An introduction to BASIC programming using the DRAGON

Data Ltd

By Richard Wadman

An Introduction to BASIC programming using the DRAGON

Data Ltd

micro computer

By Richard Wadman

First printing 1982

ISBN 0 95 08079 0 7

Acknowledgements

The author and DRAGON Data Ltd wish to thank Professor A. G. Hawkes of University College Swansea for his assistance, and Mrs. Dianne Mowat for typing a difficult manuscript.

OPERATING INSTRUCTIONS

Having opened the box and found this manual you should also have found:

1. Your Dragon 32 computer
2. A mains power unit
3. A TV connection cable.

In addition to these items you will also need an ordinary home television set. Your Dragon 32 Computer will work with either a colour, or black and white TV. However, to obtain the full colour graphic ability of your Dragon 32, you will have to connect to a colour TV. This is all that is required to get your Dragon 32 working.

You can, however, improve the capabilities of your machine by adding the following options:

1. A cassette recorder to store programs and data
2. A printer
3. Joysticks for games playing
4. Disc drives for mass storage of programs and data.

None of these options are necessary, but a cassette recorder will save you a lot of repetitive typing.

KEY

1. **TV SOCKET**
   Connection to standard television serial socket.

2. **RESET BUTTON**
   Used to reset computer to initial state. Stops running program, or input/output operations, immediately. Any program currently in memory is still present after pressing reset.

3. **LEFT JOYSTICK**

4. **RIGHT JOYSTICK**
   For both 3. and 4. 5 pin DIN sockets used for connecting joysticks, available as optional accessories.

5. **CASSETTE INPUT/OUTPUT SOCKET**
   5 pin DIN socket for connection of cassette recorder. Connection lead available as accessory.

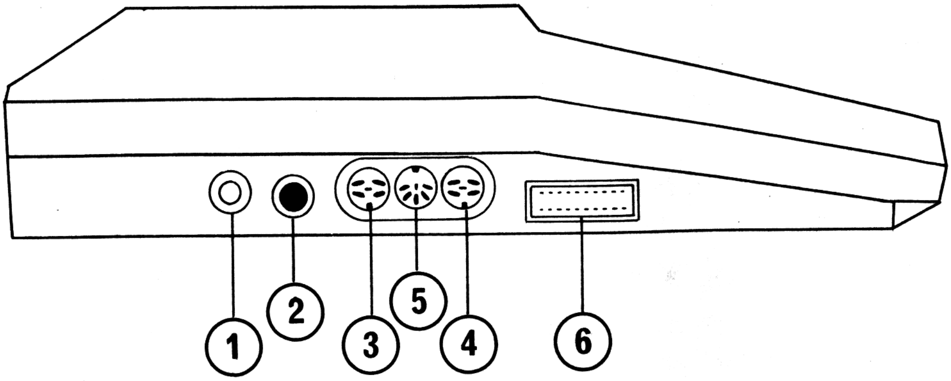
6. **PARALLEL PRINTER PORT**
   Connection for a centronics type printer via a standard centronics cable.

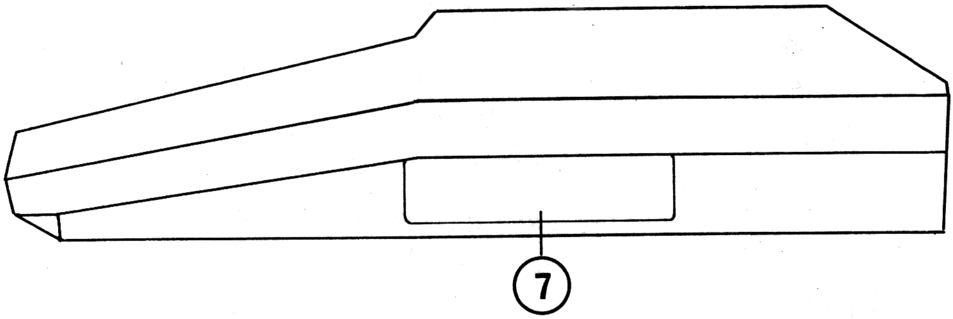
7. **PROGRAM CARTRIDGE SLOT**
   Used for games cartridges. Cartridge must be inserted with the computer switched off.

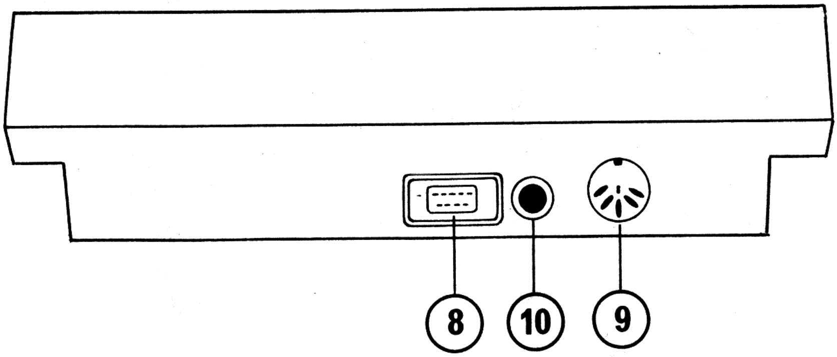
8. **POWER PACK SOCKET**
   For connection of supplied mains power unit.

9. **MONITOR SOCKET**
   For connection of colour monitor.

10. **ON/OFF SWITCH**
    Controls mains power supply to computer.


1
2
3
5
4
6


7


8
10
9

CONNECTING YOUR DRAGON 32

1. Connect the TV connection cable to the serial socket of the Television and to the TV socket 1 on the Dragon 32.

2. Connect the mains power unit to the power pack socket **8** on Dragon. The other wire from the mains unit should be plugged into a 3 pin wall socket.

3. Switch on the TV and computer.

4. Using a spare channel on the TV set, adjust the tuning until the screen shows a green square with a black border, (or light grey square with a black border if the TV is a black and white set).

   In the green square will be a message as follows:-

   ```
   (C) 1982 DRAGON DATA LTD
   l6K  BASIC INTERPRETER 1.0
   (C) 1982 BY MICROSOFT

   OK
   ```

   Your computer is now ready to use.

USING PROGRAM CARTRIDGES

Connect your Dragon 32 to the TV as above. Before switching on the computer, insert the cartridge with the label facing upwards, into the Program Cartridge slot 7

Please ensure that before inserting *or* removing a program cartridge the computer is *switched off*. Failure to do so many damage both the cartridge and your computer.

USING GAME JOYSTICKS

Various types of joysticks are available as option accessories. The joysticks for use with your Dragon computer should be of the potentiometer type. To connect the joysticks just plug into the joystick sockets **3** and **4**.

USING A CASSETTE RECORDER

Any reasonable quality cassette recorder can be used to store programs and data from your Dragon 32. The cassette recorder must have sockets for remote control, earphone, and auxiliary input. Connection leads are available as accessories for most standard cassette recorders. The connection to the computer is at socket **5**. The connections to the cassette recorder will depend on the type of cassette. See chapter 4 of the Programming Manual for instructions on how to use the cassette recorder.

TAKING CARE OF YOUR DRAGON

1. Keep all liquids well away from your computer. Your Dragon does not work as well on tea and coffee as you do.

2. Ensure all loose wires are kept out of harms way. An accidental trip could be expensive.

3. Make sure all plugs are firmly in their sockets *before* switching on.

4. Switch off everything and disconnect the machine when not in use.

5. To clean the case and keyboard, first disconnect unit completely from the power supply. Using a slightly damp cloth, wipe the case and keyboard. *Do not* use any spirit based cleaners.

CONTENTS

INTRODUCTION

This book is designed for those who wish to learn to program their DRAGON 32 computer using the BASIC language.

The BASIC language is an extremely powerful programming language, but at the same time is very easy to learn. It is made up from less than 100 statements, which is less than 1 % of the average individual's vocabulary.

While programming may appear to be difficult at first, as with any new skill, it is really solving a problem in a logical number of steps. The secret is to take your time and make sure you understand the last step before trying the next one. Do not be worried by the fact that you will make mistakes, this is part of the learning process. You will not break your computer by making a programming error, just find the mistake, correct it and carry on. Try your own ideas the "what if I do this" approach can be a very quick way of finding out just what your machine is capable of. Enter and run each example, not only to see what it does, but why it does it.

With time and patience you will find your computer is not just a cartridge-playing games machine. Programming can become an absorbing and enjoyable activity in itself, apart from the fun the results of it may give.

To aid beginners in understanding and developing BASIC programs, a small Dragon is printed in the margin next to items that should be noted carefully, as these rules are particularly important to remember.

lf you have used BASIC before then you should look out for the twin Dragon symbol; these indicate special features of Dragon BASIC.

CHAPTER ONE

GETTING STARTED

THE KEYBOARD

You have set up the computer according to the instructions and are now ready to start. So switch on, your TV screen should show a GREEN square with a message. (lf it does not, switch off and check all the connections again). The message will depend on the type of machine, whether or not disk drives are connected etc. so we will not worry about it. The last line, however, is always the same.

```
OK
```

OK is the computer's "prompt", telling you it is ready to receive instructions. You have to wait for the prompt to appear before you can type anything. Below the OK is a flashing square this is called the cursor. It shows you where you are on the line.

