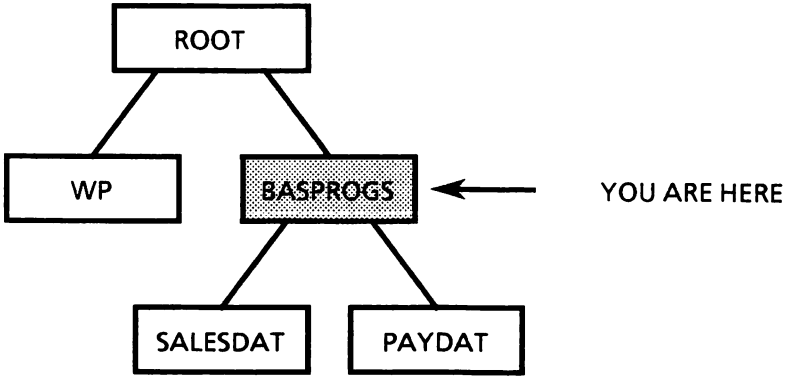
If only FILES is specified, all the files on the current directory of the MS-DOS default drive will be displayed.

A file name may contain question marks (?) or asterisks (*) used as wild cards.

If subdirectories exist, they are denoted by <DIR> following the name. (See the next page for information on how to display files in subdirectories.)

See examples on page 5-177.

To better understand how to display the file names in a subdirectory, assume you are in the BASPROGS subdirectory shown below. This is your current "working" directory.



To display the file names in one of the subdirectories of the current "working" directory, you would type the subdirectory name followed by a backslash. For example,

```
FILES "SALESDAT\"
```

will display the file and subdirectory names in the SALESDAT subdirectory of the current "working" directory (BASPROGS).

Note: If you did not put a backslash after the subdirectory name, the system will not display the file and subdirectory names in the specified subdirectory. It would only check to see if there is a file or subdirectory with that name and display the name again as confirmation.

To display the file and subdirectory names in an upper level directory which in this case is the root directory, you would type a backslash. For example, FILES "\".

If you wanted to display the file and subdirectory names in the WP subdirectory (which is located outside the working directory), you would use the command FILES "\WP\".

Examples:

FILES

Displays all file and subdirectory names in the current "working" directory of the disk in the default drive. Subdirectory names will have <DIR> displayed beside them.

FILES "B:*.*" or FILES "B:"

Displays all file and subdirectory names in the current "working" directory of the disk in Drive B

FILES "B:*.BAS"

Displays all file names with the extension of .BAS in the current "working" directory of the disk in Drive B.

FILES "TEST?.BAS"

Displays all the file names with the first four characters of TEST, with any fifth character, and an extension of .BAS in the currect working" directory of the disk in the default drive.

FILES "SALES"

If SALES is a subdirectory of the current "working" directory, this command displays SALES <DIR> and the amount of bytes free. If SALES is a file in the current "working" directory, this command displays SALES and the amount of bytes free.

FILES "SALES\MARY"

If MARY is a subdirectory of SALES, this command will display MARY <DIR> and the amount of bytes free. If MARY is a file, it will display MARY and the amount of bytes free.

Execute (run) a program file (.BAS)

The RUN command/statement loads a program file from disk into memory and runs it.

After execution, GW-BASIC returns to command level.

Syntax:

RUN *"filespec"* [,R]

where

"filespec" is a string expression which specifies the program to be loaded and run.

"filespec" is a file name or path name with an optional drive name. GW-BASIC appends the default extension .BAS, if none is specified. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

R is an option. If it is specified, all data files that were opened before loading the specified program remain open.

Before loading the specified program, RUN deletes the current contents of memory and closes all open files. However, if the R option is specified, all open data files remain open.

Examples:

```
RUN "PAYROLL"
```

Loads the program file named PAYROLL into memory from the disk in the default drive and executes it.

```
RUN "B:SALES",R
```

Loads the program file named SALES from the disk in Drive B, leaves all data files open and executes the program.

```
RUN "JOHN\INCOME"
```

Loads the INCOME program from the subdirectory named JOHN in the current "working" directory.

Load a program file (.BAS) from disk into memory

The LOAD command loads a program file (.BAS) into memory from a specified disk. You can run the program, if you specify the option R.

Syntax:

LOAD *"filespec"* [,R]

where

"filespec" is a string expression which specifies the program file to be loaded. The file name is the name that was used when the program was SAVEd.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

R is an option. If specified, it will cause the program to begin execution from the first statement after loading. In this case, all open data files are kept open.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the specified program.

If the R option is omitted, GW-BASIC returns to command level after the program is loaded.

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.

Note that

```
RUN "filespec", R
```

is equivalent to

```
LOAD "filespec", R
```

Examples:

```
LOAD "PAYROLL"
```

Loads the program file entitled PAYROLL from the disk in the default drive into memory and system returns to command level.

```
LOAD "B:STRTRK",R
```

Loads the program file entitled STRTRK from the disk in Drive B into memory and executes it, where B could be replaced by any drive name

Move a file from one directory to another

If you have multiple directories on a disk, the NAME command can be used to move or move and rename a file from one directory to another but not across disks (drives).

Syntax:

NAME *"filespec"* AS *"new path name"*

where

"filespec" is a string expression which specifies the file to be moved.

"filespec" is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

"new path name" is a string expression which specifies the path name to move the file to.

If you move a file, the file exists on the same disk with the same name but resides in a different directory.

If you move and rename a file, the file exists on the same disk with a new name and resides in a different directory.

Examples:

```
NAME "JOHN\SALES.BAS" AS "SALLY\SALES.BAS"
```

This moves the file named SALES.BAS from the subdirectory JOHN to the subdirectory SALLY. Both of these are subdirectories of the current "working" directory.

```
NAME "JOHN\SALES.BAS" AS "SALLY\JAN.BAS"
```

This moves the file named SALES BAS from the subdirectory JOHN to the subdirectory SALLY and renames the file to JAN.BAS. Both of these are subdirectories of the current "working" directory.

Rename a file

The NAME command/statement changes the name of a disk file.

Syntax:

NAME *"filespec" AS "filename"*

where

"filespec" is a string expression which specifies the file to be renamed.

"filespec" is a file or path name with an optional drive name. The file extension does not default to .BAS. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

"filename" is the new name of the file.

The *"filespec"* must exist and *"filename"* must not exist; otherwise, an error will result.

A file may not be renamed with a new drive name. If this is attempted, a Rename across disks error will be generated (e.g., NAME "PAY.BAS" AS "B:NEW.BAS").

If the *"filespec"* file is open, it must be closed before the renaming command is executed.

After a NAME command, the file exists on the same disk, with the new name. Also, the area allocated to the file will not be changed.

NAME cannot be used to rename directories.

Examples:

```
NAME "ACCTS.BAS" AS "LEDGER.BAS"
```

The file that was formerly named ACCTS.BAS on the disk in the default drive will now be named LEDGER.BAS on the same disk.

```
NAME "B:PAY.DAT" AS "B:JANPAY.DAT"
```

The file PAY.DAT on the disk in Drive B will now be named JANPAY.DAT, where B could be replaced with any drive name.

Save a program to disk

After creating or editing a program, you can store it on disk using the SAVE command.

Syntax:

SAVE *"filespec"* [,A or ,P]

where

"filespec" is a string expression which specifies where to save the program and what file name to save it under.

"filespec" is a file or path name with an optional drive name. With MS-DOS, the default extension .BAS is supplied. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

If a file with the same name already exists on the selected disk, it will be written over.

Use the A option to save the file in ASCII format. Otherwise, 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 requires an ASCII format file. Attempts to MERGE binary programs will result in a Bad file mode error. Also, some operating systems commands such as TYPE may require an ASCII format file.

Use the P option to protect 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 with an Illegal function call error.

CAUTION: No way is provided to "unprotect" such a program.

Examples:

```
SAVE "PAYROLL"
```

Saves the program PAYROLL.BAS to the disk in the default drive in binary format.

```
SAVE "B:SALES",A
```

Saves the program SALES.BAS to the disk in Drive B in ASCII format, where B could be replaced with any drive name.

```
SAVE "B:PROG",P
```

Saves the program PROG.BAS to the disk in Drive B as a protected file.

```
SAVE "JOHN\PAYROLL"
```

Saves the PAYROLL.BAS program to the subdirectory named JOHN of the current "working" directory.

Notes:

12. EDITING

All text entered while GW-BASIC is at command level is processed by the GW-BASIC Editor. This is a "screen line editor" which allows you to change a line anywhere on the screen (only one line at a time). Changes are only registered when you press the Return key on that line.

This chapter describes

Special screen editor keys

Correct the current line

Modify program lines

Special screen editor keys

The GW-BASIC Editor recognizes nine numeric keypad keys, the Backspace key (←), and the CTRL key to move the cursor, insert or delete characters.

The keys and their functions are listed below.

KEY	FUNCTION
HOME	Home: Positions the cursor in the top left hand corner of the screen.
CTRL HOME	Clear Screen: Clears the screen and moves the cursor to the "Home" position.
↑	Cursor Up: Moves the cursor up one line.
↓	Cursor Down: Moves the cursor one position (line) down.
← (keypad)	Cursor Left: Moves the cursor one position left. When the cursor is moved beyond the left limit of the screen, it appears at the right side of the screen on the preceding line.
→	Cursor Right: Moves the cursor one position right. If the cursor is moved beyond the right limit of the screen, it appears to the left side of the screen on the following line.

KEY	FUNCTION
CTRL →	Next Word: Moves the cursor to the beginning of the following word, i.e., to the next character to the right of the cursor in the set [A..Z] or [a..z] or [0..9], which follows a blank or special character. For example, in the following line: 30 IF L< = 0 THEN 20 The cursor is under the letter L. If you press CTRL →, the cursor will move to the beginning of the next word, which is 0: 30 IF L< = 0 THEN 20 If you press CTRL → again, the cursor will move to the next word, which is THEN: 30 IF L< = 0 THEN 20
CTRL ←	Previous Word: Moves the cursor to beginning of the preceding word, i.e., to the first character to the left of the cursor in the set [A..Z] or [a..z] or [0..9] which is preceded by a blank or a special character. For example: 30 IF L< = 0 THEN 20 The cursor is under the letter T. If you press CTRL ← the cursor will move to 0. Pressing CTRL ← again, it will move to L.

KEY	FUNCTION
END	End, Append: Moves the cursor from its current position to the end of the logical line. Subsequent characters are appended to the line.
CTRL END	Erase to end of line: Erases from the current cursor position to the end of the logical line, i.e., until the carriage return is found.
INS	Switch insert/overwrite mode: Switches into or out of Insert Mode. If Insert Mode is off (Overwrite Mode on), then it turns it on. If Insert Mode is on, then it turns it off (sets Overwrite Mode). Insert Mode cursor is a half-height blinking block (in Text Mode) and is a blinking triangle to the left of each character (in Graphics Mode). Overwrite mode is indicated by a different cursor, which is a slow-blinking underline. In Insert Mode, the character immediately above, together with those following the cursor move to the right as characters are entered at the current cursor position. As characters disappear off the right side of the screen, they reappear on the left of the following line. When out of Insert Mode, characters typed will replace existing characters on the line. Insert Mode is turned off when you press the INS key again, or if you press any of the cursor movement keys or the Return key.

KEY	FUNCTION
→|	Tab: When out of Insert Mode, pressing →| moves the cursor over characters until the next tab stop is reached. Tab stops occur every 8 character position starting from position 1. For example, given the line below: 20 INPUT "Length"; L If you press the →| key, the cursor will move to the 17th position as shown: 20 INPUT "Length"; L When in Insert Mode, pressing →| causes blanks to be entered from the current cursor position to the next tab stop. As characters disappear off the right side of the screen, they reappear on the left on the following line. For example, given the line below: 20 INPUT "Length"; L Blanks are entered up to the 17th position by pressing the INS key and then the →| key. 20 INPUT " Length"; L

KEY	FUNCTION
DEL	Delete Character: Erases the character located at the current cursor position. All characters which follow the deleted character shift one position left. If a logical line extends beyond one physical line, characters on subsequent lines shift left one position, and the character in the first column of each subsequent line is moved up to the end of the preceding line.
← (typewriter)	Backspace: Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of and above the cursor shift left one position. Subsequent characters and lines within the current logical line move up as with the DEL key.
CTRL RETURN	Line Feed: Causes subsequent text to start automatically on the next screen line.
ESC	Delete Line: The entire logical line containing the cursor is cleared. The line is not entered for processing. If it is an existing program line, it is **not** deleted from the program currently in memory.
CTRL BREAK	Break: Returns to Command Level, without saving any modifications that were made to the current line being edited. Unlike ESC, it does not delete the line from the screen.

Correct the current line

All text entered at GW-BASIC command level is processed by the Screen Editor. You can therefore use any of the special screen editor keys previously described.

GW-BASIC remains at command level after the prompt Ok and until a RUN command is received.

Character modification

If you make a mistake while entering a line, then proceed as follows. For example, suppose you have typed:

```
RUN "K,PROGR_
```

when you should have entered

```
RUN "B:PROG_
```

Use ←, or other cursor movement keypad keys, to move the cursor to the appropriate position:

```
RUN "K,PROG
```

Type the correct character over the wrong one:

```
RUN "B:PROG
```

Move the cursor to the end of the line using → or END keys:

```
RUN "B:PROG_
```

Continue typing if the line is not finished:

```
RUN "B:PROGRAM11"_
```

Press the Return key to pass the line to GW-BASIC. In this case, the specified program is loaded from the disk in Drive B and run.

Character insertion

If you accidentally omit characters in the line you are entering, then proceed as follows.

For example, suppose you entered:

```
10 FO  K = 1 TO__
```

instead of:

```
10 FOR K = 1 TO__
```

Use ←, or other cursor movement keypad keys, to move the cursor to the appropriate position:

```
10 FO_K = 1 TO
```

Press INS and type the letter R:

```
10 FOR_K = 1 TO
```

Note that, entering Insert Mode, the cursor becomes a half-height block.

Press INS again to return to Overwrite Mode and then press → or END to move the cursor to the end of the line:

```
10 FOR K = 1 TO__
```

Character deletion

If you accidentally type an extra character in the line you are entering, then proceed as follows.

For example, suppose you entered:

```
GOTTO_
```

instead of:

```
GOTO_
```

To erase the extra T, press ←, or other cursor movement keypad keys, to move the cursor to the appropriate position:

```
GOTTO
```

Press DEL key:

```
GOTO
```

Move the cursor using →

```
GOTO_
```

Continue typing:

```
GOTO 1000_
```

Delete part of a line

To erase a line from the current cursor position, press CTRL END.

Delete an entire line

To cancel the line you are entering, press ESC anywhere in the line. It is not necessary to press the Return key.

Modify program lines

Any line of text beginning with a number (0 to 65529) is considered to be a "program line".

Before editing a program, you need to load the program to be modified (edited) into memory and list the program contents so that you can see where the program is to be modified.

Changes to a line are recorded when a Return is entered while the cursor is somewhere on that line. The Return enters all changes for that logical line, and up to the 255 character line limitation, no matter how many physical lines are included and no matter where the cursor is located on the line.

Note that any modifications you make by using the GW-BASIC screen editor only change the program in memory. To store the updated version of your program in a disk file, you must use the SAVE command.

Add a program line

Type the new line number and the program line. For example, if you needed to add a line between the existing line numbers 20 and 30, you could use a line number between 21 and 29, inclusive.

If program memory is exhausted, and a program line is added to a program, an Out of memory error message is displayed, and the program line is not added.

Replace an existing line

Type a line number that matches an existing one, followed by the contents of the new line. The new line will replace the existing one.

Delete a program line

Type the line number of the line to be deleted and press the Return key.

An Undefined line number error is returned if an attempt is made to delete a line which does not exist.

Note: ESC *should not be used to delete program lines, since this erases from the screen only, and not from the program in memory.*

Delete a group of program lines

Use the DELETE command to delete a group of program lines. The syntax of DELETE is:

DELETE [*linenum1*][- [*linenum2*]]

where

linenum1 is the first line to be erased.

linenum2 is the last line to be erased.

If either *linenum1* or *linenum2* does not exist, an Illegal function call error occurs.

A period (.) can be used instead of the line number to indicate the current line.

GW-BASIC always returns to command level after a DELETE is executed.

Examples:

DELETE 80	Deletes line 80.
DELETE 80-120	Deletes lines 80 through 120, inclusive.
DELETE -80	Deletes all lines up to and including line 80.
DELETE 80-	Deletes all lines from line 80 through the end of the program.

Modify program line displayed on screen

Move the cursor to the appropriate position (by the cursor movement keys); modify the line using any of the techniques described previously to change, delete or insert characters to the line. Press the Return key to pass the modified line to GW-BASIC.

Modify program line not displayed on screen

Use the EDIT command to display the line, or the LIST command to display a group of lines including the line you want to modify, move the cursor to the appropriate position, modify the line and press the Return key.

Note: You can edit any line as long as it is visible on the screen. Once a direct line has been sent to the system pressing Return, there is no way to edit it. This is not the case with program lines, as they may always be recalled for editing to the screen.

If you are going to use the EDIT command, here is a description of that command. The syntax is:

EDIT { *linenum* | . }

where

linenum is a program line number. If no such line exists, an Undefined line number error message is displayed.

Alternatively a period (.) can be used instead of a line number to specify the current line.

When you enter an EDIT command, GW-BASIC displays the specified line and positions the cursor under the first digit of the line number. The line may then be modified by using the special editor keys.

13. ERROR HANDLING

This chapter describes the following

ERDEV and ERDEV$ functions

ERR and ERL functions

ERROR statement

ON ERROR GOTO statement

RESUME statement

ERDEV and ERDEV$ functions

ERDEV is an integer function which contains the error code returned by the last device to declare an error.

ERDEV$ is a string function which contains the name of the device driver which generated the error.

Syntax:

{ERDEV | ERDEV$}

ERDEV is set by the Interrupt X'24' handler, when an error within MS-DOS is detected. 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 drive name (A: , B:, C:, etc.).

For the sake of compatibility between different releases, it is advisable to perform error checking by using ERDEV rather than ERDEV$.

Example:

If a user installed a device driver "MYLPT2" caused a Printer out of paper error via INT 24, and the driver's error number for that problem was 9, ERDEV will contain the error number 9 in the lower 8 bits and the device header word attributes in the upper 8 bits, and ERDEV$ will contain "MYLPT2".

ERR and ERL functions

The ERR function returns the error code and the ERL function returns the number of the line which contains the error.

Syntax:

{ERR | ERL}

When an error handling routine is entered, the function ERR contains the error code and the function ERL contains the line number of the line in which the error was detected.

The ERR and ERL functions 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.

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 functions, neither may appear to the left of the equal sign in a LET (assignment) statement.

GW-BASIC error codes are listed in Appendix C.

To test whether an error occurred in a direct statement, use IF 65535 = ERL THEN ... Otherwise, use

IF ERR = *error-code* THEN ...

IF ERL = *linenum* THEN ...

See example on the next page.

Example:

```
LIST
10 REM RECTANGLE2
20 ON ERROR GOTO 70
30 INPUT "Length and Width";L,W
40 IF (L<0) OR (W<0) THEN ERROR 200
50 PRINT "AREA = ";L*W;" L = ";L" W = ";W
60 GOTO 30
70 IF (ERR = 200) AND (ERL = 40) THEN PRINT "L or
W<0":RESUME 30
80 ON ERROR GOTO 0
90 END
Ok
RUN
Length and Width? -2,5
L or W<0
Length and Width? 2,5
Area = 10 L = 2 W = 5
Lenth and Width? ^C
Break in 30
Ok
```

If you enter a negative value for L or W, the error handling routine is activated and the system displays:

```
L or W<0
```

Execution is resumed at statement 30 (see RESUME statement in this section). Note the use of ERR and ERL functions in the error handling routine.

ERROR statement

Simulates the occurrence of a GW-BASIC error, or generates a user-defined error.

Syntax:

ERROR *n*

where

n is an integer expression representing an error code. It must be greater than 0 and less than or equal to 255. If it is not an integer, it is rounded to the nearest integer.

ERROR can be used as a statement in a program line or as a command in direct mode.

If the value of *n* equals an error code already in use by GW-BASIC (*see Appendix C*), the ERROR is simulated, and the corresponding error message will be displayed. For example:

```
LIST
10 S = 10 : T = 5
20 ERROR  S + T
40 END
Ok
RUN
String too long in line 30
Ok
```

Or, in direct mode:

```
ERROR 15
String too long
Ok
```

If the value of *n* is greater than any error code used by GW-BASIC (*see Appendix C*), the ERROR statement will generate a user-defined error. This user-defined error code may then be handled in the error trapping routine (see the ON ERROR statement in this section).

To define your own error, use a value that is greater than any used by GW-BASIC error codes (*see Appendix C*). It is preferable to use the highest available values, in order that compatibility be maintained if more error codes are added to GW-BASIC.

Example:

```
10 ON ERROR GOTO 500
20 INPUT "WHAT IS YOUR BET";B
30 IF B > 5000 THEN ERROR 210
40 PRINT "GOOD LUCK"
50 END
500 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS
$5000"
510 IF ERL = 30 THEN RESUME 20
RUN
WHAT IS YOUR BET? 6000
HOUSE LIMIT IS $5000
WHAT IS YOUR BET? 4500
GOOD LUCK
Ok
```

In the above example, the error 210 is caused by 6000 being entered as the bet. When the error occurs, the system jumps to line 500 as instructed by line 10 that states when an error occurs GOTO 500 and display the message "HOUSE LIMIT IS $5000". Then, in line 510, system encounters the RESUME 20 which instructs the system to return to line 20 and continue.

If an ERROR statement specifies a code for which no error message has been defined, GW-BASIC responds with the message:

```
Unprintable error.
```

ON ERROR GOTO statement

Enables error trapping and specifies the first line number of a subroutine to be executed if an error occurs.

Syntax:

ON ERROR GOTO *linenum*

where

linenum is the first line number of the error trapping subroutine.

To enable error trapping, an ON ERROR GOTO *linenum* statement must be executed.

To disable error trapping, an ON ERROR GOTO 0 statement must be executed. Subsequent errors will display the associated error message and halt execution.

If error trapping is enabled and a GW-BASIC error (or a user-defined error) is found, the ON ERROR GOTO *linenum* will be executed and the corresponding routine activated. The ERL and ERR functions are usually used in IF...GOTO...ELSE or IF...THEN...ELSE statements to direct program flow within an error trapping routine.

It is recommended that the error trapping routine execute an ON ERROR GOTO 0 if an error is found for which there is no recovery action. In this case, the standard error message will be displayed and execution will stop. The RESUME statement will resume execution after the error handling routine has been entered (*see the RESUME statement in this chapter*).

If a GW-BASIC error (or a user-defined error) is found during the execution of an error trapping routine, the associated error message is displayed and execution terminates. Once an error trap takes place, all trapping is automatically disabled.

Example:

```
10 ON ERROR GOTO 1000
20 A = 20:B = 30
30 FOR I = 1 TO 20:PRINT;
40 PRINT A*B
50 END
1000 PRINT "Error number = ";ERR
1010 PRINT "Error line number = ";ERL
1100 RESUME NEXT
RUN
Error number = 26
Error line number = 30
  600
Ok
```

In this example, when the error occurs, the system goes to line 1000 and displays the error number and line number that caused the error. Then, the RESUME NEXT statement returns the system to the program line immediately following the one which caused the error.

RESUME statement

Continues program execution after an error trapping routine has been performed.

Syntax:

RESUME [0 | NEXT | *linenum*]

where

RESUME or RESUME 0 execution resumes at the statement which caused the error.

RESUME NEXT execution resumes at the statement immediately following the one which caused the error.

RESUME *linenum* execution resumes at the specified line.

Any one of the four formats shown above may be used, depending upon where execution is to resume.

A RESUME statement that is not in an error handling routine causes a RESUME without error message to be displayed.

Example:

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR = 230) AND (ERL = 90) THEN PRINT "TRY
AGAIN":RESUME 80
.
.
.
```

Notes:

14. 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 character from a communication port (ON COM(*n*) GOSUB), detection of certain keystrokes (ON KEY(*n*) GOSUB), time passage (ON TIMER(*n*) GOSUB), or emptying of the background music queue (ON PLAY(*n*) GOSUB).

Event trapping is controlled by the following statements:

Syntax 1 (to turn on trapping):

{COM(*n*) | KEY(*n*) | TIMER | PLAY} ON

Syntax 2 (to turn off trapping):

{COM(*n*) | KEY(*n*) | TIMER | PLAY} OFF

Syntax 3 (to temporarily turn off trapping):

{COM(*n*) | KEY(*n*) | TIMER | PLAY} STOP

COM(*n*) where *n* is the number (1 through 4) of the communications channel.

Typically, the COM trap routine will read an entire message from the communications port before returning. It is not recommended using the COM trap for single character messages because at high baud rates the overhead of trapping and reading for each character may allow the interrupt buffer for communications to overflow.

KEY(*n*) where *n* is a trappable key number. Trappable keys are numbered 1- 20.

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 25th line of the screen.

When GW-BASIC is in direct mode, 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. 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.

TIMER The ON TIMER(*n*) GOSUB statement (where *n* is a numeric expression representing a number of seconds since the previous midnight) can be used to perform background tasks at defined intervals.

PLAY The ON PLAY(*n*) GOSUB statement (*n* is a number of notes left in the music buffer) is used to retrieve more notes from the background music queue, to permit continuous background music during program execution.

The ON GOSUB statement

The ON GOSUB statement sets up a line number for the specified event trap.

Syntax:

ON { COM(*n*) | KEY(*n*) | TIMER(*n*) | PLAY(*n*) } GOSUB *linenum*

A *linenum* of zero disables trapping for that event.

When an event is ON and if a non-zero line number has been specified in the ON GOSUB statement, every time GW-BASIC starts a new statement it will check to see if the specified event has occurred (e.g., a communications character has been received). When an event is OFF, no trapping takes place, and the event is not remembered even if it takes place.

When an event is STOPped, no trapping takes place, but the occurrence of that event is remembered so that an immediate trap will take place when the associated event 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 once an error trap takes place, all trapping is automatically disabled. In addition, event trapping will never occur when GW-BASIC is not executing a program.

The 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(10) GOSUB 1000
```

specifies that the program go to line 1000 as soon as function key F10 is pressed. If a simple RETURN statement is executed at the end of this subroutine, program control will return to the statement following the one where the trap occurred. When the RETURN statement is executed, its corresponding GOSUB return address is cancelled.

GW-BASIC includes the RETURN *linenum* enhancement, which lets processing resume at a definable line. Normally, the program returns to the statement immediately following the GOSUB statement when the RETURN statement is encountered. However, RETURN *linenum* enables you to specify another line. If not used with care, however, this capability may cause problems. Assume, for example, that your program contains:

```
10 ON KEY(10) GOSUB 1000
20 FOR I = 1 TO 10:PRINT I:NEXT I
30 REM NEXT PROGRAM LINE
200 REM PROGRAM RESUMES HERE
1000 REM FIRST LINE OF SUBROUTINE
        :
1050 RETURN 200
```

If the function key F10 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, that was active at the time of the trap will remain active. The current FOR context will also remain active, and a FOR without NEXT error may result.

15. GRAPHICS AND SCREEN ATTRIBUTES

GW-BASIC provides, under MS-DOS, a complete range of graphic features. The design of different shapes, using a range of colors holds an exhaustive amount of possibilities. The screen can be subdivided into sections, so that several different areas can be viewed at the same time.

