

**EG2000  
BASIC  
Manual**

**COLOUR**

**GENIE**



# INTRODUCTION

You should already have read the Beginner's Manual that describes the way that the Colour Genie works and how it should be connected up to the TV and tape recorder. If you have not already read this booklet, or think that you know it all, then quickly read it now. There are things in there that are very important to know!

This book goes through most of the BASIC commands available on the Colour Genie and give examples of the syntax and sample programs. The commands themselves are given in alphabetical order, grouped in a logical order, by their operating mode. Consequently some jumping around within the text will be necessary and no attention has been taken of the order in that the commands come.

A third book is being written to complement these two. This will contain all the system information and a lot of the detail that will let serious programmers and hardware engineers have all the information that they need.

This book is *not* a BASIC programming manual. It is a reference book, and should be treated as such. There are many fine books on the market that can teach you how to program in BASIC and if you are not familiar with the language then one of these should be read along with this manual.

# VARIABLE TYPES

The Colour Genie accepts variable names which are longer than two characters; however, only the first two characters are used by the computer to distinguish between other variables. Variable names must begin with a letter (from A to Z) and followed by another letter or a digit (from 0 to 9). The following are valid and distinct variables:

A, AA, AB, AC, AO, A1, BN, BZ, B7, ZZ, Z1

Note: The user should not use any variable name which contains words with special meaning (or reserved words) in the BASIC language. For example, "CIF" cannot be used as a variable name, since it contains the BASIC keyword "IF".

A list of reserved words is in Appendix A.

There are four types of variables used with the Colour Genie: integer, single precision, double precision, and string variables. The first three types are used to store numeric values, whereas the last type is used only for character storage.

1. %: integer (whole numbers within the range -327 68 to +327 67)

Example:

A% = -30  
BB% = 8 000

2. !: single precision (6 significant digits)

Example:

A! = -50.3  
D4! = .123 456

3. #: double precision (16 significant digits)

Example:

A# = 3.141 592 653 589  
A2# = -4 567.890 123 4

4. \$: string (maximum length: 255 characters)

Example:

```
A$ = "COLOUR GENIE"  
M2$ = "THE RESULT OF  
(A * B + 15)/2.5 IS:"
```

Though A%, A!, A#, A\$ all have the same variable name "A", their types are different, that is %, !, #, \$: they are considered to be distinct variables by the computer.

The Colour Genie uses normal arithmetic symbols:

+ for addition, - for subtraction, \* for multiplication, / for division, and the ↑ key for exponentiation.

For example, the result of  $5 \times 12^{1/3}$  is equivalent to the result of  $5 * 12[(1/3)]$  in Colour Genie BASIC (Note: you will find no [key on the keyboard, it is represented by the ↑ key).

# RELATIONAL OPERATORS

Whenever a decision has to be made within a program, a relational operator is needed. The acceptable operators are:

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

Example:

```
120 IF A < B THEN PRINT "B IS GREATER  
    THAN A"
```

When the computer executes this statement, if the content of B is greater than the content of A (i.e.,  $A < B$  is true), the sentence "B IS GREATER THAN A" will be printed on the screen. Otherwise the computer will just go to the next statement.

# LOGICAL OPERATORS

AND, OR, and NOT are the only logical operators accepted by the Colour Genie.

Example:

```
10 IF A = 1 and B = 5 GOTO 50
```

The computer branches to line 50 if A = 1 and B = 5, otherwise the computer goes to the next statement following line 10.

```
20 A = (B = 2) AND (C > 10)
```

A has the value of -1, if both B = 2 and C > 10 are true. Otherwise A has the value of 0.

```
40 A = (D < 2) OR (E < 20)
```

A has the value of -1 if either D < 2 or E < 20 is true. When both D ≥ 2 and E ≥ 20 are true, then A has the value of 0.

```
70 A = NOT (F > 5)
```

Try the following 2 programs for yourself:

## PROGRAM 1

```
5 REM "AND" DEMO
10 REM PASSWORD ROUTINE
20 INPUT "ENTER FIRST PASSWORD"; P1$
30 INPUT "ENTER SECOND PASSWORD"; P2$
40 INPUT "ENTER THIRD PASSWORD"; P3$
50 IF P1$ = "SOOTY" AND P2$ = "SWEEP" AND
   P3$ = "SUE" THEN 70 ELSE 60
60 PRINT "YOU DO NOT KNOW ALL PASSWORDS TRY
   AGAIN": GOTO 20
70 PRINT "YOU KNOW THE PASSWORDS :::: ENTER ... "
```

## PROGRAM 2

```
5 REM OR DEMO. PROGRAM
10 INPUT "ENTER 3 NUMBERS"; A, B, C
20 PRINT "THE SMALLEST NUMBER IS";
30 IF A > B OR A > C THEN IF B > C THEN PRINT C
   ELSE PRINT B ELSE PRINT A
40 PRINT "THE LARGEST NUMBER IS";
50 IF A < B OR A < C THEN IF B < C THEN PRINT C
   ELSE PRINT B ELSE PRINT A
```

# STRING OPERATORS

In string operations, the relational operators are used to compare the precedence of two strings.

Note that the following operations are all true.

"B" < "C"

(THE CODE FOR B IS  
LESS THAN THE CODE  
FOR C)

"JOHN" > "JACK"

(SAME REASON AS  
ABOVE)

"STRING" = "STRING"

"LETTERS " < > "LETTERS" (SPACE ALSO COUNTS)

A\$ = "BO" + "AT"

(A\$ WILL HAVE THE  
VALUE: BOAT)



# ORDER OF OPERATORS

Operations in the innermost level of parentheses are performed first, then evaluation proceeds to the next level, etc. Operations on the same level are performed according to the following precedence rules.

- |                                |   |
|--------------------------------|---|
| 1. Exponentiation              | $A [ B$                                   |
| 2. Negation                    | $-C$                                      |
| 3. Multiplication and Division | $A \cdot B, C/D$                          |
| 4. Addition and Subtraction    | $C + D, E - F$                            |
| 5. Relational Operators        | $A < B, "C" = "C",$<br>$15 < > 16, 1 > 0$ |
| 6. Logical Operators           | $NOT, AND, OR$                            |

For example, we have a formula.

$$10 \text{ ANS} = A + B \cdot C \cdot D/2 + E [ 2$$

The computer will evaluate in the following sequence. If

$$\begin{aligned} A &= 2 \\ B &= 3 \\ C &= 4 \\ D &= 5 \\ E &= 6 \end{aligned}$$

Then apply to the formula above

$$\begin{array}{c}
 2 + 3 \cdot 4 \cdot 5/2 + 6 [ 2 \\
 \underbrace{\phantom{2 + 3 \cdot 4 \cdot 5/2 + 6 [ 2}} \\
 12 \cdot 5 \\
 \underbrace{\phantom{12 \cdot 5}} \\
 60 / 2 \\
 \underbrace{\phantom{60 / 2}} \\
 2 + 30 \quad 6 [ 2 \\
 \underbrace{\phantom{2 + 30}} \quad \underbrace{\phantom{6 [ 2}} \\
 32 \quad + \quad 36 \\
 \underbrace{\phantom{32 + 36}} \\
 68
 \end{array}$$

Therefore the answer should be 68.

The following program illustrates the use of these arithmetical operators:

```
10 REM POUNDS TO KGS CONVERT
20 INPUT "ENTER WEIGHT IN STONES, POUNDS,
   OUNCES"; S, P, O
30 W = S * 14 + P + O/16: REM WEIGHT IN POUNDS
40 KG = INT (W/2.2)
50 G = W/2.2 - KG
60 G1 = G * 1 000
70 PRINT "YOU WEIGHT"; KG; "KILOGRAMS AND";
   G1; "GRAMS"
```

# CHAPTER 1 :

## “ACTIVE” COMMANDS

When the computer is initially switched on the screen has a “MEM SIZE?” message. Once the RETURN key has been pressed the Colour Genie goes into “active” command mode.

The normal indication is the word “COLOUR BASIC READY” followed by a “ > ” sign which appears on the next line at the upper left corner on the display (monitor or TV screen). For convenience we will call this indication the “ready message”.

At this point, the user should hit the RETURN key before entering one of the following commands through the keyboard.

- |           |           |            |
|-----------|-----------|------------|
| 1. AUTO   | 6. CSAVE  | 11. RUN    |
| 2. CLEAR  | 7. DELETE | 12. SYSTEM |
| 3. CLOAD  | 8. EDIT   | 13. TROFF  |
| 4. VERIFY | 9. LIST   | 14. TRON   |
| 5. CONT   | 10. NEW   | 15. LLIST  |

Everything inside the brackets is optional. For example: AUTO (line number, increment). All the user has to do is type in the underlined portion:

AUTO 10, 5

or any numeric value to replace “line number” and “increment”. In case the option is not taken, just type in:

LIST

The computer will perform certain specified actions automatically. Notice: Every command should be followed by pressing the RETURN key.

## AUTO (1, n)

This command automatically sets the line numbers before each source line is entered. 1 specifies the beginning line number and n the increment desired between lines. If the user only types in AUTO followed by the RETURN key, the beginning line number will be set at 10, with each increment of 10. The user may enter his program statement right after the line number.

Example:

```
30 PRINT "THIS IS LINE 30"
```

Everytime the user hits the RETURN key, the computer will increment the line number. Until the BREAK key is hit, the AUTO command will remain in operation. (Note that whenever AUTO brings up a line that has been used previously, there will be an asterisk appear right next to the line number. If the user does not want to alter that line, hit the BREAK key to turn off the AUTO function).

Example:

READY  
> AUTO 1, 2  
1 LINE 1  
3 LINE 3  
5 LINE 5  
7 LINE 7  
9

BREAK

RETURN

READY  
> AUTO 2, 2  
2 SECOND LINE  
4 FOURTH LINE  
6 SIXTH LINE  
8

BREAK

RETURN

READY  
> AUTO  
10 LINE 10  
20 LINE 20  
30 LINE 30  
40

BREAK

RETURN

READY  
> AUTO 1, 1  
1\*  
2\*  
3\*  
4\*  
5\*

RETURN

## CLEAR

The CLEAR command will clear a specific number of bytes for strings storage. If the option is not used i.e., type in CLEAR followed by the RETURN key, the computer will reset all numeric variables to zero, and all string variables to null. When the option is taken, the command will perform, in addition to the first function, a second function: that is to clear a specified number of bytes for string storage. Note that when the user turns on the computer, a CLEAR 50 command is performed automatically.

Example:

**CLEAR 100**

Reset all numeric variables to zero, and all string variables to null. Then clears 100 bytes of memory for string storage.

## CLOAD "f"

CLOAD "f" will load a specified program according to the file name, "f" to the computer from a cassette. Before using this command, the user should re-wind the cassette tape, check the cables and connectors (consult the Beginner's Manual). If everything is ready, type in, for example CLOAD "A" then hit the RETURN key. Now press the PLAY button of the cassette recorder and the computer will search until the file named "A" is found. If the file is found, a stable and a blinking asterisks will appear at the top right corner of the display to indicate loading is carrying out. Once the entire program has been loaded in the computer, the READY message will appear on the display.

Example:

**CLOAD "3"**

Load from cassette the file named "3". Note that only the first character of the file name is used for CLOAD, CSAVE and VERIFY commands.

## CONT

This command continues the program execution, at the point where the execution has been stopped by the BREAK key or a STOP statement within the program.

## CSAVE "f"

This command transfers the program with file name "f" in the computer's main memory onto cassette tape. The file name must be accompanied with this command. Any alphanumeric character other than double quotes ( " ) will be acceptable as a file name. Again, before using the command, the cassette tape must be in a proper starting location (not overlapped with any useful program location). Check the cables and connectors, press the PLAY and REC buttons of the cassette at the same time, then start typing the command accordingly.

Example:

**CSAVE "C"**

Saves a program with label "C" on the cassette recorder from the main memory. Warning: keep account of the locations of the saved programs on tape. Find an empty space for the new program to be loaded, unless you want to erase the old programs. Erased programs are not recoverable.

## DELETE n1(-n2)

This command will clear the memory location that contains the specified line(s).

Example:

<b>DELETE 5</b>	Clear line 5
<b>DELETE 7 - 10</b>	Clear line 7, line, 10 and any line in between
<b>DELETE -12</b>	Clear from the first line of the program, up to and including line 12.
<b>DELETE.</b>	Clear the line currently entered, or edited.

## EDIT n

This command will cause the computer to shift from the active command mode to the editing mode. In the editing mode, the user is allowed to examine and modify the program statements in the main memory, by using a set of sub-commands. There must be a valid line number following the EDIT command, otherwise the command may not be accepted. See Chapter 2.

Example:

**EDIT 20**

Turns the computer from active command mode to editing mode – then examines line 20.

## LIST (n1–n2)

LIST tells the computer to display any specified program lines stored in the main memory. If the options are not used, the computer will scroll the entire program onto the display. In order to pause and examine the text, the user should hit the SHIFT key together with the @ key. The scrolling will continue by hitting any key.

Example:

<b>LIST 3</b>	display line 3
<b>LIST 10 – 20</b>	display line 10, line 20 and any line in between
<b>LIST –50</b>	display from the first line up to and include line 50
<b>LIST 20 –</b>	display line 20 and all following lines
<b>LIST .</b>	display the current line just entered or edited
<b>LIST</b>	display all lines in the memory.



## LLIST

Lists a program onto the printer. This command functions in a very similar way as the LIST command. If the Line printer is not properly connected, the computer will enter a dead loop and waits to print the first character. This situation can only be resolved by turning the printer on or hitting the RESET buttons.

## NEW

This command will clear all program lines; reset numeric variables to zero and string variables to null. It does not change the memory size previously set by the CLEAR command.

## RUN (n)

RUN instructs the computer to start executing (or RUN) the user's program stored in main memory. If a line number is not specified, the computer will start executing from the lowest line number. However, if a line number is provided, the computer will execute from the given line number to higher order lines. Note that an error will occur if an invalid line number is used.

Everytime a RUN is executed, a CLEAR command is also executed automatically before it.

Example:

<b>RUN 50</b>	start executing at line 50
<b>RUN</b>	start executing at the lowest number line.

RUN can also be included as part of a program line and has the effect of restarting the program from the appropriate line number.

## SYSTEM

This command turns the computer into the monitor mode. Within this mode, the user may load his own program or data file in machine code format. To load an object file from tape, type in SYSTEM and RETURN; the "\*" symbol will be displayed. Then type in the file name. The tape will begin loading. When loading is completed, another "\*" will appear. Type in a slash "/" symbol followed by the entry point address (in decimal) where the user wants the execution to start. If the user does not type in the entry address, execution will begin at the address specified by the object file.

## TROFF

This command will turn off the Trace function. Usually follows the TRON command.

## TRON

This command will turn on a Trace function that allows the user to keep track of the program flow for debugging and execution analysis. Everytime the computer executes a new program line, the line number will be displayed inside a pair of brackets.

Example: Consider the following program:

```
10 PRINT "** PROGRAM 1 **"  
20 A = 1  
30 IF A = 3 THEN 70  
40 PRINT A  
50 A = A + 1  
60 GOTO 30  
70 PRINT "END PROGRAM 1"  
80 END
```

Type in

```
> TRON RETURN  
> RUN RETURN
```

```
< 10 > ** PROGRAM 1 **  
< 20 > < 30 > < 40 > 1  
< 50 > < 60 > < 30 > < 40 > 2  
< 50 > < 60 > < 30 > < 70 > END PROGRAM 1  
< 80 >
```

In order to pause execution before its natural end, the SHIFT @ keys must be pressed. To continue, just press any key. To turn off the Trace function, enter TROFF.