Now look at the keyboard: it looks like a typewriter with some extra keys. It behaves like a typewriter too, just press a few keys and you will see the letters appear on the screen. Notice how the cursor moves over after each letter to show you where you have got to. If you press [SHIFT] and 0 (zero) together and carry on typing you will see that the letters on the screen have changed to green on a black background. Throughout the book we will use 0 to represent zero, to distinguish it from the letter O. The computer is very fussy about the difference. These are lower case letters (i.e. small a, b, c, d); BUT they will only appear as lower case on a PRINTER. ALL your commands or instructions to the computer must be in UPPER CASE leters (capital) so press [SHIFT] and 0 together again and type some more. You should now be back to black letters on a green background.

Now try the [ ] key. This is the backspace key, as you press it the cursor moves back along the line and the letter immediately to the left of it disappears. This is useful for correcting mistakes, just backspace to the wrong letter and retype.

To clear the screen completely press [CLEAR]; everything is cleared from the screen and the cursor moves to the top left hand comer. [CLEAR] only clears the screen, information stored in the computer is not affected.

Practise using the keyboard for a while to get used to the position of the keys and how to correct mistakes. Then clear the screen and make sure you are in upper case mode.

DRAGON THE CALCULATOR

Your computer will work in two different modes   *immediate*, it will obey the command at once, and *deferred*, it will store a set of instructions and then run them as a program. In the immediate mode the computer behaves like a calculator.

The computer understands a language called BASIC. In BASIC there are a number of special words to tell the computer to do certain things. For instance, PRINT, this not surprisingly means print what follows on the screen.

Try it, type PRINT 12 + 7 then press the [ENTER] key; the screen should show

```
19
OK
```

Try another, type PRINT 12 + 8/2 then press [ENTER]

```
16
OK
```

If you made a typing mistake you probably got a message

```
? SN ERROR
```

This means a 'syntax error', that is the computer does not recognise something, usually because it has been spelt wrongly. The computer will give error messages when it does not understand the command, and sometimes when it does understand the command, but feels that what it has been asked to do is illogical or impossible.

Type PRINT 3/0 then press [ENTER]

The message will be

```
?/0 ERROR
```

which means an attempt to divide by zero has been made   which is impossible.

The error messages are rather terse to save space, but a full list with probable causes is given in Appendix C.

To return to syntax errors, the computer is very fussy about the spelling of its special words in BASIC, if you spot the mistake before you press the [ENTER] key you can use the back   space [←] key to go back and correct it. Otherwise there is no alternative but to retype the line correctly, for the moment.

ARITHMETIC RULES. OK?

So far our examples have asked the computer to perform only 2 'arithmetic operations', these are add (+) and divide (/). The word 'operation' means something we are asking the computer to do. There are six simple operations the computer can do in arithmetic and it has strict rules as to how these are camed out. In the example above

```
PRINT 12 + 8/2
```

it is not clear whether the answer should be 16 or 10.

12 plus 8 divided by 2 is 10, or 8 divided by 2 is 4, plus 12 is 16.

The answer given by the computer is 16 because of the order it chooses to do its arithmetic. The operations and the priority they are given are as follows:

1. *Unary Minus*

This is when a minus sign is used to indicate a negative number

```
PRINT    3 + 2
```

The computer will first apply the minus sign to the number.
So   3 + 2 evaluates as   1. If the computer did the addition first
   3 + 2 would evaluate to   5, but it doesn't.

2. *Exponentiation*

Exponentiation means raise to the power. 5 raised to the power 4, (5^4) is 5 × 5 × 5 × 5.

After applying all minus signs the computer then does all exponentiations.

```
PRINT 4 + 3 ↑ 2
```

is evaluated by squaring three (3 x 3 = 9) then adding 4 to give at otal of 13. If there is more than one exponentiation they are evaluated from left to right. Try an example

```
PRINT 2    3    2   3
```

(the [   ] key on the left of the keyboard is used for the exponentiation operation).

this is evaluated by multiplying 2 by itself 3 times (2 x 2 x 2 = 8) then multiplying that result by itself (8 x 8 = 64) then multiplying that result by itself 3 times (64 x 64 x 64 = 262144).

3. *Multiplication*

The sign the computer uses for multiplication is * so as not to cause confusion with the letter x. This is obtained by the [SHIFT] and [*:] keys.

e.g.

```
PRINT 5 * 2 + 3
```

is evaluated as 13 (5 x 2 = 10 plus 3 = 13).

4. *Division*

The sign the computer uses for division is / so 3 ./. 2 is written as 3/2.

```
PRINT 5/2 + 3
```

is evaluated as 5.5 (5 / 2 = 2.5 plus 3 = 5.5).

Multiplication and division have equal precedence, that is the same priority. When arithmetic operators have equal precedence they are evaluated from left to right.

```
PRINT 5/2 * 3 + 4/2
```

is evaluated as 13 (2 x 3 = 6, 4 / 2 = 2, 5 + 6 + 2 = 13).

5. *Addition*

The sign for addition is +

6. *Subtraction*

The sign for subtraction is

Addition and subtraction have equal precedence, so they also are evaluated from left to right after all the higher priority operations have been done.

To summarize the computers order of precedence for carrying out mathematical operations.

FIRST		(minus sign is used to indicate negative numbers)
SECOND		(exponentiation, from left to right).
THIRD	* /	(multiplication and division, form left to right)
FOURTH	+	(addition and subtraction from left to right)

Below are some arithmetic expressions to evaluate. With each one first do it in your head (or with a pencil and paper) and then try it on the computer. If your answer is different to the computer try to find out why. Unless you have a lot of experience with the way computers evaluate expressions, you should actually do these examples. The majority of so   called 'computer errors' are caused by the programmer following a different set of rules to the computer. No answers are given. If typed in exactly as written, the computer will give the correct answer.

```
PRINT 3 + 2
PRINT 4 + 6     2 + 1
PRINT 8 * 4
PRINT 4     2 + 1
PRINT 5/4    1
PRINT 5     4/2
PRINT 6 *     2 + 6/3 + 8
PRINT 4 +     2
PRINT 2 * 2 + 3 * 4
PRINT 8/2/2/4
PRINT 20/2 * 5
PRINT 8 * 2/2 + 5 * 3 * 2     2
```

It is not necessary to type PRINT each time, there is a shorthand symbol available [?]. If you type

? 3 + 2 this is the same as PRINT 3 + 2

By now you should be used to pressing the [ENTER] key at the end of each line.
The OK prompt tells you that the computer is ready.
The [ENTER] key tells the computer you are ready.

After all this heartache about precedence, it is possible to modify the priority. This is done by using parentheses or brackets. Suppose you want to divide 14 by 4 plus 3, if you write
14/4 + 3 the answer will be 6.5, because you will get 14 divided by 4 with 3 added on. But this is not what you wanted. To accomplish this you can write

14/(4 + 3) the answer will be 2.

The parentheses modify the precedence, the rule is simple, do what is in parentheses first. If there is more than one set then work from the innermost outwards

12/(3 + (1 + 2)   2) is evaluated as follows

a) 12/(3 + 3 2)	(1 + 2) done first
b) 12/(3 + 9)	3 2 next
c) 12/12	(3 + 9) next
d) 1	division last.

Here are some more expressions to evaluate. Again, if you are not familiar with the way computers work, the few minutes you spend working them out will enable you to use your computer more effectively.

```
? 44/(2 + 2)
? (44/2) + 2
? 4 + (  5*2)
? 100(200(2*(9  5)))
? 42/((9/3) + 1.75 + (5/4))
```

PRINTING WORDS

So far we have only used numbers in our PRINT command. The layman often sees the computer as a 'number cruncher', but this is not the only function of a computer. A computer may also be used to manipulate characters. By characters we mean the letters A to Z, the digits 0 to 9, punctuation marks and other special characters we shall be meeting later. BASIC allows us to manipulate groups of characters called *strings*.

A string can be any mixture of characters   even a space can be an important character in a string. To tell the computer that it is dealing with a string, rather than a number, the string is enclosed in quotes ("") Some examples of strings are

"THE TITLE", "Z+*?!", "MR J.P.SMITH"

"AXY 479W", "01 479 6172"

The last two strings could be a car number and a telephone number, and though they both contain *numeric characters*, we would not use this collection of digits to do any serious arithmetic. As far as BASIC is concerned a collection of digits enclosed in quotes is not a number, the string "12345" is a completely different thing to the number 12345. In fact, strings and numbers are stored in completely separate places in the computers memory.

```
Try PRINT "2 + 5 = "; 2 + 5
```

The first part of the PRINT statement appears on the screen exactly as written in the string. (The quotes are used to enclose the string, they are not part of it). The second part of the statement is evaluated as a numeric expression, so the screen should show

```
2  +  5  =  7
```

As we have seen above, the space can be an important character in a string. In the BASIC statements, spaces do not make any difference, they only make the line easier to read. In strings, however, they do make a difference, as when a string is printed on the screen it is copied exactly as the group of characters appearing between the quotation marks. Throughout the book if we feel a space is necessary we will indicate it by a ▽ symbol. This is *not* a keyboard symbol, it means that a space should be typed. It will only appear in strings, as all other spaces are optional.

You should now be able to use your computer to solve simple problems, like those at the end of this chapter. Again, if you are not familiar with computers at least try some of them.