This chapter covers the following

Select screen attributes and change mode
Text mode
Graphics modes
Screen coordinates
Viewport
World coordinates
Drawing and coloring lines, rectangles, objects, circles, arcs, ellipses
Line clipping
CIRCLE statement
COLOR statement
DRAW statement
GET (graphics) statement
LINE statement
LOCATE (graphics) statement
PAINT statement
PMAP function
POINT function
PRESET statement
PSET statement
PUT (graphics) statement
SCREEN statement
VIEW statement
WINDOW statement

Select screen attributes and change mode

The SCREEN statement (*see the SCREEN statement in this chapter*) allows you through its first parameter ***mode*** to switch between text and graphics modes.

There are three different graphics modes you can select with the SCREEN statement:

Medium Resolution Mode (by entering SCREEN 1)

High Resolution Mode (by entering SCREEN 2)

Super Resolution Mode (by entering SCREEN 3)

They differ only in the number and size of the points displayed and in the number of colors allowed.

The SCREEN statement also allows you through the ***burst*** parameter to enable color in Text or Medium Resolution Mode (using a color TV set), and to select the active and display page in Text Mode through the ***apage*** and ***vpage*** parameters. For a standard monitor, the ***burst*** parameter has no meaning.

The SCREEN statement must precede any I/O statements to the screen, as it selects the "screen attributes" to be used by subsequent statements. The system assumes SCREEN 0,0,0,0 by default if no screen attributes are specified. This selects 80 columns Text Mode, B/W, and only one display page.

You can also use more than one SCREEN statement to define different screen attributes for different sections of your program.

You can also change from one graphics mode to another by the WIDTH statement. The WIDTH statement allows you to set the screen width (in Text Mode), to select a "text window", or change mode in one of the graphics modes.

Text mode (SCREEN 0)

In Text Mode you can display text, i.e., letters, numbers, and all special characters of the GW-BASIC character set. You can set the character foreground and background colors using the COLOR (Text) statement. This statement also allows you to create blinking, reverse image, invisible, highlighted, and underscore characters.

Characters are displayed in horizontal lines from top (line 1) to bottom (line 25). Each line has 80 columns, unless you specify 40 columns by the WIDTH statement.

The LOCATE statement positions the cursor on the active screen. The cursor column and line coordinates are returned by the POS(0) and CSRLIN functions.

Characters are usually displayed, using the PRINT or PRINT USING line 1 to 24. When an attempt is made to pass the cursor to line 25, lines 1 to 24 are moved one line up the screen.

Line 25 will usually display the function key values (*see KEY statement in Chapter 19*). To move the cursor to line 25 and display characters, use KEY OFF, then LOCATE and PRINT statements (*see the LOCATE (Text) statement in Chapter 23 for details*).

Multiple Display Page

Multiple display pages are allowed in Text Mode. Every statement that reads or writes from the screen is actually reading/writing from or to the active page. The visual page is the page that is shown on the screen, and may be different from the active page. This feature allows you to display a page, while writing another. The active and visual pages may be selected by the SCREEN statement (*see the SCREEN statement in this chapter*).

Statements, Commands and Functions

The statements, commands and functions available in Text Mode to display text are:

Statements/ Commands	Functions
CLS	CSRLIN
COLOR (Text)	POS
LOCATE (Text)	SCREEN
PRINT	SPC
PRINT USING	TAB
SCREEN	
VIEW PRINT	
WIDTH	
WRITE	

(*See Chapter 23 for descriptions of above.*)

In Text Mode, you can select the character foreground and background colors, and make characters blink.

If color hardware is installed, 16 different colors are available.

In a monochrome system, you can only use two colors (black and white), but you can also use shades of gray, underline characters, or display high-intensity characters. (*See the COLOR (Text) statement in Chapter 23 for details.*)

Graphics Modes

In each of the three graphics modes, you can still display text but you can also draw complex pictures.

To display text, you can use all the statements, commands and functions available in Text Mode, with the exception of the COLOR (Text) and LOCATE (Text) statements. You have to use the COLOR (Graphics) and LOCATE (Graphics) statements instead.

Note also that you can define a "text window", using the WIDTH or the VIEW PRINT statement. The WIDTH statement allows you to define a "vertical" text window by specifying the number of columns. The VIEW PRINT statement allows you to define a "horizontal" text window by specifying the top and bottom lines.

All points of the screen are addressable in medium, high or super resolution. A point on the screen is call a "pixel" (a contraction of "picture element"), and a line of pixels is called a "scanline".

The statements, functions and commands you can use in Graphics Mode to display pictures are:

Statements/Commands	Functions
CIRCLE	PMAP
COLOR (Medium Resolution Mode)	POINT
COLOR (High Resolution Mode)	
COLOR (Super Resolution Mode)	
DRAW	
GET (Graphics)	
LINE	
LOCATE (Graphics)	
PAINT	
PRESET	
PSET	
PUT (Graphics)	
SCREEN	
VIEW	
WINDOW	

Medium resolution mode (SCREEN 1)

In this mode, there are 320 pixels on the horizontal axis and 200 pixels on the vertical axis. These are numbered from left to right and from top to bottom; thus the upper left corner pixel is (0,0) and the lower right corner pixel is (319,199)

You can display four colors at a time if a color monitor is used, otherwise the four colors will appear as shades of gray.

Drawing Pictures

When you draw pictures on the screen using the graphics statements (PSET, PRESET, LINE, CIRCLE, PAINT or DRAW), you can specify a color number of 0, 1, 2, or 3. This selects the color from the current "palette" as defined by the COLOR statement.

If you do not specify a color number, the default is the graphics foreground specified by the COLOR statement, or 3 (if no graphics foreground is given).

The COLOR (Medium Resolution) statement allows you to specify both the color for color number 0 and the "palette" for the three remaining color numbers (1, 2, and 3).

Palette	Color 1	Color 2	Color 3
0	Green	Red	Yellow
1	Cyan	Magenta	White

If color is disabled, the use of memory is identical. The modes differ only in that the two bits of a pixel are interpreted differently by the hardware. Medium resolution B/W displays 4 shades of gray.

Displaying Characters

The size of the characters is the same as in 40-column Text Mode. The character foreground color is set by the *tforegound* parameter in the COLOR statement (that defaults to color number 3). The character background is set by the background parameter in the COLOR statement (that defaults to color number 0, i.e., black). If color is disabled, the character foreground will be 1 (white) and the character background 0 (black).

High resolution mode (SCREEN 2)

In this mode, there are 640 pixels on the horizontal axis and 200 pixels on the vertical axis. These are numbered from left to right and top to bottom, thus the upper left pixel is (0,0) and the lower right is (639,199).

There are only two colors: black (color number 0) and white (color number 1).

Drawing Pictures

When you draw pictures using the graphics statements, you can still specify a color number 0, 1, 2, or 3.

A color number of 0 indicates black and a color number of 1 white. A color number of 2 is treated as 0, and 3 is treated as 1.

If you do not specify a color number, the default is the graphics foreground specified by the COLOR statement, or 1 if no graphics foreground is given.

The COLOR statement allows you to specify the graphics foreground and background colors and, optionally, an XOR operation between the pixels on the screen and the pixels of your graphics picture or text.

Displaying Characters

The size of the characters is the same as in 80-column Text Mode.

The character foreground color is 1 (white) and the character background color is 0 (black).

Super resolution mode (SCREEN 3)

In this mode, there are 640 pixels on the horizontal axis and 400 pixels on the vertical axis. These are numbered from left to right and top to bottom, thus the upper left pixel is (0,0) and the lower right is (639,399).

There are only two colors: black (color number 0) and white (color number 1).

Drawing Pictures

When you draw pictures using the graphics statements, you can still specify a color number 0, 1, 2, or 3.

A color number of 0 indicates black and a color number of 1 white. A color number of 2 is treated as 0, and 3 is treated as 1.

If you do not specify a color number, the default is the graphics foreground specified by the COLOR statement, or 1 if no graphics foreground is given.

The COLOR (Super Resolution) statement allows you to specify the graphics foreground and background colors and, optionally, an OR operation between the pixels on the screen and the pixels of your graphics picture or text. The COLOR statement also allows you to specify "inverse video", when you display characters.

Displaying Characters

The size of the characters is the same as in 80-column Text Mode.

The character foreground color is 1 (white) and the character background color is 0 (black), unless you specify "inverse video" by the COLOR statement.

Screen coordinates

Graphics images are positioned on the screen in accordance with screen coordinates. These coordinates comprise two parameters generally referred to as x and y, where x defines the horizontal screen position and y defines the vertical screen position. Screen coordinates may be of two types:

absolute coordinates

relative coordinates

Whereas absolute coordinates refer to the actual position of a pixel on the screen, relative coordinates indicate the position of a pixel relative to the coordinates of the last pixel referenced. The x and y relative coordinates of the last pixel referenced (known as the "current point").

The following example illustrates the use of both types of coordinates:

```
10 SCREEN 1
20 PSET (100,50)          'absolute coordinates
30 PSET STEP (10,-5)      'relative coordinates
```

This program example illuminates two pixels on the screen. The first at coordinates (100,50) and the second at actual coordinates (110,45).

Viewport

The VIEW statement allows the definition of subsets of the viewing surface. These are called "viewports". Onto these the contents of a window are mapped. Initially RUN or VIEW, with no arguments, define the whole screen as a viewport.

World coordinates

The WINDOW statement allows you to draw lines, graphs, or objects in space not bounded by the physical limits of the screen. This is done by using programmer-defined coordinates called "world coordinates".

A world coordinate is any valid single precision floating point number pair. GW-BASIC then converts world coordinate pairs into the appropriate physical coordinate pairs for subsequent display within screen space. To make this transformation from world space to the physical space of the viewing surface (screen), you must know what portion of the unbounded (floating point) world coordinate space contains the information you want to be displayed.

You can also increase or decrease the size of the image to be displayed and clip part of the image by changing the logical dimensions of the current viewport, via the WINDOW statement.

Notes:

Displaying points

The most elementary graphics function is that of illuminating the position of a single point (or pixel) in a specified color. This is achieved using the PSET and PRESET statements. The POINT function allows you to know the color number of a specified pixel.

Drawing and coloring lines, rectangles, objects, circles, arcs, ellipses

The LINE statement permits the drawing of lines or rectangles. The DRAW statement, governed by "movement commands" such as up, down, left, and right, lets you draw any object. Circles, arcs and ellipses can be drawn using the CIRCLE statement, and the PAINT statement allows any object to be filled with color(s).

Line clipping

The graphics statements CIRCLE, LINE, PAINT, POINT, PSET, PRESET, and WINDOW use "line clipping". This simply means that lines which cross the screen or viewport are "clipped" at the boundaries of the viewing area. Only the points plotted within the screen or viewport are visible.

CIRCLE statement

Draws a circle (or an ellipse) with the specified center and radius.

Syntax:

CIRCLE [STEP](*x,y*), *radius* [,*color* [,*start,end* [,*aspect*]]]

where

x,y are numeric expressions, specifying the coordinates of the center of the circle (or ellipse). They may be given in absolute form, or in relative form if STEP is included.

radius is a numeric expression returning a positive integer value. It is the radius of the circle, or the major axis or the ellipse. It is measured in pixels in the horizontal direction if *aspect* < 1 and in vertical direction if *aspect* > 1.

color is an integer expression in the range 0 to 3. It is the color number of the circumference (or ellipse). See the COLOR graphics statement (current screen mode) for details.

start,end are numeric expressions specifying angles in radians. The range is from -2*PI to 2*PI, where PI=3.141593. They specify where the drawing of the circle (or ellipse) will begin and end.

aspect is a numeric expression returning a positive value. Due to the non-uniform distribution of the pixels on the screen, you must specify a value of *aspect* to draw a true circle with different monitors. The default value of *aspect* is 5/6 in medium and super resolution and 5/12 in high resolution. This value produces a circle with the standard monitor.

Drawing Circles and Ellipses

The CIRCLE statement draws circles if you do not specify the *aspect* parameter, and ellipses if you specify a value of *aspect* different from the default value (5/6 in medium and super resolution, and 5/12 in high resolution).

The *aspect* may be thought of as a fraction, with a separate numerator and denominator. The numerator tells GW-BASIC how many rows the CIRCLE statement should consider equivalent to the number of columns specified by the denominator.

If *aspect* is less than one, then radius is measured in pixels in the horizontal direction, i.e., it is the x-radius. In this case, GW-BASIC draws ellipses with the same width, and varies the height.

If *aspect* is greater than one, the y-radius is given, and GW-BASIC draws ellipses with the same height and varies the width. For example:

```
10 SCREEN 1
    :
100 CIRCLE (100,150),50,,,,5/18
```

will draw a horizontal ellipse with an x-radius of 50 pixels.

Drawing Arcs

The CIRCLE statement can simply draw part of a circle (or ellipse), i.e., an arc. To draw an arc, you must enter the *start* and *end* parameters. They specify the first and the second arc endpoint in radians.

The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise. For example, the following statement specifies just a quarter of a circle:

```
10 CIRCLE (100,150),50,1,0,3.141593/2
```

The angles must be measured in radians. If you have the angles in degrees, you must convert them to radians before executing the CIRCLE statement. To convert from degrees to radians, multiple by 0.0174532.

Drawing Rays

The CIRCLE statement can draw a ray from the center of the arc to either arc endpoint. A negative endpoint generates a ray to that endpoint. The endpoint -0 is not treated as a negative endpoint. To circumvent this limitation, use a small negative number (e.g., -0.001 instead of -0). When both endpoints are negative, both rays are drawn. The minus sign does not affect the arc itself, i.e., the angles will be treated as if they were positive. Note that this is different from adding 2*PI (where PI is 3.141593). The start angle may be greater or less than the end angle. For example:

```
10 SCREEN 1
   :
100 CIRCLE (100,150),50,1,-0.001,-3.141593/2
```

will draw a quarter of a circle delimited by two rays.

Last Point Referenced

The last point referenced after a circle (or ellipse) has been drawn is the center of the circle (or ellipse).

Clipping

Points that are off the screen or the graphics viewport are not drawn by the CIRCLE statement.

STEP option

Coordinates can be shown as absolutes, or the STEP option can be used to reference a point relative to the most recent point used.

For example, if the most recent point referenced was 100,50, then: either

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

or

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

will draw a circle at 100,200 with radius 50. The first example uses absolute notation; the second uses relative notation.

Example:

The following example draws three intersecting circles and colors the area of intersection.

```
5 SCREEN 1
10 COLOR 0,0,3,0
20 CLS
30 CIRCLE (100,120),90
40 CIRCLE (150,130),120
50 CIRCLE (250,120),100
60 PAINT (180,120)
```

COLOR statement (medium resolution mode)

Defines the palette background and foreground colors. In addition, the default graphics foreground and background colors, and the text foreground color can be defined.

Syntax:

COLOR [*backgrnd*][, [*palette*][, [*gforegrnd*][, [*gbackgrnd*][, *tforegrnd*]]]

where

backgrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 31. Values greater than 15 are taken modulo 15. It selects an actual color for the character background. This is also the actual color for color number 0 that may be specified in a graphics statement (palette background color). This parameter also specifies the foreground intensity (the intensity of pixels with values 1, 2, or 3). If the parameter is >15, high intensity is selected for foreground colors. It defaults to 0 (black) if unspecified.

palette is a numeric expression rounded to the nearest integer. It must be in the range 0 through 255. This selects one of two palettes for the color numbers 1, 2, and 3 that may be specified in a graphics statement.

Palette	Color 1	Color 2	Color 3
0	Green	Red	Yellow
1	Cyan	Magenta	White

Palette 0 is selected, when ***palette*** is an even number, whereas palette 1 is selected, if ***palette*** is an odd number.

gforegrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics foreground which is the default color number when no color parameter is specified in a graphics statement. If *gforegrnd* is omitted, 3 is assumed. The graphics foreground is always set to the default value (3) when the SCREEN statement selects new screen mode 1.

Any value greater than 3 will cause the bits for each pixel to be XOR's with screen memory.

gbackgrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics background; i.e., the color used when a graphics viewport is cleared with CLS 1, or with just CLS (*see the description of the CLS statement in Chapter 23*). The default value is 0, which is always selected when a new screen mode is selected. Note that the CLS 1 statement will only clear a viewport if it has been explicitly defined with a VIEW statement.

tforegrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 7. This is the bit pattern used for the foreground of characters written on the screen. It defaults to 3. Any value greater than 3 will cause the character to be XOR'd with the screen bitmap.

When you enter a CIRCLE, DRAW, LINE, PAINT, PRESET, or PSET statement in your program, you can specify a color number of 0, 1, 2, or 3. This parameter selects the color from the current *palette* as defined by the COLOR statement.

If you do not specify a color number, the default is the graphics foreground (i.e., the value of *gforegrnd*, or 3 if *gforegrnd* has not been specified).

When you display text, the character foreground will be set by *tforegrnd* that defaults to color number 3, and the character background will be set by *backgrnd* that defaults to 0, i.e., black.

Any parameter may be omitted in the COLOR statement. Omitted parameters assume the old value.

Upon initialization the default values are:

backgrnd = 0
palette = 1
gforegrnd = 3
gbackgrnd = 0
tforegrnd = 3

That is, if no COLOR statement exists in your program, the system assumes:

```
COLOR 0,1,3,0,3
```

The use of memory for color and monochrome in medium resolution mode is identical. The modes differ only in that the two bits of a pixel are interpreted differently by the hardware: B/W medium resolution displays four shades of gray.

Examples:

```
10 SCREEN 1,0
20 COLOR 10,1,2,0
```

Sets the palette background to light green, selects palette 1 (cyan, magenta, white), sets the graphics foreground to magenta, and graphics background to black.

```
100 COLOR,0
```

The background stays light green and palette 0 is selected.

COLOR statement (high resolution mode)

Defines the (default) graphics foreground, the graphics background and the text foreground colors.

Syntax:

COLOR [*gforegrnd*][, [*gbackgrnd*][, *tforegrnd*]]

where

gforegrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics foreground which is the default color number for graphics statements. If *gforegrnd* is omitted, 1 (white) is assumed.

Values of 0 (black) or 1 (white) represent actual colors; larger values specify an XOR attribute.

gbackgrnd is a numeric expression rounded to the nearest integer, whose value may be 0 or 1. This specifies the background color used for clearing graphics viewports. This defaults to 0 (black).

tforegrnd is a numeric expression, rounded to the nearest integer. A non-zero value of this specifies that characters are XOR's with the screen bitmap. Inverse video for text display is not implemented in this mode.

When you enter a CIRCLE, DRAW, LINE, PAINT, PRESET, or PSET statement in your program, you can specify a color number of 0, 1, 2, or 3. A color number of 0 indicates black and a color number of 1 white. A color number of 2 will be treated as 0, and a color number of 3 will be treated as 1.

If you do not specify a color number, the default is the graphics foreground (i.e., the value of *gforegrnd* or 1 if *gforegrnd* has not been specified).

When you display text, the character foreground will be 1 (white) and the background 0 (black).

You can also specify an XOR operation between the pixels on the screen and the pixels of your graphics picture or your text, by use of the COLOR statement.

Any parameter in the COLOR statement may be omitted. Omitted parameters assume the old values. Upon initialization default values are:

gforegrnd = 1 (white)
gbackgrnd = 0 (black)
tforegrnd = 0 (no XOR attribute for text)

That is, if no COLOR statement exists in your program, the system assumes:

```
COLOR 1,0,0
```

Example:

```
10 SCREEN 2
20 COLOR 0,1,0
```

This selects a black graphics foreground on a white background and no XOR attribute for text.

COLOR statement (super resolution mode)

Defines the (default) graphics foreground, the graphics background and the text foreground colors.

Syntax:

COLOR [*gforegrnd*][, [*gbackgrnd*][, *tforegrnd*]]

where

gforegrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics foreground which is the default color number for graphics statements. If ***gforegrnd*** is omitted, 1 (white) is assumed.

Values of 0 (black) or 1 (white) represent actual colors; larger values specify an XOR attribute.

gbackgrnd is a numeric expression rounded to the nearest integer, whose value may be 0 or 1. This specifies the background color used for clearing graphics viewports. This defaults to 0 (black).

tforegrnd is a numeric expression, rounded to the nearest integer. A value of 1 will result in "normal video"---characters written with white 1 pixels against a background of black 0 pixels.

A value of 0 will result in "inverse video"---characters written with black 1 pixels against a background of white 0 pixels.

Larger values will cause the characters to be XOR'd with the screen bitmap.

When you enter a CIRCLE, DRAW, LINE, PAINT, PRESET, or PSET statement in your program, you can specify a color number of 0, 1, 2, or 3. A color number of 0 indicates black and a color number of 1 white. A color number of 2 will be treated as 0, and a color number of 3 will be treated as 1.

If you do not specify a color number, the default is the graphics foreground (i.e., the value of *gforegrnd* or 1 if *gforegrnd* has not been specified).

When you display text, the character foreground will be 1 (white) and the character background 0 (black), unless you specify the "inverse video" by the COLOR statement (with a 1 value of *tforegrnd*).

You can also specify an XOR operation between the pixels on the screen and the pixels of your graphics picture or your text, by use of the COLOR statement.

Any parameter in the COLOR statement may be omitted. Omitted parameters assume the old values. Upon initialization default values are:

gforegrnd = 1 (white)
gbackgrnd = 0 (black)
tforegrnd = 0 (normal video, no XOR)

That is, if no COLOR statement exists in your program, the system assumes:

```
COLOR 1,0,1
```

Example:

```
10 SCREEN 3
20 COLOR 0,1,0
```

This selects a black graphics foreground on a white background, the inverse video, and no XOR.

DRAW statement

Draws a picture as specified by a sequence of single character Graphics Macro Language commands.

Syntax:

DRAW *stringexp*

where

stringexp is a string expression defining the sequence of Graphics Macro Language commands that will draw the object.

The DRAW statement combines most of the capabilities of the other graphic statements into an easy-to-use object definition language called "Graphics Macro Language". A GML command is a single character (e.g., U,D,L,R,E,F,G,H,M,B,N,A,C, S,X,P or a pair of characters (TA) with the string *stringexp*, optionally followed by one or more arguments (e.g., *n,m,x,y*).

In all GML commands, numeric arguments can be constants like "327" or =*numvar*; where *numvar* is the name of a numeric variable. The semicolon is necessary if you enter a variable this way or if you use the X command; otherwise, you can omit the semicolon between commands.

GML Movement Commands

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

Command	Action
U[*n*]	Move up. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
D[*n*]	Move down. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
L[*n*]	Move left. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
R[*n*]	Move right. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
E[*n*]	Move diagonally up and right. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
F[*n*]	Move diagonally down and right. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
G[*n*]	Move diagonally down and left. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.
H[*n*]	Move diagonally up and left. The number of points moved is *n* * scale factor (set by the S command below). If *n* is omitted, 1 is supplied.

GML Movement Commands - cont'd

Command	Action
M *x,y*	Move absolute or relative. If *x* is preceded by a plus (+) or minus (-), *x* and *y* are added to the current graphics position, and connected with the current position by a line (move relative). Otherwise, a line is drawn to point *x,y* from the current position (move absolute).
B	Move without plotting any points. B may precede any of the above mentioned movement commands.
N	Move but return to original position when finished. N may precede any of the above mentioned movement commands.

More GML Commands

Command	Action
A *n*	Set angle *n*. *n* may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180 and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with 0 or 180 degrees on a standard monitor.
TA *n*	Rotate angle *n*. *n* is equivalent to degrees in the range -360 to 360. If *n* is positive, rotation is counterclockwise. If *n* is negative, rotation is clockwise. If *n* is outside the specified range, an Illegal function call error occurs.

More GML Commands - cont'd

Command	Action
C *n*	Set color *n* (from 0 to 3 in medium resolution, and 0 or 1 in high or super resolution).
S *k*	Set scale factor. *k* may range from 1 to 255. The scale factor is defined as *k*/4. The scale factor, multipled by the distances given with U,D,L,R,E,F,G,H or relative M commands gives the actual distance travelled (in pixels). If the S command is omitted, a scale factor of 1 (i.e., k=4) is assumed.
X *stringexp*	Execute substring. This powerful command allows you to execute a second substring from a string much like GOSUB in BASIC. You can have one string execute another, which executes a third and so on. Spaces are ignored in *stringexp*.
P *n, m*	*n* is the color chosen to paint the interior of the closed figure and *m* is the border color. You must specify both parameters or an error will occur. Both parameters can range from 0 to 3 in medium resolution and from 0 to 1 in high or super resolution mode.

Examples:

To draw a square:

```
10 SCREEN 1
20 A = 40
30 DRAW "U = A; R = A; D = A; L = A;"
```

To draw a box:

```
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
```

To draw a box and color the interior:

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

To draw some spokes:

```
10 FOR D = 0 TO 360
20 DRAW "TA = D;NU50"
30 NEXT D
```

GET (graphics) statement

Reads graphic images from the screen.

Syntax:

GET [STEP] (*x1,y1*) - [STEP] (*x2,y2*), *array*

where

(x1,y1)-(x2,y2) define a rectangular area on the display screen. x1,y1 are the upper left and x2,y2 the lower right coordinates. They may be given in absolute or relative form (if the STEP option is used).

array is the name assigned to the array that will hold the image, bounded by the specified rectangle.

Characteristics

The GET statement should be used in conjunction with the PUT statement. The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array.

The GET and PUT statements are used to transfer graphic images to and from the screen. GET and PUT permit animation and high-speed object motion, (*see the PUT (graphics) statement in this chapter*).

The array must be numeric, but may be any precision. It must be dimensioned large enough to hold the entire image. Unless the array is type integer, the contents of the array after a GET will be meaningless when interpreted directly (see the next page).

Array Dimensions

The storage format in the array is as follows:

2 bytes giving x dimension in BITS
2 bytes giving y dimension in BITS
the array data itself

The data for each row of pixels is left-justified on a byte boundary, so if there are less than a multiple of 8 bits stored, the rest of the byte will be filled out with zeros. The required array size in bytes is:

```
4 + INT((x * bitsperpixel + 7) /8) * y
```

where bitsperpixel is 2 in medium resolution, and 1 in high and super resolution.

The bytes per element of an array are:

2 for integer
4 for single precision
8 for double precision

For example, you want to GET a 10 by 12 image into an integer array, in medium resolution mode. The number of bytes required is 4 + INT((10*2+7)/8)*12 or 40 bytes. So, you would need an integer array with at least 20 elements.

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The x dimension is in element 0 of the array, and the y dimension is found in element 1. It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

Example:

```
10 CLS : SCREEN 3: PSET(20,20)
20 X$ = "R20D20L20U20":DRAW X$
30 DIM BOX%(64) : GET(20,20)-(40,40), BOX%
40 CLS : PUT(100,100),BOX%
```

LINE statement

Draws either a line or a rectangle, or a filled rectangle.

Syntax:

LINE [[STEP] (*x1,y1*)] - [STEP] (*x2,y2*)[, [*color*] [,B[F]][, *style*]]

where

(x1,y1), (x2,y2) represent absolute coordinates, or relative coordinates if STEP is included. If (*x1,y1*) is omitted, the last referenced point is assumed.

color is the color number specifying the color in which the line or rectangle will be drawn (in the range 0 to 3). *Refer to the COLOR graphics statement for the current screen mode for details.*

B represents a rectangle.

F represents a rectangle to be filled (with color).

style is an optional parameter that may be defined by the user to produce varying line "styles", i.e., varieties of dotted lines.