TRON and TROFF are available for use within user programs to check if a given line is executed

Example:

```
*  
*  
*  
90 IF A = B THEN 160  
100 TRON  
110 A = B + C  
120 TROFF  
*  
*  
*
```

In this portion of a program, if A happens to be not equal to B, the line 110 should be executed. By using TRON and TROFF inside the program, the user can see precisely whether line 110 has been executed or not. The computer will display < 110 > < 120 > if these lines were executed. TRON and TROFF can be removed after a program is debugged.

## VERIFY "f

This command will compare a specific program stored on cassette tape with the one in the computer's main memory. Usually, this command is used right after the CSAVE command which stores a program from the computer's main memory to a cassette. The VERIFY command allows the user to examine whether the copying (CSAVE) operation is successful.

It is good practice to include the file name in this command, since the computer will search for that file, or program, before comparison starts. Otherwise the first file encountered on the cassette will be compared. During the operation, the program on tape and the program in memory are compared byte by byte. If any part does not match, the message "BAD" will be displayed. In this case, the user should repeat the CSAVE command again. Same as CLOAD Command, the cassette must be re-wound, cables and connectors checked, press the RETURN key, prior pressing the PLAY button on the cassette recorder.

# CHAPTER 2 : EDITING

The purpose of editing in the Colour Genie BASIC is to help you modify programs. With the Editor, you need not type in the entire program every time you make a programming mistake or typing error. The need for an editor becomes more critical when programs are long and complex.

In this chapter we discuss every editing function including subcommands, that are available in Genie BASIC. A substantial amount of descriptive examples are presented with each command. You are advised to try out each editing command before entering your first program into the system.

## EDIT n

EDIT shifts the computer from the Active Command mode to the Editing. The user must specify which line he wants to edit. If the line number is not provided, an UL error will occur (see Appendix B).

Example:

**EDIT 100** (allow to edit line 100)

**EDIT.** (allow to edit the current line just entered).

## RETURN Key

If you press the RETURN key while in the EDIT mode, the computer will record all the changes made in that line, and return back to the Active Command mode.

## n Space-bar Key

In the EDIT mode, pressing the space-bar will move the cursor one space to the right and display any character stored in the preceding position. You may type in the value of n before hitting the Space-bar, then the cursor will move n spaces to the right side.

Suppose you have entered a line into the computer by the command:

```
> AUTO 100  
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

If you want to edit this line, you should type in EDIT 100 followed by the RETURN key, like the following.

```
> EDIT 100 
```

then the display will become:

```
100 _
```

By pressing the Space-bar 12 times, the cursor will move to the right side by 12 spaces. The display should look like:

```
100 IF A = B THE _
```

You may also use the option to display more characters at once. That is, enter the number of cursor spaces desired, before hitting the Space-bar Example

Type in 8 followed by the Space-bar key:

```
100 IF A = B THE _
```

The display will become

```
100 IF A = B THEN 150 : _
```

If you want to display the next 20 positions, you may type 20 then the Space-bar again. The outcome should be:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100 _
```

## n KEY

This action will move the cursor back to the left by n spaces. Everything behind the cursor will disappear from the display; however, it is not erased from the memory.

Example:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

Hit the 5  key, the display will look like this:

```
100 IF A = B THEN 150 : A = A + 1 : GOT _
```

Then type 10  and the display changes to:

```
100 IF A = B THEN 150 : A = A
```

After this sequence of operations, if the user hits the RETURN key, the display will look like:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO100  
> _
```

That means the computer has returned back to the Active Command mode. If any further change is desired in line 100, the user must enter the Edit mode again.

## SHIFT ↑ KEY

By pressing the SHIFT and ↑ keys simultaneously, the computer will escape from any of the following Insert subcommands: H, I, X. After escaping from an Insert subcommand, you remain in the Editing mode, while the current cursor position is unchanged. Another way to escape from these Insert subcommands, is by pressing the RETURN key, which will shift the computer back to the Active Command mode.

**H****KEY**

“H” represents Hack and Insert; that is to delete remainder of the line and to let you insert materials at the current cursor position.

Example:

Consider this line:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

If you want to replace `A = A + 1` by `A = A + B`, and to delete `GOTO 100`, you should first enter the editing mode, type in 25 followed by pressing the Space-bar (move 25 spaces from the beginning of the line). The display should look like:

```
100 IF A = B THEN 150 : A = A ...
```

Now hit the H key, type in `+ B` then hit RETURN (back to the Active Command mode). Or hit SHIFT and `↑` simultaneously to return to editing mode, then hit L to display the entire line, as below:

```
100 IF A = B THEN 150 : A = A + B  
100 _
```

with anything not displayed being deleted.

**I****KEY**

“I” for Insert; that is to allow insertion of characters starting at the current cursor position, without altering any other part of the line.



Example:

You want to insert the statement "PRINT A" between "A = A + 1" and "GOTO 100" in line 100. Line 100 looks like this:

```
100 IF A = B THEN 150 : A = A + 1 : GOTO 100
```

By using the EDIT mode and the Space-bar Move the cursor to:

```
100 IF A = B THEN 150 : A = A + 1 : _
```

Now hit the I key, type in "PRINT A:", then press the SHIFT and I keys to escape from the subcommand level. At this point you can type L to list the current line. And the display should look this:

```
100 IF A = B THEN 150 : A = A + 1: PRINT A :  
GOTO 100  
100 _
```

or you can hit the RETURN key to return to the Active Command mode.



"X" represents Insert at End of Line. The command moves the cursor position to the end of the line, and shifts the line down one line. This is the Insert subcommand. You can insert new lines at the end of the line, or delete part of the

By using the  key,

Example:

Get into the Edit mode:

```
> EDIT 100  
100 _
```

Type in X without hitting RETURN. The line displayed should be

```
100 IF A = B THEN 150 : A = A + 1 : PRINT A :  
GOTO 100
```

At this point you may add some new material, or delete part of the existing line, before hitting SHIFT and t.

## **L** KEY

“L” stands for List. While the computer is in the editing mode, and is not currently executing one of the subcommands H, I, X, the L command will list the remaining part of the line onto the display.

Example:

```
> EDIT 100  
100 _
```

Hit L (without hitting RETURN), the display should be:

```
100 IF A = B THEN 150 : A = A + 1 : PRINT A :  
GOTO 100  
100 _
```

The second line allows you to edit while referencing the first line.

## **A** KEY

“A” represents Cancel and Restart. In the editing mode this command moves the cursor back to the beginning of the line, cancels all editing changes previously made on that line, and restores the former content of the line.

## **E** KEY

This command shifts the computer from editing mode back to the Active Command mode, and saves all the changes previously made. Make sure the computer is not executing any subcommand before entering E.

## **Q** KEY

This command shifts the computer from editing mode back to the Active Command mode but cancels all the changes made in the current edit mode. Just type in Q to cancel the changed mode and return to the Active Command mode.

## **n D** KEY

“D” represents delete; the command will delete n numbers of characters right after the current cursor position. The deleted characters will be enclosed in exclamation marks “!” to show you which characters are being affected.

Example:

Consider the following line:

```
100 IF A = B THEN 150 : A = A + 1 : PRINT A :  
GOTO 100
```

First enter into the editing mode and move the cursor to the following position:

```
100 IF A = B THEN 150 : A = A + 1 _
```

Now type in 15D (to delete 15 characters); the display should look like:


```
100 IF A = B THEN 150 : A = A + 1! : PRINT A : GO!
```

Then use L to list the entire line. The display should become:

```
100 IF A = B THEN 150 : A = A + 1! : PRINT A : GO!  
TO 100  
100 _
```

List again:

```
100 IF A = B THEN 150 : A = A + 1 TO 100  
100 _
```

Now use the X key and the  key to delete "TO 100". the final outcome should be:

```
100 IF A = B THEN 150 : A = A + 1
```

n  KEY

"C" means change; this command allows you to change n number of characters right after the current cursor position. If the number n is not specified the computer assumes that you only want to change a single character.

Example:

Consider the line:

```
100 IF A = B THEN 150 : A = A + 1
```

If you want to change 150 to 230, you should enter the edit mode and move the cursor to the following position:

```
100 IF A = B THEN _
```

Now type in 2C (change the next 2 characters), followed by 23 (new data), and hit the SHIFT and ↑ keys. List the line by hitting L:

```
100 IF A = B THEN 230 : A = A + 1  
100 _
```

n  c

This command searches for the n<sup>th</sup> occurrence of the character c on that line and moves the cursor to that position. If the n value is not provided, the computer will search for the first occurrence of the character specified and stop the cursor there. In case the specified character is not found, the cursor will move to the end of the line. As usual, the computer will start searching from the current cursor position towards the right end of the line.

Consider the following example:

```
100 IF A = B THEN 230 : A = A + 1
```

After entering the edit mode, the display should look like this:

```
100 _
```

Now type in 2S =, to inform the computer to search for the second occurrence of the equal sign "=", and the final display should be

```
100 IF A = B THEN 230 : A _
```

Now you can enter one of the subcommands at the current position. For example:

Type in H (hack and insert) followed by "= A + 2" (new data). Then the line will become:

```
100 IF A = B THEN 230 : A = A + 2 _
```

n **K** c

This command deletes all characters up to the n<sup>th</sup> occurrence of character C, and moves the cursor to that position. Consider the following example:

```
100 IF A = B THEN 230 : A = A + 2
```

Enter the edit mode:

```
100 _
```

Now type in 1K:, to inform the computer to search for the first occurrence of the colon ":" symbol, then delete everything in front of it on that line. The display should become

```
100! IF A = B THEN 230!
```

The " : " should also be deleted so type in D. The display will become:

```
100! IF A = B THEN 230!! : !
```

Then hit the L key to list the line on the display. The line should look like this:

```
100 A = A + 2  
100 _
```

# CHAPTER 3 :

# BASIC PROGRAMMING

# STATEMENTS

In this chapter we are going to discuss the program statements of Colour Genie BASIC.

The main part of this chapter concerns various functions of all the programming statements in BASIC which are acceptable to the Colour Genie. Since it is a very large set of statement, and each statement had its own unique and characteristics in programming, the users are advised to study each statement with the help of the examples provided.

## CLEAR n

This statement sets all variables to zero. If number n is specified, the computer sets n bytes of space for string storage. Everytime the Colour Genie is turned on, 50 bytes of space are automatically cleared and reserved for strings.

The CLEAR statement becomes critical during program execution, because an Out of String Space error will occur, if the amount of string storage cleared is less than the greatest number of characters stored in string variables.

Example:

```
10 CLEAR 1000
```

Clear 1000 bytes of memory space for string storage.

## DATA

The DATA statement allows you to store data inside the program and to access them through READ statements. The item list will be assessed by the computer sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Each item in the item list may be a string or a simple numeric value. Just like entering data from the keyboard, any string



value consisting of blanks, colons, commas, must be enclosed in a pair of quotes. For those strings having no commas or colons, quotes are not necessary. The order of values in a DATA statement must match up with the variable types in the READ statements. DATA statements may appear anywhere in a program.

For example:

```
5 REM READ DATA DEMO
10 FOR N = 1 TO 4
20 READ A$
30 PRINT A$; " ";
40 NEXT N
50 DATA THIS, IS, THE, DATA
```

Strings, and other types of data, can be mixed.

Example:

```
10 READ A$, B$, C, D
20 PRINT A$; B$; C; D
30 DATA "CHARACTERS ", "A LONG SENTENCE "
40 DATA 20, 137.54
50 END
```

When this program is run you should see the following on the screen:

```
CHARACTERS A LONG SENTENCE 20 137.54
```

## DEFINT

Variable names that begin with letters specified within the letter range, will be treated and stored as integers. However, a type declaration character (refer to the Introduction) can over-ride type definition. Defining a variable name as an integer not only saves memory space, but also saves computer time, because integer calculation is faster than single or double precision calculation. Note that integers can only take on values between  $-32,768 + 32,767$  inclusive.

Example:

```
10 DEFINT X, Y, Z
```

After the computer has executed line 10, all variables beginning with the letters X, Y or Z will be treated as integers. Therefore X2, X3, YA, YB, ZI, ZJ will become integer variables. Except that X1 #, X2 #, YB #, will be still double precision variables, because type declaration characters always over-ride DEF statements.

For example:

```
10 DEFINT A - D
```

Causes variables beginning with letter A, B, C or D to be integer variables. Note that DEFINT can be placed anywhere in a program, but it may change the meaning of variable reference without type declaration characters. Therefore, it is normally placed at the beginning of a program.

### DEFSNG letter range

Variable names that begin with those letters specified within the letter range, will be treated and stored as single precision variables. However, a type declaration character can over-ride this type definition. Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. All numeric variables are assumed to be single precision unless otherwise specified. The DEFSNG statement is primarily used to re-define variables which have previous been defined as double precision or integer.

For example:

```
10 DEFSNG A - D, Y
```

Causes variables beginning with the letter A through D, or Y to become single precision. However, A # would still be a double precision variable and Y% will still be an integer variable.

```

5 REM DEFSNG DEMO
10 DEFSNG N
20 N = 0 : M = 0
30 N = N+. 1 : M = M+. 1
40 PRINT M, N
50 IF N > 10 THEN STOP
60 GOTO 30

```

### DEFDBL letter range

Variable names that begin with those letters specified within the letter range, will be treated and stored as double precision. However, a type declaration character can over-ride this type definition. Double precision allows 17 digits of precision, while only 16 digits are displayed when a double precision variable is printed.

For example:

```
10 DEFDBL M - P, G
```

Causes variables beginning with one of the letters M through P, or G to become double precision.