1) Jim is 168 cms tall. What is his height in inches?
(1 ins = 2.54 cms).

2) A recipe requires 0.75 kgs of flour. How many pounds of flour do you need?
(1 kg = 2.2 lbs).

3) Your car uses 22.8 gallons of petrol on a journey. At the start the mileage on the clock was 10346 and at the end of the journey it was 11193. What is the m. p. g. for the trip?

4) You put 15 into a savings account which pays interest at 11 % per year How much will you have after 5 years?
(A = P(1 + R/100) N P = Principal R = Interest rate per year
N = Number of years A = Amount after N years).

5) How many turns does a 26 inch bicycle wheel make in one mile?
(1 mile = 5280 ft Circumference = PI x diameter PI = 3.14159).

ANSWERS

1) 66.14 inches
2) 1.65 lbs = 1 lb 10 ozs
3) 37.149 m.p.g.
4) 25.27
5) 775.69

CHAPTER TWO

WHAT'S IN A NAME

CONSTANTS

All the examples we have used so far have only contained *constants.* A *constants* is exactly what it says something which does not change. 3.145 is a constant, changing it to 3.146 just makes a different constant. Constant are useful in computer programs, but not as useful as *variables.*

VARIABLES

A *variable* is something which may change in value. In the equation.

X + 5 = Y

X and Y are variables as both may take many values which still make the equation true. Variables in a computer are places in the computer memory rater like a set of pigeonholes or boxes. To identify these it is necessary to label them (or give them a name like X or Y). The variables in your computer come in two flavours *numeric* or *string,* and in two sizes, *simple* or *array*. We already know the difference between numeric and string constants so numeric variables hold numeric constants (numbers) and string variables hold string constants (characters). For the moment we will only consider simple variables, array variables will be dealt with later.

NAMING VARIABLES

To name a *numeric* variable you may use any combination of a letter and a letter or number.

N, AA, X, TI, Y, Z9, L5, BZ, PQ, K9

are all examples of valid numeric variable names.

Actually your computer allows a variable name to be any length but will only recognise the first two characters of the name; so though it will accept names such as

BIRD BIRTHDAY BIGNUMBER

they will be considered as the same variable BI. The same applies to string variables

NUMERIC VARIABLE NAMES

A numeric variable can only store numbers.

The name of a numeric variable can consist of any combination of letters and numbers, but *must* start with a letter.

As the computer only recognises the first two characters of a name, names like:

STAR, STATE, STEAMER

will be considered to he the same (ST).

Variable names that are longer than two characters are useful to remind you of the contents,

NUMBER, COUNTER, SUM

will all be accepted but will take up more memory than NU, CO, SU.

FRED$ FRESHWATER$ FRIVOLOUS$

are all considered as FRS.

To name a string variable the same combinations may be used *but the name must have a $ sing on the end.*

A$, P7$, MNS, Z0S, FPS

are all examples of valid string variable names.

ASSIGNING VALUES TO VARIABLES

How do we use these variables? Type in the following example:

```
A = 5                    remember to press the [ENTER]
B = 2                    key after each line.
C = A + B
D = D + 3
PRINT A,B,C,D            (make sure you type the commas.)
```

Your screen should show

```
5        2
7        3
```

The first line means store the value 5 into a variable called A, the next line stores 2 into variable B. (The computer decides exactly where these places are in its memory, you only have to supply the name). The third line says find the values stored in variables A and B, add them together then put the results into a variable called C. After the computer has done this the variables A and B still contain the original values (5 and 2 in this case) and C contains the sum (7). The fourth line may seem a bit confusing to those of you who know algebra. This is because the equals sign (=) in the BASIC language does not mean the same thing as it does in mathematics. The = in BASIC means assign to, or take the expression on the right hand side of the equals sing, (evaluate it if necessary), and place it into the variable on the left hand side of the equals sign.

This sort of line is called an *assignment statement,* and the *left* hand side of the assignment statement must always be a variable. Something like 2 = B + C may make sense in algebra, but does not in the BASIC language.

To return to the statement D = D + 3, this means take the current content of the variable D (which happens to be 0 because we didn't put anything there), add 3 to it and then put it back into variable D (or equivalently increase variable D by 3). This may seem a little confusing but it is a very useful (and a very common) type of statement in computer programs. It also demonstrates another feature of variables   they can only hold one value at a time. If you assign a value to a variable (i. e. it appears on the *left* hand side of an assignment statement) the value over-writes the old value and the old value is lost. You may, however, copy the value in a variable (either into another variable, or by using it in an expression as in C = A + B) as many times as you like without changing it.

If you now type

```
A = B
B = 17
D = D + 2
PRINT  A,B,C,D
```

remember to press [ENTER] after each line.

Your answer will be

```
2           17
7            5
```

Variable A now holds a copy of B from the last example, variable B holds a new value (17), and the previous contents of both A and B have been lost. Variable C is unchanged. Variable D is now 5, because it held 3 from the previous example, to which has been added another 2.

String variables behave in exactly the same way (except you must remember the name must end with a $ sign). Try this example

```
A$ = "'THIS ▽ IS ▽ A ▽ "
B$ = "VERY ▽"
C$ = "LONG STRING"
D$ = A$ + B$ + B$ + B$ + BS + B$ + B$
D$ = D$ + Cs
PRINT D$
```

remember to press the [ENTER] key after each line.

Your screen will show

```
THIS IS A VERY VERY VERY VERY VE
RY VERY LONG STRING
```

In lines 1, 2 and 3 we have assigned values to the string variables A$, B$ and C$. In line 4 we add six copies of B$ to A$. With string variables the plus sign (+) does not mean the same as it does with numeric variables. It means add to end of the first string. (For those who like long words it is called *concatenation).* In line 5 we add C$ to the end of the newly constructed D$. This is one method of using a computer to construct sentences.

STRINGS AND NUMBERS DO NOT MIX

Please remember to keep numeric and string variables separate, only numbers can be stored in numeric variables and only strings can be stored in string variables; statements of the form

```
D = "STRING"
A$ = 6
B = A$ * 2
```

will give you the error message ?TM ERROR (Type mismatch error).

The plus sign (+) is the only, arithmetic operator that can be used with strings variables, all the others (, * , / ,   ) will give an error message.

COMMAND SUMMARIES

Throughout the rest of the book we will place pages in boxes. These will give details of each command as we introduce them. At the end of most boxes is a small example program to demonstrate the use of the command. Study them carefully, work through them and see if you can find out what will happen before you get the computer to run it. These programs are intended to show how a particular statement operates, but often include useful tips you may wish to include in your own programs later.

STRING VARIABLE NAMES

A string variable can only contain strings.

A string variable name can consist of any combination of letters and numbers, but must start with a letter, and end with the $ sign.

As with numeric variable names the computer will only recognise the first two characters so that,

ANSWERS$, ANI$, AN2$

will be considered to be the same (AN$).

CHAPTER THREE

A PROGRAM AT LAST

So far your computer has done little more than echo the line you have just entered. We will now start to construct a computer program.

A *program* is a set of instructions which tell a computer to do something. A BASIC program consists of a number of *lines*. A *line* has two parts: first, a line number and second, one or more *statements*. If there is more than one statement on a line, each statement must be separated by a colon (:). A *statement* is a command line such as we have already been using.

```
PRINT A$: A= 47
```

Here is a BASIC program:

```
10 CLS0
20 PRINT"WHAT IS YOUR NAME?"
30 INPUT NAME$
40 I=RND(255):J=RND(9)   1
50 CLSJ
60 PRINT @ 200+J,NAME$
70 SOUND I,2
80 GOTO 40
```

As you can see this program contains some new BASIC commands. Do not worry about these for the moment, we will explain them later. Notice the form of a BASIC program   a sequence of *lines*, each line consisting of a *line number* and at least one *statement* (line 40 as two).

PUTTING IN A PROGRAM

To put a program into the computers memory, first we must clear the memory of anything that may be there. To do this type NEW then press [ENTER]. Then enter each line exactly as above pressing the [ENTER] key at the end of each line. You will notice that after you press the [ENTER] key nothing happens. A line starting with a number is not carried out immediately, it is just stored. When a program is run, it starts with lowest line number, carries out that line, then moves to next highest line and so on. Because the sequence of the program depends on the line number, you can enter the lines in any order you want, the computer will order them into the correct sequence.

Try typing the program into the computer. If you make a mistake before pressing the [ENTER] key, use the backward arrows key [ ] as before. If the mistake becomes obvious after pressing the [ENTER] key, then retype the line again. The computer will only store the latest version of the line.

After you have entered all the program, to see the lines you have just entered as the computer has stored them, type LIST and press [ENTER]. Note there is no number before LIST. Without a number the computer will carry out the command immediately. Check the program to make sure it is correct (if not retype the lines that are wrong).

Now, at last, we are ready to get a program going. To do this type RUN and press [ENTER].

The screen will clear and a message will appear at the top of the screen asking for your name. Type in your name and press [ENTER]

The computer will spring into action

The screen will flash different colours, and your name will appear jumping about in the middle of the screen. Strange noises will accompany all this activity (if you remembered to turn up the volume control on your TV).