The following example draws a line from the last point referenced to the point specified (*x2,y2*). Since no color is specified, the default color is the foreground color.

```
LINE -(X2,Y2)
```

The examples below specify start and end points in absolute coordinates.

```
LINE (10,10)-(319,199)  'draws a diagonal line down the screen
LINE (10,100)-(319,100) 'draws a horizontal line across the screen
```

You can specify the color in which the line is drawn:

```
LINE (15,15)-(25,25),2 'draws a line in color 2
```

The B parameter is used to draw a rectangle (box) in the foreground, where the points (*x1,y1*) and (*x2,y2*) represent the opposite corners. In the following example, no color number is specified:

```
LINE (10,10)-(100,100),,B 'draws a box in foregrnd
```

color may be included as follows:

```
LINE (10,10)-(100,100),2,BF 'filled box color 2
```

The B parameter facilitates the drawing of rectangles, which would otherwise require the following lengthy programming format:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

BF fills the interior of the rectangle with the selected color.

Out-of-range coordinates are not visible on the screen. This is called "line clipping".

If the relative form is used for the second coordinate, it is relative to the first coordinate. For example:

```
LINE (50,50) - STEP (15,-13)
```

draws a line from (50,50) to (65,37).

Line Styling

LINE supports the additional argument *style*. *style* is a 16-bit integer mask used when putting down pixels on the screen. This is called "Line Styling".

Each time LINE plots a point on the screen, it will use the current circulating bit in *style*. If that bit is 0, no point is plotted. If the bit is a 1, then a point is plotted, After each point, the next bit position in *style* is selected.

Since a 0 bit in *style* does not clear out the old contents, you may wish to draw a background line before a *style*d line in order to force a known background.

style is used for normal lines and boxes, but has no affect on filled boxes. For example:

```
LINE (0,0)-(160,100),2,,&HFF00
```

Draws a dashed line from the upper left hand corner to the screen center, assuming a screen 320 pixels wide by 200 pixels high.

LOCATE (graphics) statement

Moves the cursor to the specified position. LOCATE may also turn the user cursor on and off and define the shape and blinkrate of the cursor.

Syntax 1:

LOCATE [*row*][, [*column*][, [*rate*][, [*start*][, *stop*]]]]

Syntax 2:

LOCATE [*row*][, [*column*][, [*rate*][, *line*, *map*]]]

where

The value of the fourth parameter determines which of the above applies. See below.

row is the screen line number. A numeric expression returning an unsigned integer in the range 1 to 25.

column is the screen column number. A numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon screen width.

rate is an integer expression in the range 0 to 10, which determines the state of the user cursor. (*For definition of "user cursor", see the paragraphs after the syntax definitions.*)

0 Turn the user cursor off (cursor will not appear except during INPUT statements and in direct mode).

1 Turn the user cursor on. It will not blink.

2-10 Blink the user cursor with a period of ***rate*** units of 1/18.75 seconds.

start is the cursor starting scanline. It must be an integer expression in the range 0 to 15 or 32 to 47. If *start* is in the range 0 to 15, the shapes of both the user and the overwrite cursor will be programmed. If *start* is in the range 32 to 47, only the user cursor shape will be programmed. If *start* is in the range 32 to 47, it is taken modulo 16.

stop is the cursor stop scanline. It must be a numeric expression in the range 0 to 15.

line If the value of *line* is between 50 and 50+M, byte number *line* - 50 of the cursor bitmap for the overwrite cursor is set to *map*. If the value is between 100 and 100+M, then byte number *line* 100 of the cursor bitmap for the user cursor is set to *map*. The value of M is 15 for medium resolution mode, 7 for high resolution mode, and 15 for super resolution mode.

map If *line* and *map* are specified, this value replaces the bitmap for scanline *line* of the cursor specified by *rate*. The cursor bitmap is a byte array which is XOR'd with the screen to display the cursor. For medium resolution mode, each scanline of the cursor is represented by 2 bytes; the low-order byte of each scanline is the left one on the screen. For other modes, there is one byte per scanline. The size of the array is the number of scanlines per row of text times the number of bytes per cursor scanline: this is 8 for high resolution mode and 16 for other modes. Cursor bitmaps are kept separately for screen modes 1, 2 and 3. The cursor state for each mode is restored if another screen mode is selected, and the original mode is reselected. Likewise, separate bitmaps are kept for the insert, overwrite and user cursors.

GW-BASIC includes a blinking cursor for graphics mode. The maximum height of this cursor is 8 in modes 1 and 2, and 16 in mode 3. Cursor scanlines are numbered starting with 0 for the top scanline.

There are three different cursors in graphics mode as well as in text mode (*see the LOCATE (text) statement in Chapter 23*).

The insert mode cursor will always be a rapidly-blinking small triangle at the lower left of the character cell.

The overwrite mode cursor is initially an underline, which blinks somewhat more slowly.

The user cursor is initially disabled, but its shape array is loaded with OFFH bytes, so that it can easily be made to be any underline or block shape.

The user and overwrite cursors will be programmable in shape. The blinkrate of the user cursor is programmable, but the blinkrates of the overwrite and insert cursors are fixed.

LOCATE,,0 will disable only the user cursor. Also, execution of any graphics statement (LINE, PSET, etc.) will disable the user cursor so that the cursor will be removed from screen memory while the graphics statement is executed. In this case, the user cursor must be explicitly turned on if it is used later on.

The overwrite cursor will always appear whenever an INPUT statement is being executed, or when GW-BASIC is in direct mode. At any other time, only the user cursor may appear.

See examples on next page.

Examples:

```
10 LOCATE 5,1,4,5,7
```

Moves to line 5, column 1, turns the overwrite cursor on with a blinkrate of 4/18.75 seconds and sets the height of the cursor to 3. The cursor scanlines are initialized to 0FFH, so 3 scanlines will appear unless the user has changed the bitmap).

```
100 LOCATE ,,,51,&H82
110 LOCATE ,,,103,&H01
```

These statements set bytes in the bitmaps for the overwrite and user cursor, respectively. Statement 100 sets byte 1 of the overwrite cursor to Hex 82; statement 110 sets byte 3 of the user cursor to Hex 01.

```
50 FOR X = 0 TO 7
60 LOCATE ,,,50 + X,0
70 NEXT X
```

This example clears the bitmap for the overwrite cursor for screen mode 2 (use FOR X = 0 TO 15 for modes 1 and 3). This is the only way of completely disabling the overwrite cursor for graphics modes.

PAINT statement

Fills a graphics area with a color or a pattern specified.

Syntax:

PAINT [STEP] (*x,y*) [, [*paint*][, [*border*][, *backgrnd*]]]

where

x,y are the coordinates, either absolute or relative, of a point where painting is to begin. Painting should always start on a non-border point. If painting starts within the border, the bordered figure is painted. If painting starts outside the bordered figure, the background is painted.

paint is a numeric or string expression. If it is a numeric expression in the range 0 to 3, it represents the color number to be used for painting (*see the COLOR (graphics) statement for the current screen mode, for details*). If it is a string expression, PAINT will execute "tiling". Tiling is described in detail later on in this chapter. If *paint* is omitted, the default foreground color is used for painting.

border is an integer expression in the range 0 to 3. It identifies the border color of the figure to be filled. When the border color is encountered, painting of the current line will stop. If border is omitted, the *paint* value will be used.

backgrnd is a string expression returning one character, used in "paint tiling".

Characteristics

The PAINT statement will fill in an arbitrary figure, with edges of *border* color with the specified *paint* color. The *paint* color will default to the graphics foreground color if not given, and the *border* color defaults to the *paint* color.

For example, in the medium resolution, you can fill in a circle of color 1 with color 2. Visually, this could mean a red ball with a green border (if palette 0 has been selected).

Since there are only two colors in high resolution and super resolution modes, this means "whiting out" an area until white is encountered, or "blacking out" an area until black is encountered.

PAINT must start on a non-border point; otherwise, PAINT will have no effect.

If the specified point already has the color boundary, PAINT will have no effect.

PAINTing is complete when a line is painted without changing the color of any pixel; i.e., the entire line is equal to the *paint* color.

PAINT can fill any figure, but PAINTing complex figures may result in an Out of memory error. If this happens, the CLEAR statement may be given to increase the amount of stack space available.

The PAINT statement permits coordinates outside the screen or viewport.

Tiling

Tiling is the design of a PAINT pattern that is 8 bits wide and up to 64 bytes long. Each byte in the Tile-String masks 8 bits along the x-axis when putting down points.

Use the syntax:

PAINT (*x,y*), CHR$(*n*)[+ CHR$(*n*)]...

where *n* is a number between 0 and 255, or between &H00 and &HFF in hexadecimal. It will be represented in binary across the x-axis of the "tile". Each CHR$(*n*) up to 64 will generate an image not of the assigned character, but of the bit arrangement of the code for that character. For example, the decimal number 85 is binary "01010101"; the graphic image line on a black and white screen generated by CHR$(85) is an eight pixel line, with even numbered points turned white, and odd ones black. That is, each bit containing a "1" will set the associated pixel on and each bit filled with a "0" will set the associated bit off, on a black and white system. The ASCII character CHR$(85), which is "U", is not displayed in this case.

The structure of the tile string (which is replicated uniformly over the entire screen) will then be:

	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)

Each byte of the tile string is rotated as required to align along the y-axis such that:

type-byte-mask = *y* MOD *tile-length*

In high and super resolution modes, the screen can be painted with 'x's by the following statement.

```
PAINT (320,100), CHR$(129) + CHR$(66) +
CHR$(36) + CHR$(24) + CHR$(24) + CHR$(36) +
CHR$(66) + CHR$(129)
```

or, using hexadecimal numbers for the arguments of CHR$:

```
PAINT (320,100), CHR$(&H81) + CHR$(&H42) +
CHR$(&H24) + CHR$(&H18) + CHR$(&H18) +
CHR$(&H24) + CHR$(&H42) + CHR$(&H81)
```

This pattern appears on the screen as:

```
          x increases →
0,0 1   0   0   0   0   0   0   1   CHR$(&H81) Tile byte 1
0,1 0   1   0   0   0   0   1   0   CHR$(&H42) Tile byte 2
0,2 0   0   1   0   0   1   0   0   CHR$(&H24) Tile byte 3
0,3 0   0   0   1   1   0   0   0   CHR$(&H18) Tile byte 4
0,4 0   0   0   1   1   0   0   0   CHR$(&H18) Tile byte 5
0,5 0   0   1   0   0   1   0   0   CHR$(&H24) Tile byte 6
0,6 0   1   0   0   0   0   1   0   CHR$(&H42) Tile byte 7
0,7 1   0   0   0   0   0   0   1   CHR$(&H81) Tile byte 8
```

Since there are 2 bits per pixel in medium resolution mode, each byte of the tile pattern only describes 4 pixels. In this case, every 2 bits of the tile byte describes 1 of the 4 possible colors associated with each of the 4 pixels to be put down.

If *backgrnd* color is omitted, the default value is CHR$(0). When supplied, *backgrnd* specifies the "background tile" pattern or color byte to skip when checking for boundary termination.

It may occasionally be necessary to tile paint over an 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 this parameter. Paint would stop as soon as the first red pixel was drawn. Specifying red [CHR$(&HAA)] as the *backgrnd* color, allows the red line to be drawn on the red background.

You cannot specify more than two consecutive bytes in the tile string that match the background color. Specifying more than two will result in an Illegal function call error.

Example:

```
10 SCREEN 1
20 COLOR 0,0,1,0
30 CLS
40 CIRCLE (256,128),130,2
50 PAINT (256,128),1,2
60 LINE (251,123)-STEP (10,10),0,BF
```

Statement 10 selects medium resolution mode. Statement 20 selects black for color number 0, palette 0 (green, red, yellow), green as graphics foreground, black as graphics background. Statement 30 clears the screen with the background color (in this case black). Statement 40 draws a red circumference with a radius of 130 which center is (256,128). Statement 50 paints the circle green. Statement 60 draws a black filled in box in the middle of the circle.

PMAP function

Converts physical coordinates to world coordinates or vice versa.

Syntax:

PMAP(*coordinate, n*)

where

coordinate is a numeric expression specifying either the x coordinate or the y coordinate of the point to be mapped according to the value of *n*.

n is an integer number in the range 0 to 3:

0 assumes the *coordinate* value to be the world x coordinate, and maps it to the physical x coordinate.

1 assumes the *coordinate* value to be the world y coordinate, and maps it to the physical y coordinate.

2 assumes the *coordinate* value to be the physical x coordinate, and maps it to the world x coordinate.

3 assumes the *coordinate* value to be the physical y coordinate, and maps it to the world y coordinate.

The four PMAP functions allow you to find equivalent point locations between the world coordinates created with the WINDOW statement and the physical coordinate system of the screen or viewport as defined by the VIEW statement.

See examples on next page.

Examples:

If a user had defined a WINDOW SCREEN (80,100) - (200,200) then the upper left coordinate of the window would be (80,100) and the lower right would be (200,200). The physical or screen coordinates may be (0,0) in the upper left hand corner and (639,199) in the lower right. Then:

```
X = PMAP(80,0)
```

would return the physical x coordinate of the world x coordinate 80:

```
0
```

The PMAP function in the statement:

```
Y = PMAP(200,1)
```

would return the physical y coordinate of the world y coordinate 200:

```
199
```

The PMAP function in the statement:

```
X = PMAP(619,2)
```

would return the world x coordinate that corresponds to the physical x coordinate 619:

```
199
```

The PMAP function in the statement:

```
Y = PMAP(100,3)
```

would return the world y coordinate that corresponds to the physical y coordinate 100:

```
140
```

POINT function

Returns the color number of a pixel on the screen, if two arguments (*x,y*) are given, or the current graphics coordinate if one argument (*n*) is given.

Syntax 1:

POINT(*x,y*)

Syntax 2:

POINT(*n*)

where

(x,y) are the absolute coordinates of the selected pixel. If the point is out of range, the value -1 is returned.

n may have the values 0, 1, 2, or 3 as follows:

0 returns the current physical x coordinate.

1 returns the current physical y coordinate.

2 returns the current world x coordinate if a WINDOW statement has been used; otherwise, returns the same value as the POINT(0) function.

3 returns the current world y coordinate if WINDOW is active; otherwise, returns the same value as the POINT(1) function.

Syntax 1:

v1 = POINT(*x,y*)

returns the color number of the specified pixel into the integer variable *v1*.

Syntax 2:

v2 = POINT(*n*)

returns the specified coordinate of the current point into the single (or double) precision variable *v2*.

Examples:

```
10 SCREEN 0,0
20 FOR K = 0 TO 3
30 PSET (10,10),K
40 IF POINT(10,10) <> K THEN PRINT "Broken
Basic!"
50 NEXT
```

```
10 SCREEN 2
20 IF POINT(I,I) <> 0 THEN PRESET (I,I) ELSE PSET
(I,I)
30 'Invert current state of POINT(I,I)
40 PSET (I,I),1-POINT(I,I)
50 'Another way to invert a point, if the system is
B/W
```

```
10 SCREEN 1
20 LET C = 3
30 PSET(10,10),C
40 IF POINT(10,10) = C THEN PRINT "This point is
color ";C
```

PRESET statement

Draws a point at the specified position on the screen.

Syntax:

PRESET [STEP] (*x,y*) [, *color*]

where

x,y are the coordinates of the point to be drawn. They may be in absolute or relative form (if the STEP option is included).

color is the color number to be used, in the range 0 to 3. (*See the COLOR (graphics) statement for the current screen mode, for details).* If no *color* parameter is given, the graphics background color is selected. If *color* is included, PRESET is identical to PSET.

If an out of range coordinate is given to PSET or PRESET, no action is taken nor is an error given. If a color greater than 3 is given, this will result in an Illegal function call.

Example:

PRESET(*x,y*)

is identical to:

PSET(*x,y*),0

assuming that the graphics background color is 0 (black). *See the COLOR (graphics) statement for the current mode.*

PSET statement

Illuminates a pixel at a specified position on the screen.

Syntax:

PSET [STEP] (*x,y*) [, *color*]

where

x,y are the coordinates of the pixel to be drawn. You may specify them either in absolute or relative form. If relative, the STEP option must be present.

color is the color number chosen for the point displayed. This parameter is optional; by default the graphics foreground color is taken. (*See the COLOR (graphics) statement for the current screen mode, for details.*)

Coordinates can be specified in one of two forms:

PSET STEP (*x-offset,y-offset*) or
PSET (*absolute-x,absolute-y*)

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced.

Examples are:

PSET(10,10)	absolute form
PSET STEP (10,0)	offset 10 in x and 0 in y
PSET (0,0)	origin

Note that when GW-BASIC scans coordinate values it will allow them to be beyond the edge of the screen. If an out of range coordinate is given, no action is taken and no error occurs.

PSET allows the *color* argument to be omitted and it is defaulted to the graphics foreground.

Examples:

This example draws a diagonal line to (100,100):

```
10 FOR I = 0 TO 100
20 PSET (I,I)
30 NEXT
```

This example clears out the line by setting each pixel to 0:

```
40 FOR I = 100 TO 0 STEP -1
50 PSET(I,I),0
60 NEXT
```

PUT (graphics) statement

Transfers the graphics image stored in an array to the screen.

Syntax:

PUT(*x,y*), *array*[, *action-verb*]

where

x,y represent the top left corner of the rectangle to be displayed.

array is the name of an array containing the image to be displayed. The type of the array must be numeric.

action-verb is one of: PSET, PRESET, AND, OR, XOR. The default *action-verb* is XOR.

The PUT and GET statements are used to transfer graphics images to and from the screen. PUT and GET make possible animation and high-speed object motion in graphics mode.

The *array* is used simply as a place to hold the image and can be of any type except string. It must be given dimensions large enough to hold the entire image.

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image.

The *action-verb* Parameter

The *action-verb* specifies the interaction between the stored image and the one already on the screen.

PSET transfers the data point-by-point onto the screen. Each point has the exact color it had when it was taken from the screen with a GET.

PRESET is the same as PSET except that a negative image is produced.

AND is used when the image is to be transferred over an existing image on the screen. The resulting image is the product of the logical AND expression; points that had the same color in both the existing image and the PUT image will remain the same color, points that do not have the same color in both the existing image and the PUT image, will not.

OR is used to superimpose the image onto an existing image.

XOR is a special mode often used for animation. It causes the points on the screen to be INVERTED where a point exists in the array image. This behaviour is exactly like that of the cursor. When an image is PUT against a complex background TWICE, the backgound is restored unchanged. This allows you to move an object around the screen without erasing the background.

In medium resolution AND, OR and XOR have the following effects on color:

AND screen

array value	0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

OR screen

array value	0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR screen

array value	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Animation

One of the most useful things that can be done with GET and PUT is animation.

Animation can be performed as outlined below:

1. PUT the object(s) on the screen with the XOR option.

2. Recalculate the new position of the object(s).

3. PUT the object(s) on the screen with the XOR option a second time at the old location(s) to remove the old image(s).

4. Go to step 1, but this time PUT the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET *action-verb*. The idea is to leave a border around the image when it is first received as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points left by the previous PUT. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

Possible Errors

An Illegal function call error occurs, if the image to be transferred is too large to fit on the screen.

SCREEN statement

Sets the specifications for the display screen.

Syntax:

SCREEN [*mode*][, [*burst*][, [*apage*][, *vpage*]]]

where

mode is a numeric expression resulting in an integer value in the range 0 to 255. It defines either Text Mode (0), Medium Resolution Graphics Mode (1), High Resolution Graphics Mode (2), or Super Resolution Graphics Mode (3 to 255).

burst is a numeric expression resulting in an integer value of 0 or 1. It enables color on a color TV set. In Text Mode, a 0 value disables color, and a 1 value enables color. In Medium Resolution Graphics Mode, a 0 value enables color, and a 1 value disables color. Both in High Resolution and Super Resolution Graphics Modes, the *burst* value is ignored, as these two modes only support monochrome.

For a standard monitor, this parameter has no meaning.

apage (Text Mode only) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the active page, i.e., the page to be written to by output statements to the screen. If omitted, the active page defaults to 0.

vpage (Text Mode only) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the visual page, i.e., the page to be displayed on the screen which may be different from the active page. If you omit this parameter, the visual page will default to the active page.

Mode and Burst Parameters

In the following table, the first two columns are the *mode* and *burst* parameters of a SCREEN statement.

The *burst* parameter enables color on color TV sets. For systems with standard monitors, this parameter has no real meaning. For example, a *burst* value of 0 or 1 in medium resolution mode will have the same effect if a color monitor is used; likewise, it will have the same effect if a monochrome monitor is used (in this case the four colors will appear as shades of gray).

mode	*burst*	Description
0	0	80 column x 25 row B/W Text Mode
0	1	80 column x 25 row Color Text Mode
1	0	320 hor. pixels x 200 vert. pixels Color Medium Resolution Graphics Mode (40 column x 25 row)
1	1	320 hor. pixels x 200 vert. pixels B/W Medium Resolution Graphics Mode (40 column x 25 row)
2	x (ignored)	640 hor. pixels x 200 vert. pixels B/W High Resolution Graphics Mode (80 column. x 25 row)
3-255	x (ignored)	640 hor. pixels x 400 vert. pixels B/W Super Resolution Graphics Mode (80 column x 25 row)

Default Values

If you do not enter a SCREEN statement, the system assumes the following default values:

mode = 0 (Text Mode)
burst = 0 (B/W)
apage = 0 (active page 0)
vpage = 0 (visual page 0)

It would be the same, if you entered:

```
SCREEN 0,0,0,0
```

The SCREEN statement must precede any I/O statement to the screen, but you can use more than one SCREEN statement to define different screen attributes for different sections of your program.

apage and vpage Parameters

If Text Mode is selected, you can specify two more parameters (*apage* and *vpage*) to select the active and visual page. There are eight display pages (numbered 0 to 7) in 40-column Text Mode, and four display pages (numbered 0 to 3) in 80-column Text Mode. Only one display page is available in any of the three graphics modes.

Only one cursor is shared between the pages, thus, if you select a new active page, you must save the cursor position (by POS(0) and CSRLIN) before changing to the new page. If you return to the original active page, you must restore the cursor position by the LOCATE (Text) statement. If you use the SCREEN statement only to change the pages, you can omit the first two parameters (*mode* and *burst*).

Screen Width

At initialization the width is 80 columns, thus you should use the WIDTH statement to select a 40-column screen. If you select the medium resolution mode by the SCREEN statement, this also causes the number of columns to be 40 without using the WIDTH statement.

While in Text Mode, the WIDTH statement may be used to select between the 40-column mode and the 80-column mode. Likewise, the WIDTH statement may be used to select between modes 1 and 2 (medium or high resolution mode).

Selecting Text Mode (*mode*=0) after selection of one of the graphics modes will select either a 40-column screen or an 80-column screen, depending on the width used in the graphics mode. For example:

```
SCREEN 1  'set screen to medium res. mode (WIDTH = 40)
SCREEN 0  'changes screen to 40x25 Text Mode
```

See the WIDTH statement in this chapter.

Remarks

If all parameters are valid, the new screen mode is saved, the screen is erased, the foreground and the background colors are set to their default values.

If all parameters are identical to the preceding ones, nothing is altered.

If you omit a parameter, it assumes the preceding value except for the visual page that defaults to the active page.

Examples:

```
10 SCREEN 0,1,0,0      'select text mode with color,
                       'active and visual page to 0.

20 SCREEN,,1,2         'mode and color burst
                       unchanged,
                       'use active page 1,
                       'visual page 2.

30 SCREEN 2            'switch to high res. graphics
                       mode.

40 SCREEN 1,1          'switch to medium res. color
                       graphics.

50 SCREEN ,0           'medium res. graphics, color
                       off.
```

Possible Errors

If you enter a value outside the specified ranges, an Illegal function call error is returned.

VIEW statement

Defines screen limits for graphics activity.

Syntax:

VIEW [[SCREEN] [(vx1,vy1)-(vx2-vy2)[, [color] [, border]]]]

where

(vx1,vy1)(vx2,vy2) represent the x and y coordinates within the physical boundary of the screen that graphics will map into. (*vx1,vy1*) are the upper left and (*vx2,xy2*) are the lower right coordinates to the viewport defined.

color permits the viewport to be filled with a specified color. If *color* is omitted, then the viewport is not filled-in.

border permits the drawing of a border-line surrounding the viewport if the necessary space for a border is available. If *border* is omittted, no border-line is drawn.

Characteristics

VIEW defines a "Physical Viewport" limit from *vx1,vy1* (upper left x,y coordinates) to *vx2,vy2* (lower right x,y coordinates). The x and y coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which graphics may be mapped (*see also the WINDOW statement*).

Initially, RUN, SCREEN, and VIEW with no arguments define the entire screen as the viewport.

SCREEN Option

The SCREEN option dictates that the x and y coordinates are absolute to the screen, not relative to the border of the physical viewport, and only graphics within the viewport will be plotted.

For the form:

VIEW(*vx1,xy1*)-(*vx2,xy2*)

all points plotted are relative to the viewport. That is, *vx1* and *vy1* are added to the x and y coordinates before putting down the point on the screen.

If

VIEW(10,10)-(200,100)

were executed, then the point set down by the statement PSET(0,0),3 would actually be at the physical screen location 10,10.

For the form:

VIEW SCREEN (*vx1,vy1*)-(*vx2,vy2*)

all coordinates are absolute and may be inside or outside of the screen limits, but only those within the VIEW limits will be plotted.

If

VIEW SCREEN (10,10)-(200,100)

were executed, then the point set down by the statement PSET(0,0),3 would actually not appear because 0,0 is outside of the viewport. PSET(10,10),3 is within the viewport, and places the point in the upper left hand corner of the viewport.

Multiple Viewports

Each time a VIEW statement is executed, a viewport is defined; this is the "current" viewport. Thus, to change the "current" or "active" viewport, you have to execute another VIEW statement.

A number of VIEW statements may be executed, if the newly described viewport is not wholly within the previous viewport, the screen can be reinitialized with the VIEW statement. Then the new viewport may be stated.

Examples:

This example opens three viewports, each smaller than the previous one. In each case, a line that is defined to go beyond the borders is programmed, but appears only within the viewport border.

```
260 CLS:SCREEN 1
280 VIEW 'Make the viewport the entire screen.
300 VIEW(10,10)-(300,180),,1
320       CLS
330       LINE(0,0)-(310,190),1
360       LOCATE 1,11:PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400       CLS 'clears only viewport
420       LINE(300,0)-(0,199),1
440       LOCATE 9,9:PRINT "A medim viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480       CLS
500       CIRCLE(150,100),20,1
520       LOCATE 11,9:PRINT ""A small viewport"
```

This example defines two viewports, plots two pixels, both in relative and absolute coordinates, and draws two circles within each viewport. Note that one of the circles is clipped within each viewport. The WINDOW statement changes the coordinate system and causes the "zooming" effect. *See the WINDOW statement in this chapter.*

```
10 CLS:SCREEN 3
20 VIEW(30,30)-(300,300),,3
30 PSET(30,30),3 'relative coordinates
40 CIRCLE(135,135),8:CIRCLE(135,0),20
50 VIEW SCREEN(340,30)-(610,300),,3
60 PSET(370,60),3 'absolute coordinates
65 rem zooming for new range values
70 WINDOW SCREEN(40,30)-(320,300)
80 CIRCLE(270,200),8:CIRCLE(300,250),35
```

This example defines a viewport in the top left hand corner of the screen, draws one large rectangle, which is overlapped by a small rectangle within the viewport.

```
10 CLS:SCREEN 2
20 VIEW(1,1)-(100,100),,3
30 LINE(50,50)-(80,80),3,BF
35 WINDOW SCREEN(1,1)-(200,200)
40 LINE(150,150)-(180,180),3,BF
```

WIDTH statement

Sets the line width in characters. GW-BASIC adds a carriage return after outputting the specified number of characters.

Syntax:

WIDTH *size* or WIDTH "SCRN:",*size*

where

size is an integer expression in the range 0 to 255. It specifies the new width.

Sets the screen width (in Text Mode), selects a text window or changes mode (in Graphics mode). Changing the screen or text window width, or the mode, causes the screen to be cleared.