```

5 REM DEFDBL DEMO
10 DEFDBL N
20 N = 0 : M = 0
30 N = N+. 1 : M = M+. 1
40 PRINT M, N
50 IF N > 10 THEN STOP
60 GOTO 30

```

### DEFSTR letter range

Variables that begin with those letters specified within the letter range, will be treated and stored as string.

However, a type declaration character can over-ride this type definition. Each string can store up to 255 characters, if there is enough string storage space cleared.

For example:

```
10 DEFSTR A - D
```

Causes variables beginning with any letter A through D to be string variables, unless a type declaration character is added. Therefore, after the execution of line 10, the assignment  $B3 = \text{"A STRING"}$  is valid.

DIM name (dim 1, dim 2, . . . . . , dim n)

The statement defines the variable name to be an array or list of arrays. The number of elements in each dimension may be specified through dim 1, dim 2, etc. If dim n is not specified, 11 elements in each dimension is assumed in each array. The number of dimensions is limited only by the memory size available.

Example:

```
10 DIM A(5), B(3, 4), C(2, 3, 3)
```

This statement defines the one dimensional array with 6 elements (from 0 to 5); the two dimensional array B with 20 elements (4 x 5); the three dimensional array C with 48 elements (3 x 4 x 4).

DIM statements may be placed anywhere in a program, and the number of subscripts may be an integer or an expression.

Example:

```
10 INPUT "NUMBER OF ITEMS", N  
20 DIM A (N + 2, 4)
```

The number of elements in array A may vary according to N. To re-dimension an array, you must use a CLEAR statement either with or without the argument n. Otherwise an error will occur.

Example:

```
10 X (2) = 13.6
20 PRINT "THE SECOND ELEMENT IS: ";X (2)
30 DIM X (15)
40 PRINT X (2)
50 END
```

if you run the above program an error, type DD (see ERROR codes) will occur.

## END

This statement causes a normal termination of program execution. The END statement is primarily used to cause execution to terminate at some point other than the logical end of the program.

Example:

```
5 B = 3 : C = 14
10 A = C + B
20 GOSUB 70
30 D = X + Y
40 PRINT "THE RESULTS ARE:";
50 PRINT A, D
60 END
70 X = 50
80 Y = A * X
90 RETURN
```

When RUN you should get:

```
THE RESULTS ARE: 17      900
```

The END statement in line 60 prevents the computer from executing into line 70. Therefore the subroutine that starts at line 70 can be accessed only by line 20.

## ERR . . ERL

These commands are used to return the error code relating to the displayed error.

ERR returns the code of the error and ERL the line number in which the error occurs.

These commands are usually used with the BASIC statement ON . . . ERROR . . . GOTO. (viz)

```
1 REM ERL DEMO
5 ON ERROR GOTO 50
10 INPUT "ENTER TWO NUMBERS"; A, B
20 C = A/B
30 PRINT C
40 END
50 IF ERL = 20 THEN PRINT "DIVISION BY ZERO":
    RESUME 10
60 IF ERL = 10 THEN PRINT "YOUR NUMBERS
    CAUSED AN OVERFLOW": RESUME 10
```

## ERROR CODE

This statement is used for testing an ON ERROR GOTO routine. When the ERROR code statement is encountered, the computer will proceed exactly as if that kind of error has occurred.

Example:

```
30 ERROR 1
?NF ERROR IN 30
```

## EXPLANATION OF ERROR MESSAGES

<i>Code</i>	<i>Abbreviation</i>	<i>Error</i>
1	NF	NEXT without FOR: NEXT is used without a matching FOR statement. This error also occurs if NEXT variable statements are reversed in a nested loop.
2	SN	Syntax Error: This usually is the result of incorrect punctuation, open parenthesis, an illegal character or a mis-spelled command.

- 3 RG RETURN without GOSUB: A RETURN statement was encountered before a matching GOSUB was executed.
- 4 OD Out of Data: A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape or DATA.
- 5 FC Illegal Function Call: An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative matrix dimension, negative or zero LOG arguments, etc. Or USR call without first POKEing the entry point.
- 6 OV Overflow: The magnitude of the number input or derivative is too large for the Computer to handle. NOTE: There is no underflow error. Numbers smaller than  $\pm 1.701411E-38$  single precision or  $\pm 1.701411834544556E-38$  double precision are rounded to 0. See/0 below.
- 7 OM Out of Memory: All available memory has been used or reserved. This may occur with very large matrix dimension, nested branches such as GOTO, GOSUB, AND FOR-NEXT Loops.
- 8 UL Undefined Line: An attempt was made to refer or branch to a non-existent line.
- 9 BS Subscript out of Range: An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.

- 10 DD Redimensional Array: An attempt was made to DIMension a matrix which had previously been dimensioned by DIM or by default statements. It is a good idea to put all dimension statements at the beginning of a program.
- 11 /Ø Division by Zero: An attempt was made to use a value of zero in the denominator. NOTE: If you can't find an obvious division by zero check for division by numbers smaller than allowable ranges. See OV above and RANGES.
- 12 ID Illegal Direct: The use of INPUT as a direct command.
- 13 TM Type Mismatch: An attempt was made to assign a non-string variable to a string or vice-versa.
- 14 OS Out of String Space: The amount of string space allocated was exceeded.
- 15 LS String Too Long: A string variable was assigned a string value which exceeded 255 characters in length.
- 16 ST String Formula Too Complex: A string operation was too complex to handle. Break up the operation into shorter steps.
- 17 CN Can't Continue: A CONT was issued at a point where no continuable program exists eg. after program was ENDEd or EDITed.
- 18 NR No RESUME: End of program reached in error-trapping mode.
- 19 RW RESUME without ERROR: A RESUME WAS encountered before ON ERROR GOTO was executed.



20	UE	Unprintable Error: An attempt was made to generate an error using an ERROR statement with an invalid code.
21	MO	Missing Operand: An operation was attempted without providing one of the required operands.
22	FD	Bad File Data: Data input from an external source (i.e. tape) was not correct or was in improper sequence, etc.

## FKEY

This function programs the user definable keys F1 to F4. Eight possible functions are available with the use of the SHIFT key. A string of up to seven characters can be stored for future use.

Format 1 : FKEY n = "string"

Format 2 : FKEY n = "string"

The second format is treated as direct command.

There are eight default functions preprogrammed into the keys that are available after power on. (See Beginner's manual.)

Fn	Function
1	LIST
2	RUN
3	AUTO
4	EDIT
5	RENUMber
6	DELETE
7	CLOAD
8	CSAVE

Key functions 5 to 8 are accessed by the SHIFTEd keys 1 to 4 respectively. The preprogrammed functions take effect only after the RETURN key has been pressed.

## FOR ..... = ..... TO ..... STEP ..... NEXT

These statements form an interactive loop so that a sequence of program statements may be executed over a specified number of times.

The general form is:

For counter = initial value TO final value STEP increment.

\*  
\*  
\*

Statements

\*  
\*  
\*

NEXT counter

In the FOR statement, initial value, final value and increment can be constants, variables or expressions. The first time the FOR statement is executed, these three are evaluated and the values are saved; if these values are changed inside the loop, they will have no effect on the loop's operation. However, the counter value must not be changed or the loop will not operate normally.

The FOR-NEXT loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value". Execution proceeds until a NEXT statement is encountered. At this point, the counter is incremented by the amount specified in the STEP increment. If STEP increment is not used, an increment of 1 is assumed. However, if the increment has a negative value, then the counter is actually decremented.

The counter is then compared with the final value specified in the FOR statement. If the counter is greater than the final value, the loop is completed and execution continues with the statement following the next statement. (if increment was a negative number, loop ends when counter is less than the final value).

If the counter has not yet reached the final value, control is passed back to the first statement after the FOR statement.

Example:

```
10 FOR K = 0 TO 1 STEP 0.3
20 PRINT "THE VALUE OF K:"; K
30 NEXT K
40 END
```

When run we get:

```
THE VALUE OF K: 0
THE VALUE OF K: .3
THE VALUE OF K: .6
THE VALUE OF K: .9
```

When  $K = 1.2$ , it is greater than the final value 1, therefore the loop ends without ever printing 1.2.

If we now try counting "backwards".

Example:

```
10 FOR N = 5 TO 0
20 PRINT "THE VALUE OF N:"; N
30 NEXT N
40 END
```

All we get is the first value

```
THE VALUE OF N: 5
```

We need to tell the computer how much to step "down" by.

```
10 FOR N = 5 TO 0 STEP - 1
20 PRINT "THE VALUE OF N:"; N
30 NEXT N
40 END
```

This will now give:

```
THE VALUE OF N: 5
THE VALUE OF N: 4
THE VALUE OF N: 3
THE VALUE OF N: 2
THE VALUE OF N: 1
THE VALUE OF N: 0
```

Since no STEP was specified, so STEP 1 is assumed. N is incremented the first time, and its value becomes 6. Because 6 is greater than the final value 0, the loop ends. This is remedied by adding STEP-1, as you can see.

Example:

```
10 FOR A = 0 TO 3
20 PRINT "THE VALUE OF A:"; A
30 NEXT
40 END
```

This gives:

```
THE VALUE OF A: 0
THE VALUE OF A: 1
THE VALUE OF A: 2
THE VALUE OF A: 3
```

Note here that instead of using NEXT A in line 30, you may simply write NEXT. However, this can lead to trouble if you have nested FOR-NEXT loops.

Here is an example of nested loops, showing how it is advisable to identify the counter variable in each NEXT statement:

```
10 I = 1
20 J = 2
30 K = 3
40 FOR N = I + 1 TO J + 1
50 PRINT "FIRST LOOP"
60 FOR M = I TO K
70 PRINT "SECOND LOOP"
80 NEXT M
90 NEXT N
100 END
```

This gives:

```
FIRST LOOP
  SECOND LOOP
  SECOND LOOP
  SECOND LOOP
FIRST LOOP
  SECOND LOOP
  SECOND LOOP
  SECOND LOOP
```

## GOTO

This statement transfers program control to the specified line number. If used independently, an unconditional branch will result. However, test statements may precede the GOTO statement to create a conditional branch.

Example:

```
10 A = 10
20 B = 45
30 C = A + B
40 C = C + 3.4
50 GOTO 100
60 .....
70 .....
80 .....
90 .....
100 PRINT "A ="; A, "B ="; B, "C ="; C
110 END
```

When run we should get:

```
A = 10    B = 45    C = 187
```

When line 50 is executed, control will unconditionally jump to line 100, and lines 60, 70, 80 and 90 are ignored.

Example:

```
10 IF A = 2 GOTO 120
```

When line 10 is under execution, if A equals 2 then control will jump to line 120, otherwise it will just go to the next statement. You may use GOTO in the Active Command mode as an alternative to RUN.GOTO line number causes execution to begin at the specified line number, but without the automatic CLEAR.

## GOSUB

GOSUB transfers program control to the specified line number where a subroutine starts. Only if the computer encounters a RETURN statement will it then jump back to the statement that immediately follows the GOSUB. Just like GOTO, GOSUB may be preceded by a test statement such as:

```
IF A = B THEN GOSUB 100
```

Example:

```
10 PRINT "MAIN PROGRAM."  
20 GOSUB 50  
30 PRINT "END OF PROGRAM."  
40 END  
50 PRINT "SUBROUTINE."  
60 RETURN
```

This should give:

```
MAIN PROGRAM.  
SUBROUTINE.  
END OF PROGRAM.
```

## IF

This statement instructs the computer to test a logical or relational expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line. In numerical terms, if the expression has a non-zero value, it is always equivalent to a logical true.

Example:

```
10 INPUT "ENTER A VALUE (MAX. 20)"; A
20 IF A > 20 GOTO 60
30 A = A * 3.1416 * 2
40 PRINT "THE CIRCUMFERENCE IS:"; A
50 END
60 PRINT "NUMBER TOO BIG! (MAX 20)": GOTO 10
```

When RUN we get:

```
ENTER A VALUE (MAX. 20)? 24
NUMBER TOO BIG! (MAX. 20)
ENTER A VALUE (MAX. 20)? 18
THE CIRCUMFERENCE IS: 113.098
```

In this example if A is greater than 20 then a warning is printed and another input is expected. However, if A is equal to or less than 20, the computer will go to the next line and compute the value of A, without passing through the warning message and the GOTO statement.

Example:

```
120 INPUT A: IF A = 10 AND A > B THEN 160
120 INPUT A: IF A = 10 AND A > B GOTO 160
```

The two statements above have the same effect.

### IF . . . THEN

Initiates the "action clause" of an IF – THEN type statement. THEN is optional except when it is used to specify a branch to another line number, as in IF A > D THEN 100. THEN should be used in IF – THEN – ELSE statements.

## IF ... THEN ... ELSE

This statement must be used after the IF statement, and acts as an alternative action in case the IF test fails.

Example:

```
10 IF A = 1 THEN 60 ELSE 40
```

In this example, if  $A = 1$  then control branches to line 60, otherwise it branches to line 40. If the ELSE clause is not used and  $A$  is not equal to 1, the computer will go to the next statement instead of branching to line 40.

IF-THEN-ELSE statements may be nested, but the number of IFs and ELSEs must match with each other.

Example:

```
10 INPUT "ENTER THREE NUMBERS"; X, Y, Z
20 PRINT "THE LARGEST NUMBER IS:";
30 IF X < Y OR X < Z THEN IF Y < Z THEN
    PRINT Z ELSE PRINT Y ELSE PRINT X
40 END
READY
> RUN
ENTER THREE NUMBERS ? 30, 75, 73
THE LARGEST NUMBER IS: 75
```

This program accepts three numbers and prints out the one that has the highest value.



Another use of the IF ... THEN ... ELSE command is shown below:

```
5 REM IF THEN ELSE DEMONSTRATION
10 PRINT "SET TIME OF CLOCK TO START"
20 INPUT "HRS, MINS, SECS"; H, M, S
30 CLS
40 PRINT @ 400, "HRS", "MINS", "SECS"
50 PRINT @ 440, H, M, S
60 FORD = 1 TO 320: NEXT D
70 IF S = 59 THEN IF M = 59 THEN 90 ELSE 100
   ELSE 80
80 S = S + 1: GOTO 50
90 H = H + 1: M = 0: S = 0: GOTO 50
100 M = M + 1: S = 0: GOTO 50
```

RUN this, and follow the screen instructions!

### INKEY \$

This command makes the computer scan the keyboard and returns a one character string determined by the key pressed. If no key is pressed during execution of the command then a null string is returned.

```
5 REM INKEY$ DEMO
10 CLS
20 A$ = INKEY$
30 PRINT @ 460, A$;
40 IF A$ = "?" THEN STOP
50 GOTO 10
```

### INPUT

This statement causes the computer to suspend execution of a program and wait until you input the specified number and type values through the keyboard. Input values can be string or numeric according to the variable type. The items (if more than one) in the list must be separated by commas.

Example:

```
10 INPUT A$, B$, A, B
```

This statement permits you to input two string values, followed by two numeric values. The input sequence must be consistent. When the computer executes this statement, it sends a signal onto the display:

```
?
```

And waits for the inputs. You may enter all four values at once (separated by commas). In this case, the inputs could be as follows:

```
orange, apple, 59, 47 RETURN
```

The computer then assigns the values accordingly.

```
A$ = "ORANGE"
```

```
B$ = "APPLE"
```

```
A = 59
```

```
B = 47
```

The other way to input those values would be by entering the items on separate lines. In this way, the computer will remind you to input the next data for the remaining variables by displaying:

```
??
```

Until all variables are set, the computer then advances to the next statement. Input must be compatible to the variable type specified. In other words, you should not input a string value to a numeric variable. If such an invalid entry occurs, the computer will send the message:

```
? REDO
```

```
?
```

This indicates that input does not match with the current variable type. However, the computer gives you a second chance to input the correct data starting with the first value expected by the INPUT statement.

Example:

```
10 INPUT A$, A
20 PRINT A$, A
30 END
```

We then get:

```
? STRING, 10
STRING 10
```

or even:

```
? THIS IS A STRING, 13.5
THIS IS A STRING 13.5
```

or even:

```
? ABCD, IJK
? REDO
? ABCDE
?? 25
ABCDE 25
```

If an input string consists of blanks, the entire string must be enclosed by quotes.

In order to provide a clearer indication to the operator, the user may include a "prompting message" in the INPUT statement. This helps to input correct data type to each variable. The prompting message must immediately follow INPUT, enclosed in quotes, and followed by a semi-colon.

Example:

```
100 INPUT "INPUT ITEM NAME AND QUANTITY";
    N$, Q
```

When RUN the screen will display :

```
INPUT ITEM NAME AND QUANTITY?
```

Try RUNNING these 3 programs to see how INPUT works.

### Program 1

```
5 REM INPUT DEMO #1
10 INPUT "HOW MANY NUMBERS"; T
20 PRINT "ENTER YOUR NUMBERS ONE BY ONE"
30 C = 0
40 FOR D = 1 TO T
50 INPUT N
60 C = N + C
70 NEXT D
80 PRINT "AVERAGE EQUALS"; C/T
```

### Program 2

```
5 REM INPUT DEMO #2
10 INPUT "LENGTH (CM)"; L
20 INPUT "WIDTH (CM)"; W
30 INPUT "HEIGHT (CM)"; H
40 V = L * W * H
50 PRINT "THE VOLUME IS"; V
60 S = 2 * (L * W + L * H + H * W)
70 PRINT "THE SURFACE AREA IS"; S
```

### Program 3

```
5 REM INPUT DEMO #3
10 PRINT "INPUT NO. OF";
20 INPUT "DEGREES F"; F
30 C = (5/9) * (F - 32)
40 PRINT F; "DEGREES F IS"; C; "DEGREES C"
```

### INPUT # – cassette number, item list

This statement tells the computer to input the specified number of values stored on the cassette tape and to assign them to the variables. You must specify the cassette drive number from which data is expected.

Example:

```
10 INPUT # - 1, A$, B, C, D$
```

This statement inputs data from cassette drive number 1. The first value is assigned to A\$, the second value to B, etc. The cassette deck must be in PLAY mode. If a string is encountered when a numeric value is expected by the INPUT statement, a bad file data error will occur. An Out-of-Data error will also occur if there is not enough data on the tape for all the variables in an INPUT statement.

### LET

This statement is used to assign a value to a variable. The word LET is not required in assignment statements by the Colour Genie BASIC interpreter. However, you may use the word LET in order to make program compatible with other systems.

Example:

```
10 LET A = 5.67
20 B% = 20
30 S$ = "CHARACTERS"
40 LET D% = D% + 1
50 PRINT A, B%, S$, D%
60 END
```

This gives:

```
5.67      20      CHARACTERS      1
```

In all the assignments above, the variable on the left of the equal sign is assigned with the value of the constant or expression on the right side. All these statements are acceptable.

## LPRINT

LPRINT prints a file onto the printer. This command (and statement) functions in a similar way to a PRINT statement (print on the display). If the line printer is not properly connected, the computer will enter a dead loop and will wait to print the first character. This situation can only be resolved by turning the printer on or hitting the RESET buttons.