This will carry on forever unless you stop it. One way to stop the program is to switch off the power, but this is not very satisfactory you will lose the program. The best way to stop this program is to press the red BREAK key.

STEP BY STEP

Now you have seen what the program does, we will now explain how it does it, a line at a time.

```
10 CLS
```

As this is the lowest numbered line, it is the first to be obeyed. The command CLS means clear the screen, and set the background to the usual colour (which is green).

```
20 PRINT "WHAT IS YOUR NAME"
```

This is the PRINT statement we have seen before. This line prints the message at the top of the screen.

```
30 INPUT NAME$
```

The command INPUT tells the computer to stop and wait for you to type something in, which it will put into the variable following the command.

LIST

The LIST command displays the current program in memory, on the screen.
It is not preceded by a line number.

If the program is too long to fit on the screen, the listing can be stopped by pressing the [SHIFT] and [@] keys together, (but you have to be quick). The listing can be restarted by pressing any other key on the keyboard.

To LIST only part of the program you can use

LIST n_1 n_2

where n1 and n2 are two line numbers (n2 must be greater than n1).

```
LIST 40   100
```

will display all the program lines between line number 40 and line number 100

```
LIST   80
```

will display all program lines from the start of the program to line number 80.

```
LIST 120
```

will display all program lines from line number 120 to the end of the program.

RUN

The RUN command is used to start a program.

It does not have a number in front of it.

lf you wish to start a program from any place other than the beginning you may do so by typing

RUN *line number*

where *line number* is the number of the line at which you wish to start.

RUN *250*

NEW

The command NEW clears the memory, and sets all variables to zero.

Note that it does not have a line number.

It is a good idea to type NEW before entering a program to ensure that none of the old program is left to interfere with the current program.

ASSIGNMENTS STATEMENT

The *assignment statement* is used to place a value into a *variable*.

The form of the assignment statement is:-

LET *Variable = expression*

The LET part of the statement is part of standard BASIC, but it is not necessary on your computer, so it will not appear in any of the program listings in this book.

The *variable* part of the statement can be any variable name.

The *expression* part of the statement can be a constant, another variable or a mixture of both connected by operators (+ ,   , * , etc.). As string and numeric variables cannot be mixed, both the *variable* and the *expression* must be of the same type.

The equals sign (=) does not mean the same as the equals sign in algebra, it is better to interpret it as 'assign to'. This means the same variable can appear on both sides of the assignment, as in

```
40 X = X + 1
```

which tells the computer to add 1 to the current value of X and put it back into X.

```
10 S = 0:N = 0:CLS5
20 PRINT @ 72,"ENTER A NUMBER";
30 INPUT X:CLS5
40 S = S + X:N = N + 1
50 PRINT @ 194, "YOU HAVE ENTERED";N;"NUMBERS.";
60 PRINT @ 262,"THE AVERAGE IS";S/N;
70 GOTO 20
```

(Remember the computer will call the variable in this case NA$, and ignore any other letters. It is, however, often useful to use a variable name that is longer to make it more understandable to people. A longer name also acts as a reminder of what the variable is being used for).

```
40 I = RND(255):J = RND(9)     1
```

line 40 shows how more than one statement may appear in a line. (Note the colon (:) separating them). This line also introduces another new command, RND. This command generates a random number. A random number is like picking a number out of a hat. The number in brackets after the RND tells the computer which range of numbers to select from. In the first statement I = RND(255),RND(255) means select a random whole number from the range 1 to 255 and then place this number into a variable called I. In the second statement the range for the random number is from 1 to 9, however, after the number has been selected, 1 is subtracted before it is put into J. This means that J will be a number between 0 and 8.

```
50 CLS(J)
```

This line clears the screen. This time however, the background colour will depend on the value of J. There are nine colours available, which are numbered from 0 to 8. This is the line that makes the screen flash different colours.

```
60 PRINT @ 200 + J,NAME$;
```

This is a more sophisticated version of our old friend the PRINT statement. It instructs the computer to print the value of NAME$ (in this case your name), starting at a specified position on the screen. The position in this case is 200 + J, which is somewhere between 200 and 208. Position 200 is at line 7 and 8 spaces in. (Read the PRINT @ box to see how the screen is divided up). Because the value of J changes, your name appears to be jumping about on the line.

```
70 SOUND I, 2
```

This is the line which makes the strange noises. The SOUND command tells the computer to use its tone generator to make a sound; which sound, and how long to make it are decided by the two numbers following the command.

```
80 GOTO 40
```

The GOTO command simply means go to the line number following the command (40 in this program).

PRINT

The PRINT command is used to display output

It can be used to output constants, the value of variable, strings and also to evaluate expressions.

If more than one item is included in a PRINT statement, the items should be separated by either a comma (,) or a semi-colon (;).

The comma will cause the output to be printed in two columns, each fifteen characters wide. (If the length of the first item is more than 15 characters it will he printed in full. The next item will appear on the next line).

The semi-colon causes the output to be compressed   Strings will be printed next to each other, and numeric items will have a space either side. The semi-colon holds the print head at its last position ready to continue printing when the program reaches the next PRINT statement.

A PRINT statement with no items prints a blank line.

```
10 CLS
20 PRINT
30 PRINT "▽▽▽▽▽▽▽▽THE PRINT COMMAND"
40 PRINT "▽▽▽▽▽▽▽▽▽A DEMOSTRATION"
50 PRINT
60 PRINT"COLUMN ONE","COLUMN TWO"
70 PRINT 14.2,13.7
80 PRINT 1,2,7,11
90 PRINT
100 A$ = "COMPRESSED": B = 3
110 PRINT;"SECOND LINE"
130 PRINT A$;"OUTPUT ON LINE";B
140 PRINT
150 PRINT "THIS WILL APPEAR";
160 PRINT "AS ONE LINE"
170 PRINT "DEMOSTRATION",
180 PRINT "FINISHED"
```

INPUT

When a program comes to an INPUT statement, it stops and waits for something to be entered from the keyboard. Following the INPUT command must be one or more variable names separated by commas.

```
25 INPUT A,B,F$,H7
```

The above statement required you to enter 4 items. This can be done one at a time by pressing the [ENTERI key after each one, or as a list separated by commas, e.g.

```
146.2, 78.1, STRING,3[ENTER]
```

It is always a good idea to print a message before an INPUT statement, to remind you what is needed. This can be done with a PRINT statement, or included in the INPUT statement as follows:

```
35 INPUT "TWO NUMBERS PLEASE";N1,N2
```

Note the semi-colon separating the string from the input list.

You must ensure that the correct type of entry is made (strings to string variables, numbers to numeric variables), or else the program will halt with a ?REDO, and you will have to re-enter.

It is not necessary to enclose strings in quotes when inputing to a string variable both "STRING" and STRING are acceptable.

In the following demonstration program note:

a) lines 100 and 170. Uses the INPUT command to stop the program until ready. A$ will not contain anything.

b) the use of string variables (C$, P$ and T$) to avoid typing the same message more than once.

```
10 CLS:T$ = "THIS IS AN INPUT DEMOSTRATION"
20 P$ = "PRESS THE ENTER KEY TO CONTINUE"
30 C$ = "ENTER 4 NUMBERS"
40 PRINT:PRINT T$:PRINT
50 PRINT C$;"PRESS THE"
60 PRINT "ENTER KEY AFTER EACH ONE"
70 INPUT A,B,C,D
80 PRINT:PRINT "YOU INPUT THESE VALUES"
```

```
90   PRINT A;B;C;D:PRINT
120 PRINT "NOW▽";C$;"▽SEPARATED"
130 PRINT "'BY COMMAS AND PRESS ENTER."
140 INPUT A,B,C,D
150 PRINT: PRINT"THIS TIME THE NUMBERS WHERE:   "
160 PRINT A;B;C;D
170 PRINT: PRINT P$:INPUT A$:CLS
180 PRINT: PRINT T$:PRINT
190 INPUT "ENTER A STRING AND A NUMBER";B$,N
200 PRINT: PRINT "THE STRING YOU ENTERED WAS:-";PRINT
210 PRINT B$
220 PRINT: PRINT"AND THE NUMBER WAS:-";N
230 PRINT: PRINT"DEMOSTRATION ENDED."
```

So the program returns to line 40 where it selects a different random number for 1 and also for J. As the background colour is decided by J, that changes when the program reaches line 50. And as 1 is also different the sound changes in line 70. When the program reaches line 80 again it goes back to line 40 where it selects .... (forever, or until you press the BREAK key).

MAKING CHANGES

That was the first program, it didn't do very much but it's a start. It got us going and introduced some new commands. A more detailed explanation of the commands can be found in the 'boxes' throughout the text. (Each box also contains a small program to demonstrate the command, you should try running these).

If you were not very impressed with the first program, you may like to make some changes. To change a program you have already typed in is simple. Because a BASIC program sequence is decided by the line number, a line can be changed by typing a line with the same number as the one in the program. The new line replaces the old. Try entering a new line 70.

```
70 SOUND I,K
```

A new line can be inserted into the program by giving it a number that will place it in the correct order.

Try typing

```
45 K = RND(20)
```

As this line is numbered 45 it will come between line 40 and line 50.

(When you write your own programs you can number lines with any number between 0 and 63999. It is usually a good idea to number lines in 10's, i.e. 10, 20, 30, so that you have room to insert extra lines if necessary).