In Text Mode, *size* may only have the values 40 or 80, selecting either a 40-column or an 80-column screen.

In Graphics Mode you can either change mode or select a text window to the left of the screen of width less than or equal to 40 (Medium Resolution Mode) or less than or equal to 80 (High or Super Resolution Mode).

The width of the function key display will correspond to the selected width. If the number of columns displayed is less than 80 columns, a CTRL T may be entered to scroll the function key display horizontally.

The table on the next page summarizes all possible cases.

IF *mode* is ...	AND *size* is ...	THEN you ...
0 (text)	40	select a 40-column screen
	80	select an 80-column screen
1 (medium-res)	80	place the system in high-resolution (mode 2)
	8< = *size*< = 40	create a text window of width *size*
2 (high res)	40	place the system in medium resolution (mode 1) with *burst* in whatever state the system was when a text or medium resolution mode was last used
	8< = *size*< = 39 or 41< = *size*< = 80	create a text window of width *size*
	size = 4	create a text window of width 40
3-255 (super res)	8< = *size*< = 80	create a text window of width *size*
	8< = *size*-80< = 80	create a text window of width *size*

When the WIDTH statement causes a change in the screen mode, colors are set to their default values.

You should turn the function key display off when changing the window width by a KEY OFF statement; otherwise, if the width is decreased, part of the old (wider) function key display may be left on the screen.

If *size* is outside the specified ranges, an Illegal function call error is returned. The previous value is retained.

Examples:

SCREEN 1,0	set screen to medium res. color graphics
WIDTH 80	change screen to high res. graphics
WIDTH 40	changes screen back to medium res.
SCREEN 0,1	changes screen to 40x25 text color mode
WIDTH 80	changes screen to 80x25 text color mode

WINDOW statement

Defines the logical dimensions of the current viewport.

Syntax:

WINDOW [[SCREEN] (*wx1,wy1*)-(*wx2,wy2*)]

where

(*wx1,wy1*)-(*wx2,wy2*) are the world coordinates. (*wx1,wy1*) represent the lower-left and (*wx2,wy2*) the upper-right coordinates of the screen border. The SCREEN option inverts the y-axis of the world coordinates, so that (*wx1,wy1*) represent the upper-left and (*wx2,wy2*) the lower-right coordinates of the screen border.

Characteristics

WINDOW allows you to draw lines, graphs, or objects in the space not bounded by the physical dimensions of the screen. This is done by using arbitrary programmer-defined coordinates called "world coordinates". When you have redefined the screen, graphics can be drawn within a customized mapping system.

GW-BASIC converts world coordinates into physical coordinates for subsequent display within the current viewport, as defined by the VIEW statement. To make this transformation from world space to the physical space of the viewing surface (screen), you must know what portion of the (floating point) world coordinate space contains the information to be displayed.

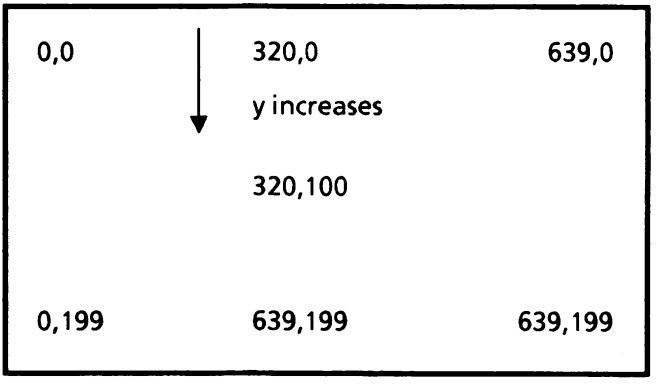
This rectangular region in world coordinate space is called a "window".

Initially, RUN, SCREEN, or WINDOW with no arguments, disables "window" transformation.

If you enter:

```
NEW
SCREEN 2
```

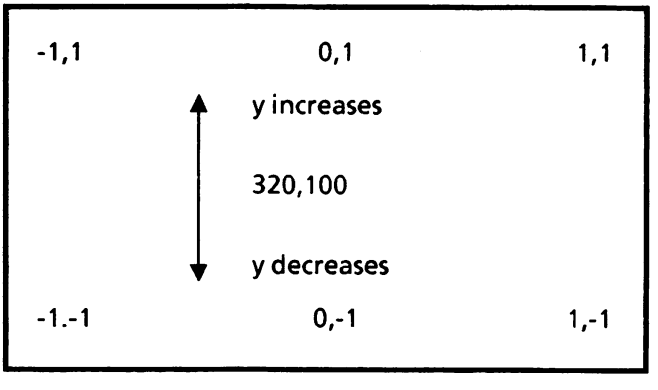
the screen will appear as:



Now enter:

```
WINDOW (-1,-1)-(1,1)
```

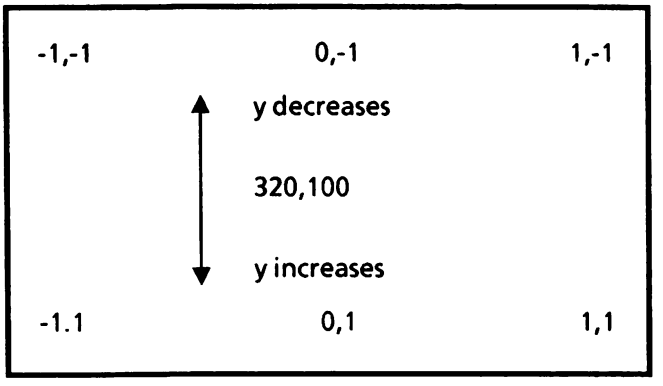
the screen will appear as:



If the variant:

WINDOW SCREEN (-1,-1)-(1,1)

is executed, then the screen appears as:



You can increase or decrease the size of the image to be displayed and clip part of the image by changing the logical dimensions of the current viewport via the WINDOW statement.

Examples:

The following example illustrates two lines with the same end point coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
200 LINE(100,100)-(150,150),1
220 LOCATE 2,20:PRINT "The line on the default screen"
240 WINDOW SCREEN (100,100)-(200,200)
260 LINE (100,100)-(150,150),1
280 LOCATE 8,18
300 PRINT "& the same line on a redefined window"
```

The following example draws three concentric circles with the center in the middle of the screen. The WINDOW statement defines three different coordinate systems with the same origin in the middle of the screen and the same x-, y-axis orientation. Note the zooming effect produced by the WINDOW statement.

```
5 SCREEN 3:CLS
10 DATA -30, 30, -20, 20, -10, 10
20 FOR I = 1 TO 3
30        READ X,Y
40        WINDOW SCREEN (X,X)-(Y,Y):GOSUB 100
50        CIRCLE (0,0),1
55 NEXT
60 END
100 'Draw a Carthesian system of coordinates
110 LINE(X,0)-(Y,0)
120 LINE(0,X)-(0,Y)
130 RETURN
```

16. GW-BASIC AND CHILD PROCESSES

Through the use of the SHELL command, GW-BASIC is able to use one of the most powerful features of MS-DOS: the ability to create child processes. SHELL enables you to run part of a GW-BASIC program, temporarily exit to MS-DOS to perform a specified function, and return to the GW-BASIC Program at the statement after the SHELL command to proceed with the rest of the program.

GW-BASIC will produce a child program when it uses the SHELL command. It is not possible for GW-BASIC to totally protect itself from its children. When a SHELL command is executed, many things may be going on. For example, files may be OPEN and devices may be in use.

The following guidelines will help to prevent child processes from harming the GW-BASIC environment.

Hardware

In general, it is recommended that the state of all hardware be preserved during a SHELL command. The implementation interface provides a way for performing this task. However, it may be necessary to request that you refrain from using certain devices within child processes which are executed using the GW-BASIC SHELL command. Specific areas of concern are as follows:

1. Screen Device - Child processes might modify screen mode parameters. However, useful information may be displayed by a child process.

2. Interrupt Vectors - Save and restore interrupt vectors the child intends to use.

3. Other hardware - Many devices are placed in a specific state by GW-BASIC. Such devices may include an Interrupt Controller, Counter Timers, DMA Controller, I/O Latch, and Uarts. These devices may be utilized by the child process without the user being aware of any limitations.

The File System

A child that alters any file open in the GW-BASIC parent may have disastrous results.

If it is necessary to update such files, they should be CLOSEd in the parent before doing a SHELL, then re-OPENed upon return to the GW-BASIC parent. (*See "Redirection of Standard Input and Output" under the GWBASIC command in Chapter 19.*)

Memory Management

1. Before GW-BASIC "Shells" to COMMAND, it will try to free any memory it is not using with one exception: when GW-BASIC is run with the /M: switch. In this case, GW-BASIC must assume that you intended to load something in the top of GW-BASIC's Memory Block. This prevents GW-BASIC from "compressing the workspace" before doing the SHELL. For this reason, SHELL may fail on an Out of memory error when using the /M: switch.

   The preferred method is to load machine language subroutines before GW-BASIC is run. This can be accomplished by placing "Pocket Code" at the end of machine language subroutines that allows them to exit to MS-DOS and stay resident. See example on next page.

```
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 before GW-BASIC by running them. The AUTOEXEC.BAT file is very useful for this.

2. A child should never "terminate and stay resident". Doing so may not leave GW-BASIC enough room to expand it's workspace to the original size. If GW-BASIC cannot restore the workspace, all files are closed, the error message SHELL can't continue is displayed, and GW-BASIC exits to MS-DOS.

3. There is no restriction in the machine independent portion of GW-BASIC which prohibits GW-BASIC from running as a child of GW-BASIC. However, due to the complications which arise from this configuration, it may not be advisable to use this capability.

SHELL command

Loads and executes another program (.EXE or .COM. or .BAT).

Syntax:

SHELL [*stringexp*]

where

stringexp is a string expression containing the name of a program to run and optionally command arguments.

Characteristics

When the program finishes, control returns to the GW-BASIC program at the statement following the SHELL command. A program executed under control of GW-BASIC is referred to as a "child process".

Child processes (or "children") are executed by SHELL loading and running a copy of COMMAND with the /C switch. By using COMMAND this way, command line parameters are passed to the child. Standard Input and Output may be redirected, and built-in commands such as DIR, PATH, and SORT may be executed.

Rules

1. The program name in *stringexp* may have any extension you want since COMMAND has to worry about it. If no extension is supplied, COMMAND will look for a .COM file, then .EXE file, and finally, a .BAT file. If COMMAND is not found, SHELL will issue a File not found error. No error is generated if COMMAND cannot find the file specified in *stringexp*.

2. Any text in *stringexp* separated from the program name by at least one blank, will be processed by COMMAND as program parameters.

3. GW-BASIC remains in memory while the child process is running. When the child finishes, GW-BASIC continues.

4. SHELL with no *stringexp* will give you a new COMMAND shell. You may now do anything that COMMAND allows. When ready to return to GW-BASIC, enter the MS-DOS command EXIT.

Examples:

```
Ok
SHELL (get a new COMMAND)
A>DIR (user enters DIR to see files)
A>EXIT    (user enters EXIT to return to GW-
           BASIC)
Ok
```

Write some data to be sorted, SHELL executes SORT to sort it, then read the sorted data to write a report.

```
900 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
...
950 REM write data to be sorted
...
1000 CLOSE 1
1010 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
1030 REM Process the sorted data
...
```

```
10 SHELL "DIR|SORT >FILES"
20 OPEN "FILES" FOR INPUT AS #1
...
```

Possible Errors

File not found
: A file name (.EXE or .COM or.BAT) could not be found.

Out of memory
: There was not enough memory to run the child.

Can't continue after SHELL
: There is not enough memory for GW-BASIC to continue. All files are closed and GW-BASIC returns to MS-DOS. This may happen when a child process "Terminates and stays Resident".

Internal error
: Either GW-BASIC or MS-DOS is not functioning correctly.

17. INPUT DATA

This chapter describes the following

DATA, READ, RESTORE statements

INKEY$ function

INPUT statement

INPUT$ function

LET statement

LINE INPUT statement

DATA, READ, RESTORE statements

DATA statement

Creates an "internal" file, i.e., a sequence of data belonging to the program. Each data item will then be assigned to a program variable by a READ statement.

Syntax:

DATA *constant*[, *constant*]...

where

constant is a numeric or string constant. Any numeric format (i.e., integer, hexadecimal, octal, single or double precision) is acceptable for numeric constants. 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.

DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program.

A DATA statement in a program need not correspond to a specific READ statement. This is because before program execution, a data file (the "internal file" as it is often called) is created. It contains all the values of all the DATA statements in the program in line number sequence. When the program is executed, READ takes its values from this file.

The data-type of an entry in the data sequence must correspond to the type of the variable to which it is to be assigned; i.e., numeric variables require numeric constants as data (conversion from one numeric type to another is allowed, for example, you may have a single precision floating point constant associated with an integer variable) and string variables require quoted or unquoted strings as data.

A quoted string is required if the string contains commas (e.g., "BIRMINGHAM,") or initial or final blanks (e.g., " BIRMINGHAM").

DATA statements may be reread from the beginning by use of the RESTORE statement.

See examples in the READ statement description beginning on the next page.

READ statement

Reads values from one or more DATA statements and assigns them to variables.

Syntax:

READ *variable*[, *variable*]...

where

variable is a numeric or string variable. The type of the *variable* must agree with the type of the associated value in the DATA sequence.

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.

If the data type (numeric or string) of an entry in the data sequence does not correspond to the type of the associated variable, a Syntax error will result. However, any numeric data type (integer, single or double precision) may be assigned to any numeric variable.

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 the list of variables exceeds the number of elements in the DATA statement(s), an Out of DATA message is displayed.

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.

Examples:

This program READs the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, A(2) is 5.19, and so on.

```
Ok
10 FOR I = 1 TO 5
20 READ A(I)
30 PRINT "A(";I;") = ";A(I)
40 NEXT I
50 DATA 3.08,5.19
60 DATA 3.12,3.98,4.24
RUN
A( 1 ) = 3.08
A( 2 ) = 5.19
A( 3 ) = 3.12
A( 4 ) = 3.98
A( 5 ) = 4.24
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

```
Ok
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
RUN
CITY                STATE           ZIP
DENVER,             COLORADO         80211
Ok
```

RESTORE statement

Permits DATA statements to be reread either from the beginning of the internal data file or from a specified line.

Syntax:

RESTORE [*linenum*]

where

linenum must be the line number of a DATA statement.

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program.

If *linenum* is specified, the next READ statement accesses the first data item in the specified DATA statement.

Example:

```
Ok
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 PRINT A;B;C;D;E;F
50 DATA 57, 68,79
RUN
 57  68  79  57  68  79
   Ok
```

INKEY$ function

Returns either a one- or two-character string read from the keyboard or a null string if no character is pending at the keyboard.

Syntax:

INKEY$

INKEY$ returns one of the following values:

a null string if no character is read from the keyboard

a one-character string in accordance with a single character read from the keyboard

a two-character string if a key (or a key combination) is entered, that cannot be associated with a standard ASCII code. The first character is hex zero (00); the second indicates the extended code.

Although more than one character may be pending in the keyboard buffer, a single character only will be read. This value must then be assigned to a variable before it is considered by the GW-BASIC program.

No characters will be displayed. All characters are passed to the program, except for the following control characters:

PRT SC	(prints the screen)
CTRL NUMLOCK	(sets the system to pause)
CTRL BREAK	(stops the program)
ALT CTRL DEL	(resets the system)

Note that a carriage return is passed to the program like any other character.

Examples:

In this example, if the Return key is pressed before the loop reaches the final value of 200, the message Return key pressed is displayed on the screen and execution ends. If the Return key is not pressed, the program executes 200 times, then the message Return key not pressed is displayed at the end of the program.

```
Ok
10 FOR X = 1 TO 200
20 PRINT X
30 A$ = INKEY$
40 IF LEN(A$) = 0 THEN 60
50 If ASC(A$) = 13 then print "Return key
pressed":END
60 NEXT X
70 PRINT "Return key not pressed"
RUN
 1
 2
 3
Return key pressed
Ok
```

The following example program will display in hex the value of the key pressed.

```
10 S$ = INKEY$
20 IF LEN(S$) = 0 THEN GOTO 10
30 IF LEN(S$) = 2 THEN GOTO 100
40 REM display 1-byte codes
50 PRINT HEX$(ASC(S$))
60 GOTO 10
100 REM display 2-byte codes - second byte in hex
(and decimal)
110 S1$ = MID$(S$,1,1):S2$ = MID$(S$,2,1)
120 PRINT HEX$(ASC(S1$)); "  "; HEX$(ASC(S2$));
"(";ASC(S2$);")"
130 GOTO 10
```

To exit the above program, use CTRL BREAK.

INPUT statement

Allows input from the keyboard during program execution.

Syntax:

INPUT [;] [*prompt*;] *variable* [, *variable*]...

where

prompt is a string constant enclosed in quotation marks which prompts you for the values you have to enter from the keyboard.

variable is a numeric or string variable to which is assigned the value entered from the keyboard.

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. For example:

```
Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2
RUN
? 5                          (user types 5)
 5 SQUARED IS 25
Ok
```

If ***prompt*** is included, the string is displayed before the question mark. The required data is then entered from the keyboard. A comma may be used instead of a semicolon after the ***prompt*** string to suppress the question mark. See example at top of next page.

```
Ok
10 PI = 3.14
20 INPUT "What is your name";N$
30 INPUT "Enter the radius ",R
40 A = PI*R^2
50 PRINT N$ ", the area of the circle is";A
RUN
What is your name? TOM
Enter the radius 7.4
TOM, the area of the circle is 171.9464
Ok
```

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not display a carriage return/line feed sequence. For example:

```
Ok
10 INPUT;N
20 PRINT " MAPLE AVENUE"
RUN
? 1120 MAPLE AVENUE
(user types 1120 and presses Return key)
Ok
```

The data that is entered is assigned to the variable(s) given in the *variable* list. The number of data items supplied must be the same as the number of variables in the list. Data items must be 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 (string instead of numeric, etc.) causes the message ?Redo from start to be displayed. No assignment of input values is made until an acceptable response is given.

You may use all the GW-BASIC screen editor features (described in Chapter 12) in responding to INPUT and LINE INPUT statements.

INPUT$ function

Returns a string of characters read from the keyboard.

Syntax:

INPUT$(*length*)

where

length is an integer expression specifying the number of characters to be read from the keyboard.

No characters will be displayed on the screen. All characters including control chararters are passed through except CTRL BREAK, which is used to interrupt the execution of the INPUT$ function.

Example:

```
Ok
10 Print "Do you wish to continue? (Y or N)";
20 X$ = INPUT$(1)
30 Print
40 If X$ = "N" then print "This is the end":end
50 If X$ = "Y" then print "Continuing" else 10
RUN
Do you wish to continue? (Y or N)
(user types an uppercase N)
This is the end
Ok
```

In this example, only one character can be input from the keyboard. This character can be either an uppercase N or an uppercase Y.

If the INPUT$ statement had read X$ = INPUT$(3), then three characters could be entered.

LET statement

Assigns a value to a variable.

Syntax:

[LET] *variable* = *expression*

where

variable is a numeric or string variable which will receive the value of the ***expression***.

expression is evaluated and assigned to the line variable on the left side of the equal sign.

The word LET is optional. The equal sign is sufficient when assigning an ***expression*** to a ***variable*** name.

The type of the ***expression*** (numeric or string) must match the type of the ***variable***; if not, a Type mismatch error occurs. However, in numeric assignments the type of the ***expression*** (integer, single precision or double precision) may be different from the type of the destination ***variable***. In this case, GW-BASIC converts the ***expression*** value to the type of the ***variable***. Rounding or overflow may occur in this conversion.

Example:

```
10 LET A = 12
20 LET B = 12*2
30 C = 12/2
40 SUM = A + B + C
```

Note that the word LET is optional (see statements 30 and 40 above).

LINE INPUT statement

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Syntax:

LINE INPUT [;] [*prompt*;] *stringvar*

where

prompt is a string constant (enclosed in double quotation marks) that is displayed on the screen before input is accepted.

stringvar is the name of a string variable to which the line will be assigned.

A question mark is not displayed unless it is part of the "*prompt* string"

All input from the end of the *prompt* to the carriage return is assigned to *stringvar*. If a linefeed/carriage return sequence (this order only) is encountered, both characters are displayed; but the carriage return is ignored, the linefeed is put into *stringvar*, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return pressed to end the input line does not display a carriage return/linefeed sequence on the screen.

A LINE INPUT statement may be escaped by pressing CTRL BREAK. GW-BASIC will return to command level. Typing CONT resumes execution at the LINE INPUT.

See example on next page.

Example:

```
Ok
10 Line Input "Description of item? ";D$
20 PRINT D$
RUN
Description of item? ASCII keyboard
ASCII keyboard
Ok
```

18. LOOPING

This chapter describes the following

FOR...NEXT statements

WHILE...WEND statements

FOR...NEXT statements

Allows a series of statements to be performed in a loop a specified number of times.

Syntax:

FOR *control-variable* = *initial-value* TO *final-value* [STEP *increment*]

.
.
.

[loop statements]

.
.
.

NEXT [*control-variable*][,*control-variable*]...

where

control-variable is an integer or single precision variable used as a counter.

initial-value is a numeric expression specifying the first value assigned to the *control-variable*.

final-value is a numeric expression specifying the limit of the *control-variable*.

increment is a numeric expression specifying the value to be added (with its algebraic sign) to the *control-variable* when the NEXT statement is encountered.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the *control-variable* is incremented by the amount specified by STEP *increment*. A check is performed to see if the value of the *control-variable* is now greater than the *final-value*. If it is not greater, GW-BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the *increment* is assumed to be one. If *increment* is negative, the *final-value* of the *control-variable* must be less than the *initial-value*. The *control-variable* is decreased each time through the loop. The loop is executed until the *control-variable* is less than the *final-value*.

The *control-variable* must be an integer or single precision numeric variable. If a double precision numeric variable is used, a Type mismatch error will result.

The body of the loop is skipped if either the *increment* is positive, and the *initial-value* exceeds the *final-value*, or the *increment* is negative, and the *initial-value* is less than the *final-value*.

If a NEXT statement is encountered before its corresponding FOR statement, a NEXT without FOR error message is displayed and execution is terminated.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. When using nested loops, the variables(s) in each NEXT statement **must** be specified.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them. Note that a statement of this form:

```
100 NEXT V1, V2, V3
```

performs the same action as the sequence of statements:

```
100 NEXT V1
110 NEXT V2
120 NEXT V3
```

Examples:

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30     PRINT I;
40     K = K + 10
50     PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

In the above example, the control-variable (I) advances +2 on each cycle. Each time through the loop the control-variable value is displayed, the value of K is calculated and displayed.

```
10 J = 0
20 FOR I = 1 TO J
30     PRINT I
40 NEXT I
RUN
Ok
```

In the above example, the loop does not execute because the initial-value of the loop exceeds the final-value.

```
10 I = 5
20 FOR I = 1 TO I + 5
30     PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
```

In the above example, the loop executes ten times. The final-value for the loop variable is always set before the initial-value is set.

WHILE...WEND statements

Execute a series of statements in a loop as long as a given *condition* remains true.

Syntax:

```
WHILE condition
        .
        .
        .
[loop statements]
        .
        .
        .
WEND
```

where

condition is a numeric, relational or logical expression.

GW-BASIC determines whether the *condition* is true or false by testing the result of the expression for non-zero and zero, respectively. A non-zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non-zero or zero by merely specifying the name of the variable as a condition.

If *condition* is not zero (true), loop statements are executed until the WEND statement is encountered. GW-BASIC then returns to the WHILE statement and checks *condition*. If it is still true, the process is repeated. If it is zero (false), 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.

Do not direct program flow into a WHILE...WEND loop without entering through the WHILE statement.

Example:

```
10 FOR C = 1 TO 9
20        READ CUST$(C)
30 NEXT C
40 DATA SHARON,MARILYN,FELIX
50 DATA BARBARA,SHERRY,TOM
60 DATA LORNA,KARL,ELISE
100 rem bubble sort of cust$ array
110 FLIPS = 1 'forces one pass thru loop
120 WHILE FLIPS
130       FLIPS = 0
140       FOR I = 1 TO 8
150           If cust$(I) > cust$(I + 1) then
              swap cust$(I),cust$(I + 1):
              FLIPS = 1
160       NEXT I
170 WEND
200 rem display sorted cust$ array
210 FOR X = 1 TO 9
220       PRINT CUST$(X)
230 NEXT X
RUN
BARBARA
ELISE
FELIX
KARL
LORNA
MARILYN
SHARON
SHERRY
TOM
Ok
```

19. MISCELLANEOUS STATEMENTS, COMMANDS AND FUNCTIONS

This chapter describes the following

BEEP statement
CLEAR command
DATE$ function
DATE$ statement
DEFINT/SGN/DBL/STR statements
ENVIRON statement
ENVIRON$ function
FRE function
GWBASIC command
KEY statement
RANDOMIZE statement
REM statement
RND function
SWAP statement
TIME$ function
TIME$ statement
TIMER function
TIMER and ON TIMER GOSUB statements

BEEP statement

Activates the bell.

Syntax:

BEEP

The **BEEP** statement sounds the ASCII bell character. This statement has the same effect as PRINT CHR$(7).

Example:

```
10 X = 15
20 IF X < 20 THEN BEEP
RUN
(speaker sounds)
Ok
```

CLEAR command

Clears all numeric variables to zero, all string variables to null, and closes all open files. Options set the highest memory location available for use by GW-BASIC, and the amount of stack space.

Syntax:

CLEAR [, [*memory*][, *stack*]]

where

memory is an integer expression representing a memory location which; if specified, sets the top of memory (i.e., the maximum extension of the GW-BASIC Data Segment).

stack is an integer expression whose value sets aside stack space for GW-BASIC. The default is 128 bytes or one-eighth of the available memory, whichever is smaller.

The ***memory*** parameter should be specified to reserve space in storage for assembly language routines.

The ***stack*** parameter to use several nested GOSUBs, FOR...NEXT loops, or PAINT to paint complex pictures.

GW-BASIC allocates string space dynamically. An Out of string space error occurs only if there is no free memory left for GW-BASIC to use.

If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 128 bytes, and the default top of memory is the current top of memory.

The CLEAR command performs the following actions:

closes all files

clears all COMMON variables

resets the stack and string space

resets all simple numeric variables and numeric array elements to zero

resets all simple string variables and string array elements to null

releases all disk buffers

resets all DEF FN, DEFINT/SNG/DBL/STR, DEF SEG and DEF USR statements

Examples:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

DATE$ function and statement

Retrieves the date (as a function), or sets the date (as a statement).

Syntax 1 (as a function):

stringvar = DATE$

Syntax 2 (as a statement):

DATE$ = *stringexp*

As a function, the current date is fetched and assigned to the string variable *stringvar*. The DATE$ function may also be used in any string expression in a LET or PRINT statement.

As a statement, the current date is set. In this case, DATE$ is the target of a string assignment.

The date may also have been set by MS-DOS prior to entering GW-BASIC.

Rules

1. If *stringexp* is not a valid string, a Type mismatch error will result. Previous values are retained.

2. For *stringvar* = DATE$, DATE$ returns a 10-character string in the form "mm-dd-yyyy" where mm is the month (01 to 12), dd is the day (01 to 31) and yyyy is the year (1980 to 2099).

3. For DATE$ = *stringexp*, *stringexp* may be one of the following forms:

   "mm-dd-yy" or "mm/dd/yy"
   or
   "mm-dd-yyyy" or "mm/dd/yyyy"

If the month or day is specified by the use of only one digit, GW-BASIC assumes a 0 (zero) in front of it. If the year is specified by the use of one digit (y), GW-BASIC assumes the year to be 200y; if two digits are specified (yy), the year will be 19yy.

If any of the values are out of range or missing, an Illegal function call error is issued. Any previous date is retained.

Function examples:

```
10 PRINT DATE$

Ok
PRINT DATE$
07-01-1985
```

Statement examples:

```
10 DATE$ = "07-01-1985"

Ok
DATE$ = "07-01-1985"
Ok
```

DEFINT/SNG/DBL/STR statements

Declare the variable type in accordance with the letter(s) specified.

Syntax:

DEF*type* *letter*[- *letter*][, *letter*[- *letter*]]...

where

type is INT, SNG, DBL, or STR. No space should be entered between DEF and INT, SNG, DBL, or STR.

letter represents a letter from the alphabet (A-Z).

Any variable names beginning with the letter(s) specified in range of *letter*s will be considerd the type of variable specified by the *type* portion of the statement.

A type declaration character (%, !, #, $) always takes precedence over a DEFtype statement.

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

DEFtype statements must precede the use of the defined variables.

Examples:

```
10 DEFDBL L-P
```
(All variables beginning with the letters L, M, N, O, and P will be double precision variables.)