```
10 FOR X = 1 TO 0 STEP -.25
20 LPRINT "THE VALUE OF X:"; X
30 NEXT X
40 END
```

If a printer is connected then it will print.

```
THE VALUE OF X: 1
THE VALUE OF X: .75
THE VALUE OF X: .5
THE VALUE OF X: .25
THE VALUE OF X: 0
```

## ON ERROR GOTO

This statement allows you to set up an error-trapping routine to recover a program from an error and to continue, without any break in execution. Without this statement, the computer will stop execution and print out an error message, once it encounters any kind of error in the user's program. Normally, you have a particular type of error in mind when an ON ERROR GOTO statement is used.

For example, suppose that a program performs some division operations and the user has not ruled out the possibility of division by zero. You could write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

Example:

```
5 B = 15: C = 0
10 ON ERROR GOTO 120
20 A = B/C
30 PRINT A, B, C
40 END
120 PRINT "DIVIDED BY ZERO!!"
130 END
```

When RUN the screen shows:

```
DIVIDED BY ZERO!!
```

In this example, C has a value of zero, so a divide-by-zero error will occur when the computer attempts to execute line 20. But because of line 10, the computer will simply ignore line 20 and branch to the error-handling routine beginning at line 120. Please note that the ON ERROR GOTO statement must be executed before the error occurs, otherwise it has no effect. Note also that the error handling routine must be terminated by a RESUME statement.

The following programs illustrate the use of error values with ON . . . ERROR . . . GOTO:

#### Program 1

```
5 REM ON ERROR GOTO DEMO. # 1
10 ON ERROR GOTO 60
20 INPUT "ENTER TWO NUMBERS"; A, B
30 C = A/B
40 PRINT C
50 GOTO 10
60 IF ERR = 20 THEN PRINT "DIVISION BY ZERO":
  RESUME 20
70 IF ERR = 10 THEN PRINT "YOU NUMBERS
  CAUSED AN OVER FLOW": RESUME 20
```

## Program 2

```
5 REM ON ERROR GOTO DEMO #2
10 FOR C = 1 TO 10
20 ON ERROR GOTO 70
30 READ A
40 PRINT A
50 NEXT C
60 END
70 READ A$
80 PRINT A$
90 RESUME 50
100 DATA THE, NUMBERS, ARE, 1, 2, 3, 4, IS,
    THAT, OK
```

### ON n GOSUB

This works like On n GOTO, except control branches to one of the subroutines specified by the line numbers in the line number list.

Example:

```
10 PRINT "** FUNCTION SUBROUTINES **"
20 PRINT "1. FUNCTION A"
30 PRINT "2. FUNCTION B"
40 PRINT "3. FUNCTION C"
50 INPUT "ENTER 1, 2, OR 3"; N
60 ON N GOSUB 150, 100, 250
70 END
100 PRINT "THIS IS FUNCTION B": RETURN
150 PRINT "THIS IS FUNCTION A": RETURN
250 PRINT "THIS IS FUNCTION C": RETURN
```

If this is RUN once, let's choose 2.

```
** FUNCTION SUBROUTINES **
```

1. FUNCTION A
2. FUNCTION B
3. FUNCTION C

```
ENTER 1, 2, OR 3 ? 2
THIS IS FUNCTION B
```



Now 1

**\*\* FUNCTION SUBROUTINES \*\***

1. FUNCTION A
2. FUNCTION B
3. FUNCTION C

ENTER 1, 2, OR 3 ? 1  
THIS IS FUNCTION A

Try RUNning this program:

```
5 REM ON GOSUB DEMO
10 INPUT "ENTER A NUMBER BETWEEN 1 AND 4";
  N
20 ON N GOSUB 100, 200, 300, 400
30 GOTO 10
100 PRINT "YOU ENTERED ONE": RETURN
200 PRINT " YOU ENTERED TWO": RETURN
300 PRINT "YOU ENTERED THREE": RETURN
400 PRINT "YOU ENTERED FOUR": RETURN
```

ON n GOTO

This statement allows multi branching to the line numbers specified according to the value of n. The general format for ON n GOTO is:

ON expression GOTO 1st line number, 2nd line number, . . . , mth line number. The value of the expression must be between 0 and 255 inclusive.

When an ON-GOTO statement is executed, first, the expression is evaluated and the integer position, that is INT (expression) is obtained. Then the computer assigns this integer to N, and counts over to the Mth element in the line number list, and then branches to the line number specified by that element. If N is greater than the available line number M, the control falls through to the next statement in the program. If the expression or number is less than zero, an error will occur.

The line number list may contain any number of items.

Example:

```
10 INPUT "ENTER COMMAND"; C
20 ON C GOTO 100, 120, 130, 150, 130
30 PRINT "END OF PROGRAM": END
100 PRINT "THIS IS LINE 100": GOTO 10
120 PRINT "THIS IS LINE 120": GOTO 10
130 PRINT "THIS IS LINE 130": GOTO 10
150 PRINT "THIS IS LINE 150": GOTO 10
```

Let's RUN this a few times

```
ENTER COMMAND ? 5
THIS IS LINE 130
ENTER COMMAND ? 4
THIS IS LINE 150
ENTER COMMAND ? 1
THIS IS LINE 100
ENTER COMMAND ? 2
THIS IS LINE 120
ENTER COMMAND ? 3
THIS IS LINE 130
ENTER COMMAND ? 0
END OF PROGRAM
```

Now RUN again.

```
ENTER COMMAND ? 4
THIS IS LINE 150
ENTER COMMAND ? 6
END OF PROGRAM
```

The ON-GOTO statement is a more elegant way of achieving the same result than the equivalent IF-GOTO statements.

```
10 IF C = 1 GOTO 100
20 IF C = 2 GOTO 120
30 IF C = 3 GOTO 130
40 IF C = 4 GOTO 150
50 IF C = 5 GOTO 130
60 IF C < 1 OR C > 5 GOTO 70: REM GO TO THE
    NEXT STATEMENT.
```

## PRINT

The command prints an item or a list of items on the display. An item may be any of the following:

- a) Numeric constants (numbers such as 0.368 72, 0.2, -34)
- b) Numeric variables (names representing numeric values, such as X, Y, Z, etc.)
- c) String constants (characters enclosed in quotes, such as "HOME COMPUTER", "3003", etc.)
- d) String variables (names representing string or character values, such as A\$, B\$, etc.)
- e) Expressions (a sequence of any combination of the above, such as  $(X + 10)/Y$ , "BALL" + "PEN", etc.)

Items in the item list may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next printing zone before printing the next item. If semi-colons are used, no space is inserted between alphabetic items before printing on the display, but one space is inserted before each numeric item.

Example:

```
10 N = 25 + 7
20 PRINT "25 + 7 IS EQUAL TO"; N
30 END
```

This prints on the screen.

```
25 + 7 IS EQUAL TO 32
```

Example:

```
10 H$ = " HOME "
20 C$ = "COMPUTER"
30 PRINT "TRY OUR"; H$; C$
40 END
```

This displays:

```
TRY OUR HOME COMPUTER
```

When commas are used to separate items, 4 columns are acceptable per line. Each column consists of a maximum of 10 characters. Any string beyond this bound will be printed on the next line.

Example:

```
10 PRINT "COLUMN 1", "COLUMN 2", "COLUMN 3",  
"COLUMN 4", "COLUMN 5"  
20 END
```

When RUN gives:

```
COLUMN 1  COLUMN 2  COLUMN 3  COLUMN 4  
COLUMN 5
```

If two or more commas are applied together, each comma will still occupy 10 characters. (Blank spaces).

Example:

```
10 PRINT "COLUMN 1",,, "COLUMN 2"  
20 END
```

This gives:

```
COLUMN 1          COLUMN 2
```

Note the following examples:

```
10 PRINT "LINE ONE"  
20 PRINT "LINE TWO"  
30 END
```

This program gives:

```
LINE ONE  
LINE TWO
```

```
10 PRINT "LINE ONE",  
20 PRINT "LINE TWO"  
30 END
```

This one:

```
LINE ONE  LINE TWO
```

## PRINT @

This statement print out items in the item list at the screen location specified. The "@" sign must follow PRINT immediately, and the location specified must be a number of value from 0 to 959. For more details on the display map, please refer to the Beginner's Manual

Example:

```
20 PRINT @100 "LOC 100"
```

If you construct a PRINT @ statement to print on the bottom line of the display, there will be an automatic line-feed, causing everything display to move up one line.

To suppress this action, add a semi-colon at the end of the statement.

Example:

```
10 PRINT @ 959, "BOTTOM LINE";
```

For use of PRINT @ with graphics characters refer to Chapter 6.

## PRINT TAB

This allows you to print at any specified cursor position within a line. More than one TAB in a PRINT statement is acceptable. However, the value in the expression should be between 0 and 39 inclusive.

Example:

```
10 PRINT TAB (10) "POSITION 10" TAB (30)  
    "POSITION 30"  
20 END
```

The screen displays:

POSITION 10

POSITION 30

Example:

```
10 N = 4
20 PRINT TAB (N) "POS."; N TAB (N + 10)
   "POS."; N + 10 TAB (N + 20) "POS."; N + 20
30 END
```

This displays:

```
POS. 4      POS. 14      POS. 24
```

Example:

```
5 REM TAB DEMO
10 T = 38
20 PRINT TAB (T); "&"
25 IF T = 2 THEN STOP
30 T = T - 2
40 GOTO 20
```

## PRINT USING

This statement allows you to print the data with a pre-defined format. The data can be numeric or string values.

The format and item list in PRINT USING statement can be expressed as variables or constants. The statement prints the item list according to the format specified.

The following specifiers may be used in the format field.

- # This sign represents the proper position of each digit in the item list (for numeric value). The number of # signs used forms the format desired. If the format field is greater than the numeric value (in the item list), the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

The decimal point can be placed anywhere in the format field established by the # signs. Rounding off will take place if the digits to the right of the decimal point are suppressed.

The comma – when it is placed at any position between the first digit and the decimal point, a comma will be displayed to the right of every three digits.

Let us consider the following examples:

```
10 REM PRINT USING DEMO.  
20 INPUT "ENTER FORMAT"; F$  
30 IF F$ = "STOP" THEN STOP  
40 INPUT "ENTER A NUMBER"; N  
50 PRINT USING F$; N  
60 GOTO 10
```

This program requests inputs for the format field and item list (in this case with numeric value). The program will stop only if the user inputs the word "STOP" as the value for F\$.

Now RUN this program.

```
ENTER FORMAT ? # #. # #  
ENTER A NUMBER ? 12.34  
12.34
```

```
ENTER FORMAT ? # # #. # #  
ENTER A NUMBER ? 12.34  
12.34
```

```
ENTER FORMAT ? # #. # #  
ENTER A NUMBER ? 123.45  
%123.45
```

```
ENTER FORMAT ? STOP
```

The % sign will be automatically printed out if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal point will be displayed after the % sign.

Now RUN the program again.

```
ENTER FORMAT ? # #. # #  
ENTER A NUMBER ? 12.345  
12.35
```

```
ENTER FORMAT ? STOP
```

Since only two decimal places were specified, the numeric value will be rounded-off before displaying to the screen.

- (i) \*\* Two asterisks placed at the beginning of the format field will cause all unused positions to the left of the decimal point to be filled with asterisks. The two asterisks will establish two more positions in the field.
- (ii) \$\$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is: A dollar sign will occupy the first position preceding the number.
- (iii) \*\*\$ Combines the effects of \*\* and \$\$ . Any empty position to the left of the number will be filled by the \* sign and the \$ sign will also occupy the first position preceding the number.

Let us use the same example as before:

```
ENTER FORMAT ? * * # # . # #  
ENTER A NUMBER ? 12.3  
**12.30
```

```
ENTER FORMAT ? $ # # . # #  
ENTER A NUMBER? 12.34  
$12.34
```

```
ENTER FORMAT ? * * $ # # # . # #  
ENTER A NUMBER ? 12.34  
***$12.34
```

```
ENTER FORMAT ? STOP
```

- (iv) + When a "+" sign is placed at the beginning or at the end of the format field, the computer will print a + sign for a positive number or a - sign for a negative number at the specific position accordingly.
- (v) - When a "-" sign is placed at the end of the format field, it will cause a negative sign to be printed after any negative number, and will display as a blank for positive numbers.



Examples (using the same program as above):

```
ENTER FORMAT ? + # # . # #  
ENTER A NUMBER ? 12.34  
+12.34
```

```
ENTER FORMAT ? + # # . # #  
ENTER A NUMBER ? -12.34  
-12.34
```

```
ENTER FORMAT ? # # . # # +  
ENTER A NUMBER ? -12.34  
12.34-
```

```
ENTER FORMAT ? # # . # #  
ENTER A NUMBER ? 12.34  
12.34
```

```
ENTER FORMAT ? # # . # # #  
ENTER A NUMBER ? 123 456  
% 123 456.000
```

```
ENTER FORMAT ? STOP
```

(vi) % space %

To define a string field of more than one character. The length of the format field will be 2 plus the number of spaces between the percentage signs. An exclamation mark (!) informs the computer to use only the first character of the current string value.

Consider the following program example:

```
10 INPUT "ENTER FORMAT"; F$  
20 IF F$ = "STOP" END  
30 INPUT "ENTER A STRING"; C$  
40 PRINT USING F$; C$  
50 GOTO 10
```

This program performs similarly to the ones we just used. The only difference is that, the user has to input a string value instead of a numeric value for the second data entry. This is, the variable C\$.

Now let us run the program and test its function.