Try running the changed program (type RUN), the new line 45 selects another random number, this time between 1 and 20. The changed line 70 uses this random number, (in variable K), to change the duration of the sound.

PROGRAM CONSTRUCTION

Though it is easy to sit at the keyboard and type in program lines, the difficulties tend to appear afterwards, especially when programs start to get longer. Typing in the program should be the *last* part of creating a new program.

RND

The command RND generates random numbers.

RND is a *function*. A function, in BASIC is something that takes one or more numbers and performs some operation on them which results in a single value. The numbers used by the *function* are called *arguments* and are always put in brackets after the *function* name. The result of a *function* is said to be returned to the program.

The RND *function* returns a random number, which depends on the value of the function argument.

- If the value of the argument is 0 (RND(0)) the *function* returns a value between 0 and 1.
- If the value of the argument is greater than 0 (RND(6)) the function returns a value that is an integer (whole number) between 1 and the value of the argument. (RND(6 will return either 1, or 2, or 3, or 4, or 5, or 6, you don know which because it is random!)

```
10 CLS
20 PRINT @ 8,"ENTER A NUMBER";: INPUT N
30 CLS: PRINT @ 194,"3 RANDOM NUMBERS FOR N = ";N
40 PRINT @ 270,RND(N)
50 PRINT @ 302,RND(N)
60 PRINT @ 334,RND(N)
70 GOTO 20
```

CLS

The command CLS is used to clear the screen, and set the background colour. The normal background colour is green. If you use CLS on its own this is the colour that will be set.

To change the background colour add a number between 0 and 8 after the CLS command, (CLS 2).

The available colours are:-

0	Black	1	Green	2	Yellow
3	Blue	4	Red	5	Buff
6	Cyan	7	Magenta	8	Orange

The actual hue of these colours will depend upon your television set.

You will notice, however, that whatever the background colour, the computer will print all text as black or green.

```
10 CLS
20 PRINT @ 0,"BACKGROUND COLOUR DEMOSTRATION.";
30 PRINT @ 192,"ENTER NUMBER BETWEEN 0 and 8";
40 INPUT C
50 CLS C
60 PRINT @ 288,"THIS IS BACKGROUND COLOUR:-";C
70 GOTO 20
```

Start with a pencil and paper, then write down what you intend to do. Break the problem into distinct different sections. Many problems to be done on a computer tend to divide easily into at least three parts,

1) Preparation, titles, instructions and data entry,
2) Calculations,
3) Displaying the results.

Now tackle each section separately, further dividing into sub-sections until you are left with a simple action like 'add 1 to counter'. Order these actions into a sensible sequence, write them out in full, then start with the next section. When you have finished you have a series of steps (some of which may be very primitive) inside each section. Anyone reading through the result should be able to solve the problem, whatever it may be, by using only simple arithmetic. (This plan of action is called an algorithm in computerese). All that is left now is to convert your plan into a language the computer can understand. Ideally, each step in your plan should translate into one line in a BASIC program.

After the translation you should test each section separately and ensure that it does what it is supposed to before you assemble the complete program. When the program is running successfully, do not celebrate by having a bonfire of all the pieces of paper lying around. Keep the final plan, parts of which may be useful in other programs, also some errors may not appear until long after you have forgotten what it was that you did. It is also useful to put comments into the program itself to remind yourself of what is going on. To do this BASIC has the REM statement. This statement does not actually do anything. BASIC ignores anything after the REM command, so if more than one statement appears on a line make sure the REM statement is last. There is a shortened version of the REM statement which uses the single quote ('), (this is obtained by [SHIFT] and $\left[\begin{smallmatrix}'\\7\end{smallmatrix}\right]$).

```
10   REM PROGRAM TO FIND AVERAGE
140  A = B * C:GOTO 15: REM RETURN TO START
48   K = K + 1:'INCREMENT COUNTER
```

While REM statements do take up some of the memory, it is unwise to do without them completely. Trying to understand your own programs a year after you have written them can be a frustrating experience if there are no comments at all.

All this section may seem terribly pedantic, but it is a well known fact in computer circles that more time is spent on debugging (locating errors in a

Print @ Grid

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0																																
32																																
64																																
96																																
128																																
160																																
192																																
224																																
256																																
288																																
320																																
352																																
384																																
416																																
448																																
480																																

PRINT @

The PRINT @ command is used to place output at a specified place on the screen.

For this purpose the screen is divided into a 16 × 32 grid, giving 512 positions. See the diagram opposite to show how the grid is numbered.

The form of the PRINT @ command is

PRINT @ *expression, print list*

The *expression* can be a number, a variable or an arithmetic expression, as long as the value is between 0 and 511.

The *print list* is the same as used in the PRINT command, it can be numbers, variables, strings or expressions, separated by commas or semi   colons.

lf you considered your screen as having 16 lines ruled on it, then the statement,

```
PRINT @ 32 * (LINE     1), A
```

will print the value of A at the beginning of the imaginary line on your screen. Each line will depend on the value of the variable LINE, (a number from 1 to 16)

The following example is a deliberately confusing way of producing a fairly simple result. Try and sort out what it is doing, run the program, then write a more straightforward version to produce the same result.

Note the use of the semi-colon at the end of the statements to stop the rest of the line being erased. (If you don't believe it, try entering line 1 lo without the semi-colon on the end).

```
10 CLS: P$ = "PRINT @ ":N = 1
20 ROW = 12:A$ = "ON THE SCREEN"
30 PRINT @ 32 * (ROW    1)+19, "TO PLACE";
40 PRINT @ 448  +  9, "ANYWHERE"
50 PRINT @ 262,"THIS";: PRINT @ 267,"SHOWS";
60 PRINT @ 32*(ROW   1)+11,"BE USED";:PRINT @
    13,"PAGE";
70 PRINT @ 448 + 18, A$:
90 PRINT @ 273, "HOW'▽"; P$
```

```
80   PRINT @ 32*(ROW   1)+7,"MAY"
100 PRINT @ 134,P$;"▽DEMOSTRATION"
110 PRINT @ 451,"ITEMS"
120 PRINT @ 18,N
130 GOTO 130
```

The last line (130) puts the computer into an endless loop, which doesn't do anything. This stops the OK prompt appearing when the program has finished. Press the BREAK key to stop the program.

SOUND

The SOUND command generates a tone or specified pitch and duration. It requires two arguments:

```
50 SOUND P,D
```

P is a number between 1 and 255. The lowest tone is i, the highest 255. Middle C on the piano is P = 89.

D can be any number between 1 and 255. D = 16 gives a tone of about 1 second duration

```
10  CLS
20  PRINT  @ 6,"SOUND DEMOSTRATION."
30  PRINT  @ 64,"ENTER A NUMBER BETWEEN 1 AND 255";
40  PRINT  @ 96,"FOR THE PITCH OF THE NOTE";:INPUT P
50  PRINT  @ 192,"ENTER A NUMBER FOR THE DURATION OF
    NOTE.";
60  INPUT D: CLS(RND(9)   1)
70  SOUND P,D
80 GOTO 10
```

GOTO

The GOTO command has the form

GOTO *line number*

The *line number* must be a number (not a variable), and must exist somewhere in the program. If the *line number* is not found the program will stop with an ?UL ERROR (undefined line).

The GOTO statement is executed immediately, there is therefore no point in putting another statement on the same line, after a GOTO statement. The program will never reach it.

```
20 CLS
30 GOTO 60
40 PRINT "AT LINE 40"
50 GOTO 80
60 PRINT "AT LINE 60'
70 GOTO 40
80 PRINT "END OF PROGRAM"
```

program) than on writing the program. If you work as suggested there is less chance of errors getting into the program, and if they do, they are easier to find.

A PROGRAM EXAMPLE

PROBLEM: Use the computer to simulate the throw of 2 dice.

SECTION 1 Display titles and instructions.

a) Clear the screen
b) Print the title
c) Print instructions
d) Print headings, 1st dice, 2nd dice.

SECTION 2 Find the values for two dice.

(When you roll a dice any of the 6 sides appear randomly)
a) First dice is a random number between 1 and 6
b) Second dice is a random number between 1 and 6

SECTION 3 Print answer.

a) Print the value of 1st dice, 2nd dice

SECTION 4 Halt the program and repeat if needed.

a) Stop program
b) Ask for another roll
c) Repeat sections 1,2 and 3 as necessary

Translating into BASIC (with added comments) we get:

```
10   ' DICE SIMULATION PROGRAM
20   '
30   ' FIRST SECTION
40   CLS: REM CLEAR THE SCREEN
50   PRINT"DICE SIMULATION":'TITLE
60   PRINT
70   PRINT "USE BREAK KEY TO END PROGRAM":'INSTRUCTIONS
80   PRINT
90   PRINT "1ST DICE","1ND DICE":'HEADINGS
100 PRINT
110 REM END OF FIRST SECTION
```

REM

The REM command is used for inserting comments into a program. The computer ignores anything following a REM (or its shorthand form ') on that line.

```
10  REM THIS IS A COMMENT LINE
35  D = B*B   4*A*C: 'FIND DETERMINANT
```

```
120 REM
130 REM SECOND SECTION
140 D1 = RND(6): ' ROLL 1ST DIE
150 D2 = RND(6): ' ROLL 2ND DIE
160 'END OF 2ND SECTION
170 '
180 'THIRD SECTION
190 PRINT D1,D2: ' DISPLAY RESULT
200 'END OF 3RD SECTION
210 '
220 'FOURTH SECTION
230 INPUT"PRESS ENTER TO ROLL DICE"; A$
240 GOTO 40: ' RETURN TO START
250 'END OF 4TH SECTION
```

This program does not really need ali the comments, write your own version of the program.