```
10 DEFSTR A
```
(All variables beginning with the letter A will be string variables.)

```
10 DEFINT I-N, W-Z
```
(All variables beginning with the letters I,J,K,L,M,N,W,X,Y,Z will be integer variables.)

ENVIRON statement

Allows modification of parameters in GW-BASIC's Environment String Table. This is similar to the MS-DOS SET command.

Syntax:

ENVIRON *parm*

where

parm is a valid string expression containing the new Environment String parameter. The string expression must only include uppercase letters.

The ENVIRON statement may be used, for example, to change the "PATH" parameter for a child process. Parameters may also be passed to a child process by inventing a new environment parameter.

Rules

1. *parm* must be of the form *parm-id* = *text* where:

   parm-id is the name of the parameter such as "PATH".

   parm-id must be separated from *text* by an equal sign (=) or " " (blank), such as "PATH= ". ENVIRON takes everything to the left of the first blank or "=" as the *parm-id*, and everything to the right as *text*.

   text is the new parameter text. If *text* is a null string, or consists only of ";" (a single semicolon, such as "PATH=;") then the parameter (including *parm-id*=) is removed from the Environment String table and the table is compressed.

2. If *parm-id* does not exist in the Environment String Table, then *parm-id* is added at the end of the Enviornment String Table.

3. If *parm-id* does exist, it is deleted, the Environment String Table is compressed and the new *parm-id* is added at the end.

Examples:

The following MS-DOS command will create a default "PATH" to the root directory on Disk A.

```
A>PATH = A:
A>
```

The PATH may be changed while in GW-BASIC to a new value by:

```
Ok
ENVIRON "PATH = A:SALES;A:ACCOUNTS"
Ok
```

A new parameter may be added to the Environment String Table:

```
ENVIRON "SESAME = PLAN"
```

The Environment String Table now contains:

```
PATH = A:SALES;A:ACCOUNTS
SESAME = PLAN
```

If you then enter:

```
ENVIRON "SESAME = ;"
```

then you would have deleted SESAME, and you would have a table containing:

```
PATH = A:SALES;A:ACCOUNTS
```

Possible Errors

Type mismatch	If *parm* is not a string.
Out of memory	If the Environment Table is full and no more can be allocated.

ENVIRON$ function

Allows you to retrieve the specified Environment String from GW-BASIC's Environment String Table.

Syntax:

ENVIRON$ { (*parm*) | (nth-parm) }

where

parm is a valid string expression containing the parameter to be retrieved. The string expression must only include uppercase letters.

nth-parm is an integer expression returning a value in the range 1 to 255.

Rules

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 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*.

4. If there is no *nth-parm*, then a null string is returned.

See example on next page.

Possible Errors

Illegal function call
If *nth = parm* is out of range.

Type mismatch
If *parm* is not a string.

String too long
If the string is longer than 255 characters.

Example:

This program will read the Environment String Table and display the contents on the screen.

```
5 rem read contents into an array
10 DIM A$(50)
20 X = 0
30 X = X + 1
40 A$(X) = ENVIRON$(X)
50 IF A$(X) <> "" GOTO 30
60 rem display the contents of the array
70 X = X - 1
80 FOR CNT = 1 TO X
90      PRINT A$(CNT)
100 NEXT CNT
110 END
```

FRE function

Returns the number of bytes in memory not being used by GW-BASIC.

Syntax:

FRE(dummy)

The argument to FRE is a dummy argument. Any value may be supplied.

FRE("") forces a garbage collection before returning the number of free bytes.

GW-BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:

```
PRINT FRE(0)
 14542
Ok
```

GWBASIC command

Initializes GW-BASIC and the operating environment (GWBASIC is an MS-DOS command, not a GW-BASIC command).

Syntax:

GWBASIC [*filespec*] [<*stdin*][[>]>*stdout*] [/F: *number-of-files*] [/S: *lrecl*] [/C: *buffer-size*] [/M: *highest-memory*][, *max-blocksize*]] [/D] [/I]

where

Options beginning with a slash (/) are called switches. A switch is a means used to specify parameters.

filespec is a string literal (not included in quotation marks) that specifies a GW-BASIC program file. If the file is present, GW-BASIC proceeds as if a RUN "*filespec*" command was given after initialization is complete.

filespec is a file or path name with an optional drive name. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

A default extension of .BAS is used if none is supplied and the file name is less than 9 characters long. The *filespec* option allows GW-BASIC programs to be run in batch by putting this form of the command line in an AUTOEXEC.BAT file. GW-BASIC programs which run this way will need to exit via the SYSTEM command in order to allow the next command from the AUTOEXEC.BAT file to be executed.

stdin is a literal string (not included in quotation marks) for the standard input file specification. GW-BASIC input is redirected from the file specified by *stdin*. When present, this syntax must appear before any switches. (*See "Redirection of Standard Input and Output" section below.*)

stdout is a literal string (not included in quotation marks) for the standard output file specification. GW-BASIC is redirected to the file specified by *stdout*. When present, this syntax must appear before any switches. (*See "Redirection of Standard Input and Output" below.*)

/F: *number-of-files* is a switch that sets the maximum number of files (from 1 to 15) that may be open simultaneouly during the execution of a GW-BASIC program. It is ignored unless the /I switch is specified on the command line. *Refer to the /I switch below.*

If this switch and the /I switch are present, then the maximum number of files 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 via the /S: option switch. 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. It is recommended that FILES = 10 for GW-BASIC. Remember that the first 3 are taken by *stdin*, *stdout*, *stderr*, *stdaux*, and *stdprn*. One additional file handler is needed by GW-BASIC for LOAD, SAVE, CHAIN, NAME, and MERGE. This leaves 6 for GW-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 handlers have been exhausted will result in a Too many files error.

/S: *lrecl* is a switch that sets the maximum record length allowed with random files. It is ignored unless the */I* switch is specified on the command line (*refer to the /I switch below*). If this switch and the */I* switch are present, then the maximum record length is set to *lrecl*. The record length option (*record-length*) on the OPEN statement cannot exceed this value. If the */S:* option is omitted, the record length defaults to 128 bytes. The maximum value permitted for *lrecl* is 32767 bytes.

/C: ***buffer-size***, if present, controls RS232 Communications. If RS232 cards are present, */C:0* disables RS232 support. Any subsequent I/O attempts will result in a Device Unavailable error. Specifying /C:*n* allocates *n* bytes for the receive buffer for each RS232 card present. If the /C: option is omitted, GW-BASIC allocates 256 bytes for the receive buffer of each card present. 128 bytes are always allocated to the transmit buffer. GW-BASIC ignores the /C: switch when RS232 cards are not present. The maximum value permitted for ***buffer-size*** is 32767.

/M:[*highest-memory*][,*max-blocksize*], when present, ***highest-memory*** sets the maximum number of bytes that will be used as GW-BASIC workspace. GW-BASIC will attempt to allocate 64K of memory for the data and stack segment. If machine language subroutines are to be used with GW-BASIC programs, use the /M: switch to set the highest memory location that GW-BASIC can use. When omitted or 0, GW-BASIC attempts to allocate all it can up to a maximum of 65536 bytes.

In order to load programs above the GW-BASIC workspace, you must use the optional parameter ***max-blocksize*** to reserve areas for the workspace and your programs. This is necessary if you intend to use the SHELL command. Failure to do so will result in COMMAND being loaded on top of your routines when a SHELL command is executed.

max-blocksize must be in Paragraphs (multiples of 16 bytes). When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes (65536 = 4096 x 16) for GW-BASIC's Data and Stack segment. If you require 65536 bytes for GW-BASIC and 512 bytes for machine language subroutines, then use /M:&H1010 (4096 paragraphs for GW-BASIC + 16 paragraphs for your routines).

This option can also be used to shrink the GW-BASIC block in order 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 but GW-BASIC will only use the lower 32000. This leaves 768 bytes available for program space.

/D, if present, causes the Double Precision Transcendental maths package to remain resident. The functions that will be calculated in double precision if this package is resident are: ATN, COS, EXP, LOG, SIN, SQR, and TAN. If omitted, this package is discarded and the space is freed for program use. The amount of memory required by this package is approximately 3000 bytes.

/I GW-BASIC is able to dynamically allocate space required to support file operations. For this reason, GW-BASIC does not need to support the /S: and /F: switches. However, some applications are written in such a manner that certain BASIC internal data structures must be static. In order 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.

Note

number-of-files, lrecl, buffer-size, highest-memory and *max-blocksize* are numbers that may be Decimal, Octal (preceded by &O), or Hexadecimal (preceded by &H).

Examples:

A>GWBASIC PAYROLL
Uses 64K of memory and 3 files, loads and executes PAYROLL.BAS

A>GWBASIC INVENT/F:6
Uses 64K of memory and 6 files, loads and executes INVENT.BAS.

A>GWBASIC /C:0/M:32768
Disables RS232 support and uses only the first 32K of memory.

A>GWBASIC /F:4/S:512/I
Uses 4 files and allows a maximum record length of 512 bytes.

A>GWBASIC TTY/C:512
Uses 64K of memory and 3 files, allocates 512 bytes to RS232 receive buffers, load and execute TTY.BAS.

Redirection of Standard Input and Output

Under GWBASIC you can redirect your Input and Output. Generally, standard input is read from the keyboard, but this can be redirected to any file specified on the GWBASIC command line. Standard output, generally written to the screen, can be redirected to any device or file specified on the GWBASIC command line.

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

2. All PRINT statements will write to *stdout* instead of to the screen. If a pair of greater than symbols (>>) are entered before the *stdout*, data output by PRINT statements will be "appended" to the specified file.

3. Error messages go to standard output.

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

5. File output to SCRN: still outputs to the screen.

6. GW-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 BREAK will cause GW-BASIC to close any open files, issue the message Break in line nnnnn to standard output, exit GW-BASIC, and return to MS-DOS.

9. When input is redirected, GW-BASIC will continue to read from this 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 an attempt is made to read past end-of-file by an INPUT# statement, then any open files are closed. The message Read past end is then written to standard output, and GW-BASIC returns to MS-DOS.

10. Because of the way in which MS-DOS handles text files, it is not recommended to execute a program in GW-BASIC with output rerouted and appended to a file created previously, either with EDLIN or in sequential mode in GW-BASIC: commands such as TYPE will only be able to show you the original contents.

Examples:

A>GWBASIC MYPROG >MYOUT.DAT
Data read by INPUT and LINE INPUT will continue to come from the keyboard. Data output by PRINT goes into the file MYOUT.DAT.

A>GWBASIC MYPROG <MYIN.DAT
Data read by INPUT and LINE INPUT will come from MYIN.DAT. Data output by PRINT goes to the screen.

A>GWBASIC MYPROG <IN.DAT >OUT.DAT
Data read by INPUT and LINE INPUT comes from the file IN.DAT and data output by PRINT goes into OUT.DAT.

A>GWBASIC MYPROG <\SALES\ED\TRANS.DAT >>\SALES\SALES.DAT
Data read by INPUT and LINE INPUT will now come from the file TRANS.DAT that is in the \SALES\ED directory. Data output by PRINT will be appended to the SALES.DAT files in the SALES directory.

KEY statement

Sets a function key to automatically type any sequence of characters. Other options allows you to enable or disable the function key display from the 25th line, or to list the function key values.

Syntax:

KEY { OFF | ON | LIST | n , stringexp }

where

n is the key number. An expression returning an unassigned integer in the range 1 to 10.

stringexp is a string expression assigned to the key. String constants should be enclosed in quotation marks. The *stringexp* value may be up to 15 characters long. Longer strings are truncated to 15 characters.

Characteristics

The KEY statement enables the designation of a function key as a softkey. This means that you can set any function key to generate any sequence of characters.

KEY OFF erases the softkey display from the bottom line, making this line available for your GW-BASIC program. In this case, you can use LOCATE 25,1 followed by PRINT to display data on the bottom line of the screen. KEY OFF does not disable the function keys.

KEY ON causes the softkey values to be displayed on the bottom line of the screen. If the screen width is 80, all ten softkeys are displayed, but only five softkeys are displayed if the width is 40. In either case, only the first 6 characters of each key value are displayed. If fewer than the total number of function keys are displayed, you may scroll the function key display (increasing the number of the leftmost key displayed by one each time) by pressing CTRL T. ON is the default state.

KEY LIST displays all softkey values on the screen, with all 15 characters of each key displayed.

KEY *n, stringexp* sets function key *n* equal to *stringexp*. Any one or all of the ten function keys may be assigned up to a 15 byte string by KEY *n,stringexp*. When the key is pressed, the associated string will be input to GW-BASIC.

Initially, the softkeys default to the following values:

F1 - LIST
F2 - RUN (carriage return)
F3 - LOAD"
F4 - SAVE "
F5 - CONT (carriage return)
F6 - ,"LPT" (carriage return)
F7 - TRON (carriage return)
F8 - TROFF (carriage return)
F9 - KEY space
F10 - SCREEN 0,0,0 (carriage return)

Rules:

1. If the function key number is not in the range 1 to 10, an Illegal function call error is returned. The previous key string expression is retained.

2. The key assignment string may be 1 to 15 characters in length. If the string is longer than 15, characters, the first 15 characters are assigned.

3. Assigning a null string (string of length 0) to a softkey disables the function key as a softkey.

4. When a softkey is assigned, the INKEY$ function returns one character of the softkey string per invocation.

Examples:

```
40 rem display the softkeys on the bottom line.
50 KEY ON


55 rem erase softkey display.
60 KEY OFF

65 rem assigns the string "MENU" and a carriage
return to softkey 1. Such assignments might be
used for rapid data entry.
70 KEY 1,"MENU" + CHR$(13)


75 rem disables softkey 2 as a softkey.
80 KEY 2,""
```

The following routine initializes the first five softkeys:

```
1 KEY OFF ' turn off key display during init.
10 DATA "EDIT", "LET", "SYSTEM", "PRINT",
"LPRINT"
20 FOR C = 1 TO 5: READ SOFTKEYS$(C)
30 KEY C,SOFTKEYS$(C)
40 NEXT C
50 KEY ON ' now display new softkeys.
```

Defining Keys 15-20

Defining keys 15 to 20 allows you to trap any CTRL-key, SHIFT-key, or Super-SHIFT (ALT)-key. These keys are defined by the statement:

KEY *n*, CHR$(*shift*) + CHR$(*scan-code*)

where

n is an integer expression in the range 15 to 20.

shift is a numeric value corresponding to the following hex values:

CAPS LOCK	&H40 (CAPS LOCK is active)
NUM LOCK	&H20 (NUM LOCK is active)
ALT	&H08 (ALT key is pressed)
CTRL	&H04 (CTRL key is pressed)
Right SHIFT	&H01
Left SHIFT	&H02

Both the left and right SHIFT keys can be used, where values of &H01, &H02, or &H03 (the sum of hex 01 and hex 02) denote a SHIFT key.

It is also possible to add multiple shift states, such as CTRL and ALT keys together, by adding the associated shift state values.

scan-code is a decimal number in the range 1 to 83. It represents the scan code (in decimal) of the key to be trapped.

If several function keys are pressed while temporarily de-activated by an explicit or implicit KEY(*n*) STOP statement, the corresponding trap routines will be called in the order that the keys are re-activated. In the case of simultaneous re-activated of several trapped keys, they are processed in the following order:

1. CTRL PRT SC. Note that CTRL PRT SC, even if defined as a trappable key, will still produce a printed copy of the screen.

2. The function keys F1 to F10, and the cursor direction keys. Defining a function key or cursor movement key as a user-defined key trap will have no effect as they are considered predefined.

3. Finally, the user-defined keys are examined (15-20).

Any key that is trapped is not passed to GW-BASIC, i.e., it does not go into the keyboard buffer. This applies to any key, including CTRL BREAK or CTRL ALT DEL. This makes it possible to prevent GW-BASIC users from accidentally interrupting a program or rebooting the system.

ON TIMER(n) GOSUB and TIMER statements

Causes an event trap every *n* seconds.

Syntax:

ON TIMER(*n*) GOSUB *linenum*

TIMER ON
TIMER OFF
TIMER STOP

where

n is an integer expression in the range 1 through 86400 (1 second through 24 hours). Values outside this range will result in an Illegal function call error.

linenum is the entry point line number of the TIMER event trap subroutine.

The TIMER ON statement enables real time event trapping by an ON TIMER(*n*) GOSUB statement. While trapping is enabled, GW-BASIC checks between every statement to see if the time has reached the specified level. If it has, the ON TIMER(*n*) GOSUB statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent TIMER ON is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an ON TIMER(*n*) GOSUB statement will be executed as soon as trapping is enabled.

The ON TIMER(*n*) GOSUB statement will only be executed if a TIMER ON statement has been executed to enable event trapping. If event trapping is enabled, and if the *linenum* in the ON TIMER(*n*) GOSUB statement is not zero, GW-BASIC checks between statements to see if the timer has been reached. If it has, a GOSUB will be performed to the specified line.

If a TIMER OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a TIMER STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a TIMER ON statement is executed.

If an ON TIMER(*n*) GOSUB is performed, an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a TIMER ON statement unless an explicit TIMER OFF was performed inside the subroutine.

If an error trap occurs, all trapping will be disabled including ERROR trapping.

The RETURN *linenum* 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.

Example:

```
10 TIMER ON
20 ON TIMER(1) GOSUB 100
30 FOR X = 1 TO 10000:NEXT
40 CLS
50 END
100 CLS
110 LOCATE 12,30
120 PRINT TIME$
130 RETURN
```

When this program is executed, every second the screen is cleared and the time is displayed in the middle of the screen until the FOR...NEXT loop is exhausted.

RANDOMIZE statement

Reseeds the random number generator.

Syntax:

RANDOMIZE [*numexp*]

where

numexp is any numeric expression. The value of the expression will be used to seed the random numbers.

If ***numexp*** is omitted, GW-BASIC suspends program execution and asks for a value by displaying:

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

To get a new random seed without prompting the user, use the numeric TIMER function. For example:

```
RANDOMIZE TIMER
```

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 RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

See example on next page.

Examples:

```
10 RANDOMIZE
20 FOR I = 1 TO 3
30 PRINT RND,
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3
 .2226007       .594141419      .2414202
Ok
RUN
Random Number Seed (-32768 to 32767)? 4
 .628988        .765605         .5551561
Ok
RUN
Random Number Seed (-32768 to 32767)? 3
 .2226007       .594141419      .2414202
Ok
```

Note that the numbers your program produces may not be the same as the ones shown here.

REM statement

Allows explanatory remarks to be inserted in a program.

Syntax:

REM *remark*

where

remark represents a sequence of characters.

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. 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. The single quotation mark may also be entered just after the line number, like REM.

Do not use remarks in a DATA statement, because it would be considered legal data.

Examples:

```
110 rem calculate average velocity
120 FOR I = 1 TO 20
130     SUM = SUM + V(I)
140 NEXT I
```

or

```
120 FOR I = 1 TO 20
130 SUM = SUM + V(I) 'calculate average velocity
140 NEXT I
```

or

```
110 'calculate average velocity
120 FOR I = 1 TO 20
130     SUM = SUM + V(I)
140 NEXT I
```

RND function

Returns a random number between 0 and 1.

Syntax:

RND[(*numexp*)]

where *numexp* is a numeric expression which modifies the returned value.

RND returns a uniformly distributed random number in the open interval between zero and 1. Unless you write a RANDOMIZE statement before the RND function the same sequence of random numbers is generated on every run.

RND acts differently depending upon whether the *numexp* evaulates to a positive number, negative number, or zero:

RND(positive number) returns the next number in the current sequence.

RND(negative number) reseeds the random number generator and returns the first random number in the new sequence.

RND(0) returns the last random number generated, without affecting the current sequence.

The *numexp* is optional. If you do not give one, RND acts as if you had given a positive expression as an argument.

To return integer random numbers in the range 0 to N, use:

```
INT(RND * (N + 1))
```

Example:

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 12  65  86  72  79
```

SWAP statement

Exchanges the values of two variables.

Syntax:

SWAP *variable1, variable2*

where *variable1* and *variable2* are two variables of the same type (integer, single precision double precision, or string).

The two variables to be exchanged must be of the same type or a Type mismatch error occurs.

If the second variable is not already defined when SWAP is executed, an Illegal function call error will result.

Example:

```
Ok
10 A$ = " ONE " : B$ = " ALL " : C$ = "FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

After line 30 is executed, A$ has the value " ALL " and B$ has the value " ONE ".

TIME$ statement and function

The TIME$ statement sets the current time.

The TIME$ function retrieves the current time.

Syntax 1: (as a statement)

TIME$ = *stringexp*

Syntax 2: (as a function)

stringvar = TIME$

where

stringexp is a string expression indicating the time to be set.

stringvar is a string variable in which the current time (8 character string) is returned.

As a statement (to set the time):

stringexp is a string expression indicating the time in the form:

hh (sets the hour; minutes and seconds default to 00), or

hh:mm (sets the hour and minutes; seconds default to 00), or

hh:mm:ss (sets the hour, minutes and seconds)

A 24 hour clock is used; therefore 8:00 p.m. would be entered as 20:00:00.

You may omit a leading zero to specify the values of hours, minutes and seconds, but you must enter at least one digit (*see the examples below*).

Note that the time may also have been set by MS-DOS prior to entering GW-BASIC.

As a function (to retrieve the time):

The TIME$ function 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 again used.

Example:

```
Ok
TIME$ = "8:0"
Ok
PRINT TIME$
08:00:04
Ok
```

TIMER function

Returns a single precision number indicating the seconds that have elapsed since midnight or system reset.

Syntax:

```
TIMER
```

TIMER is a numeric function. It calculates fractional seconds to the nearest degree possible. It may not be used as a user variable.

Example:

```
10 FOR K = 1 TO 10
20 PRINT "TIMER = ";TIMER
30 NEXT
```

VARPTR$ function

Returns a character form of the memory address of the variable.

Syntax:

VARPTR$(*variable*)

VARPTR$ is primarily used to execute substrings with the DRAW and PLAY statements in programs that will later be compiled. With programs that will not be later compiled, the standard system of the DRAW and PLAY statements will be sufficient to produce the desired ecffects.

For example:

```
PLAY "XA$;"
```

and

```
PLAY "X" + VARPTR$(A$)
```

produce the same effect.

The *variable* must have been defined prior to the execution of the VARPTR$ function; otherwise, an Illegal function call error results. Variables are defined by executing any reference to the variable. Both numeric and string variables 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 that type specifies the type of the variable, as follows:

2 integer
3 string
4 single precision
8 double precision

Because array addresses, string addresses and file data blocks change whenever a new variable is assigned, it is unsafe to save the result of a VARPTR$ function in a variable. It is recommended that VARPTR$ is executed before each use of the result.

20. MULTIPLE DIRECTORIES

This chapter describes the following:

Directory paths

Current "working" directory

Make a directory

Change current "working" directory

Remove a directory

Directory paths

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

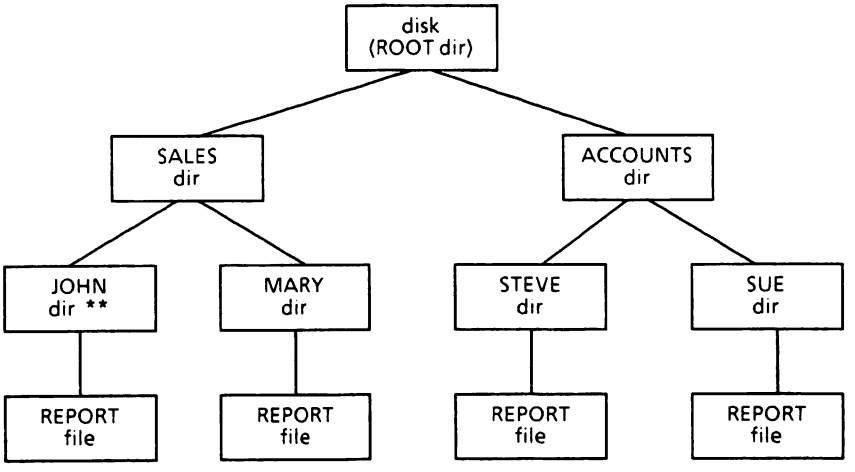
Previously, only a single directory was supported that contained all files on a disk. MS-DOS extends this concept to allow a directory to contain both files and directories and to introduce the notion of the current "working" directory.

To specify a file, you could use one of two methods, either specify a path from the root directory to the file, or specify a path from the current "working" directory to the file. A path name is a series of directory names separated by "\" and ending with a file name. A path name that starts at the root begins with the "\".

There are two special directory entries in each directory, denoted by "." and "..". They specify the directory itself (".") or the parent of the directory ("..."). The root directory's parent is itself.

Let us take a hypothetical example.

In a particular business, both sales and accounting share a computer with a hard 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:



Using a directory structure like the hierarchy above, and assuming that the current "working" directory is at point [**] (directory JOHN), to reference the REPORT under JOHN, the following are equivalent:

REPORT

\SALES\JOHN\REPORT

To refer to the REPORT under MARY, supposing that JOHN is still the current "working" directory, the following are equivalent:

..\MARY\REPORT

\SALES\MARY\REPORT

To refer to the REPORT under SUE, supposing that JOHN is still the current "working" directory, the following are equivalent:

..\..\ACCOUNTS\SUE\REPORT

\ACCOUNTS\SUE\REPORT

There is no restriction on the number of directories except in the number of allocation units available.

The root directory will have a fixed maximum number of entries for a diskette. There is no limit to the number of files and/or subdirectories in the root directory on a hard disk, other than the size of the MS-DOS partition.

Other "subdirectories" can also be accessed via the root directory, and these in turn can branch off to further files and subdirectories. The only limit being the space available on the disk.

Each directory can also contain the file and directory names that also appear in other directories.

Path names can be used with the following commands:

BLOAD	GWBASIC (*)	NAME
BSAVE	KILL	OPEN
CHAIN	LOAD	RMDIR
CHDIR	MERGE	RUN
FILES	MKDIR	SAVE

(*) Used to initialize GW-BASIC. This is an MS-DOS command (not a GW-BASIC command).

A path name may be considered as an extension of filespec and a string expression of the form:

[*device:*][\][*directory*][*directory*]...[\]*filename*

or

[*device:*][\][*directory*][*directory*]...[*directory*][\]

All characters that are valid for a file name are also valid for a directory name.

A path name may not contain more than 63 characters. Path names longer than 63 characters will give a Bad filename error.

Current "working" directory

If you enter a ***filename***, in a GW-BASIC statement or command, without specifying ***pathname***, the current "working" directory is searched. A single directory is created on a disk when it is formatted. That directory is called the "root" directory, and it is the current "working" directory, initially. You can create other directories by entering the MKDIR command, or remove directories by entering the RMDIR command. The CHDIR command allows you to change the current "working" directory. (*See this chapter for details on these commands.*)

If a ***pathname*** begins with a backslash (\), GW-BASIC starts its search from the "root"; otherwise, it starts its search from the current directory. The ***pathname*** you specify can be either a sequence of directory names starting with the "root", or with the current "working" directory. If the file belongs to the current "working" directory you only need to specify the file.

Make a directory

The MKDIR command permits the creation of a new directory on a specified disk.

Syntax:

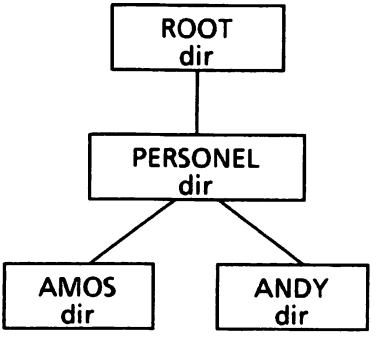
MKDIR *pathname*

where

pathname is a string expression specifying the name of the directory to be created.

Examples:

Assume your directory structure is like to this:



To create a subdirectory MARKETNG from the root on the current drive, enter:

MKDIR "MARKETNG"

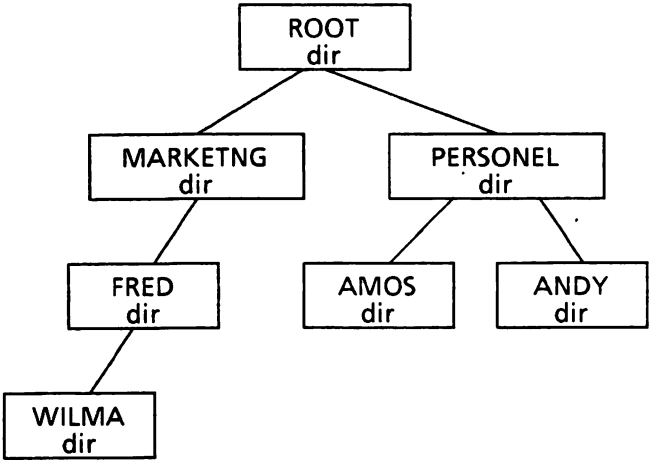
To create a subdirectory called FRED under the directory MARKETNG, enter:

MKDIR "MARKETNG\FRED"

To create a subdirectory called WILMA under the directory FRED, enter ;

MKDIR "MARKETNG\FRED\WILMA"

The directory structure will now look like this:



Possible Errors

Bad file name

Path/File Access error

Change current "working" directory

The CHDIR command changes the current "working" directory.

Syntax:

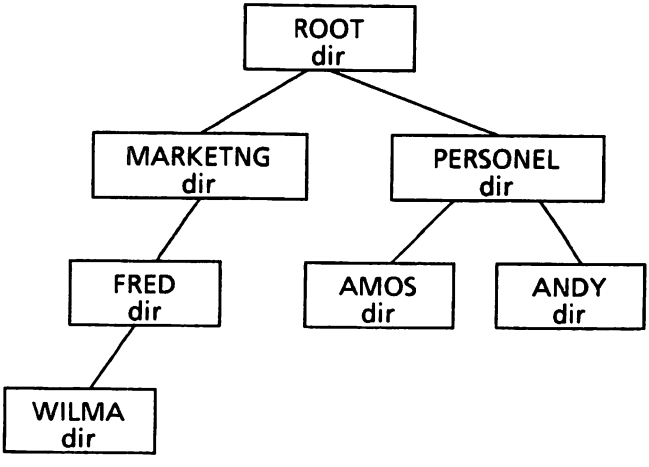
CHDIR *pathname*

where

pathname is a string expression identifying the new directory which is to be the current "working" directory.

Examples:

Assume your directory structure is like to this:



To change the current "working" directory from ROOT to PERSONEL, enter:

CHDIR "PERSONEL"

PERSONEL is now the current "working" directory.

To change the current "working" directory from PERSONEL to ANDY use:

CHDIR "ANDY"

Avoid nesting directories too deeply, as result, of using MKDIR and CHDIR repeatedly.

Possible Errors

Bad file name

Path not found

Path/File Access error

Remove a directory

The RMDIR command removes an existing directory.

Syntax:

RMDIR *pathname*

where

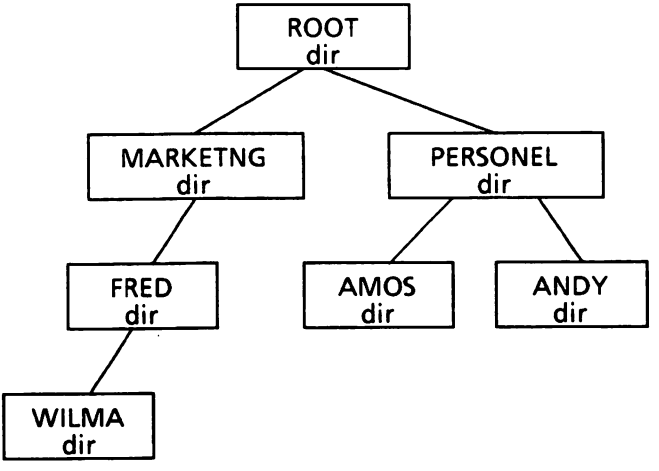
pathname is a string expression identifying the directory which is to be deleted.

RMDIR works exactly like the MS-DOS command RMDIR. The directory to be deleted must be empty of all files and subdirectories except the working directory (".") and the parent directory ("..") entries, or a Path not found error is given.

See example on next page.

Examples:

Assume your directory structure is like to this:



You decide that you no longer want the subdirectory ANDY. Assume that the current "working" directory is ROOT. Then:

RMDIR "PERSONEL\ANDY"

deletes the directory ANDY.

On the other hand, if you want to make PERSONEL the current "working" directory and remove the directory called AMOS then:

CHDIR "PERSONEL"
RMDIR "AMOS"

Possible Errors

Bad file name

Path/File Access error usually indicating that the directory is not empty.

Notes:

21. MUSIC

This chapter describes the following:

ON PLAY(*n*) GOSUB and PLAY {ON | OFF| STOP} statements

PLAY statement

PLAY(*n*) function

SOUND statement

ON PLAY(n) GOSUB and PLAY {ON|OFF|STOP} statements

Specifies the first line number of a subroutine to be executed when the music buffer contains fewer than *n* notes. This permits continuous background music during program execution.

Syntax:

ON PLAY(*n*) GOSUB *linenum*

PLAY ON
PLAY OFF
PLAY STOP

where

n is an integer expression in the range of 1 to 32. Values outside this range result in an Illegal function call error.

linenum is the first line number of the associated trap routine. A *linenum* of 0 disables play trapping.

If a PLAY ON statement is executed, PLAY(*n*) trapping will be enabled.

If a PLAY OFF statement is executed, the PLAY(*n*) trapping will be disabled.

If a PLAY STOP statement is executed, PLAY(*n*) trapping will be suspended, i.e., the GOSUB is not performed, but it will be performed as soon as a PLAY ON is executed.

If PLAY(*n*) trapping is enabled, and the background music buffer has gone from *n* to *n-1* notes, ON PLAY(*n*) GOSUB *linenum* will be executed, and the corresponding routine activated. To avoid recursive traps, a PLAY STOP is automatically executed when the trap occurs. A RETURN from the trapping subroutine will automatically perform a PLAY(*n*) ON unless an explicit PLAY(*n*) OFF was performed within the trap routine. The RETURN *linenum* form may also be used. Use this form with care, because any other active GOSUBs, WHILEs, or FORs will remain active and errors such as FOR without NEXT may result.

If the program is running, PLAY(*n*) trapping is enabled, and the background music buffer is empty, no PLAY(*n*) trapping routine will be executed.

A PLAY event trap is only effective when playing Background Music (PLAY "MB..."). PLAY event traps have no effect when running in Music Foreground (PLAY "MF...").

A PLAY event trap is ineffective if the Music Background buffer is already empty when a PLAY ON is executed.

Care should be taken in selecting values for *n*. If *n* is a large number, event traps will occur frequently enough to reduce program execution speed.

Example:

```
10 PLAY ON
20 ON PLAY(8) GOSUB 1000
      .
      .
      .
1000 REM SUB PLAY(8) TRAP
      .
      .
      .
1050 RETURN
```

PLAY statement

Plays music in accordance with a string which specifies the notes to be played, and the way in which the notes are to be played.

Syntax:

PLAY *stringexp*

where

stringexp is a string expression containing a series of single-character commands.

PLAY uses a concept similar to that in DRAW (*see the "DRAW statement" in Chapter 15*) by embedding a Music Macro Language into one statement. A set of subcommands, used as part of the PLAY statement, specifies the particular action to be taken.

The subcommands used for *stringexp* are:

COMMAND	ACTION
A-G[#|+|-]	Plays a note in the range A-G. The suffixes (#) or (+) after the note specifies sharp; suffix (-) specifies flat.
O*n*	Sets the current octave. There are seven octaves, numbered 0 through 6.
>*n*	Increments the octave and plays note *n*. The octave is progressively incremented, each time note *n* is played, until octave 6 is reached. Note *n* is subsequently played at octave 6.

COMMAND	ACTION
<*n*	Decrements the octave and plays note *n*. The octave is progressively decremented, each time note *n* is played, until octave 0 is reached. Note *n* is subsequently played at octave 0.
N*n*	Plays one of 84 notes within the 7 possible octaves. The *n* parameter ranges from 0 to 84. 0 indicates a rest. This command is an alternative to specifying notes using the note name (A-G) and octave number commands.
P*n*	Specifies a pause. The *n* parameter ranges from 1 to 64 and corresponds to the length of each note, set by L*n*.
L*n*	Sets the length of each note. The *n* parameter ranges from 1 to 64, where n = 1 is equivalent to a whole note; n = 4 is equivalent to a quarter note, etc. The length may also follow the note when a change of length only is required for a particular note. In this case, A16 is equivalent to L16A.
. (period)	A period after a note causes the note to be played 3/2 times the length determined by L multiplied by T (tempo). Multiple periods may appear after a note. The period is scaled accordingly; e.g., A. is 3/2, A.. is 9/4, A... is 27/8, etc. Periods may appear after a pause (P). In this case, the pause length may be scaled in the same way notes are scaled.

COMMAND	ACTION
T*n*	Sets the tempo, or number of quarter notes in one minute. The *n* parameter ranges from 32 to 255, with a default value of 120.
MF	Sets Music Foreground. Music (PLAY statement) and SOUND are to run in Foreground. Each successive note or sound will not start until the preceding note or sound has finished. This is the default setting.
MB	Sets Music Background. Music (PLAY statement) and SOUND are to run in Background. That is, each note or sound is placed in a buffer allowing the GW-BASIC program to continue executing while the note or sound plays in the "background". Up to 32 notes can be played in the background at a time.
MN	Sets "music normal", so that each note will play 7/8 of the time determined by 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 set by length (L).
X*string*	Executes the specified string.

The *n* parameter may be constant or variable, where a variable is written as *=variable;*. The semicolon is necessary when a variable is used in this way, or when the X command is used, but it is not allowed after MF, MB, MN, ML or MS. In all other cases, a semicolon is optional between commands.

Examples:

```
100 PLAY "<<"  'decrement by two octaves

200 PLAY ">"  'increment by an octave

300 PLAY "A>"  'increment by an octave and play
an A note

400 PLAY "XSONG$"
```

This example will play the beginning of the first movement of Beethoven's Fifth Symphony.

```
10 LISTEN$ = "T180 O2 P2 P8 L8 GGG L2 E-"
20 FATE$ = "P24 P8 L8 FFF L2 D"
30 PLAY LISTEN$ + FATE$
```

PLAY(n) function

Returns the number of notes remaining in the music background buffer.

Syntax:

PLAY(*dummy*)

where

dummy is a dummy argument. Any value may be supplied.

If the program is running Music Foreground mode, PLAY(*n*) returns 0.

If the program is running in Music Background mode, PLAY(*n*) returns the number of notes currently in the Music Background buffer. The maximum value that PLAY(*n*) may return is 32.

Example:

```
210 IF PLAY (0) = 6 GOTO 500
```

SOUND statement

Produces sound via a speaker.

Syntax:

SOUND *frequency, duration*

where

frequency is a numeric expression from 37 to 32767. It represents the frequency in Hertz.

duration is the duration in clock ticks. Clock ticks occur 18.2 times per second. ***duration*** is an integer expression from 0 to 65535.

If the ***duration*** is zero, any SOUND statement that is running will be turned off. If no SOUND statement is currently running, a SOUND statement with a ***duration*** of zero will have no effect.

See "Notes and Frequencies" on the next page.

Example:

This statement creates random sounds.

```
100 SOUND RND * 1000 + 37,2
```

Notes and Frequencies

This table displays the frequencies of musical notes (two octaves below and two octaves above middle C).

1975.5	1760.0	1568.0	1396.9	1318.5	1174.7	1046.5
B	A	G	F	E	D	C
987.77	880.00	783.99	698.46	659.26	587.33	523.25
B	A	G	F	E	D	C
493.88	440.00	392.00	349.23	329.63	293.66	261.63
B	A	G	F	E	D	C
246.94	220.00	196.00	174.61	164.81	146.83	130.81
B	A	G	F	E	D	C

Tempos and Beats/Minute

Tempos	Beats/Minute	Ticks/Beat
Larghissimo		
Largo	40-60	28.13-18.75
Larghetto	60-66	18.75-17.05
Grave		
Lento		
Adagio	66-76	17.05-14.8
Adagietto		
Andante	76-108	14.8-10.42
Andantino		
Moderato	108-120	10.42-9.38
Allegretto		
Allegro	120-168	9.38-6.7
Vivace		
Veloce		
Presto	168-208	6.7-5.41
Prestissimo	greater than 208	less than 5.41

22. NUMERIC FUNCTIONS

This following functions are described in this chapter:

ABS
ATN
COS
EXP
FIX
INT
LOG
SGN
SIN
SQR
TAN

ABS function

Returns the absolute value of a numeric expression.

Syntax:

ABS(*numexp*)

The returned value will always be positive or zero.

Examples:

```
Ok
PRINT ABS(8*(-6))
 48
Ok
```

```
110 DISTANCE = ABS(START-FINISH)
```

```
320 IF ABS(DELTA) < = LIMIT THEN STOP
```

ATN function

Returns the arctangent of the argument.

Syntax:

ATN(*numexp*)

The evaluation of ATN is performed in single precision, unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

The result is expressed in radians and falls in the range -PI/2 to PI/2 (where PI = 3.141593)

Examples:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.249046
Ok
```

```
100 IF ATN(N) < PI/2.0 THEN PRINT "ANGLE 90
DEGREES"
```

COS function

Returns the cosine of the argument.

Syntax:

COS(*numexp*)

The argument *numexp* represents the angle in radians.

The calculation of the COS function is performed in single precision, unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

Example:

```
10 X = 2 * COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

EXP function

Returns **e** (base of natural logarithms) to the power of the argument.

Syntax:

EXP(*numexp*)

numexp must be <= 87.3365. If EXP overflows, the Overflow error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXP is calculated in single precision unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

Example:

```
10 X = 5
20 PRINT EXP(X-1)
RUN
 54.59815
Ok
```

FIX function

Returns the truncated integer part of the argument.

Syntax:

FIX(*numexp*)

FIX(*numexp*) is equivalent to SGN(*numexp*) * INT(ABS(*numexp*)). The major difference between FIX and INT is that FIX does not return the next lower number for a negative argument.

Examples:

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

INT function

Returns the largest integer that is equal to, or less than the argument.

Syntax:

INT(*numexp*)

Refer to the "FIX function" in this chapter and the "CINT function" in Chapter 7, which also return integer values.

Examples:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

LOG function

Returns the natural logarithm of a positive argument.

Syntax:

LOG(*numexp*)

LOG is calculated in single precision unless /D option is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

Examples:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

SGN function

Returns 1 if the argument is positive, 0 if the argument is zero, and -1 is the argument is negative.

Syntax:

SGN(*numexp*)

If *numexp* > 0, SGN(*numexp*) returns 1.

If *numexp* = 0, SGN(*numexp*) returns 0.

If *numexp* < 0, SGN(*numexp*) returns -1.

Example:

```
Ok
10 READ X
20 PRINT "X = ";X, "SGN(X) = "; SGN(X)
30 GOTO 10
40 DATA 10,-5,0
RUN
X =  10          SGN(X) =  1
X = -5           SGN(X) = -1
X =  0           SGN(X) =  0
Out of DATA in 10
Ok
```

SIN function

Calculates the sine of the argument.

Syntax:

SIN(*numexp*)

The SIN function is calculated in single precision, unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

Example:

```
PRINT SIN(1.5)
 .9974951
Ok
```

See also the COS function in this chapter.

SQR function

Returns the square root of a positive numeric expression.

Syntax:

SQR(*numexp*)

The SQR function is calculated in single precision, unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

An Illegal function call error results if the argument is negative.

Example:

```
Ok
10 FOR X = 10 TO 25 STEP 5
20        PRINT X, SQR(X)
30 NEXT
RUN
 10              3.162278
 15              3.872984
 20              4.472136
 25              5
Ok
```

TAN function

Returns the tangent of the argument.

Syntax:

TAN(*numexp*)

numexp is a numeric expression representing the angle in radians.

The TAN function is calculated in single precision, unless /D is supplied in the GWBASIC command line (*see "GWBASIC command" in Chapter 19*).

If TAN overflows, the Overflow error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
PRINT TAN(3.14/4)
 .999204
Ok
```

23. OUTPUT TO SCREEN OR PRINTER

The following are described in this chapter:

CLS statement
COLOR (Text Mode) statement
CRSLIN function
LCOPY command
LOCATE (Text Mode) statement
LPOS function
LSET and RSET statements
POS function
PRINT and LPRINT statements
PRINT USING and LPRINT USING statements
SCREEN function
SCREEN statement
SPC function
TAB function
VIEW PRINT statement
WIDTH statement
WRITE statement

CLS statement

Erases all or part of the screen.

Syntax:

CLS [*n*]

where

n is an integer expression in the range 0 to 2.

CLS without a parameter clears the entire screen to the current text background color, unless a graphics viewport has been defined, and resets the function key line (if the function key display is enabled).

If a viewport has been defined, the current viewport only will be cleared to the graphics background color. Outputting a formfeed character (typing CTRL L or issuing the command PRINT CHR$(12) will have the save effect).

If there is a text window, and no graphics viewport (no VIEW statement in effect), then CLS will clear only the text window.

CLS 0 clears the entire screen, resetting the function key display.

CLS 1 clears the graphics viewport to the graphics background color (in one of the graphics modes). If no viewport has been defined, this will have no effect.

CLS 2 clears the text window to the text background color, without resetting the function key display.

CLS not only erases all or part of the screen, but also returns the cursor to the upper left-hand corner of the screen in Text Mode.

If you are in Graphics Mode, CLS makes the "last referenced point" the center of the screen.

The screen can also be cleared by pressing CTRL HOME, or by modifying the screen mode using the SCREEN statement, or the width using the WIDTH statement.

Examples:

```
10 CLS  'clears the screen (or the current viewport
or the text window)

60 CLS 0 'clears whole screen

90 CLS 1 ' clears the graphics viewport to graphics
background color

110 CLS 2  'clears the text window to text
background color.
```

COLOR (Text Mode) statement

Sets the text foreground and background colors in Text Mode.

Syntax:

COLOR [*foregrnd*][, [*backgrnd*][, *dummy*]]

where

foregrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 31. It selects the character foreground color.

backgrnd is a numeric expression rounded to the nearest integer. It must be in the range 0 to 15, but it is interpreted modulo 8, thus only values from 0 to 7 are taken into consideration.

dummy allows for compatibility with other BASICs. It will have no effect. It may specify border color on other systems.

Characteristics (Color Text Mode)

If you enable color (*see the SCREEN statement in this chapter*) or the color hardware is installed (standard monitor), the following colors are allowed for *foregrnd*:

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	White	15	High-intensity White

To make characters blink for a specific color, you should set *foregrnd* equal to 16 plus the color number.

Only colors 0 through 7 are allowed for *backgrnd*.

Characteristics (B/W Text Mode)

In a monochrome system, the following values can be used for *foregrnd*:

0	Black
1	Underline character with white foreground
2-6	Shades of gray
7	White

Adding 8 to the number of the selected color, you will get the color in high-intensity. For example, 15 will be high-intensity white. It is not possible to get high-intensity black.

To make characters blink, add 16 to the number of the desired color.

The following values are allowed for *backgrnd*:

0-1	Black
2-6	Shades of gray
7	White

Remarks

Foreground color may be the same as the background color. In this case, any character displayed is invisible.

Any parameter can be omitted. If a parameter is omitted, the previous value is retained.

Upon initialization, the default values are:

foregrnd = 7 (white)

backgrnd = 0 (black)

That is, if no COLOR statement exists in your program, the system assumes: COLOR 7,0.

Examples:

This sets a black foreground on a green background in color mode and a black foreground on a black background, i.e., invisible characters, in B/W mode.

```
100 COLOR 0,2
```

This sets a high-intensity white on a blue background in color mode, and a high-intensity white on a black background in B/W mode.

```
150 COLOR 15,1
```

Possible Errors

If you enter a comma (,) at the end of a COLOR statement, a Missing operand error is returned. For example:

```
COLOR 2,
```

is invalid.

Any parameters outside the specified ranges will result in an Illegal function call error. In this case, previous values are retained.

CSRLIN function

Returns the current line (row) position of the cursor.

Syntax:

```
CSRLIN
```

CSRLIN returns a value in the range 1 to 25. To return the current column position use the POS function. (*See the POS function in this chapter.*)

Example:

```
Ok
5 CLS 'clears screen
10 Y = CSRLIN 'record current line
20 X = POS(0) 'record current column
30 LOCATE 12,38 'move cursor
40 PRINT "HELLO"
50 LOCATE X,Y 'restore cursor position to old line
and column
RUN
```

(The screen is cleared, HELLO is displayed in the middle of the screen, and Ok displays at the top of the screen with the cursor on the next line. Use the CLS command to clear this screen.)

```
Ok
```

LCOPY command

Dumps the screen (text and graphics) to the line printer. The MS-DOS GRAPHICS command must be executed before entering GW-BASIC to allow you to dump graphics.

Syntax:

LCOPY [*n*]

where

n is a dummy argument. Any value may be supplied. This parameter is allowed only for compatibility with other BASICs, where it may specify to copy either text or graphics.

LOCATE (Text Mode) statement

Moves the cursor to the specified position on the active page. LOCATE may also turn the cursor on and off and define the size of either the user cursor, or both the user and overwrite cursors.

Syntax:

LOCATE [*row*][, [*column*][, [*cursor*][, [*start*][, *stop*]]]]

where

row is the screen line number. A numeric expression returning an unsigned integer in the range 1 to 25.

column is the screen column number. A numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending upon screen width.

cursor is a boolean value indicating whether the user cursor is visible or not. A 0 (zero) value turns the user cursor off, a nonzero value (say 1) turns the cursor on.

start is a numeric expression whose integer value represents the cursor top (starting) scanline. If *start* is in the range 0-31, *start* and *stop* will affect the overwrite cursor. If *start* has a larger value, it will be interpreted modulo 32, and *start* and *stop* will change the size of the user cursor.

stop is a numeric expression whose integer value represents the bottom (*stop*) cursor scanline. If this parameter is omitted, and *start* is given, *stop* defaults to the same value as *start*, and the height of the cursor will be one scanline.

In GW-BASIC, there are three cursors.

- The insert-mode cursor which appears when insert-mode is in effect.
- The overwrite cursor which appears when overwrite mode is in effect (during command entry and input with the INPUT statement).
- The user cursor which appears during program execution when an INPUT statement is not being executed.

The overwrite cursor is the one which appears most of the time.

The overwrite cursor is initialized to a 2-scanline underline.

The insert-mode cursor is initialized to a half-height block.

The user cursor is initially disabled (but its size is initialized to a full-size block).

The insert-mode cursor has a fixed size. The sizes of the overwrite and user cursors may be changed.

Following a LOCATE statement, I/O statements to the screen begin placing characters at the specified location. The user cursor is normally off during program execution, but can be turned back on using LOCATE,,1.

Note that *start* and *stop* parameters enable you to define the size of the cursor by indicating the starting and ending scanlines. The scanlines are numbered from 0 at the top of the character position. The bottom scanline is 7 if a color monitor has been installed and 13 if a B/W monitor is used. If you specify *start* and omit *stop*, this assumes the value of *start*.

Normally, GW-BASIC will not print to line 25 because of the softkey display. This can be turned off, however, using KEY OFF; then use LOCATE 25,1:PRINT... to display characters on line 25. PRINT statements on line 25 must end with a semicolon; otherwise, the screen may scroll under certain circumstances.

Also, the following sequence of statements will result in a 25-line scrolling display, without any function key line:

```
KEY OFF
VIEW PRINT
```

Any parameter may be omitted, and will then assume the current value.

Any values entered outside of the ranges indicated will result in an Illegal function call error. Previous values are retained.

Examples:

```
100 LOCATE 1,1
```
(Moves the cursor to the home position in the upper left-hand corner.)

```
200 LOCATE,,1
```
(Makes the cursor visible, its position remains unchanged.)

```
300 LOCATE,,0
```
(Turns both the user and overwrite cursors off. This is useful during a program which displays text or graphics and only uses INPUT to input keyboard data (INPUT uses the screen editor).)

```
400 LOCATE 6,1,1,0,7
```
(Moves the overwrite cursor to line 6, column 1. Makes the cursor visible, covering the entire character cell, starting at scan line 0 and ending on scanline 7 (if a color monitor is installed).)

LOCATE,,1,13
(Makes the overwrite cursor visible, its position remains unchanged, its shape will be a thin horizontal line at the bottom of the character cell (in monochrome).)

LOCATE,,1,45
(Makes the user cursor visible, its position remains unchanged, its shape will be a thin horizontal line at the bottom of the character cell (in monochrome).)

LPOS function

Returns the current position of the print head within the printer buffer.

Syntax:

LPOS(*printer*)

where

printer is an integer expression whose value (1, 2, or 3) indicates which printer is to be tested (LPT1:, LPT2:, or LPT3:).

LPOS does not necessarily give the physical position of the print head.

Example:

When the print head is greater than 30, a carriage return is executed.

```
10 FOR I = 1 TO 100
20     LPRINT I;
30     IF LPOS(1) > 30 THEN LPRINT CHR$(13)
40 NEXT
```

LSET and RSET statements

LSET left-justifies a string value in a string variable.

RSET right-justifies a string value in a string variable.

Syntax 1:

LSET *stringvar* = *stringexp*

Syntax 2:

RSET *stringvar* = *stringexp*

where

stringvar represents a non-fielded string variable.

stringexp represents the string to be left- or right-justified in a given field.

Example:

This program right-justifies the string N$ in a 20-character field. This can be very handy for formatting displayed/ printed output.

```
Ok
10 N$ = "SHARON"
20 A$ = SPACE$(20)
30 RSET A$ = N$
40 PRINT A$
RUN
                    SHARON
Ok
```

POS function

Returns the current horizontal (column) position of the cursor.

Syntax:

POS(*dummy*)

where

dummy is a dummy argument. Any value is accepted.

The current horizontal (column) position of the cursor is returned. The leftmost position is 1. The rightmost position may be 40 or 80, depending on the current screen width. To return the current vertical line position of the cursor, use the CSRLIN function (*see CSRLIN function and the LPOS function in this chapter*).