```
ENTER FORMAT ?!  
ENTER A STRING ? ABCDE  
A  
ENTER FORMAT ?% %  
ENTER A STRING ? ABCDE  
ABC
```

```
ENTER FORMAT ?% %  
ENTER A STRING ? ABCDEF  
ABCDE
```

```
ENTER FORMAT ? STOP
```

(viii) ! By using the ! sign we can also concatenate, or join strings together.

Example:

```
10 INPUT "ENTER THREE STRINGS"; A$, B$, C$  
20 PRINT "THE RESULT IS: " PRINT USING  
"!!!"; A$; B$; C$;
```

Now run the program.

```
ENTER THREE STRINGS ? ABC, XYZ, IJK  
THE RESULT IS: AXI
```

```
ENTER THREE STRINGS ? A, COMPUTER,  
PROGRAM  
THE RESULT IS: ACP
```

By using more than one "!" signs the first letter of each string will be printed with spaces inserted corresponding to the space inserted between the "!" signs.

Try to follow this example:

```
10 INPUT "ENTER THREE STRINGS"; A$, B$, C$  
20 PRINT "THE RESULT IS: " PRINT USING  
"! ! !"; A$; B$; C$;  
30 END
```

We now get:

```
ENTER THREE STRINGS ? XYZ, FGH, ABC  
THE RESULT IS: X F A
```

and again:

```
ENTER THREE STRINGS ? A, COMPUTER,  
PROGRAM  
THE RESULT IS: A C P
```

The following program shows how the DEFDBL command and PRINT USING command can be used to overcome the problems inherent in Colour Genie BASIC.

Example:

```
5 REM DEFDBL & PRINT USING DEMO  
10 DEFDBL N  
20 N = 0: M = 0  
30 N = N + .1: M = M + .1  
40 PRINT M, USING "###. #####"; N  
50 IF N > 10 THEN STOP  
60 GOTO 30
```

### PRINT # – CASSETTE NUMBER, ITEM LIST

This statement prints the value of the specified variables onto cassette tape. The recorder must be properly set in record mode before executing this statement.

Example:

```
10 A$ = "BEGIN TAPE"  
20 B = 3.141 6  
30 C = 50  
40 D$ = "DATA"  
50 PRINT # -1, A$, B, C, D$, "END OF FILE"  
60 END
```

This program assigns various data to variables AS, B, C, and DS respectively, then PRINT these data on tape through cassette drive No. 1. Note that the string constant "END OF FILE", can be printed on tape as well as variables. Once the data are stored on tape, you may input these data into the computer again, just like playing music tapes with a cassette. Please note that the INPUT statement must be identical to the PRINT statement in terms of number and types of variables. However, the variable names may be different in any case.

Important:

The total number of characters represented in all the variables mentioned in the "item list" must not exceed 255; otherwise anything after the 255th character will be truncated or lost.

Example:

```
10 PRINT = -1, AS, BS, CS, DS, ES
```

If the total number of characters in AS, BS, CS, DS, are 250 and ES has a length of 35 characters, then ES will not be saved on tape. And if you try to INPUT the value of ES, an Out-of-Data error will occur.

## READ

This statement instructs the computer to read in a value from a DATA statement and assign that value to the specified variable. The values in the DATA statement will be read sequentially by the READ statement. After all the times in the first DATA statement have been read, the next READ statement encountered will access the second DATA statement for the next variable. If there is no more value in the DATA statement available for a READ statement an Out-of-Data error will occur.

Consider the following example.

```
10 READ C$
20 IF C$ = "EOF" GOTO 60
30 READ Q
40 PRINT C$, Q
50 GOTO 10
60 PRINT; PRINT "END OF LIST.": END
70 DATA BOOKS, 4, PENCILS, 12
80 DATA BALL PENS, 5, COMPASSES, 2
90 DATA GLASSES, 5, EOF
```

This program when RUN displays:

```
BOOKS          4
PENCILS        12
BALL PENS      5
COMPASSES     2
GLASSES       5
END OF LIST
```

## RESTORE

RESTORE allows the next READ statement to access the first item in the first DATA statement, and the subsequent items.

Example:

```
10 READ A$, A
20 PRINT A$, A
30 RESTORE
40 READ B$, B
50 PRINT A$, A, B$, B
60 DATA "JOHN WHITE", 25, "JOE HUDSON",
      32, "BILL ADAMS", 30
70 END
```

We get:

```
JOHN WHITE    25
JOHN WHITE    25
JOHN WHITE    25
```

This program shows that the RESTORE statement not only allows the READ statement to access the first item in the first DATA statement, but also it has no effect on the previous assignments.

## RETURN

This statement ends a subroutine and returns control to the statement that immediately follows the GOSUB. An error will occur if RETURN is encountered without execution of a matching GOSUB.

## RESUME

This statement terminates an error handling routine by specifying where normal execution is to resume.

RESUME 0 or RESUME without a line number causes the computer to return to the statement in which the error occurred. IF RESUME is followed by a line number, it causes the computer to branch to the line number provided.

RESUME NEXT causes the computer to branch to the statement following the point at which the error occurred.

Example:

```
10 ON ERROR GOTO 80
20 PRINT "SIMPLE DIVISION."
30 INPUT "ENTER TWO NUMBERS": A, B
40 IF A = 0 THEN END
50 C = A/B
60 PRINT "THE QUOTIENT IS"; C
70 GOTO 20
80 PRINT "ATTEMPT TO DIVIDE BY ZERO!"
90 PRINT "TRY AGAIN ..."
100 RESUME 20
```

Let's RUN this

```
SIMPLE DIVISION
ENTER TWO NUMBERS ? 6, 2
THE QUOTIENT IS 3
SIMPLE DIVISION
ENTER TWO NUMBERS ? 7, 3
THE QUOTIENT IS 2.333 33
SIMPLE DIVISION
ENTER TWO NUMBERS ? 5, 0
ATTEMPT TO DIVIDE BY ZERO!
TRY AGAIN . . .
SIMPLE DIVISION
ENTER TWO NUMBERS ? 9, 4
THE QUOTIENT IS 2.25
SIMPLE DIVISION
ENTER TWO NUMBERS? 0, 0
```

## REM

REM represents remarks. This statement informs the computer that the rest of the line only consists of comments, and should be ignored. The statement also allows you to have more comments in your program for better documentation. If REM is used in a multi-statement program line, it must be the last statement.

Example:

```
10 REM * VARIABLE REPRESENTATIONS *
20 REM * A = AMOUNT *
30 REM * B = NUMBER OF ITEMS *
40 REM * C = UNIT COST *
50 REM *
60 A = B * C: REM ** AMOUNT = NO. OF ITEMS x
   UNIT COST
```

## STOP

This statement is essentially a debugging aid. It sets a break point in a program during execution, and allows you examine or modify variable values. A message will be printed out as "BREAK IN line number" once the computer executes the STOP statement. The active command CONT can then be used to re-start execution at the point where it breaks.

Example:

```
5 INPUT B, C
10 A = B + C
20 STOP
30 X = (A + D)/0.74
40 IF X < 0 GOTO 70
50 PRINT A, B, C
60 PRINT X
70 END
```

Let's enter some numbers:

```
? 2, 4
```

The STOP statement allows the user to examine the value of A before line 30.

```
BREAK in 20
READY
> PRINT A
6
READY
> CONT
6 2 4
8.10811
```



# CHAPTER 4 : ARRAYS AND STRINGS

## DIM ARRAY (dim 1, . . . , dim n)

This dimensions any arrays used. Colour Genie BASIC allows you to use arrays with up to 11 dimensions without declaring their size. Any more than 11 they must be declared. It is a good idea to dimension arrays all the time as this brings that array to the users' attention each time the program is listed.

Default dimensioning is dynamic so that no array space is lost by not declaring them.

Example:

Let us write a program that allows you to enter up to 45 names in a class list and then PRINT them out when it is RUN. It could be something like this!

```
5 CLEAR 1000: REM CLEAR 1000 BYTES FOR  
  STRING STORAGE  
10 DIM AR$ (44): REM ARRAY AR$ HAS 45  
  ELEMENTS  
15 REM ** INPUT ARRAY SECTION **  
20 FOR N = 0 TO 44: REM LOOPS 45 TIMES  
30 INPUT "ENTER THE NAME OF THE STUDENT":  
  AR$(N)  
40 REM ASSIGN THE NAMES TO EACH ELEMENT  
  IN THE ARRAY  
50 NEXT N  
55 REM ** PRINT ARRAY SECTION **  
60 FOR N = 0 TO 44: REM LOOPS 45 TIMES  
70 PRINT AR$(N): REM PRINTS THE N TH  
  ELEMENT OF THE ARRAY  
80 NEXT N  
90 END
```

## STRINGS

String operations are the essence in data processing and Colour Genie BASIC allows many useful string operations in addition to arithmetic operations.

### STRING COMPARISON

By using a relational operator, two strings may be compared for equality or alphabetic precedence. If they are checked for equality, every character, including any leading or trailing blanks, must be identical otherwise the test fails.

Example:

```
100 IF A$ = "YES" THEN 250
```

Strings are compared character by character from left to right. In Colour Genie BASIC the ASCII code representations for the characters are compared. A character with the lower code number is considered to precede the other character. In other words, "AB" precedes "AC". When strings of different lengths are compared, the shorter string is given precedence even if its characters are identical as those in the longer string. Therefore, "B" precedes "B ". The following relational operators may be used to compare strings.

<, <=, >=, >, =, <>

### STRING OPERATION

There is really only one string operation that is concatenation which is represented by the plus sign "+".

Example:

```
10 S1$ = "THE SUN IS"  
20 S2$ = " SHINING"  
30 S3$ = " , "  
40 C$ = S1$ + S2$ + S3$ + S2$  
50 PRINT C$  
60 END
```

This program gives us: THE SUN IS SHINING, SHINING.

## ASC (STRING)

This statement returns the ASCII code (in decimal) for the first character of the specified string. The string specified must be enclosed in parentheses. A null-string will cause an error to occur.

```
100 PRINT "ASCII CODE FOR 'H' IS:"; ASC ("H")
105 SS = "HOME": PRINT "THE STRING IS:"; SS
110 PRINT "THE ASCII CODE FOR THE FIRST
    LETTER IS:"; ASC (SS)
120 END
```

giving:

```
THE ASCII CODE FOR 'H' IS: 72
THE STRING IS: HOME
THE ASCII CODE FOR THE FIRST LETTER IS: 72
```

Both lines will print the same number.

A complete set of control, graphics, and ASCII codes is listed in Appendix C.

## CHR\$

This statement is essentially the inverse of the ASC function. It returns the character of the specified ASCII, control or graphics code. The argument may be any number from 0 to 255, or any variable expression with a value within that range. The argument must be enclosed in parentheses.

So to print an exclamation mark (code 33) enter:

```
100 PRINT CHR$ (33)
```

Example:

```
5 REM CHR$ DEMO
10 FOR G = 30 TO 255
20 PRINT CHR$ (G);
30 PRINT " ";
40 NEXT G
```

This program points out all the Colour Genie BASIC characters.

## FRE (STRING)

This function returns the amount of available memory for string storage. The string is a dummy variable.

```
5 REM FRE DEMO
10 CLEAR 75
20 PRINT STRING$ (60, "#")
30 PRINT FRE ("#")
```

## LEFT\$ (STRING, n)

This statement returns the first *n* characters of the specified string. The argument must be enclosed in parentheses. The string may be a constant or an expression, and *n* may be a numeric expression.

Example:

```
10 A$ = "ABCDEFGH"
20 B$ = LEFT$ (A$, 4)
30 PRINT B$
40 END
```

If RUN we get: ABCD

## RIGHT\$ (STRING, n)

This returns the last *n* characters of a string and *n* must be enclosed in parentheses. String may be a string constant or variable, and *n* may be a numerical constant or variable. If the length of the string is less than or equal to *n*, the entire string is returned.

Example:

```
10 A$ = "ABCDEFGH"
20 B$ = RIGHT$ (A$, 3)
30 PRINT B$
40 END
```

This program prints the right hand letters: EFG.

## LEN (STRING)

This returns the length of the specified string. The string may be a variable, expression or constant and must be enclosed in parentheses.

Example:

```
5 REM LEN DEMO.  
10 AS = "UVWXYZ"  
20 FOR C = 1 TO LEN (AS)  
30 SS = LEFTS (AS, C)  
40 PRINT SS  
50 NEXT C
```

## MIDS (STRING, p, n)

This statement returns a substring of string starting at position p, with length n. The string, position and length must be enclosed in parentheses. String may be a constant or an expression, p and n may be numeric expressions or constants.

Example:

```
10 AS = "ABCDEFGH"  
20 BS = MIDS (AS, 3, 4)  
30 PRINT "THE NEW STRING IS: "; BS  
40 END
```

When RUN gives: THE NEW STRING IS: CDEF

## STR\$ (EXPRESSION)

This converts a constant or numeric expression into a string of characters. The expression or constant must be enclosed in parentheses.

Example:

```
5 REM STRING DEMO  
10 A = 12  
20 BS = STR$ (A)  
30 CS = BS+ " STONE"  
40 PRINT CS
```

## STRINGS (n, CHARACTER OR NUMBER)

This returns a string which is composed of n (number) of the specified characters.

Example:

```
10 PRINT STRINGS (10, "*" )
20 END
```

which displays:

```
*****
```

In this statement the character may be a number from 0-255. In the next example it will be treated as an ASCII, control or graphics code.

```
10 PRINT STRINGS (10, 33)
20 END
```

33 is the ASCII code for an exclamation mark so we get:

```
!!!!!!!!!!
```

## VAL (STRING)

This is the inverse of the STR\$ function. It returns the numeric value of the characters in a string argument.

Example:

```
10 A$ = "56"
20 B$ = "23"
30 C = VAL (A$ + "," + B$)
40 PRINT "THE RESULTS ARE:"; C; " , "; C + 100
50 END
```

which displays:

```
THE RESULTS ARE: 56.23, 156.23
```

Here are some more programs illustrating the string functions discussed above:

Example 1:

```
10 REM BINARY TO DEC. CONVERT; VAL DEMO
20 F = 0: E = 0
30 PRINT "ENTER BINARY NUMBER WITH
    LEADING ZEROS INCLUDED."
40 INPUT "EIGHT DIGITS ARE REQUIRED"; B$
50 FOR D = 1 TO 8
60 READ A
70 E = VAL (MID$ (B$, D, 1))
80 F = F + E * A
90 NEXT D
100 PRINT F
110 RESTORE
120 DATA 128, 64, 32, 16, 8, 4, 2, 1
130 GOTO 10
```

Example 2:

```
10 REM MID$ DEMO. ONES COMPLEMENT
20 INPUT "ENTER BINARY NUMBER"; B$
30 C$ = " "
40 FOR N = 1 TO LEN (B$)
50 IF MID$ (B$, N, 1) = "1" THEN D$ = "0"
    ELSE D$ = "1"
60 C$ = C$ + D$
70 NEXT N
80 PRINT C$
```

# CHAPTER 5 :

## DOING ARITHMETIC

In this chapter, we discuss the built-in functions available in Colour Genie BASIC. In most cases, it is necessary to pass an argument (initial value) to the function, before a desired value (result) is returned. The argument may be a constant, a numeric variable, or an expression. The general format is:

result = function (argument)

Example:

```
10 A = RND (3)
20 B = INT (C)/D
30 E = SQR (F * G - H)
```

### ABS (X)

Returns the absolute value of the argument X.

### ATN (X)

Returns the arctangent function (in radians) of the argument. To get the arctangent in degrees, multiply ATN (X) by 57.295 78.

### CDBL (X)

Returns the double-precision representation of the argument. The value returned contains 17 digits, however, only the digits contained in the argument will be significant.

### CINT (X)

Returns the largest integer that is not greater than the argument. The argument must be within the range of  $-32\,768 + 32\,767$ . For example, CINT (2.6) returns 2; CINT (-2.6) returns -3.



## COS (X)

Returns the cosine function of the argument (in radians). In order to obtain the cosine of X when X is in degrees, use  $\text{COS}(X * .0174533)$ .

## CSNG (X)

Returns a single-precision representation of the argument. It returns a 6 significant digit number with 4/5 rounding for a double precision argument.

## EXP (X)

Returns the "natural exponential" of X, that is  $e^x$ . This is the inverse of the LOG function.

## FIX (X)

Returns a truncated representation of the argument with all digits on the right of the decimal point being truncated or chopped off. For example,  $\text{FIX}(1.5)$  returns 1,  $\text{FIX}(-1.5)$  returns -1.

## INT (X)

Returns an integer representation of the argument, using the largest integer that is not greater than the argument. The argument is not limited to the range -32 768 to +32 767. For example,  $\text{INT}(3.5)$  returns 3,  $\text{INT}(-3.5)$  returns -4.