There is no 'correct' version of a program, the correct version is one that works. There may be more elegant or efficient ways of doing the same problem. Generally, a shorter program is faster and uses less memory.

We finish this section with a 'fancier' version of the same program, note how breaking the problem into sections sometimes allows the program to be assembled in a different order but with a similar result.

```
10   ' DICE SIMULATION PROGRAM
20   '
30   ' FIRST SECTION
40   CLS0: PRINT @ 8,"DICE SIMULATION";
50   PRINT @ 167,"FIRST▽";:PRINT @ 178,"SECOND";
60   PRINT @ 200,"DICE";:PRINT @ 211,"DICE";
70   PRINT @ 450,"USE THE BREAK TO END PROGRAM";
80   ' FOURTH SECTION
90   PRINT @ 358,"PRESS ENTER TO ROLL";
100 INPUT A$
110 ' SECOND SECTION
120 D1 = RND(6):D2 = RND(6)
130 ' THIRD SECTION
140 PRINT @ 265,D1;:PRINT @ 276,D2;
150 GOTO 90
```

CHAPTER FOUR

GOOD HOUSEKEEPING

SETTING UP THE RECORDER

Some of the programs are beginning to get quite long and it is a nuisance to have to type in a program every time you want to run it. However, it is fairly simple to store your programs on tape and call them back into memory when needed. To do this you need a casette recorder and a connection lead.

Any cassette recorder of reasonable quality can be used, provided it has the ability to

a) record from an outside source (a jack socket usually marked AUX, or LINE IN)
b) output to a loudspeaker or earphone (a jack socket marked EAR, or MONIT, or L/S, or SPKR)
c) stop and start by remote control (a small jack socket usually marked REM and next to the microphone socket).
d) operate from mains power, This is not essential, but weak batteries can seriously affect your success in storing and retrieving programs.

To connect the recorder to the computer, put the DIN plug into the socket market TAPE on the left side of the computer.

The three plugs on the other end of the lead are connected to the recorder as follows:

1) The smallest jack fits into the socket marked REM (the remote on/off switch)
2) The large jack with the grey wire goes into the socket marked AUX
3) The other large jack with the black wire goes into the socket marked EAR (the earphone socket)

Turn on the recorder, insert a tape and rewind back to the beginning of the tape. Now set the volume control at 6 (or just over halfway between ON and FULL). You are now ready to store your programs.

STORING A PROGRAM ON TAPE

Type in a program and run it to ensure that it works correctly. Then proceed as follows:

1) Press the PLAY and RECORD buttons together until they lock.
2) Type the command

CSAVE "PROGRAM 1" press [ENTER]

The name 'PROGRAM1' can be replaced with any name you like, (it must begin with a letter and not be larger than 8 characters). When you press [ENTER] the cassette motor will start and the program will be recorded. After a while the OK prompt will return to the screen, and the cassette motor will stop.

The program will still be in the computers memory, only a copy has gone on to the tape. You have saved a program called PROGRAM 1, (or whatever you called it) onto the tape.

LOADING PROGRAMS INTO MEMORY

To return a saved program into the computer, first type NEW to clear any existing program from memory. The process is as follows:

1) Rewind the tape to the beginning.
2) Press the PLAY button until it locks.
3) Type the command.

CLOAD "PROGRAM1" press [ENTER]

The cassette motor will start and a letter S will appear in the top left hand corner of the screen. This indicates the computer is searching for the program. When found the S will change to

F PROGRAM 1

When the OK prompt appears, and the cassette motor stops, the program has been loaded into the computer memory. To check it is there, type LIST

If the program is not there, or I/O ERROR appears on the screen, maybe your cassette recorder requires a different setting. Check the connections are correct, then repeat the saving and loading sequence with different settings of the volume control until successful.

CSAVE
CLOAD
SKIPF

The command CSAVE saves a program onto cassette tape. The program name must be eight characters or less

```
CSAVE "PROGRAM"
```

To save data onto cassette use the extra parameter A, the information will then be stored in ASCII format. It may then be read by an INPUT#   1 command.

```
CSAVE "DATA",A
```

The CLOAD command loads a specified program file from cassette into memory.

```
CLOAD "PROGRAM"
```

The SKIPF command skips to the next program file after the program specified, or to the end of the specified program.

```
SKIPF "PROGRAM"
```

SAVING MORE THAN ONE PROGRAM

To save more than one program on tape, you want to avoid recording on top of those already there. This requires the tape to be positioned past the last program. This is done as follows: -

1) First rewind the tape to the beginning.
2) Press PLAY until it locks.
3) Type the command

```
SKIPF "PROGRAM 1"
```

The motor will come on, the computer will search (S) for the program, find (F) it, read past it, then stop the motor and give the OK prompt.

4) Press the STOP button.
5) Press PLAY and RECORD buttons together, name the new program, then CSAVE it.

After saving this program, you are past the end of it, so may type in and save others if you want.

HINTS ON RELIABLE RECORDING

1) Operate the recorder on mains power to ensure constant speeds.
2) Use new, high quality cassettes. While longer playing cassettes (C120) may appear to be more convenient, it is better to use shorter lengths (C30 or C12).
3) Always start the search at the beginning of the tape, do not rely on the tape counter.
4) Do not leave the PLAY or RECORD buttons down, press STOP after you finish saving or loading.
5) Rewind cassettes before putting away.
6) Label cassettes, immediately after saving, and remove the erase protect tab from the back of the tape for important programs.

Even if you are careful, accidents can happen! Some programs can be a considerable investment of your time and effort so make another copy on another cassette.

THE EDITOR

As the program lines get longer, there is more chance of typing errors. Up to this point the only remedy has been to retype the whole line. This is no longer necessary, as we are going to introduce the EDITOR. The EDITOR allows you to move backwards and forwards along a line, changing, deleting or inserting characters. To call the EDITOR type,

EDIT *linenumber* [ENTER]

where *linenumber* is the number of the program line you wish to work on. The line will be displayed on the screen in full. Then the line number will be printed with the flashing cursor next to it.

MOVING DOWN THE LINE

The cursor is now at the start of the line. To move forward along the line press the spacebar. The cursor will move, displaying the characters it has passed over. To move backwards we can use the backspace [ ] key. You can speed up the movement by typing a number followed by the key, (i.e. [5] [SPACE] will move the cursor 5 forward, and [3] [ ] will move it 3 back). Using these two keys you can position the cursor over any character in the line. You will not see the character because the cursor will be flashing over it. Two other commands allow you to jump directly to a particular place in the line. To add to the end of a line type [X], for extend. The cursor will jump to the end of the line, and you just type in the characters you wish to append. By using the search facility you can move directly to a target character. TYPE [S] followed by the character you wish to move to. ([S][A] will move the cursor to the *first* occurrence of the letter A in the line). If there is more than one A in the line and you want to move to the third A, then [3][SI[A], means search for the third occurence of A.

ALL CHANGE

Once the cursor has been positioned by using the spacebar, backspace or [S] key, you can:

a) *Delete* a character by typing [D]. This will delete the character under the cursor. To delete more than one character, [5][D] will delete the next five characters starting from the cursor position.

b) *Change* a character by typing [C], followed by the new character. ([C][F] will change the character under the cursor to F). As with delete [3][C] will change the next three characters to the three typed in.

c) *Insert* characters by typing [I] followed by the characters you wish to insert. Once you type [I], the editor goes into its insert mode. In this

EDIT

The EDIT command is used to alter the contents of the specified line.

EDIT *linenumber*

Once in the EDIT mode any of the following editor commands may be used.

L	List the current state of the line
C *character*	Changes the current character
n C *character*	Changes the next *n* character to new characters
I	Inserts characters
D	Deletes current character
n D	Deletes next n characters
H	Deletes rest of line from current position, and goes into insert mode
X	Extends the line, moves to end and goes into insert mode
S *character*	Searches for first occurrence of character
n S *character*	Searches for nth occurrence of character
K	Deletes rest of line from current position
n K *character*	Deletes line to nth occurrence of character
n [SPACEBAR]	Advances cursor n spaces. If n omitted 1 is assumed
n [ ]	Backspaces cursor n spaces. If n omitted 1 is assumed
[SHIFT][ ]	Leave insert mode and return to edit mode
[ENTER]	Leave editor, store line and return to keyboard

mode everything that is typed is inserted into the line. To leave the insert mode and return to the normal editor mode you must type [SHIFT] and [   ] together.

List the current state of the line by typing [L]. The line will be displayed with any changes you have made so far. After displaying the line the cursor returns to the beginning.

Leave the editor by pressing the [ENTER] key. This will place the altered line back into the program in memory and then give the OK prompt. You can leave the editor at any time or in any mode just by pressing [ENTER].

Here is an example of an editing session. It may look a little complicated at first, but you will be surprised how quickly you can become proficient with very little practice. The keys you press are in square brackets.