Example:

When the print head or cursor position is greater than 30, a carriage return is executed.

```
10 FOR I = 1 TO 25
20   PRINT I;
30   IF POS(X) > 30 THEN PRINT CHR$(13)
40 NEXT
```

PRINT and LPRINT statements

PRINT displays data on the screen.

LPRINT prints data on a printer. It assumes a 132-character wide printer.

Syntax:

PRINT [*list-of-expressions* separated by commas or semicolons or blanks]

LPRINT [*list-of-expressions* separated by commas or semicolons or blanks]

where

list-of-expressions may be numeric and/or string expressions. String constants must be enclosed in quotation marks. Each expression should be separated from the next by a comma, semicolon, or blank.

If *list-of-expressions* is omitted, a blank line is displayed/printed.

If *list-of-expressions* is included, the values of the expressions are displayed/printed.

When entering a program line, a question mark may be used in place of the word PRINT in a PRINT statement. It will be interpreted as the word "PRINT" and will appear as "PRINT" in subsequent listings. This expansion may cause the end of a line to be truncated if the line length is close to 255 characters.

Print Positions

The position of each displayed/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 displayed/printed at the beginning of the next zone. A semicolon causes the next value to be displayed/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/LPRINT statement begins displaying/printing on the same line, spacing accordingly. If the *list-of-expressions* terminates without a comma or a semicolon, a carriage return is entered at the end of the line and the next PRINT/LPRINT statement is displayed/printed on the next line. If the displayed/printed line is wider than the screen/printer width, GW-BASIC goes to the next physical line and continues displaying/printing.

Displayed/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 7 or fewer digits in the unscaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-17 is output as .0000000000000001 and 1D-16 is output as 1D-16.

Examples:

NOTE: *In these examples, the* **PRINT** *statement is used. But it could be replaced with the* **LPRINT** *statement if the data was to be output to a printer.*

In this example, the commas in the PRINT statement cause each value to be displayed at the beginning of the next print zone.

```
10 X = 5
20 PRINT X + 5, X-5, X*(-5), X^5
RUN
 10          0          -25         3125
Ok
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be displayed on the same line and line 40 causes a blank line to be displayed.

```
10 INPUT X
15 IF X = 999 THEN END
20 PRINT X "Squared is" X^2 "and";
30 PRINT X "Cubed is" X^3
40 PRINT
50 GOTO 10
RUN
? 9
 9 Squared is 81 and 9 Cubed is 729

? 999
Ok
```

In this example, the semicolons in the PRINT statement causes the values to be displayed on the same line. In line 30, a "?" is used instead of the word PRINT.

```
10 FOR X = 1 TO 5
20  J = J + 5:K = K + 10
30  ?J;K;
40 NEXT X
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

PRINT USING and LPRINT USING statements

PRINT USING displays data using a specified format on the screen.

LPRINT USING prints data using a specified format on the printer. It assumes a 132-character wide printer.

Syntax:

PRINT USING *format-string; list-of-expressions* separated by commas or semicolons or blanks.

LPRINT USING *format-string; list-of-expressions* separated by commas or semicolons or blanks.

where

format-string is a string expression (usually a constant or variable) composed of special formatting characters. These formatting characters (see below) determine the field and the format of the displayed/printed strings or numbers.

list-of-expressions is comprised of the string expressions or numeric expressions that are to be displayed/printed, separated by commas, semicolons, or blanks. String contants must be enclosed in quotation marks. If a comma or semicolon terminates the list of expressions, the next PRINT or PRINT USING statement begins printing on the same line, spacing accordingly.

NOTE: In the examples on the following pages, the PRINT USING statement is used. It could be replaced with the LPRINT USING statement if the data was to be output to a printer.

String Fields

When PRINT USING/LPRINT USING is used to display/print strings, one of three formatting characters may be used to format the string field:

- The "!" specifies that only the first character in the given string is to be displayed/printed.

- The "\n *spaces*\" specifies that 2+n characters from the string are to be displayed/printed. If the backslashes are typed with no spaces, two characters will be displayed/printed; with one space, three characters will be displayed/ 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. For example:

```
10 A$ = "LOOK":B$ = "OUT"
20 PRINT USING "!";A$;B$
30 PRINT USING "  ";A$;B$
40 PRINT USING "   ";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
Ok
```

(*Note: In line 30, there are two spaces between the backslashes. In line 40, there are three spaces between the backslashes.*)

- The ampersand sign (&) specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

```
Ok
10 A$ = "LOOK":B$ = "OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
Ok
```

Numeric Fields

When PRINT USING/LPRINT USING is used to display/print numbers, the following special characters may be used to format the numeric field:

- A number sign (#) is used to represent each digit position. Digit positions are always filled. If the number to be displayed/printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

  If the number of digit positions specified in the format string exceeds 24, an Illegal function call error will result.

- 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 displayed/printed (as 0, if necessary). Numbers are rounded as necessary.

```
Ok
PRINT USING "##.##";.78
 0.78
Ok
PRINT USING "###.##";987.654
 987.65
Ok
Print Using "##.##   ";5.3,66.789,.234
 5.30   66.79   0.23
Ok
```

  In the last example, three spaces were inserted at the end of the format string to separate the displayed 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 displayed/printed before or after the number. (*See examples in the minus sign description.*)

- A minus sign (-) at the end of the format field will cause negative numbers to be displayed/printed with a trailing minus sign.

```
Ok
Print Using " + ##.##   ";-68.95,2.4,-.9
-68.95   + 2.40   -0.90
Ok
Print Using "##.##-   ";-68.95,22.449,-7.01
68.95-   22.45   7.01-
Ok
```

- 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.

```
Ok
Print Using "**#.#   ";12.39,-0.9,765.1
 *12.4   *-0.9   765.1
Ok
```

- A double dollar sign ($$) causes a dollar sign to be displayed/printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
Ok
Print Using "$$###.##";456.78
 $456.78
Ok
```

- The **$ at the beginning of a format string combines the effects of the previous two symbols. Leading spaces will be asterisk-filled and a dollar sign will be displayed/printed before the number. **$ specifies three more digit positions, one of which is the dollar sign. The exponential format cannot be used with **$. When negative numbers are displayed/printed, the minus sign will appear immediately to the left of the dollar sign.

```
Ok
Print Using "**$##.##";2.34
 ***$2.34
Ok
```

- A comma (,) that is to the left of the decimal point in a formatting string causes a comma to be displayed/printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is displayed/printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential format.

```
Ok
PRINT USING "####,.##";1234.5
 1,234.50
Ok
PRINT USING "####.##,";1234.5
 1234.50,
Ok
```

- Four carats (^^^^) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+XX or D + XX to be displayed/printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to display/print a space or a minus sign. See examples on next page.

```
Ok
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok
PRINT USING ".####^^^^-";-888888
 .8889E+06-
Ok
PRINT USING " + .##^^^^";123
 + .12E+03
Ok
```

- An underscore (_) in the format string causes the next character to be output as a literal character.

  The literal character itself may be an underscore by placing two underscores together "_ _" in the format string.

```
Ok
PRINT USING "_!##.##_!";12.34
!12.34!
Ok
PRINT USING "_ _##.##_ _";12.34
  12.34
Ok
```

- If the number to be displayed/printed is larger than the specified numeric field, a percent sign (%) is displayed/printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be displayed/printed in front of the rounded number.

```
Ok
PRINT USING "##.##";111.22
%111.22
Ok
PRINT USING ".##";.999
%1.00
Ok
```

SCREEN function

Returns either the ASCII code (0-255) or the color number for the character at the specified screen location, depending on the value of a given condition.

Syntax:

SCREEN(*row, column*[, *condition*])

where

row is a numeric expression returning an unsigned integer in the range 1 to 25.

column is a numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80 depending on the screen width.

condition is a valid numeric, relational or logical expression returning a boolean result (0 or 1). If condition is given a non-zero, the color number for the character is returned instead of the ASCII code.

The SCREEN function returns zero if the system is in graphics mode and the specified screen location contains graphics data.

If you enter a value outside the above mentioned ranges, an Illegal function call error is returned.

Refer to Appendix A for a complete list of ASCII codes.

Examples:

```
100 X = SCREEN(10,10) 'If the character at 10,10 is A then return 65.

110 X = SCREEN(1,1,1) 'returns the color number of the character in the upper left hand corner of the screen.
```

SCREEN statement

Sets the specifications for the display screen.

Syntax:

SCREEN [*mode*][, [*burst*][, [*apage*][, *vpage*]]]

where

mode is a numeric expression resulting in an integer value in the range 0 to 255. It defines either Text Mode (0), Medium Resolution Graphics Mode (1), High Resolution Graphics Mode (2), or Super Resolution Graphics Mode (3 to 255).

burst is a numeric expression resulting in an integer value of 0 or 1. It enables color on a color TV set. In Text Mode, a 0 value disables color, and a 1 value enables color. In Medium Resolution Graphics Mode, a 0 value enables color, and a 1 value disables color. Both in High Resolution and Super Resolution Graphics Modes, the *burst* value is ignored, as these two modes only support monochrome.

For a standard monitor, this parameter has no meaning.

apage (Text Mode only) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the active page, i.e., the page to be written to by output statements to the screen. If omitted, the active page defaults to 0.

vpage (Text Mode only) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the visual page, i.e., the page to be displayed on the screen which may be different from the active page. If you omit this parameter, the visual page will default to the active page.

Mode and Burst Parameters

In the following table, the first two columns are the *mode* and *burst* parameters of a SCREEN statement.

The *burst* parameter enables color on color TV sets. For systems with standard monitors, this parameter has no real meaning. For example, a *burst* value of 0 or 1 in medium resolution mode will have the same effect if a color monitor is used; likewise, it will have the same effect if a monochrome monitor is used (in this case the four colors will appear as shades of gray).

mode	*burst*	Description
0	0	80 column x 25 row B/W Text Mode
0	1	80 column x 25 row Color Text Mode
1	0	320 hor. pixels x 200 vert. pixels Color Medium Resolution Graphics Mode (40 column x 25 row)
1	1	320 hor. pixels x 200 vert. pixels B/W Medium Resolution Graphics Mode (40 column x 25 row)
2	x (ignored)	640 hor. pixels x 200 vert. pixels B/W High Resolution Graphics Mode (80 column x 25 row)
3-255	x (ignored)	640 hor. pixels x 400 vert. pixels B/W Super Resolution Graphics Mode (80 column x 25 row)

Default Values

If you do not enter a SCREEN statement, the system assumes the following default values:

mode = 0 (Text Mode)
burst = 0 (B/W)
apage = 0 (active page 0)
vpage = 0 (visual page 0)

It would be the same, if you entered:

```
SCREEN 0,0,0,0
```

The SCREEN statement must precede any I/O statement to the screen, but you can use more than one SCREEN statement to define different screen attributes for different sections of your program.

apage and vpage Parameters

If Text Mode is selected, you can specify two more parameters (*apage* and *vpage*) to select the active and visual page. There are eight display pages (numbered 0 to 7) in 40-column Text Mode, and four display pages (numbered 0 to 3) in 80-column Text Mode. Only one display page is available in any of the three graphics modes.

Only one cursor is shared between the pages, thus, if you select a new active page, you must save the cursor position (by POS(0) and CSRLIN) before changing to the new page. If you return to the original active page, you must restore the cursor position by the LOCATE (Text) statement. If you use the SCREEN statement only to change the pages, you can omit the first two parameters (*mode* and *burst*).

Screen Width

At initialization the width is 80 columns, thus you should use the WIDTH statement to select a 40-column screen. If you select the medium resolution mode by the SCREEN statement, this also causes the number of columns to be 40 without using the WIDTH statement.

While in Text Mode, the WIDTH statement may be used to select between the 40-column mode and the 80-column mode. Likewise, the WIDTH statement may be used to select between modes 1 and 2 (medium or high resolution mode).

Selecting Text Mode (*mode*=0) after selection of one of the graphics modes will select either a 40-column screen or an 80-column screen, depending on the width used in the graphics mode. For example:

```
SCREEN 1  'set screen to medium res. mode (WIDTH = 40)
SCREEN 0  'changes screen to 40x25 Text Mode
```

See the WIDTH statement in this chapter.

Remarks

If all parameters are valid, the new screen mode is saved, the screen is erased, the foreground and the background colors are set to their default values.

If all parameters are identical to the preceding ones, nothing is altered.

If you omit a parameter, it assumes the preceding value except for the visual page that defaults to the active page.

Examples:

```
10 SCREEN 0,1,0,0      'select text mode with color,
                       'active and visual page to 0.

20 SCREEN,,1,2         'mode and color burst unchanged,
                       'use active page 1,
                       'visual page 2.

30 SCREEN 2            'switch to high res. graphics mode.

40 SCREEN 1,1          'switch to medium res. color graphics.

50 SCREEN ,0           'medium res. graphics, color off.
```

Possible Errors

If you enter a value outside the specified ranges, an Illegal function call error is returned.

SPC function

Skips spaces in a PRINT, LPRINT, or PRINT# statement.

Syntax:

SPC(*n*)

where

n is an integer expression from 0 to 255. It specifies the number of spaces to be inserted in the output line.

SPC may only be used with PRINT, LPRINT and PRINT# statements.

If *n* is greater than the defined width, then the value used is *n* MOD *width*.

A semicolon (;) is assumed to follow the SPC function; thus GW-BASIC does not add a carriage return, if you enter an SPC function at the end of a list of data.

If *n* is outside the specified range, an Illegal function call error is returned.

Example:

```
Ok
PRINT "OVER" SPC(15) "THERE"
OVER            THERE
Ok
```

See also the SPACE$ function in Chapter 26.

TAB function

Tabs the cursor or the print head to a specified position, PRINT, LPRINT, or PRINT# statements.

Syntax:

TAB(*n*)

where

n is an integer expression from 1 to 255.

If the current cursor or print position is already beyond the specified value n, TAB goes to that position on the next line.

Space 1 is the leftmost position, and the rightmost position is the width minus one.

If the value of *n* exceeds the defined width, the modulo operation is applied. For example, PRINT TAB(243) on a 40-column screen is the same as PRINT TAB(3), because 243 MOD 40 = 3.

A semicolon is assumed to follow the TAB function, thus GW-BASIC does not add a carriage return if you enter a TAB function at the end of a list of data.

Example:

```
10 PRINT "Account" TAB(25) "Amount"
15 PRINT
20 READ ACCT$,AMT$
30 PRINT ACCT$ TAB(25) AMT$
40 DATA "G. T. JONES","$25.00"
RUN
Account                   Amount

G. T. JONES               $25.00
Ok
```

VIEW PRINT statement

Sets the boundary of the text window.

Syntax:

VIEW PRINT *[line1* TO *line2]*

where

line1 is the top line of the text window.

line2 is the bottom line of the text window.

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:

VIEW PRINT 1 TO 5

creates a text window of 5 lines on the top of the screen.

WIDTH statement

Sets the line width in characters. GW-BASIC adds a carriage return after outputting the specified number of characters.

Syntax 1:

WIDTH [LPRINT] *size*

Syntax 2:

WIDTH*#filenum, size*

Syntax 3:

WIDTH *device, size*

where

size is an integer expression in the range 0 to 255. It specifies the new width.

filenum is the number under which the file was opened.

device is a string expression indicating the device that is to be used. Valid devices are: SCRN:, LPT1:, LPT2:, LPT3:, COM1:, COM2:, COM3:, or COM4:.

WIDTH LPRINT *size*

Sets the line width at the line printer.

WIDTH *size* or WIDTH "SCRN:",*size*

Sets the screen width (in Text Mode), selects a text window or changes mode (in Graphics mode). Changing the screen or text window width, or the mode, causes the screen to be cleared.

In Text Mode, *size* may only have the values 40 or 80, selecting either a 40-column or an 80-column screen.

In Graphics Mode you can either change mode or select a text window to the left of the screen of width less than or equal to 40 (Medium Resolution Mode) or less than or equal to 80 (High or Super Resolution Mode).

The width of the function key display will correspond to the selected width. If the number of columns displayed is less than 80 columns, a CTRL T may be entered to scroll the function key display horizontally.

The table on the next page summarizes all possible cases.

WIDTH #*filenum*, *size*

If the file is open, the width is immediately changed to the specified *size*. This allows the width to be changed while the file is open.

WIDTH *device*, *size*

The default line width for the specified device is set to *size*. The line widths of currently open files are not modified.

Stores the new *size* without changing the current *width*, if the device is already open. A subsequent OPEN device FOR OUTPUT AS #n will use the specified value for width initially.

IF *mode* is ...	AND *size* is ...	THEN you ...
0 (text)	40	select a 40-column screen
	80	select an 80-column screen
1 (medium-res)	80	place the system in high-resolution (mode 2)
	8< = *size*< = 40	create a text window of width *size*
2 (high res)	40	place the system in medium resolution (mode 1) with *burst* in whatever state the system was when a text or medium resolution mode was last used
	8< = *size*< = 39 or 41< = *size*< = 80	create a text window of width *size*
	size = 4	create a text window of width 40
3-255 (super res)	8< = *size*< = 80	create a text window of width *size*
	8< = *size*-80< = 80	create a text window of width *size*

When the WIDTH statement causes a change in the screen mode, colors are set to their default values.

You should turn the function key display off when changing the window width by a KEY OFF statement; otherwise, if the width is decreased, part of the old (wider) function key display may be left on the screen.

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. WIDTH 255 is the default for communications files.

If you alter the width for a communications file, you do not modify the receive or the transmit buffer: GW-BASIC will insert a carriage return after the number of characters, equal to the specified size, has been received or sent.

If *size* is outside the specified ranges, an Illegal function call error is returned. The previous value is retained.

See examples on next page.

Examples:

```
10 Print "abcdefghijklmnopqrstuvwxyz"
RUN
abcdefghijklmnopqrstuvwxyz
Ok
WIDTH 18          This changes line width to 18
                  characters.
Ok
RUN
abcdefghijklmnopqr
stuvwxyz
Ok
WIDTH 255         This changes the line width
                  back to 255 characters.
Ok
```

```
10 WIDTH "LPT1:",5
20 OPEN "LPT:" FOR OUTPUT AS 1
30 PRINT 1, "1234567890"
35 PRINT 1
40 WIDTH 1, 6
50 PRINT 1, "1234567890"
RUN
```

will yield on the printer

```
12345
67890

123456
7890
```

SCREEN 1,0	set screen to medium res. color graphics
WIDTH 80	change screen to high res. graphics
WIDTH 40	changes screen back to medium res.

SCREEN 0,1	changes screen to 40x25 text color mode
WIDTH 80	changes screen to 80x25 text color mode

WRITE statement

Writes data to the screen.

Syntax:

WRITE [*list-of-expressions*]

where

list-of-expressions is a list of numeric and/or string expressions. They must be separated by commas.

The values of the expressions are output to the screen. If no expression is indicated, a blank line is output.

Each item displayed is separated from the last by a comma. Strings are delimited by quotation marks. Numeric values are displayed using the same format as the PRINT statement, but they are not followed by blanks. After the last item in the list is displayed, GW-BASIC inserts a carriage return, line feed.

Example:

```
Ok
10 A = 80:B = 90:C$ = "THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,"THAT'S ALL"
Ok
```

24. PROGRAM INTERRUPTS

The following are described in this chapter:

Manual interrupts

Automatic interrupts

Programmable interrupts

Manual interrupt

If you press CTRL BREAK, the program is interrupted, Break in nnnnn message displays, GW-BASIC enters command level and displays Ok.

CTRL BREAK does not close any data files.

You can resume execution by entering a CONT command (*see CONT command in this chapter*). You can display program variables by direct PRINT or PRINT USING statements or change their values by direct LET or SWAP statements. You can also display program lines by an EDIT or LIST command, and modify them.

If you modify lines, you cannot continue execution via a CONT command. You can only rerun the program by entering RUN.

Automatic interrupt

Syntax Error

If a Syntax error is found, the program is interrupted, GW-BASIC displays the error message, enters command level, and displays the line that caused the error positioning the cursor under the first digit of the line number.

You can modify the line, and then rerun the program by entering RUN. You cannot continue execution by entering CONT.

If you want to examine the contents of some variables before making any modifications, you should press CTRL BREAK to return to command level. After examining the contents of the variables, you can edit the line and run the program. For example:

```
10 A = 2$6
RUN
Syntax error in 10
Ok
10 A = 2$6
```

Other Errors

If an error other than a Syntax error is found, the program is interrupted, GW-BASIC displays the error message, enters command level, and displays Ok.

You can either display program variables or display program lines by an EDIT or LIST command and then modify them. You cannot continue execution by entering a CONT command, but you can rerun the program by entering RUN. For example, running a program which contains:

```
100 FOR K =
```

will cause:

```
Missing operand in 100
Ok
```

Programmable interrupts

The END, STOP and SYSTEM statements can be use as program interrupts.

END statement

Terminates program execution, closes all open data files, and returns to command level.

Syntax:

```
END
```

END statements may be placed anywhere in the program to terminate execution.

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

An END statement at the end of a program is optional.

GW-BASIC always returns to command level after an END is executed.

Example:

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

STOP statement

Interrupts program execution then returns to command level.

Syntax:

```
STOP
```

A STOP statement may be used anywhere in a program.

When a STOP is encountered, the following message is displayed:

```
Break in nnnnn
```

The STOP statement does not close files, unlike the END statement.

GW-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example:

```
Ok
10 INPUT A,B,C
20 K = A^2*5.3:L = B^3/.26:PRINT L
30 STOP
40 M = C*K + 100:PRINT M
RUN
? 1,2,3
 30.76923
Break in 30
Ok
CONT
 115.9
Ok
```

CONT command

Resumes program execution after a CTRL BREAK has been typed or a STOP or END.

Syntax:

CONT

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 not be used to continue execution after an error has occurred. CONT is also invalid if the program has been modified during the break.

Example:

```
10 INPUT A, B
20 TEMP = A*B
30 STOP
40 FINAL = TEMP + 300:PRINT FINAL
RUN
? 32, 2.4
Break in 30
Ok
PRINT TEMP
 76.8
Ok
CONT
 376.8
Ok
```

SYSTEM command

Closes all open data files and returns control to MS-DOS.

Syntax:

```
SYSTEM
```

When a SYSTEM command is executed, all open files are closed, the current program is lost, and control is returned to MS-DOS.

If GW-BASIC has been entered through a Batch file from MS-DOS, SYSTEM returns control to the Batch file.

Notes:

25. PROGRAM HANDLING

This chapter describes

Automatic program line numbers
Clear memory
Display/print contents of a program
Execute a program in memory
Renumber program lines
Save a program to disk
Send a program listing to a device or file

Automatic program line numbers

When entering a new program or adding program lines to an existing program, you can use the AUTO command to automatically enter the program line numbers. AUTO is used only in direct mode.

Syntax:

AUTO [*linenum*][, [*increment*]]

AUTO generates a line number automatically after every carriage return.

If given, AUTO begins numbering with *linenum* and increments each subsequent line number by the *increment*.

AUTO 100,50	generates line numbers 100, 150, 200, 250, . . .

The default for both values is 10.

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

If a *linenum* is given but no *comma* or *increment* is specified, line numbering begins with the *linenum* specified and increments by 10.

AUTO 100	generates line numbers 100, 110, 120, 130, . . .

If a *comma* and *increment* are specified but no *linenum* is given, line numbering starts with 0 and increments as specified.

AUTO ,5	generates line numbers 0, 5, 10, 15, . . .

If *linenum* is followed by a *comma* but no *increment* is specified, the last *increment* specified in an AUTO command is assumed.

```
AUTO 50,
```

If the last command was AUTO 10,20, this would generate line numbers 50, 70, 90, 110, . . .

If no preceding AUTO command was given, an *increment* of 10 is assumed.

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

```
50*
```

You can use the command **AUTO .** to start line numbering with the current line in memory.

```
AUTO .,5
```

If the current line number was 100, this would generate line numbers 100*, 105, 110, . . .

If a carriage return is entered immediately after a program line number, numbering will resume there.

To exit the AUTO command, press the CTRL C or CTRL BREAK. The line in which CTRL C or CTRL BREAK is pressed is not saved. The system returns to command level.

Clear memory

The NEW command deletes the program currently in memory and clears all variables so that you may enter a new program.

Syntax:

NEW

NEW is entered at command level (direct mode) to clear memory before entering a new program.

GW-BASIC always returns to command level after a NEW command is executed.

NEW closes all files and switches off the trace flag in the same way as TROFF.

Example:

```
Ok
NEW
Ok
```

Display/print a program listing

LIST displays all or part of the program currently in memory on the screen or sends it to a specified file or device (*see "Send program listing to a device or file" in this chapter*).

LLIST prints all or part of the program currently in memory to the printer. It assumes a 132-character wide printer.

After execution, GW-BASIC returns to command level.

Syntax 1:

```
LIST [linenum]
LLIST [linenum]
```

Syntax 2:

```
LIST [linenum1] - [linenum2]
LLIST [linenum1] - [linenum2]
```

where

linenum is the program line to display/print.

linenum1 is the beginning program line to display/print.

linenum2 is the ending program line to display/print.

Syntax 1

If *linenum* is omitted, the entire program is displayed/printed beginning at the lowest line number. Listing is terminated either by the end of the program or by pressing CTRL BREAK.

If *linenum* is included, only the specified program line will be displayed/printed.

You may use a period (.) for the *linenum* to indicate the current line.

Examples:

LIST or LLIST

Displays/prints entire program currently in memory.

LIST 500 or LLIST 500

Displays/prints line 500.

LIST . or LLIST .

Displays/prints the current program line.

Syntax 2

This format allows the following options:

-- If only *linenum1* is specified, that program line and all higher-numbered program lines are displayed/printed.

- If only *linenum2* is specified, all program lines from the beginning of the program through that program line are displayed/printed.

- If both line numbers are specified, the entire range is displayed/printed.

You may use a period (.) for either line number to indicate the current line.

You can stop the listing by pressing CTRL BREAK at any time.

Examples:

LIST 150- or LLIST 150-

Displays/prints all program lines from 150 to the end of the program.

LIST -1000 or LLIST -1000

Displays/prints all program lines from the lowest number through 1000.

LIST 150-1000 or LLIST 150-1000

Displays/prints program lines 150 through 1000, inclusive.

Execute a program in memory

The RUN command/statement is used to execute (run) a program that is currently in memory or a program stored on disk (*see "Execute a program file (.BAS)" in Chapter 11*).

After execution, GW-BASIC returns to command level.

Syntax:

RUN [*linenum*]

where

linenum is the beginning program line.

If *linenum* is specified, execution begins with that program line. Otherwise, execution begins at the lowest program line number.

Examples:

RUN

Executes the program in memory.

RUN 500

Executes the program in memory starting at line 500.

Renumber program lines

The RENUM command allows you to change the line numbers of the current program.

RENUM is used only in direct mode.

Syntax:

RENUM [*newnum*][,[*oldnum*] [,*increment*]]

where

newnum is the first program line number to be used in the new sequence. The default is 10.

oldnum 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 program line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, RESTORE, RESUME and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message Undefined line xxxxx in yyyyy is displayed. The nonexistent program 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, so they start with line number 1000 and are numbered in increments of 20.

Save a program to disk

After creating or editing a program, you can store it on disk using the SAVE command.

Syntax:

SAVE *"filespec"* [,A or ,P]

where

"filespec" is a string expression which specifies where to save the program and what file name to save it under.

"filespec" is a file or path name with an optional drive name. With MS-DOS, the default extension .BAS is supplied. If the drive name is omitted, the default drive is assumed. If the path name is omitted, the current "working" directory is assumed.

If a file with the same name already exists on the selected disk, it will be written over.

Use the A option to save the file in ASCII format. Otherwise, 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 requires an ASCII format file. Attempts to MERGE binary programs will result in a Bad file mode error. Also, some operating systems commands such as TYPE may require an ASCII format file.

Use the P option to protect 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 with an Illegal function call error.

CAUTION: No way is provided to "unprotect" such a program.

Examples:

```
SAVE "PAYROLL"
```

Saves the program PAYROLL.BAS to the disk in the default drive in binary format.