```
5 REM INT DEMO
10 X = 10 * 192.6
20 Y = INT(10 * 192.6)
30 PRINT X
40 PRINT Y
50 PRINT X - Y
```

## LOG (X)

Returns the natural logarithm of the argument, that is  $\log_e (X)$ . To find the logarithm of a number of another base  $b$ , use the formula  $\log_b (X) = \log_e (X) / \log_e (b)$ .

## RANDOM

This function causes the computer to generate a new set of random numbers every time when the computer is turned on and runs a program which has RND functions. No argument is needed in this function.

## RND (X)

Returns a pseudo-random using the current pseudo-random number (generated internally and has not access to the user). RND (0) returns a single-precision value between 0 and 1, RND (X) returns an integer between 1 and X inclusive. However, X must be positive and less than 32 768.

## SGN (X)

The "sign" function, that is to return -1 if X is negative, 0 if X is zero, and +1 if X is positive.

## SIN (X)

Returns the sine function of the argument (in radians). To obtain the sine of X when X is in degrees use SIN (X\*, 0.174533)

## SQR (X)

Returns the square root of the argument.

## TAN (X)

Returns the tangent function of the argument (in radians).  
To obtain the tangent of X and X is in degree, use  
`TAN (X * .017 453 3)`.

# CHAPTER 6 :

## GRAPHICS

The Colour Genie is able to do some quite amazing things with graphics. As you saw in the Beginner's Manual there are two pages that can be used with this computer. One is referred to as the low graphics page, or LGR, and is used for text and the standard graphics characters. The other page is for full graphics and is called FGR. It is not possible to use text or the keyboard graphics on the FGR page nor the graphics commands on the LGR page. They are two distinct entities and must be treated as such.

This chapter will look at each page in turn. It is possible to switch between the two pages both from the keyboard and within a program. It is recommended that you familiarise yourself with the graphics section of the Beginner's Manual before going any further in this one!

### LGR PAGE

This page is divided into 960 sections, or pixels, on a 24 by 40 matrix. The LGR screen is 40 columns wide and 24 rows high. It is possible to address each one of the pixels individually and place characters from the Colour Genie graphics set into one of these positions. They are numbered 0 to 959 as shown on the diagram in the Beginner's Manual.

It is possible to supplement the Colour Genie characters with another 128 programmable ones. As there are 128 upper and lower case characters in the ASCII character set, and another 64 graphics characters available from the keyboard some method must be used to allow us to tell the computer what set to use.

## CHAR

CHAR defines the character set being used. The Colour Genie has four character sets so n can have a value 1 to 4. That appropriate set is given by the following table:

n	1	2	3	4
ASCII code				
0 – 127	alpha	alpha	alpha	alpha
128 – 191	prog	prog	graph	graph
192 – 255	prog	spec	prog	spec

alpha — These are alphanumeric characters and are shown on the Keytops.

prog — These are user definable graphics.

spec — These are the graphics characters shown on the front of the Keys.

graph — These are special graphics characters and are used for non English alphabets.

The CHAR n command is mainly for use with system commands although it can be used within a program. If it is then do not count on the default character set i.e. alphanumeric to be available when the program is finished.

## CLS

CLS clears the screen both when used directly from the keyboard or within a program. It is especially useful when used at the beginning of a program using graphics.

The cursor is returned to the top left hand corner of the display.

## COLOUR n

The Colour Genie has the ability to show up to 8 colours on the LGR page. This is the one that is normally used for text entry. These colours are indicated on the keys and the n refers to the key used to get that colour.

Format: COLOUR n

where n has a value between 1 and 8

n	colour
1	white
2	green
3	red
4	yellow
5	orange
6	blue
7	cyan
8	magenta

Colour can be changed within programs. The system stays in the last colour used even if this was in a program.

It is possible to change the colour of the LGR page directly from the keyboard. If the CTRL key is pressed before one of the number keys then the following text or graphics characters will be in the appropriate colour. They will stay in that colour until changed.

Example 1:

```
1 REM COLOUR DEMONSTRATION
5 CLS
10 FOR L = 1 TO 12
20 C = RND (8)
30 COLOUR C
40 READ D$
50 PRINT @ 400 + L, D$
60 NEXT L
70 DATA C, O, L, O, U, R, , G, E, N, I, E
80 RESTORE
90 GOTO 10
```

Example 2:

```
5 REM ANOTHER COLOUR DEMO
10 FOR C = 1 TO 8
20 COLOUR C
30 PRINT STRING$(10, "■"); C
40 NEXT C
```

## LGR

This command is usually used within a program and instructs the computer to revert to displaying the LGR page.

## POS (DUMMY ARGUMENT)

This command returns the current cursor position, between 0 and 39. The dummy argument is usually 0.

Example:

```
10 FOR X = 1 TO 20
20 PRINT STRING$(X, "*");
30 PRINT POS(0)
40 NEXT X
```

## PRINT @ n, ITEM

PRINT @ n, item or even? @ n, item, can be used in conjunction with FOR . . . NEXT loops to give something equivalent to the PLOT command available on the FGR page.

'Item', if a variable can be left as that variable but if text must be within quotation marks. Other rules of printing also apply to this command i.e. ', ' and '; '.

Example:

```
10 REM GROWTH, PRINT @ DEMO
20 CLS
30 P = RND (900)
40 PRINT @ P, "#"
50 D = RND (5)
60 ON D GOTO 70, 80, 90, 100, 110
70 N = -40: GOTO 120
80 N = -1: GOTO 120
90 N = 0: GOTO 120
100 N = 1: GOTO 120
110 N = 40: GOTO 120
120 IF P < 40 THEN N = 40
130 IF P > 800 THEN N = -40
140 P = P + N
150 PRINT @ P, "#"
160 GOTO 50
```

Run this program and the # sign is printed at random locations.

## PROGRAMMABLE GRAPHICS CHARACTERS

The Colour Genie can have up to 128 graphics characters that can be programmed by the user. A fairly complicated shape table needs to be drawn up that involves rather complex use of POKEing and Octal to decimal conversion. To make life easier we have developed the following BASIC program that lets you create your characters in a simple way!

The graphics on the LGR page are made up of a matrix of dots which are 8 dots wide and 8 dots high. To create a graphics character draw a diagram on an 8 x 8 grid. Put ones in the squares where you want a dot to appear on the screen and zeros where you want the screen dark.

This data will be transferred to the conversion program. The example on page 88 shows the character PI.



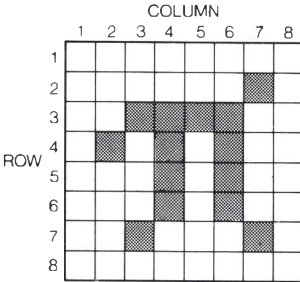
When the program is RUN it will ask you for the data for the first line. Just enter the ones and zeros when reading from the left for each row. So for the pi character this would be:

```

0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 1 1 1 0 0
0 1 0 1 0 1 0 0
0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 0
0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0

```

DIAGRAM TO CREATE 'PI'



You also need to have decided what code you are going to allocate this character to. The program line 30 is where this data is entered and you need to know that ASCII code for that character. The ASCII code number must be between 128 and 255. The ensuing character will not be available from the keyboard and must be called up using the CHR\$ command. The program displays the character as it is being formed.

Let's allocate the code 128 for it. To access it we then type

```
PRINT CHR$(128)
```

and the graphic character of PI will be displayed.

Example:

```
10 CLS: CHAR1
20 REM PROG. GRAPHICS PROGRAMMER
30 REM BY D.C. EVENDEN
40 S = -3072: L = 8: C = 128: N = 1
50 PRINT "ENTER ASCII CODE REQUIRED"
60 INPUT "BETWEEN 128 AND 255"; A
70 PRINT "ENTER DATA FOR CHARACTER"; A;
   " "; CHR$(A)
80 S1 = S + (A - C) * 8
90 FOR ADDR = S1 TO S1 + 7
100 PRINT "ENTER DATA FOR LINE"; N;
110 INPUT D$
120 GOSUB 200
130 POKE ADDR, D
140 N = N + 1
150 NEXT ADDR
160 PRINT "THE CHARACTER AT ASCII CODE";
   A; "IS "; CHR$(A)
170 INPUT "PRESS ANY KEY TO CONTINUE"; F$
180 GOTO 10
190 REM BINARY TO DECIMAL
200 D = 0
210 FOR P = 1 TO 8
220 READ B
230 D = D + B * VAL (MID$(D$, P, 1))
240 NEXT P
250 DATA 128, 64, 32, 16, 8, 4, 2, 1
260 RESTORE
270 RETURN
```

## FULL GRAPHICS PAGE FGR

The full graphics page FGR is accessed from the keyboard by pressing CTRL and MODSEL at the same time. It can be accessed from within a program by the FGR command.

The FGR page has a resolution of 160 columns by 96 rows. Each point, or pixel, is accessed on an x, y grid as shown in the diagram on page 12 of the Beginner's Manual. All drawing and other similar commands make use of this grid and it is very useful if you become familiar with the boundaries of the FGR area before you go much further. At the start of any program that uses the FGR page the various parameters referring to that page must be set up. These include background colour, the fact that the FGR page is to be used etc. These commands are described and illustrated below.

### BGRD

When used in a program this command automatically colours the high resolution display background PINK. The following example shows a blue circle on a pink background:

```
10 FCLS : FGR
20 BGRD
30 FCOLOUR 2
40 CIRCLE 80, 47, 30
50 GOTO 50
```

### NBGRD

This command disables the BGRD command; when used in a program, the PINK background colour reverts back to a black (or blank) background. Let's use this command in the previous example – by pressing any Key, the background colour is changed from PINK to BLACK.

Example:

```
10 FCLS : FGR
20 BGRD
30 FCOLOUR 2
40 CIRCLE 80, 47, 30
50 IF INKEY$ = "" THEN 50
60 NBGRD
70 GOTO 70
```

### CIRCLE x, y, r

This command draws circles on the FGR page x, y are the coordinates of the centre of the circle and r the radius. x has a value between 0 and 159 and y a value between 0 and 95.

Illustration: `CIRCLE 40, 40, 40`

draws a circle in the top left hand corner of the FGR screen.

### CPOINT (x, y)

This is a useful programming tool as it allows you to read the colour of any point (X, Y) in the high resolution display by the following code relationship:

CODE	COLOUR
0	BLACK
1	BLUE
2	RED
3	GREEN

Example:

```
10 FILL 3
20 A = CPOINT (30, 30)
30 PRINT A
```

Run this program and you will see that A = 2  
i.e. the colour of A (POINT 30, 30) is RED.

## FCOLOUR (n)

This command sets the colour of any graphics drawn on the FGR page. n can have a value between 1 and 4. The colour defaults to black if a value of n is not specified.

<u>n</u>	<u>colour</u>
1	black
2	blue
3	red
4	green

This command is usually used within a program and with the FILL command too.

Illustration:     5 FCLS: FGR  
                  10 FOR R = 1 TO 46  
                  20 FCOLOUR (RND (4))  
                  30 CIRCLE 79, 47, R  
                  40 NEXT R  
                  50 GOTO 10

## FCLS

FCLS clears the FGR page and is usually used within a program.

## FGR

FGR switches the display to the full graphics page (FGR).

This command is used with FCLS and FCOLOUR within a program as illustrated below.

```
5 REM FCOLOUR DEMONSTRATION
10 FCLS: FGR
20 FOR R = 1 TO 46
30 FCOLOUR (RND (4))
40 CIRCLE 79, 47, R
50 NEXT R
60 GOTO 10
```

## FILL n

FILL fills the FGR screen with background colour. n has a value between 1 and 4.

<u>n</u>	<u>colour</u>
1	black
2	blue
3	red
4	green

When using FILL and FCOLOUR the user must remember not to have the background and graphics colour the same! FILL1 defaults to black, which is the background colour when the computer is powered up.

Example:

```
10 FCLS : FGR
20 FOR N = 1 TO 4
30 FILL N
40 IF INKEYS= "" THEN 40
50 NEXT
60 GOTO 20
```

Run this program and by pressing any Key, the colour of the screen display will change to the 4 different colours.

## NPLOT x, y

This command is similar to PLOT x, y except it unplots the line drawn with PLOT x, y.

## NSHAPE

This command is functionally similar to the SHAPE command except that NSHAPE will blank the picture or a drawn by SHAPE.

Example:

```
10 FCLS : FGR
20 A = 7F00
30 FOR S = 0 TO 3
40 READ D
50 POKE A + S, D
60 NEXT
70 SCALE 3
80 SHAPE 30, 30
90 IF INKEY$= "" THEN 90
100 NSHAPE 30, 30
110 GOTO 110
120 DATA 3, 68, 119, 68
```

## PAINT x, y, c, b

The PAINT command fills in simple shapes created by CIRCLE or PLOT for example. Complex shapes are not easily filled in with a single PAINT command and may take a number of repeated uses. x and y represent a point somewhere inside the shape to be PAINTed. The shape itself must be a closed area. If it is not some very strange things happen! c is the colour that is to fill the shape as defined in the FCOLOUR command. b is the boundary colour taken from the same table. It is possible to have b and c the same.

The example below shows how the PAINT command can be used. Concentric circles, for example, must be drawn by "superimposing" one on top of the other. This command will not fill the "donut" area in one go. It is not advisable to have the starting point on the boundary of the shape to be PAINTed.

```
5 REM PAINT DEMO.  
10 FGR: FCLS: FCOLOUR 3,  
20 CIRCLE 50, 50, 20  
30 PAINT 50, 50, 4, 3  
40 GOTO 40
```

### PLOT x1, y1 TO x2, y2 TO x3, y3, . . . xn, yn

The PLOT command draws a line between the coordinates specified. These can be joined with the TO command. The line is plotted in the colour defined in the FCOLOUR statement.

Try RUNNING the following programs to get the hang of the commands:

#### Program 1:

```
5 REM FGR PAGE COLOUR DEMO  
10 FCLS: FGR  
20 FCOLOUR (RND (3) + 1)  
30 FORX = 0 TO 159 STEP 4  
40 PLOT A, B TO X, 0  
50 PLOT A, B TO X, 95  
60 NEXT X  
70 FOR Y = 0 TO 95 STEP 3  
80 PLOT A, B TO 0, Y  
90 PLOT A, B TO 159, Y  
100 NEXT Y  
110 A = RND (128)  
120 B = RND (90)  
125 FOR D = 1 TO 300: NEXT  
130 GOTO 10
```



Program 2:

```
5 REM WRITING "COLOUR" ON FGR PAGE
10 FGR: FCLS
20 FCOLOUR (RND (3) + 1)
30 CIRCLE 20, 50, 10
40 FCOLOUR (1)
50 FOR P = 2 TO 0 STEP -1
60 PLOT 27 + P, 40 TO 27 + P, 60
70 NEXT P
80 FCOLOUR (RND (3) + 1)
90 CIRCLE 40, 50, 10
100 FCOLOUR (RND (3) + 1)
110 PLOT 56, 40, TO 56, 60 TO 65, 60
120 FCOLOUR (RND (3) + 1)
130 CIRCLE 79, 50, 10
140 FCOLOUR (RND (3) + 1)
150 CIRCLE 97, 45, 5
160 PLOT 94, 40 TO 94, 60
170 PLOT 94, 50 TO 104, 60
180 FOR F = 3 TO 1 STEP -1
190 FCOLOUR (1)
200 PLOT 94 - F, 40 TO 94 - F, 50
210 NEXT F
220 GOTO 20
```

Here is a rather good program that shows the FGR page off to effect. It plots a three dimensional pattern on the screen. Be patient as it takes a few minutes to finish!