[E] [D] [I] [T] [1] [0] [ENTER]

10 PRINT▽ "THEIR ▽ ARE ▽ MANY ▽ MISTOOK ▽ IN ▽ LINE

10	(denotes cursor position)
[1] [0] [SPACE]	move ten forward, (or [21 [S] [1])
[C] [R] [C] [E	change I to R and R to E
[S] [M]	move to start of MANY
[2] [D]	delete M and A
[SPACE] [C] [0]	skip over N and change Y to 0
[S] [O]	move to first 0 in Mistook
[D] [C] [A]	delete first O and change second O to A
[SPACE] [1] [E] [S]	pass over K and insert ES
[SHIFT] [ ]	leave insert mode
[S] [L]	move to L in LINE
[I] [T] [H] [I] [S] [SPACE]	insert THIS and a space before LINE
[SHIFT] [ ]	leave insert mode
[X]	move to the end of line (you are now in insert mode)
[SPACE] [N] [O] [W] ["]	add NOW and quotation mark to close string
[SHIFT] [ ]	leave insert mode
[L]	list the current state of the line

10 PRINT▽ "THERE ARE NO MISTAKES IN THIS LINE NOW"
10

[ENTER]	store the line and leave EDITOR.

There are two other editor commands, which should be used with care,

a) The *kill* character. Typing [K] deletes the rest of the line from the cursor position. [K] followed by a character will delete everything from the cursor to the first occurrence of that character. [3] [K] [A] will delete up to the third occurrence of A.

b) The *hack* character. Typing [H] deletes the rest of the line from the cursor position and then goes into insert mode. It is useful for retyping the end part of a line.

If your typing is perfect and you never make programming mistakes, you may ignore this section. For the rest of us, the EDITOR will make life much easier from now on. Type in one of the examples you have been given, save it on tape if you like, then turn into a different program by using the EDITOR.

MORE SYSTEM COMMANDS

Commands like LIST and RUN are system commands. They are not part of the program but a direct command to your computer to do something now. Here are some others which make the business of programming easier.

DELeting program lines

To remove a line from a program we have been typing the line number followed by [ENTER]. This is alright for one or two lines, but what about 30 or 40 lines?

DEL *linenumber*   *linenumber*

will delete a whole block of lines starting with the first line-number up to and *including* the second linenumber.

DEL 100   250

will remove all lines from 100 to 250, inclusive, from the program currently in memory.

The DEL command can be used in other forms as well,

DEL 20	will delete line 20 only
DEL 30	will delete all lines from 30 to the end of the program.
DEL 200	will delete all lines from the start of the program up to and including 200
DEL	will delete the entire program.

RENUMbering program lines

DEL

The DEL command is used to delete specified lines from the program currently in memory.

DEL *linenumber1*   *linenumber2*

The command will delete all lines from *linenumber* 1 up to and including *linenumber* 2. Both linenumbers are optional and the command can be used in any of the following forms:

DEL	Delete the entire program
DEL 100	Delete from the start to line 100
DEL 300	Delete from the line 300 to the end
DEL 40	Delete line 40 only
DEL 100 200	Delete all lines between 100 and 200

RENUM

The RENUM command allows all or part of the program line numbers to be changed. The RENUM command also changes the line numbers in branching Statements (GOTO etc.) to ensure that program flow continues at the same place.

RENUM *newline, starline, increment*

The command will remember all lines from startline beginning with newline and a gap between the lines of increment. All parameters are optional and the statement can be used in any of the following forms:

RENUM	Renumber entire program. Lines will be numbered 10,20,30 ....
RENUM 100	Renumber entire program. Lines will be numbered 100,110,120 ....
RENUM 100,50,5	Renumber starting at old line 50. Lines will be numbered 100,105,110 ....
RENUM,,20	Renumber entire program. Lines will be numbered 10,30,50 ....

Note that if a parameter is omitted and a following parameter is to be used, the comma must he present. RENUM cannot be used to alter the line sequences.

The RENUM command will remember all, or some of, the line numbers in your program. It will also change the line number in GOTO, GOSUB, IF THEN, ON GOTO, and ON GOSUB statements to make sure they still branch to the same place, we will meet these statements later.

RENUM *newline, startline, increment*

Newline is the new line number of the first line to be renumbered. *Startline* is where you want to start renumbering from, and *increment* is the increment to be used between each renumbered line. Any, or all, of these parameters can be omitted. If you omit *newline,* 10 is assumed. Omitting *startline* causes the entire program to be renumbered, and omitting *increment* will cause the line numbers to increase in tens.

RENUM	will number the entire program as 10, 20, 30....
RENUM 100,50,5	will number the lines from 50 as 100, 105, 110 ... All the lines before 50 will be unchanged.
RENUM 110,,2	will renumber the entire program as 110, 112, 114 ....
RENUM,,5	will renumber the entire program as 10, 15, 20 ....

Note that if you omit a parameter, but wish to use one that comes after it in order, you must include the comma.

TRacing a program flow

Sometimes when you have difficulty with a program it is useful to know where it is going to. The trace facility allows you to do this. By typing TRON *before* you run the program, you switch the trace on. The line number will now be printed on the screen as the program comes to it. This enables you to see if the program is branching to the correct place. To switch the trace off, type TROFF.

STOPing and starting

A program can be halted during a program run by including a program line containing the STOP command

```
185 STOP
```

This line will cause the program to halt when it reaches line 185. A message is printed on the screen telling you the program has stopped and which line it sopped at. You can now look at the contents of any variable by using PRINT or ? To restart the program type CONT (means continue) and the program will carry on from the next line after the STOP.

You can now renumber programs, delete unwanted lines, change the contents of any line, and save the result into a cassette tape. As we are now in a postion to maintain our programs, we will now start to construct some that do something more interesting than flash screens and made weird noises.

TRACE

The program flow may be followed by using the trace. As each line is reached the line number is printed on the screen. The trace must be switched on before the program is run.

TRON	switches on the trace
TROFF	switches off the trace

Both are direct commands and do not require a line number.

END
STOP
CONT

The END command terminates program execution and returns control to the keyboard.

The STOP command halts execution of the program at the line containing the STOP. A message BREAK AT N appears on the screen to indicate the halt has taken place at line number N. To restart the program use CONT (continue), without a line number. The program will continue operation at the next line after the STOP. A program will not continue after an END statement, it must be rerun.

CHAPTER FIVE

GOING PLACES

At the end of chapter 3, we were discussing how to construct a program by considering it as a number of separate sections. Next we shall cover how to direct the path the program takes to join up these sections. This is called branching. We have already met the branching statement, GOTO. This is an unconditional branch, because as soon as the program reaches the GOTO statement it jumps immediately to the specified line number and continues from there. The program has no choice in the matter, GOTO means go to at once, not maybe or sometimes. So far we have mainly used the GOTO to return to the beginning of the program. While the ability to repeat part of a program over and over is extremely useful, it is unlikely we would want to do so forever. (It is also not a good practice to have to rely on the BREAK key to stop the program). Fortunately, the BASIC language provides us with a number of statements that allow us to control the flow of the program.

SELECTING OPERATIONS

The first of these statements is an extension of our old friend the GOTO statement. It is the ON... GOTO statement and has the following form:

ON *numeric expression*   GOTO *list of line numbers*

The numeric expression is evaluated and, if necessary, truncated to a whole number (i.e. the numbers after the decimal point are dropped). Control is then transferred to one of the line numbers in the list. If the expression evaluates to 1, it goes to the first line number, if 2, the second, and so on. If the value of the expression is less than 1, or greater than the number of line numbers in the list, the computer will ignore the statement completely and carry on at the next line. A negative value for the expression, however, will cause the program to halt with an error message. It is usually a good idea to check that the value is within the intended range before reaching the ON. . . GOTO statement. Here are some examples of the statement.

```
140 ON P GOTO 200,300,400
210 ON X    4 GOTO 20,40,700,10,690
185 ON B*C/D    E GOTO 115,285,900,40
```

The ON... GOTO statement is a useful way of selecting from a number of options and can be considered as a conditional branch. We will use it this way in the example in the next section.

ON. . .GOTO

The ON GOTO command performs a multiway branch to specified line numbers.

ON *expression* GOTO *line number list*

The expression is evaluated (and truncated if not an integer). The program then branches to the number in the list with a position equal to the value of the expression. If the expression value is four, the ON GOTO command will select the fourth item in the line number list. The value of the expression must not be negative, an error wili result. If the expression value is zero or greater than the number of line numbers in the list, the statement will be ignored and the program will continue on the next line.

```
10 CLS: PRINT"SOLUTION OF QUADRATIC EQUATION"
20 INPUT "A,B,C"; A,B,C:IF A = 0 THEN 20
30 R =    B(2*A):D = B*B   4*A*C:S = SGN(D)
40 P = SQR(D*S)/(2*A)
50 ON S + 2 GOTO 80, 60, 70
60 PRINT "PERFECT ROOTS": PRINT R,R:END
70 PRINT "REAL ROOTS": PRINT R+P, R   P:END
80 PRINT "COMPLEX ROOTS": PRINT R,R: PRINT P,    P: END
```

DECISIONS

A much more useful form of a conditional branch is available using the IF ... THEN statement. This is probably the single most powerful statement in the BASIC language, and as such can range from very simple to extremely complex. In its simplest form:

IF *condition* THEN *linenumber*

it has the meaning IF the condition is true THEN, go to line number, otherwise continue on the next line.

```
120 IF D > 9 THEN 250
180 IF A$ = "YES" THEN 600
```

In line 120, above, the program will transfer control to line 250 if, *and only if*, the value of D is greater than 9. If D is less than, or equal to, 9 the program will continue at the next line. In line 180 the branch to line 600 will occur only if the string variables A$ contains the characters YES. The match in this case must be exact. YESS, YEH, YEA, YEP, Y, OK, (or even 'yes', remember lower case characters), will cause the program to ignore the branch.