```
SAVE "B:SALES",A
```

Saves the program SALES.BAS to the disk in Drive B in ASCII format, where B could be replaced with any drive name.

```
SAVE "B:PROG",P
```

Saves the program PROG.BAS to the disk in Drive B as a protected file.

```
SAVE "JOHN\PAYROLL"
```

Saves the PAYROLL.BAS program to the subdirectory named JOHN of the current "working" directory.

Send a program listing to a device or file

The LIST command can be used to send all or part of the program currently in memory to a specified file or device.

Syntax:

LIST [*linenum1*][-[*linenum2*]] , *device*

where

linenum1 is the beginning program line.

linenum2 is the ending program line.

device is a device designation string, such as SCRN: or LPT1:, or a file specification.

device allows the listing to be directed to a device such as a printer or communications device.

Or, it allows the listing to be sent to a file on a disk. If the file does not exist, it is created. If the file does exist, the list is stored over the existing program. The existing program is destroyed so be careful when using this application. When you direct a listing to a disk file, the program is saved in ASCII format, thus you may later use this file with MERGE.

You cannot interrupt (stop) a listing directed to a file or device. In this case, the listing will continue until the range is exhausted.

Examples:

```
LIST 150-1000 "PAYROLL"
```

The above command stores program lines 150 through 1000 of the program currently in memory to a file named PAYROLL on the disk in the active drive.

```
LIST , LPT1:
```

Lists the program to the line printer.

26. STRING MANIPULATION

The following are described in this chapter:

INSTR function
LEFT$ function
LEN function
MID$ function
MID$ statement
RIGHT$ function
SPACE$ function
STRING$ function

INSTR function

Searches for the first occurrence of a given substring in a string, and returns the position at which the match is found.

Syntax:

INSTR([*start*,] *string*, *substring*)

where

start is an integer expression in the range 1 to 255, which specifies where the search is to begin. If omitted, 1 is assumed.

string is a string expression (in particular a string constant or variable) whose value is the string to be searched.

substring is a string expression in particular a string constant or variable whose first occurrence is to be searched for.

Optional offset *start* sets the character position for starting the search. It must be in the range 1 to 255. If it is greater than the number of characters in *string* (LEN(*string*)) or if *string* is null or if *substring* cannot be found, INSTR returns 0. If *substring* is null, INSTR returns *start* or 1, and if no *start* was specified, then INSTR returns 1. If *start*=0 is specified, error message Illegal function call will be displayed.

Example:

```
10 A$ = "ABCDEB"
20 B$ = "B"
30 PRINT INSTR(A$,B$)
40 PRINT INSTR(4,A$,B$)
RUN
 2
 6
Ok
```

LEFT$ function

Returns a substring extracting a number of characters to the left of a given string, as specified by the *length* parameter.

Syntax:

LEFT$(*string,length*)

where

string is a string expression whose value is the string from which the substring is to be returned.

length is an integer expression (from 0 to 255) which specifies the number of the characters to be returnd.

If *length* is greater than LEN(*string*), the entire original string will be returned.

If *length* = 0, the null string (length zero) will be returned.

Refer to the MID$ and RIGHT$ functions in this chapter.

Example:

```
10 A$ = "GW-BASIC"
20 B$ = LEFT$(A$,6)
30 PRINT B$
RUN
GW-BAS
Ok
```

LEN function

Returns the length of a given string.

Syntax:

LEN(*stringexp*)

where

stringexp is a string expression whose length will be returned.

Unprintable characters and blanks are counted in the number of characters.

If the argument *stringexp* is a null string, LEN returns zero.

Example:

```
10 A$ = "PORTLAND, OREGON"
20 PRINT LEN(A$)
RUN
 16
Ok
```

MID$ function

Returns a substring from a specified string.

Syntax:

MID$(*string, start*[, *length*])

where

string is a string expression whose value is the string from which the substring is to be returned.

start is an integer expression whose value specifies the character position of the beginning of the returned string. It must be >=1.

length is an integer expression from 0 to 255 which represents the length of the returned string.

The function returns a substring from a specified ***string***, starting from a specified character position (***start***). The ***length*** of the returned substring can be specified. If ***length*** is omitted or if there are fewer than ***length*** characters to the right of the specified character position, all rightmost characters beginning with the specified character position are returned. If ***length*** is equal to zero, or if ***start*** is greater than LEN(***string***), then MID$ returns a null string. *Also see LEFT$ and RIGHT$ functions in this chapter.*

Example:

```
10 A$ = "Good "
20 B$ = "morning evening afternoon"
30 PRINT A$;MID$(B$,9,7)
RUN
Good evening
Ok
```

MID$ statement

Replaces a part of a string with another string.

Syntax:

MID$(*string, start*[, *length*]) = substring

where

string is a string expression whose value is the string from which a substring is to be replaced.

start is an integer expression from 1 to 255, whose value specifies the character position where the replacement has to begin. *start* must be < = LEN(*string*).

length is an integer expression from 0 to 255 which represents the length of the returned string.

substring is a string expression which replaces the characters in *string*, beginning from *start* positon.

The characters in string, beginning from *start* position, are replaced by the characters in *substring*. The optional *length* refers to the number of characters from *substring* that will be used in the replacement. If *length* is omitted, all of the characters of *substring* are used. However, regardless of whether *length* is omitted or included, the replacement of characters never goes beyond the original length of *string*. If either *start* or *length* is out of the specified range, an Illegal function call error will be returned.

Example:

```
10 A$ = "KANSAS CITY, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
RUN
KANSAS CITY, KS
Ok
```

RIGHT$ function

Returns a substring from a specified string, extracting its rightmost characters.

Syntax:

RIGHT$(*string,length*)

where

string is a string expression whose value is the original string from which a substring is to be returned.

length is a numeric expression rounded to the nearest integer, whose value (from 0 to 255) represents the length of the returned string.

If *length* is greater than or equal to LEN(*string*), then the entire original string is returned. When length = 0, the null string (length of zero) is returned.

Also see the LEFT$ and MID$ functions in this chapter.

Example:

```
10 A$ = "DISK GWBASIC"
20 PRINT RIGHT$(A$,7)
RUN
GWBASIC
Ok
```

SPACE$ function

Returns a string of a specified number of spaces.

Syntax:

SPACE$(*length*)

where

length is an integer expression from 0 to 255. It specifies the number of spaces, i.e., the length of the returned string.

If *length* is outside the specified range, an Illegal function call error is returned.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
 2
  3
   4
    5
Ok
```

STRING$ function

Returns a string of specified length whose characters all have the same ASCII code (Appendix A) or equal to the first character of a given string.

Syntax 1:

STRING$(*length, code*)

Syntax 2:

STRING$(*length, stringexp*)

where

length is an integer expression from 0 to 255. It specifies the length of the resulting string.

code is an integer expression in the range 0 to 255. It specifies the ASCII code whose corresponding character is used to form the resulting string.

stringexp is a string expression whose first character is used to form the resulting string.

Examples:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "SALES REPORT" X$
RUN
----------SALES REPORT----------
Ok

10 A$ = "DALLAS"
20 X$ = STRING$(8,A$)
30 PRINT A$
RUN
DDDDDDDD
Ok
```

Notes:

27. USER-DEFINED FUNCTIONS

The DEF FN statement is used to define and name a function that is written by the user.

Syntax:

DEF FN*name*[(*argument*[, *argument*]...)] = *expression*

where

name is a legal variable name beginning with FN. No blanks may be inserted between FN and the remainder of the name and the first character after FN must be a letter.

argument is a "dummy" variable that is to be replaced by the corresponding argument value when the function is called.

expression is an expression that performs the operation of the function.

The type of ***expression*** must agree with the type (numeric or string) of the function, specified by ***name***.

In the DEF FN statement, variable names 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 argument list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the program variable is used.

The variables in the ***argument*** list represent, on a one-to-one basis, the argument variables or values that are to be given in the function call.

User-defined functions may be numeric or string. The type of the function is specified by ***name***. The type of the ***expression*** must match the type of the function; otherwise, a Type mismatch occurs. If the function is numeric, the value of the ***expression*** is forced to that type before the function value is returned.

If a DEF FN statement has not been executed before the function it defines is called, an Undefined user function error occurs.

Example:

```
400 X = 10:Y = 20
410 DEF FNAB(X,Y) = X*3 + Y*2
420 I = 10:J = 20
430 T = FNAB(I,J)
440 PRINT T
RUN
 70
Ok
```

Line 410 defines the function FNAB. The function is called in line 430.

ASCII CODE

APPENDIX A

Notes:

ASCII CODE

This table shows the 256 elements of the standard ASCII character set, together with their decimal and hexadecimal equivalents.

DEC	HEX	CHARACTER
000	00	BLANK (NULL)
001	01	☺ (SOH)
002	02	● (STX)
003	03	♥ (ETX)
004	04	♦ (EOT)
005	05	♣ (ENQ)
006	06	♠ (ACK)
007	07	• (BEL)
008	08	■ (BS)
009	09	○ (HT)
010	0A	◙ (LF)
011	0B	♂ (VT)
012	0C	♀ (FF)
013	0D	♪ (CR)
014	0E	♫ (SO)
015	0F	☼ (SI)

DEC	HEX	CHARACTER
016	10	► (DLE)
017	11	◄ (DC1)
018	12	↕ (DC2)
019	13	!! (DC3)
020	14	¶ (DC4)
021	15	§ (NAC)
022	16	▬ (SYN)
023	17	↨ (ETB)
024	18	↑ (CAN)
025	19	↓ (EM)
026	1A	→ (SUB)
027	1B	← (ESC)
028	1C	∟ (FS)
029	1D	↔ (GS)
030	1E	▲ (RS)
031	1F	▼ (US)

DEC	HEX	CHARACTER
032	20	BLANK (SPACE)
033	21	!
034	22	"
035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	—
046	2E	.
047	2F	/

DEC	HEX	CHARACTER
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?

DEC	HEX	CHARACTER
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O

DEC	HEX	CHARACTER
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	^
095	5F	_

DEC	HEX	CHARACTER
096	60	‘
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

DEC	HEX	CHARACTER
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	Δ

DEC	HEX	CHARACTER
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å

DEC	HEX	CHARACTER
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	Ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	Pt
159	9F	f

DEC	HEX	CHARACTER
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	1/2
172	AC	1/4
173	AD	¡
174	AE	«
175	AF	»

DEC	HEX	CHARACTER
176	B0	▒
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

DEC	HEX	CHARACTER
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧

DEC	HEX	CHARACTER
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	■
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

DEC	HEX	CHARACTER
224	E0	α
225	E1	β
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	μ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	∅
238	EE	∈
239	EF	∩

DEC	HEX	CHARACTER
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	n
253	FD	2
254	FE	■
255	FF	BLANK 'FF'

MATHEMATICAL FUNCTIONS

APPENDIX B

Notes:

DERIVED FUNCTIONS

Functions that are not intrinsic to GW-BASIC may be calculated as follows.

FUNCTION	GW-BASIC EQUIVALENT
SECANT	SEC(x) = 1/COS(x) when x <>1.570796
COSECANT	CSC(x) = 1/SIN(x) when x<>0
COTANGENT	COT(x) = 1/TAN(x) when x<>0
INVERSE SINE	ARCSIN(x) = ATN(x/SQR(1-x*x))
INVERSE COSINE	ARCCOS(x) = 1.570796-ATN(x/SQR(1-x*x)) when ABS(x)<1
INVERSE SECANT	ARCSEC(x) = ATN(SQR(x*x-1)) + SGN(SGN(x)-1)*1.570796 when ABS(x) >= 1
INVERSE COSECANT	ARCCSC(x) = ATN(1/SQR(x*x-1)) when ABS(x)>1 + (SGN(x)-1)*1.570796
INVERSE COTANGENT	ARCCOT(x) = 1.570796-ATN(x)
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)) when x<>0

FUNCTION	GW-BASIC EQUIVALENT
HYPERBOLIC COTAGENT	COTH(x) = (EXP(x) + EXP(-x)) / (EXP(x)-EXP(-x)) when x<>0
INVERSE HYPERBOLIC SINE	ARCSINH(x) = LOG(x + SQR(x*x + 1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(x) = LOG(x + SQR(x*x-1)) when x >= 1
INVERSE HYPERBOLIC TANGENT	ARCTANH(x) = LOG((1 + x) / (1-x)) / 2 when ABS(x)<1
INVERSE HYPERBOLIC SECANT	ARCSECH(x) = LOG((SQR(1-x*x) + 1) / x) when 0<x <= 1
INVERSE HYPERBOLIC COSECANT	ARCCSH(x) = LOG((SGN(x)*SQR(x*x + 1) + 1) / x) when x>0
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(x) = LOG((x + 1) / (x-1)) / 2 when ABS(x) >1
LOGARITHM TO BASE 'a'	LOGA(x) = LOG(x) / LOG(a) when a>0 and x>0

You can define a derived function in your program by use of a DEF FN statement, to avoid coding the formula each time you need it.

Note that both 'x' and 'a' can be any numeric constant, variable, array element, function or expression. Any values of 'x' or 'a' that would cause error messages are noted.

ERROR CODES AND ERROR MESSAGES

APPENDIX C

Notes:

ERROR MESSAGES

NUMBER	MESSAGE
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error A line is encountered which includes an incorrect sequence of characters (misspelled keyword, unmatched parenthesis, incorrect punctuation, etc). 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.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a numeric or string function. This FC error may also occur as the result of:

NUMBER	MESSAGE
	1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR. 4. A negative mantissa with a noninteger exponent. 5. A call to a USR function for which the starting address has not yet been given. 6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO. 7. A negative record number used with GET(Files) or PUT(Files) statements.
6	Overflow The result of a calculation is too large to be represented in 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 big, or has too many loops, subroutines, variables, or has expressions that are too complicated to evaluate.
8	Undefined line A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

NUMBER	MESSAGE
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; or an OPTION BASE is given after an array has been dimensioned.
11	Division by zero A division by zero is encountered in an expression; or, the value zero has been raised to a negative power. In the former case, the result is machine infinity (with the appropriate sign); in the latter case, the result is positive machine infinity. In both cases 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.

NUMBER	MESSAGE
14	Out of string space String variables have caused GW-BASIC to exceed the amount of free memory remaining. GW-BASIC will allocate string space dynamically, until it runs out of memory.
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 to be processed. It should be broken into smaller expressions.
17	Can't continue An attempt is made to continue a program that: 1. Has halted due to an error. 2. Has been modified during a break in execution. 3. Does not exist.
18	Undefined user function A USR function is called before the function definition (DEF statement) is given.

NUMBER	MESSAGE
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.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt has been made to input a line that has too many characters.
24	Device Timeout GW-BASIC did not receive information from an I/O device within a predetermined amount of time.
25	Device Fault An incorrect device designation has been entered.
26	FOR without NEXT A FOR statement was encountered without a matching NEXT.

NUMBER	MESSAGE
27	Out of paper The printer is out of paper or is not switched on. Insert paper, ensure power is switched on and continue.
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.
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.
52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

NUMBER	MESSAGE
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(Files), GET(Files), 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 A sequential output mode OPEN statement is issued for a file that is already open; or a KILL command is given for a file that is open.
57	Device I/O Error An I/O error occurred on a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.
58	File already exists The filename specified in a NAME command is identical to a filename already in use on the disk.
61	Disk full All disk storage space is in use.

NUMBER	MESSAGE
62	Input past end An INPUT statement is executed 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(Files) or GET(Files) statement, the record number is either greater than the maximum allowed (16,777,215) or equal to zero.
64	Bad file name An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement/command (e.g., a filename with too many characters).
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 An attempt was made to open a file to a non-existent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:,

NUMBER	MESSAGE
	or was disabled by the user. This occurs if an OPEN "COM1:... statement is executed after the user has disabled RS232 support via the /C:0 switch directive on the GWBASIC command line.
69	Communication buffer overflow Occurs when a communication input statement is executed but the input queue was already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs' will attempt to clear this fault, unless characters continue to be received faster than the program can process them. In this case several options are available: 1. Increase the size of the COM receive buffer via the /C: switch. 2. Implement a "hand-shaking" protocol with the host/satellite such as XON/XOFF to turn transmit off long enough to catch up. 3. Use a lower baud rate for transmit and receive.
70	Disk Write Protected This is one of 3 "hard" disk errors returned from the disk controller. This occurs when an attempt is made to write to a diskette that is write protected. Use an ON ERROR GOTO statement to detect this situation and request user action. Errors 71, 72, and 74 are other possible "hard" disk errors.

NUMBER	MESSAGE
71	Disk not ready Occurs when the diskette drive door is open, or a diskette is not in the drive. Again use an ON ERROR GOTO statement to recover.
72	Disk media error Occurs when the FDC controller detects a hardware or media fault. This usually indicates damaged media. Copy any existing files to a new diskette and reformat the damaged diskette. FORMAT will flag the bad tracks and place them in a file "badtrack". The remainder of the diskette is now usable.
74	Rename across disks An attempt was made to rename a file with a new drive designation.
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.

NUMBER	MESSAGE
**	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 GW-BASIC to continue. GW-BASIC closes any open files and exists to MS-DOS.

Notes:

GW-BASIC RESERVED WORDS

APPENDIX D

Notes:

RESERVED WORDS

GW-BASIC comprises a set of statements, commands, function names, and operator names which are treated as reserved words, and which cannot be used as variable names. The total list of GW-BASIC reserved words is as follows:

ABS
AND
ASC
ATN
AUTO
BEEP
BLOAD
BSAVE
CALL
CALLS
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSGN
DEFSTR
DELETE
DIM
DRAW
EDIT
ELSE
END
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
EXP
FIELD
FILE
FIX
FNxxxxxxxx
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
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$
MKDIR
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
PAINT

PEEK
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT
RANDOMIZE
READ
REM.
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SAVE
SCREEN
SGN
SHELL
SIN
SOUND
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TIMER
TIME$
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR$
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#
XOR

HEXADECIMAL CONVERSION TABLES

APPENDIX E

Notes:

HEXADECIMAL CONVERSION TABLES

BYTE				BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15
4		3		2		1	

BINARY TO HEXADECIMAL CONVERSION TABLE

The following table shows the decimal (base 10), binary (base 2), octal (base 8), and hexadecimal (base 16) representations for the numbers 0 to 16.

DECIMAL	BINARY	OCTAL	HEX
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	8
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

TECHNICAL INFORMATION

APPENDIX F

Notes:

HOW GW-BASIC ALLOCATES VARIABLES

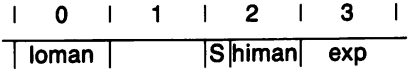
Offset	Length (in bytes)	Description	Comments
0	1	Type	The value of the variable type is: 2 (integers) 3 (strings) 4 (single-precision numbers) 8 (double-precision numbers)
1	1 to n	Name	- Bytes 1 and 2 contain the first 2 characters of the name - Byte 3 contains a number indicating how many more characters are in the name - Bytes 4 to n contain the additional characters
4+n	2	Integer	Integers are stored low byte first, then high byte
	3	String Descriptor	- First byte contains the string length (0-255) - Second byte is the low byte of the offset into the GW-BASIC's data segment - Third byte is the high byte of the offset into the GW-BASIC's data segment
	4	Single Precision	In floating point format
	8	Double Precision	In floating point format

Note that 3 bytes are reserved for a variable name, even if the name is one or two characters long. In this case the data item itself (or the string descriptor) begins with an offset of 4.

INTERNAL REPRESENTATION OF FLOATING POINT NUMBERS

The following section describes the internal representation of numbers in GW-BASIC.

Single Precision - 24 bit mantissa



where loman = the low mantissa
S = the sign
himan = the high mantissa
exp = the exponent
man = himan:...:loman

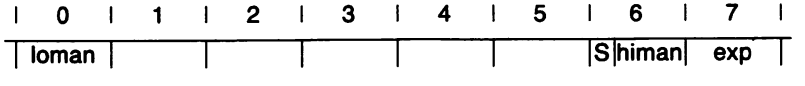
- If exp = 0, then number = 0

- If exp <>0, then the mantissa is normalized and number = sign * .1 man * 2 ** (exp -80h)

  That is, in single precision (hex notation - bytes low to high)

  00000080 = .5
  00008080 = -.5

Double Precision - 56 bit mantissa



MEMORY MAP

The map below illustrates the system memory when GW-BASIC is loaded (address values are approximate).

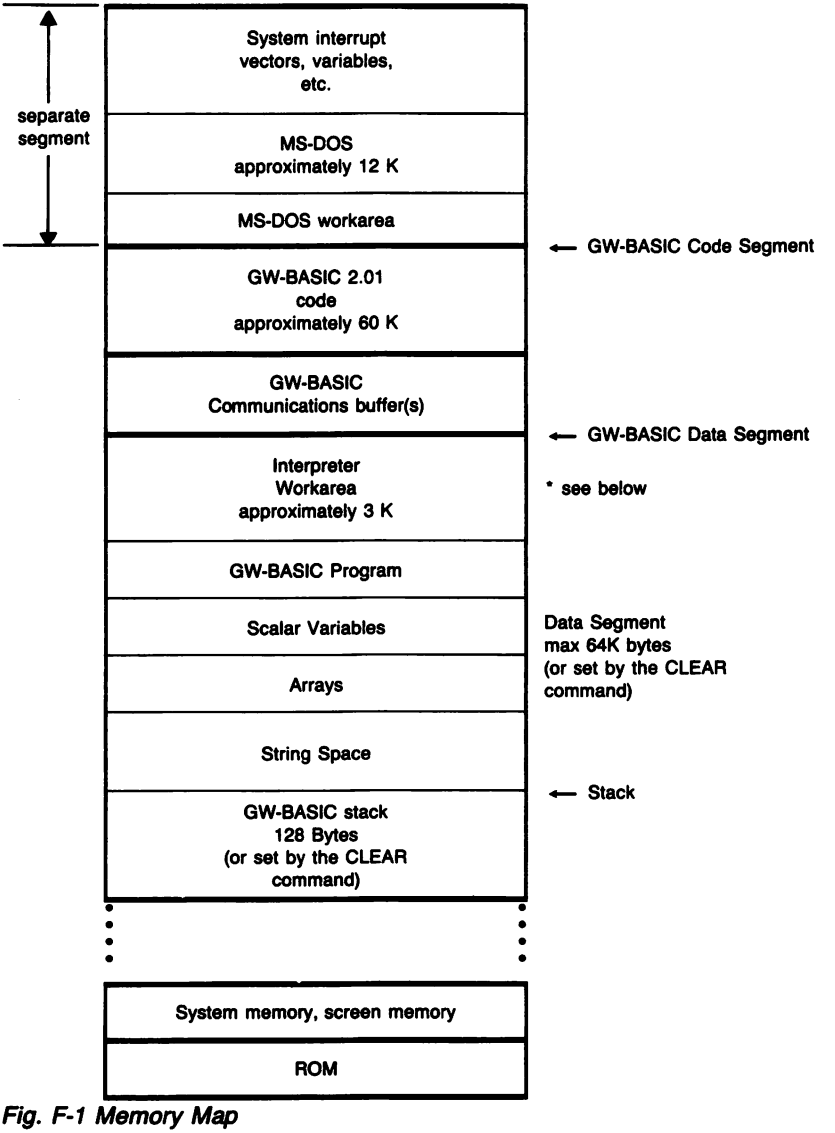


Fig. F-1 Memory Map

* In the GW-BASIC workarea there are some variables available to the user, which may be accessed via the PEEK or POKE. These variables are not at the actual addresses specified: the PEEK and POKE map the specified addresses to the actual locations.

The paragraph address of the GW-BASIC Data Segment may be obtained by examining the word at absolute location 0:510H. (This is an actual address).

This information is not required to execute PEEKs and POKEs.. a DEF SEG statement sets up the right segment. In this case, the following locations are of significance:

Address (Decimal)	**Offset** (Hex)	**Meaning**
46	2EH	Current line number
839	347H	Current Error line number
48	30H	Offset of start of Program
856	358H	Offset of start of Variables
106	6AH	Keyboard buffer PEEK/POKE location

The first four locations are word variables which are mapped to the given addresses when the PEEK function and POKE statements are used. Note that these are 2-byte variables; thus, execution of the program line

```
1234 X = PEEK(46) + 256 * PEEK(47)
```

sets X = to the value 1234, which is the current line number.

POKE-ing a 0 to the location 106 (decimal) will clear the GW-BASIC line buffer. The GW-BASIC line buffer holds characters received from the system keyboard buffer. These characters are handled by the GW-BASIC screen editor. Any other value POKE-ed to this address will have no effect.

PEEK(106) will return a 0 if no keyboard characters are available, and a 1 if any are available.

Notes:

CONVERSION OF PROGRAMS TO GW-BASIC

APPENDIX G

INTRODUCTION

GW-BASIC bears a similarity to many BASICs. Your personal computer will support programs written for an extensive variety of microcomputers. For programs written in a BASIC other than GW-BASIC, some minor adjustments may be necessary before running them. This appendix highlights some specific areas to examine when converting programs.

STRING DIMENSIONING

LENGTH OF STRINGS

GW-BASIC strings are of variable lengths. Therefore, all statements that declare the length of strings should be deleted. For example, in a statement which dimensions a string array for 'J' elements of lengths 'I' such as:

```
DIM A$(I,J)
```

the conversion for GW-BASIC would be:

```
DIM A$(J)
```

SUBSTRINGS

In GW-BASIC the following functions are used to take substrings of strings:

LEFT$
MID$
RIGHT$

Other forms, such as:

A$(I) (to access the Ith character in A$) or,
A$(I,J) (to take a substring of A$ from position I to J) should be changed as follows:

Other BASICs	GW-BASIC
X$ = A$(I)	X$ = MID$(A$,I,1)
X$ = A$(I,J)	X$ = MID$(A$,I,J-I + 1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, then the conversion should be carried out as follows:

Other BASICs	GW-BASIC
A$(I) = X$	MID$(A$,I,1) = X$
A$(I,J) = X$	MID$(A$,I,J-I + 1) = X$

CONCATENATION

GW-BASIC uses a plus (+) sign to denote string concatenation. Other BASICs use a comma (,) or an ampersand (&) which should be altered accordingly.

MAT FUNCTIONS

Some BASICs incorporate MAT functions for array handling. To convert a program which uses these functions to the GW-BASIC environment, it is necessary to rewrite the program including FOR...NEXT loops.

MULTIPLE ASSIGNMENTS

Some BASICs allow the following syntax:

```
10 LET D=E=0
```

to set D and E equal to zero. GW-BASIC interprets the second equal sign as a logical operator and sets D equal to -1 if E was equal to 0. This statement should therefore be broken up into two assignment statements as follows:

```
10 D=0:E=0
```

MULTIPLE STATEMENTS

Multiple statements on a GW-BASIC line must always be separated by colons (:), unlike some other BASICs which use a backslash (\) instead.

PEEKs AND POKEs

The execution of programs containing PEEK and POKE instructions may vary from machine to machine. It is therefore necessary to analyse the purpose of these instructions in other BASIC programs before translating the same functions into GW-BASIC.

IF...THEN...[ELSE...]

Not all BASICs feature the optional ELSE clause which is performed in the event of a test proving false.

For example, a BASIC statement may originally be:

```
10 IF D=E THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

The above statement sequence will work correctly, but it may be optimized in GW-BASIC as follows:

```
10 IF D=E THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

FILE I/O

In some BASICs, the I/O to disk may differ from GW-BASIC.

GRAPHICS

Selecting screen attributes and drawing objects on the screen can vary from BASIC to BASIC.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W