```
10 REM, 3D, PLOT
20 FGR: FCLS: FCOLOUR 3
30 H = 159
40 V = 95
50 D1 = H/2
60 D2 = D1 * D1
70 E1 = V/2
80 E2 = V/4
90 FOR D = 0 TO D1
100 D4 = D * D
110 M = -E1
120 A = SQR (D2 - D4)
130 FOR I = -A TO A STEP 5
140 S = SQR (D4 + I * I)/D1
160 E = I/5 + F * E2
170 IF E <= M THEN GOTO 1 000
180 M = E
190 E = E1 + E
200 X = D1 - D
210 Y = (3 * E) - 75
220 GOSUB 2 000
230 X = D1 + D
240 GOSUB 2 000
1 000 NEXT I
1 010 NEXT D
1 020 END
2 000 PLOT X, 96 - (Y/2)
2 010 RETURN
```

There are lots of other things that you can do with SIN and COS, like draw spirals and other interesting shapes. Try RUNning the next few programs.

```
5 REM SPIRAL
10 FCLS: FGR
20 FOR X = 0 TO 50 STEP .15
30 IF COS (X) > .9 GOSUB 70
40 PLOT 20 * COS (X) + 65, X * SIN (X) + 50
50 NEXT X
60 GOTO 20
70 FCOLOUR (RND (3) + 1)
80 RETURN
```

This one also does some interesting things!

### Program 1

```
10: FCLS: FGR
20 FCOLOUR 3
30 PLOT 0, 0 TO 0, 96
40 PLOT 0, 48 TO 157, 48
50 FOR X = 0 TO 15.8 STEP .1
60 LET X1 = X * 10
70 Y = 40 * SIN (-X) + 48
80 Z = -40 * COS (X) + 48
90 FCGLOUR 2
100 PLOT X1, Y
110 FCOLOUR 4
120 PLOT X1, Z
130 NEXT X
140 GOTO 140
```

### Program 2

```
5 REM PLOT DEMO.
10 FGR: FCLS
20 FOR A = 1 TO 60 STEP 2
30 FCOLOUR (RND (3) + 1)
40 FOR X = 0 TO 20 STEP .1
50 PLOT 5 * X + A, 10 * SIN (X) + 10 + A
60 NEXT
70 NEXT
80 GOTO 80
```

### Program 3

```
5 REM PLOT & RND DEMO.  
10 FCLS: FGR  
20 M = 0: N = 0  
30 X = RND (159)  
40 Y = RND (95)  
45 FCOLOUR (RND (3) + 1)  
50 PLOT M, N TO X, Y  
60 M = X: N = Y  
70 GOTO 30
```

### Program 4

```
5 REM DAMPED OSC. DEMO  
10 FGR: FCLS  
20 FOR A = 0 TO 20 STEP 2  
30 FCOLOUR (RND (3) + 1)  
40 FOR X = 0 TO 30 STEP .2  
50 PLOT 5 * X + A, X * SIN (X) + 35 + A  
60 NEXT X  
70 NEXT A  
80 GOTO 80
```

### SCALE n

SCALE is used with the SHAPE command to define the scale at which the shape created should be plotted. For example SCALE 2 produces a shape that is twice the size of SCALE 1. When power-on, the system defaults SCALE to 1. SCALE 0 will cause FC error.

### SHAPE x, y

This command defines a shape which can be used on the FGR page. As text from the alphanumeric character set cannot be used on the FGR page this is one means of overcoming the problem.

The shape itself has to be stored at memory location 32 512 for 16 K memory system (or -16 640 for 32 K system) in the computer memory. This involves the use of the POKE command. Unfortunately there is no simple way of using this command and we hope that the example will make things clear!

The numbers that are stored in the memory tell the computer a number of things. First the colour of the pixel to be saved at the starting point defined by the coordinates x, y and then the direction to move in to plot the next pixel. This only takes four out of the eight bits of information that can be stored at each memory location, so it is repeated again to fill up the full eight bits available. If you're getting lost at this point jump to the program, enter it, and just sit back and enjoy what happens when you RUN it!

As each byte, or group of eight bits, consists of two four bit nibbles, we will consider these in turn. The first two bits in the first nibble define the colour the second two the direction of movement. The actual number to be entered into the memory is easily worked out from the table below:

#### UPPER NIBBLE

#### LOWER NIBBLE

colour	direction			colour	direction	
00	00	= 0	} + }	0	black {	
00	01	= 16		1		right
00	10	= 32		2		down
00	11	= 48		3	left	
01	00	= 46		4	blue {	
01	01	= 72		5		right
01	10	= 96		6		down
01	11	= 112		7	left	
10	00	= 128		8	red {	
10	01	= 144		9		right
10	10	= 160		10		down
10	11	= 176		11	left	
11	00	= 192		12	green {	
11	01	= 208		13		right
11	10	= 226		14		down
11	11	= 240		15	left	
					up	

To use this table decide from the left hand columns the effects that you want e.g. if you want the pixel to be green and the next pixel to be to its right enter 192 + 12, or 204, into the memory. Diagonal directions are achieved by painting a pixel in the background colour as you move, say, up and then left.

Here's the example. The length of the shape table is stored at 32 512 (or -16 640) and the shape from 32 513 (or -16 639) upwards.

This first program draws a simple square on the FGR page and then uses the SCALE command to increase its size.

```
10 FCLS: FGR
20 FOR N = 32 512 TO 32 517
30 READ D
40 POKE N, D
50 NEXT N
60 SCALE 1
70 SHAPE 50, 50
80 GOTO 80
90 DATA 5, 136, 153, 170, 187, 204
```

Now let's play with the SCALE command; enter this into the computer a number of times with different values after SCALE.

## SCALE 2: SHAPE 50, 50

The next program is slightly more complicated and puts up the EACA logo (EACA are the people who make the Colour Genie!)

```
10 FCLS: FGR
20 FOR I = 0 TO 56
30 READ A
40 POKE 32512 + I, A
50 NEXT I
60 SCALE 1
70 SHAPE 112, 16
80 GOTO 80
90 DATA 57, 170, 170, 170, 170, 170, 170, 170,
    170, 146, 153, 41, 146, 153, 41, 146, 153, 152
100 DATA 136, 136, 136, 136, 136, 136, 136, 136,
    136, 131, 118, 102, 102, 102, 102, 102, 102,
    102, 103, 112, 119
110 DATA 68, 68, 68, 68, 68, 68, 68, 67, 51, 118,
    102, 102, 102, 102, 102, 102, 82, 85
```

It is possible to POKE into the colour memory directly. This is explained in detail in the Technical Manual.

## XSHAPE

This command changes the colour of the picture drawn by the SHAPE command according to this relationship:

	Picture by SHAPE	Picture by XSHAPE
Colour	PINK/BLANK	GREEN
	BLUE	RED
	RED	BLUE
	GREEN	PINK/BLANK

Example:

```
10 FCLS : FGR
20 A = &HF00
30 FOR S = 0 TO 3
40 READ D
50 POKE A + S,D
60 NEXT
70 SCALE 3
80 SHAPE 30, 30
90 IF INKEY$="" THEN 90
100 XSHAPE 30, 30
110 GOTO 110
120 DATA 3, 68, 119, 68
```



# CHAPTER 7 :

## SOUND

The Colour Genie has a programmable sound generator, or PSG. This is a very complex chip inside the computer and handles not only sound but also the parallel output facilities of the system. A full explanation of the PSG is given in the Technical Manual so we are only going to show you how to use the most straight forward facilities in this chapter. These will allow you to generate quite interesting sound effects for use with games, for example: We have also included a program which shows you how to play some rather sweet music.

The sound comes through the television loudspeaker so it is necessary to have your tv correctly tuned in so that both sound and vision are possible at the same time. Although the computer has been designed for most tvs on the market it is possible that you may have one of the few that will not give sound and vision together. If this is the case then see your local tv dealer or rental company to have it tuned properly. If a bit of judicious twiddling with the knobs on the front don't work, then it needs getting at inside and this should be done by a technical qualified person.

There are two basic sound commands; PLAY and SOUND.

### PLAY (ch #, oct, note, amp)

PLAY is by far the most powerful of the two commands and the simplest to use. The PSG has three sound channels when used with the PLAY command. These are turned on by the use of the channel number, ch #. This has a value between 1 and 3. There are eight octaves available for each channel, with the value of 'oct' being between 1 and 8. A value of 4 will give a note in the octave containing middle C. The value of 'note' is between 1 and 12 as shown in the table below. The amplitude, 'amp', is between 1 and 15.

All these parameters can be put into, and read from, DATA statements as shown in the program below. The relationship between the value of 'note' and the actual note is:

value	note
0	rest
1	C
2	D
3	E
4	F
5	G
6	A
7	B
8	C#
9	D#
10	F#
11	G#
12	A#

The program below uses two of the sound channels and assumes that each channel keeps playing until it is told to play a new note. This gives the effect of two part harmony.

```
10 REM MUSIC
20 FOR X = 1 TO 34
30 READ C, O, N, V
40 PLAY (C, O, N, V)
50 FOR D = 1 TO 150: NEXT D
60 NEXT X
70 DATA 1, 4, 1, 10
80 DATA 2, 4, 3, 12
90 DATA 1, 4, 5, 10
100 DATA 1, 4, 6, 12
110 DATA 1, 3, 3, 10
120 DATA 2, 2, 6, 12
130 DATA 2, 4, 7, 12
140 DATA 1, 3, 4, 8
150 DATA 2, 3, 6, 10
160 DATA 2, 3, 1, 12
170 DATA 1, 4, 1, 10
180 DATA 2, 4, 3, 12
190 DATA 1, 4, 5, 10
200 DATA 2, 4, 6, 12
210 DATA 1, 4, 1, 10
220 DATA 2, 4, 3, 12
230 DATA 1, 3, 4, 8
240 DATA 2, 3, 6, 10
250 DATA 2, 3, 1, 12
260 DATA 1, 4, 1, 10
270 DATA 2, 4, 3, 12
280 DATA 1, 4, 5, 10
290 DATA 2, 4, 7, 12
300 DATA 1, 2, 3, 10
310 DATA 2, 2, 6, 10
320 DATA 1, 4, 6, 10
330 DATA 2, 4, 2, 12
340 DATA 1, 4, 5, 10
350 DATA 2, 4, 7, 12
360 DATA 1, 3, 5, 10
370 DATA 2, 3, 7, 12
380 DATA 2, 3, 1, 10
390 DATA 1, 2, 2, 0
400 DATA 2, 3, 3, 0
```

## SOUND R, N

The PSG has sixteen registers which can be individually programmed to give different types of sound. The actual attributes of each register, and how to program them, are given in the Technical Manual. The registers are listed below for reference, and some programs using the more commonly used registers listed.

<u>Register</u>	<u>Attribute</u>
R0	Tone period 1-fine tune
R1	Tone period 1-coarse tune
R2	Tone period 2-fine tune
R3	Tone period 2-coarse tune
R4	Tone period 3-fine tune
R5	Tone period 3-coarse tune
R6	Noise period
R7	Enable R8, R9, R10
R8	Channel 1 amplitude
R9	Channel 2 amplitude
R10	Channel 3 amplitude
R11	Envelope period-fine tune
R12	Envelope period-coarse tune
R13	Envelope shape/cycle

Registers 14 and 15 are used for parallel I/O control and do not affect the sound generator directly.

Most of the following programs, and most sound effects, use registers 6, 7, 8, 9, 10, 11, and 13. R0 and R1 are occasionally used to change the time period of one channel. As stated before full details of the PSG are available in the Technical Manual.

As R7 is the most important register at this level of programming a short explanation of how it works is useful here. The number appearing in the SOUND statement when R has the value 7 tells the PSG what sound channels to enable. These are dependent on the last three bits of the number held in register 7. The table below shows you how each channel is turned on.

<u>Number in R7</u>	<u>Effect</u>
248	=1 on, =2 on, =3 on
249	=1 off, =2 on, =3 on
250	=1 on, =2 off, =3 on
251	=1 off, =2 off, =3 on
252	=1 on, =2 on, =3 off
253	=1 off, =2 on, =3 off
254	=1 on, =2 off, =3 off
255	=1 off, =2 off, =3 off

Numbers in R7 below 248 do strange and interesting things with the noise generator!

The programs below show how SOUND R, n works.

### Program 1

```

5 REM CIRCLE AND SOUND DEMO
10 FCLS: FGR: FCOLOUR 3
20 SOUND 7, 254: SOUND 8, 15
30 FOR I = 1 TO 30
40 SOUND 1, I
50 CIRCLE 80 - I, 70 - I, I
60 CIRCLE 80 + I, 40 + I, I
70 NEXT I
80 GOTO 10

```

### Program 2

```

10 REM SIREN
15 CLS
20 SOUND 0, 254
30 SOUND 1, 0
40 SOUND 7, 248
50 SOUND 8, 15
60 FOR D = 1 TO 100: NEXT D
70 SOUND 0, 86
80 SOUND 1, 1
90 FOR T = 1 TO 100: NEXT T
100 GOTO 10

```

### Program 3

```
10 REM GUNSHOT
15 CLS
20 SOUND 6, 15
30 SOUND 7, 7
40 SOUND 8, 16
50 SOUND 9, 16
60 SOUND 10, 16
70 SOUND 12, 16
80 SOUND 13, 0
90 INPUT "PRESS RETURN TO FIRE"; F$
100 GOTO 10
```

### Program 4

```
10 REM EXPLOSION
15 CLS
20 SOUND 6, 0
30 SOUND 7, 7
40 SOUND 8, 16
50 SOUND 9, 16
60 SOUND 10, 16
70 SOUND 12, 56
80 SOUND 13, 0
90 INPUT "PRESS RETURN TO DETONATE"; F$
100 GOTO 10
```

### Program 5

```
10 FOR X = 48 TO 192
20 SOUND 7, 254
30 SOUND 8, 20
40 SOUND 0, X
50 NEXT X
60 SOUND 6, 0
70 SOUND 7, 7
80 SOUND 8, 30
90 SOUND 9, 30
100 SOUND 10, 30
110 SOUND 12, 56
120 SOUND 13, 0
130 FOR X = 0 TO 1000: NEXT X
140 SOUND 7, 255
```

## Program 6

```
5 REM BEAT GENERATOR
10 DIM R (20)
20 CLS
30 FOR D = 1 TO 20
40 PRINT @400, "ENTER BEAT VALUE FOR
   LOCATION"; D
50 INPUT V
60 R (D) = V
70 PRINT @80 + S, V
80 S = S + 4: REM DISPLAYING DATA
90 NEXT D
100 SOUND 7, 247
105 CLS
110 FOR P = 1 TO 20
120 SOUND 8, R (P)
130 FOR D = 1 TO 10: NEXTD
140 IF INKEYS < > " " THEN 170
150 NEXT P
160 GOTO 110
170 SOUND 7, 255
```

## Program 7

```
10 FOR X = 48 TO 192
20 SOUND 0, X
30 SOUND 1, 0
40 SOUND 2, 0
50 SOUND 3, 0
60 SOUND 4, 0
70 SOUND 5, 0
80 SOUND 6, 0
90 SOUND 7, 254
100 SOUND 8, 15
110 SOUND 9, 0
120 SOUND 10, 0
130 SOUND 11, 0
140 SOUND 12, 0
150 SOUND 13, 0
160 FOR C = 0 TO 20: NEXT C
170 NEXT X
180 SOUND 7, 255
```

# CHAPTER 8 : OTHER GOODIES

There are some other commands that the Colour Genie executes, although these are not fully applicable until you have some other bits and pieces to plug into the computer. You may have noticed some sockets on the side of the machine. These are for extra keypads and joysticks. The Technical Manual has full details of how to use these and the keypad, joystick and light pen available from your supplier will have instructions included. However, to accomodate the inquisitive we have included a brief summary of the commands below:

## CALL

This command provides a direct linkage between BASIC and the machine language. Calling a four digit Hex address will execute the subroutine at the address and return to BASIC.

## KEYPAD n

n = 1 or 2

There are two keypads with twelve keys each. These are software controlled through the two i/o register of the PSG.

## JOY n N

n = 1 or 2, N = X or Y

There are two joysticks available. The BASIC command JOY n N, where n is 1 or 2, will return the x, y position of the joystick.



### INP (port – number)

This inputs an 8-bit value from the specified port. The Colour Genie is capable of handling 256 ports, numbered from 0 to 255. Usually this function is used only when the expansion box is installed.

Example:

```
10 A = INP (124)
```

### OUT (port – number, value)

Outputs an 8-bit value to the specified port. This statement requires two arguments: port-number and the value. The Colour Genie is capable of handling 256 ports, numbered from 0 to 255.

Example:

```
30 OUT 14, 240
```

This outputs the value 240 to port 14. Both arguments are limited to single byte values, that is 0 – 255.

### PEEK (address)

This function returns the 8-bit value stored at the specified decimal address in the computer's memory, and displays the value in decimal form. The value will be between 0 – 255.

Example:

```
20 B = PEEK (30000)
```

Returns the value stored at location 30 000 and assign that value to the variable B.

## POKE (address value)

This statement sends a 8-bit value to the specified (decimal) memory address location. It requires two arguments: address and value. The value must be between 0 – 255.

Example:

```
10 A = 250
20 POKE 19000, A
30 B = PEEK (19000)
40 PRINT "THE RESULT IS: "; B
50 END
```

This program sends the value of A to address 19 000 and then prints the value at that memory location. When RUN we get: THE RESULT IS: 250.

## MEM

Returns the number of unused and unprotected bytes in memory.

Example:

```
200 IF MEM = 180 THEN 700
```

When used as a command, it must be accompanied with the PRINT command. That is PRINT MEM, to find out the amount of memory not being used to store program, variables, strings, arrays, etc.

## USR (argument)

This calls a machine language subroutine and passes the argument to the subroutine. Such a subroutine could be loaded from tape or created by POKEing Z80-machine code into the memory. Users who are not familiar with machine language programming are not recommended to use this command.

The subroutine entry address should be POKEd into location 16 526 – 16 527. The last significant byte should be in location 16 526.

To pass the argument to the subroutine, the subroutine should immediately execute a CALL 0A7FH (2 687 dec.). The argument will then be placed in registers HL.

To return to your BASIC program without passing any value back, a RET instruction should be executed.

To return a value, load the value into the HL register pair as a two-byte signed integer and execute a JP 0A9AH. (0A9AH = 2 714 Decimal).

USR routine reserves 8 stack levels for the users' subroutine.

Example:

```
10 INPUT 1%: REM * INPUT ARGUMENT *
15 REM * PREPARE ENTRY ADDRESS *
20 POKE 16 526, 0: POKE 16 527, 120
30 A = USR (1%): REM * RETURN ARGUMENT A *
```

The subroutine should place on top of the memory map. To protect that region of memory, the user should input the highest location available for his BASIC program storage when the machine asks READY? at power up.

Further details of this, and other machine code routines, can be found in the Technical Manual.

## VARPTR (variable name)

An address – value of the variable name will be returned.

If K is the returned address, the variables will be stored in the following structures: –

- (i) 2 = byte integer  
\*(K) = LSB  
(K + 1) = MSB
  
- (ii) single precision variable  
(K) = LSB  
(K + 1) = Next MSB  
(K + 2) = MSB  
(K + 3) = Exponent value
  
- (iii) double precision value  
(K) = LSB  
(K + 1) = Next MSB  
  
\*            \*  
\*            \*  
\*            \*  
(K + 6) = MSB  
(K + 7) = Exponent value
  
- (iv) string variable  
(K) = length of string  
(K + 1) = LSB of string starting address  
(K + 2) = MSB of string starting address

Note: \*(K) signifies "contents of address K"

## & H – hex number

This command returns the decimal equivalent of a hexadecimal number in the range 0H to 7FFFH. Hex numbers between 8000H and FFFFH are returned as the signed complement i.e., 4000H = 16 384 and FFFFH = -1.

Example:

```
10 A = &H3FFF  
20 PRINT A
```

When RUN gives:

```
16 383
```

### & O – octal number

This command returns the decimal equivalent of an octal number between 0 and 255.  
(i.e. octal number between 000 and 377)

Example:

```
PRINT &O101
```

gives: 65.

# APPENDIX A

## COLOUR GENIE BASIC RESERVED WORDS

None of these words can be used inside a variable name.

ABS	DIM	LEN	RESUME
AND	EDIT	LIST	RETURN
ASC	ELSE	LLIST	RIGHT\$
ATN	END	LOG	RND
AUTO	ERL	LPRINT	RUN
BGRD	ERR	MEM	SCALE
CALL	ERROR	MID\$	SGN
CLOAD	EXP	NBGRD	SHAPE
CDBL	FCLS	NEW	SIN
CHAR	FCOLOUR	NEXT	SOUND
CSAVE	FGR	NOT	SQR
CSNG	FILL	NPLOT	STEP
CHR\$	FKEY	NSHAPE	STOP
CINT	FIX	ON	STRING\$
CIRCLE	FOR	OR	STR\$
CLEAR	FRE	OUT	SYSTEM
CLOSE	GOSUB	PEEK	TAB
CLS	GOTO	PAINT	TAN
COLOUR	IF	PLAY	THEN
CONT	INKEY\$	PLOT	TROFF
COS	INP	POKE	TRON
CPOINT	INPUT	POS	TO
DATA	INT	PRINT	USING
DEFDBL	JOY	RANDOM	USR
DEFINT	KEYPAD	READ	VAL
DEFSNG	LEFT\$	RENUM	VARPTR
DEFSTR	LET	REM	VERIFY
DELETE	LGR	RESTORE	XSHAPE

# APPENDIX B

## PROGRAM LIMITS AND MEMORY OVERHEAD

### Ranges

Integers                    – 32 768 + 32 767 inclusive  
Single Precision       – 1.701 411E + 38 to +1.701 411E + 38  
                                  inclusive  
Double Precision       – 1.701 411 834 544 556E + 38 to  
                                  + 1.70 141 183 454 455 6E + 38 inclusive

String Range                Up to 255 characters  
Line Numbers Allowed      0 to 65 529 inclusive  
Program Line Length       Up to 240 characters

### Memory Overhead

Program lines require 5 bytes minimum, as follows:

Line Number                – 2 bytes  
Line Pointer               – 2 bytes  
Carriage Return           – 1 byte

In addition, each operator, variable name, special character and constant character requires one byte.

However the reserved word requires one or two bytes.

## DYNAMIC (RUN-TIME) MEMORY ALLOCATION

Integer variables (2 for value, 3 for variable name)	5 bytes each
Single-precision variables (4 for value, 3 for variable name)	7 bytes each
Double-precision variables (8 for value, 3 for variable name)	11 bytes each
String variables (3 for variable name, 3 for stack and variable pointers, 1 for each character)	6 bytes minimum
Array variables (3 for variable name, 2 for size, 1 for number of dimensions, 2 for each dimension, and 2, 3, 4, or 8 (depending on array type) for each element in the array)	12 bytes minimum

































































Each active FOR-NEXT loop requires 16 bytes.







































Each active (non-returned) GOSUB requires 6 bytes.



























Each level of parentheses requires 4 bytes plus 12 bytes for each temporary value.



# APPENDIX C

ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS
128		146		164		182	
129		147		165		183	
130		148		166		184	
131		149		167		185	
132		150		168		186	
133		151		169		187	
134		152		170		188	
135		153		171		189	
136		154		172		190	
137		155		173		191	
138		156		174			
139		157		175			
140		158		176			
141		159		177			
142		160		178			
143		161		179			
144		162		180			
145		163		181			

ASCII CODE	GRAPHICS	KEY	ASCII CODE	GRAPHICS	KEY (SHIFT)
193		<	198		'
194		=	199		-
195		>	200		.
196		?	201		/
197		+	192		;
202		@	229		\
203		A	230		a
204		B	231		b
205		C	232		c
206		D	233		d
207		E	234		e
208		F	235		f
209		G	236		g
210		H	237		h
211		I	238		i
212		J	239		j
213		K	240		k
214		L	241		l
215		M	242		m

ASCII CODE	GRAPHICS	KEY	ASCII CODE	GRAPHICS	KEY (SHIFT)
216		N	243		n
217		O	244		o
218		P	245		p
219		Q	246		q
220		R	247		r
221		S	248		s
222		T	249		t
223		U	250		u
224		V	251		v
225		W	252		w
226		X	253		x
227		Y	254		y
228		Z	255		z

National Colour Genie Users' Group free cassette to new Subscribers

This is a brief set of instructions to get you using your free cassette tape. The tape consists of:

TCOPY	- Machine language tape copier	SYSTEM
CGEN	- Character Generator Editor / Saver	Basic
PLATO	- An "OTHELLO" type game	Basic
MAGIC CARPET	- A High Res demonstration program	Basic
DEMO1	- A Demonstration program	Basic
DEMO2	- " " "	Basic
DEMO3	- " " "	Basic

There is no need to explain how the later six programs work because they either have the instructions built in or they are demonstration programs for you to watch and enjoy. Feel free to manipulate the programs yourselves!!

TCOPY is a program which will backup most machine language program tapes. Please note: this program is intended for backup purposes only. Do not give away programs away to your friends. NCGUG is attempting to get dealers to lower their software prices. You can help by paying for your programs.

To load TCOPY, you must type SYSTEM followed by pressing the Return key. Then type T followed by the return key. The program will load both graphics and letters on the screen. After a successful load, most of the screen will clear and two lines of text will be displayed on the top of the screen. If this did not happen, then try again with a different volume setting on the tape recorder.

To load a system tape into TCOPY, press the L key. Then put a system (Machine Language like TCOPY) into the tape recorder and press the Play button. Then press the Return button on the Computer. The tape will then load in. If an error was discovered, then try again at a different volume rate. To write a tape, prepare a blank tape into the tape recorder. Press the W key, then press Play and Record on the tape recorder. Then, press the Return key on the computer. A new copy will be written to the tape. Copies can be tested by using the V command which will test a system tape with what is contained in memory.

CGEN is the character generator editor. LOAD this program in. This program is extremely useful for designing pictures or characters with the programmable character generator.

The seven programs on the free cassette tape may not be sold.

TCOPY, PLATO and MAGIC CARPET were written by Marc J. Leduc of Nottingham  
CGEN was written by Tony Parkin of Nottingham  
DEMO1, DEMO2 and DEMO3 were written by Dave Doohan of Nottingham

This cassette was prepared and distributed by GUMBOOT SOFTWARE, the software house of the National Colour Genie Users' Group.

### COLOUR GENIE ROM ENHANCEMENTS

All the enhancements are upward compatible with the existing R.O.M.S.

The following features have been either added or improved:-

1. The text screen is now 40 characters by 25 lines.
2. The Graphics screen is now 160 by 102. To keep compatible with Text Screen.
3. SHIFT-F2 now equals SYSTEM
4. SHIFT-F4 now equals CSAVE "
5. The Paint command has been vastly improved to cope with difficult shapes. There is also now 3 modes of operation as follows:

PAINT X, Y, C

PAINT X, Y, C, B

PAINT X, Y, C, B, B

Where X and Y = the start co-ordinates

C = the colour to paint in

B = the Boundry colour

6. The play command can now be used with expressions/commands within it, i.e. PLAY (1, KEYPAD1, KEYPAD2, (JOYIX-1) AND !5).

To be able to take advantage of the various envelope shapes, you can now use the Play command with the volume equal to 16 which will allow whatever envelope shape has been set up to be used.

E.G. SOUND 13, 8 : set envelope to 8

PLAY (1, 5, 8, 16)

Further by adding 16 to the old note number a better C-Major scale can be produced.

Cont'd .....

7. The Plot command has been speeded up.
8. The Verify command can now be used with a program name,  
i.e. VERIFY "N"
9. The &H and &O now allow spaces to follow the number  
within statements. Also it is now valid to use any Hex  
or Octal number without leading zeros, i.e. &HF.
10. The FILL command is now replaced with FCLS n. n = 1 to 4.
11. The SOUND command can now be used to read back the contents  
of the PSG chip, i.e. PRINT SOUND (n) where n = 0 - 15.
12. The KEYPAD command can now be used with a variable, i.e.  
A = 1 : PRINT KEYPAD (A).
13. Scale now returns the scale factor in use, i.e. PRINT SCALE

All the following are new commands added to the Colour Genie.

The latter three are involved with Bit manipulation.

14. SWAP var, var: allows you to swap variables, which is  
especially useful for strings for use in sorts, etc. and  
doesn't cause any string "hangups", usually known as the  
"GARBAGE Collection Cycle".

E.G. (the normal way to swap) = TEMP\$=A\$ : A\$=B\$ : B\$ = TEMP\$

(using the Swap command) = SWAP A\$, B\$

15. SET b, addr Sets bit b in address addr
16. RESET b, addr Resets " " " " "
17. CHECK(b, addr)Checks if bit b is set in address addr returns - 1  
as true or set and 0 as false or reset.

Please note that some errors are in the manual. They are corrected as follows:

Page 14

Error: > (2+3) \*4

Correct: > PRINT (2+3) \*4

Page 20

Error: 30 COLOR N: PRINT A, N\*N

Correct: 30 COLOUR N: PRINT N, N\*N

Page 27

Error: CYAN

Correct: BLUE

Page 28

Error: FCLS: PLOT 0,0 TO 0,159 TO 95,159 TO 95,0 TO 0,0

Correct: FCLS: PLOT 0,0 TO 159,0 TO 159,95 TO 0,95 TO 0,0



Published by  
**EACA INTERNATIONAL LTD.**  
13 Chong Yip St.,  
Kwun Tong,  
Kowloon, Hong Kong.

Copyright (c) 1982 EACA International Ltd. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photo copying, recording or otherwise without the prior written permission of Eaca International Ltd.