In its full form the IF ... THEN statement appears as follows:

IF *condition* THEN *action 1* ELSE *action 2*

which has the meaning IF the condition is true THEN perform action 1, if the condition is not true then perform action 2.

```
210 IF P = 3 THEN PRINT "TRUE" ELSE PRINT "FALSE"
```

In this example TRUE will be printed only if P is equal to 3, if P has *any* other value, FALSE will be printed. In both cases the program will continue on the next line. It is possible for both action 1 and action 2 to consist of more than# one statement.

```
210 IF P = 3 THEN PRINT "TRUE": R = R + 1:GOTO 560
      ELSE PRINT"FALSE":L = L + 1
```

If P is equal to 3 the computer will print TRUE, add 1 to variable R and continue the program at line 560. For any other value, it prints FALSE, adds 1 to L and continues on the next line.

The computer will accept for *action 1* and *action 2* any legitimate BASIC statements, including other IF ... THEN statements, if you wish. The only constraint is that the complete IF ... THEN statement must fit onto one line. (One line may contain a maximum of 256 characters including the line number).

Up to this point we have only considered what happens after the condition has been evaluated. The decision part of your computation lies in testing the condition. A condition can be TRUE or FALSE, nothing else. (The computer gives TRUE the value 1, and FALSE the value 0). A simple condition has the form

expression1   relation   expression2

expression 1 and *expression 2* are the usual BASIC expressions, such as appear in the assignment statement. Both expressions must be of the same type, (both numeric, or both string). A *relation* can be any of the following:-

MEANING	SYMBOL	EXAMPLE
Equal to	=	60 IF X=Y+2 THEN 100
Less than	<	110 IF A*B+2<C/2 THEN 20
Greater than	>	185 IF A$>B$ THEN PRINT A$
Less than or equal to	<=	220 IF 4*W9<=B/Z9 THEN A = A 1
Greater than or equal to	>=	415 IF B7>=0 THEN X = 0
Not equal to	<>	80 IF A$<>'"YES" THEN 999

Note that a relation can be applied to strings as well as numeric expressions. When strings are compared each character is checked in turn, and so IF. . . THEN statements can be used to compare for alphabetic order.

The condition "A" < "B" is true, because the letter A comes before the letter B in the alphabet. The condition "AAA" < = "AA" is false, because AAA would appear after AA in a dictionary, for instance.

Conditions can be extended by combining two or more conditions using the operators AND, OR

```
270 IF A = 4 AND B = 7 THEN 500
```

The AND operator means both conditions must be true at the same time for the whole conditional expression to be true. In the example, if A is equal to 4 and B has any other value but 7, the whole will be false and the program will continue on the next line.

The OR operator means that if any one of the conditions is true, the whole expression is true.

```
320 IF D<3 OR F+G>25 THEN 100
```

The program will branch to line 100 if D is less than 3, no matter what the value of F+ G may be. Alternatively, it will branch to line 100 if F+ G is greater than 25, irrespective of the value of D.

IF. . .THEN. . .ELSE

The full form of the IF command is,

IF *condition* THEN *action1* ELSE *action2*

The statement tests the condition which will be either TRUE or FALSE. lf TRUE then action 1 is carried out, if FALSE then action 2.

A condition is made up of an expression, a relation and an expression. The expressions may be any BASIC expressions of the same type (i.e. both numeric or both string). A relation is any of the following operators,

=	Equal to	<>	Not equal to
>	Greater than	<	Less than
>=	Greater than or equal to	<=	Less than or equal to.

Conditions may also be combined by the logical operators AND, OR, NOT.

condition AND *condition*	*only* TRUE *if both conditions* are true
condition OR *condition*	TRUE if either *condition* is TRUE
NOT *condition*	TRUE if *condition* is FALSE.

Action 1 and *action 2* may be any BASIC statement including another IF statement.

The ELSE part of the command is optional and may be omitted, in which case the program continues on the following line if the condition is FALSE.

```
10 CLS: PRINT @ 9,"GUESSING GAME":N=RND(100):T=0
20 INPUT "GUESS MY NUMBER";G: T = T+1
30 IF G = N THEN PRINT "CORRECTIN";T;"TRYS":END
40 IF G > N THEN PRINT "NO IT'S SMALLER" ELSE PRINT
   "NO IT'S LARGER
50 GOTO 20
```

INKEY$

The INKEY$ command is a function. It checks the keyboard to see if a key is being pressed, if so returns the string character of the key.

It can be used to place a single character into a string variable, and does not require the [ENTER] to follow the entry.

```
10 CLS0:PRINT @ 5,"PRESS ANY KEY AND I WILL";
20 PRINT @ 38,"TELL YOU WHAT IT WAS. ▽"
30 B$ = "YOU ARE NOT PRESSING A KEY▽▽"
40 C$ = "THE KEY YOU PRESSED WAS:    ▽"
50 A$ = INKEY$
60 IF A$="" THEN PRINT @ 193,B$; ELSE PRINT @ 193,C$;A$
70 FOR D = 1 to 600: NEXT D: GOTO 50
```

The program which follows is a simplified form of a common educational program. The comments in the program indicate what each section is doing. Note the use of ON ... GOTO to select the option and IF . . . THEN statements to check the arithmetic. At line 800 we introduce another new word, INKEY$. This command scans the keyboard to see if a key has been pressed. If it has the character pressed will be stored in A$. lt does not require the [ENTER] key to be pressed after the input. See the box marked INKEY$.

The program looks a lot longer than it really is, as over half of it is comment lines. To save space in the future we will not be quite so liberal with comments.

```
10   REM   ARITHMETIC PRACTIQUE PROGRAM
20   REM
30   REM    ZERO COUNTERS & FIND DIFFICULTY LEVEL
40   REM
50   R = 0:W = 0:CLS
60   T$ = "ARITHMETIC PRACTICE"
70   PRINT @ 6,T$
80   PRINT @ 64,"▽▽ENTER LEVEL OF DIFFICULTY"
90   PRINT @ 128,"▽▽A NUMBER BETWEEN 1 AND 10":INPUT
     L:L1 = 10*L   1
100 REM
110 REM  DISPLAY OPTIONS
120 REM
130 CLS:PRINT @ 6,T$
140 PRINT @ 71,"1. ▽ADDITION."
150 PRINT @ 103,"2.▽SUBTRACTION."
160 PRINT @ 135,"3.▽MULTIPLICATION."
170 PRINT @ 167,"4. ▽DIVISION."
180 REM
190 REM  SET PRINT POSITION
200 REM
210 P = 224: Q = 352
220 REM
230 REM  SELECT OPTIONS
240 REM
250 PRINT @ P,"▽▽WHICH DO YOU WANT TO TRY▽";:INPUT A

260 REM
270 REM    SELECT 2 NUMBERS FOR PROBLEM
280 REM
```

```
290 N1 = RND(L1):N2=RND(L1)
300 REM
310 REM  BRANCH TO OPTION
320 REM
330 ON A GOTO 390,460,530,600
340 REM
350 REM  OPTION WRONG     TRY AGAIN
360 REM
370 CLS: SOUND 160,3:GOTO 130
380 REM
390 REM  ADDITION SECTION
400 REM
410 PRINT @ P," ▽▽▽▽▽▽▽▽▽▽▽▽ADDITION."
420 PRINT @ Q,"▽▽WHAT IS";N1;"PLUS   ";N2;:INPUT N4
430 N3 = N1 + N2
440 IF N4 = N3 THEN 690 ELSE 730
450 REM
460 REM SUBSTRACTION SECTION
470 REM
480 PRINT @ P,"▽▽▽▽▽▽▽▽SUBTRACTION."
490 PRINT @ Q,"WHAI IS";N1;"TIMES";N2;:INPUT N4
500 N3 = N1     N2
510 IF N4 = N3 THEN 690 ELSE 730
520 REM
530 REM  MULTIPLICATION SECTION
540 REM
550 PRINT @ P,"▽▽▽▽▽MULTIPLICATION."
560 PRINT @ Q,"WHAT IS";N1;"TIMES";N2;:INPUT N4
570 N3 = N1*N2
580 IF N4 = N3 THEN 690 ELSE 730
590 REM
600 REM  DIVISION SECTION
610 REM
620 PRINT @ P,"▽▽▽▽▽▽▽▽DIVISION."
630 PRINT @ Q,"WHAT IS2;N1;"DIVIDED BY";N2;:INPUT N4
640 N3 = N1/N2
650 IF N3 = N4 THEN 690 ELSE 730
660 REM
670 REM  CORRECT ANSWER
680 REM
```

```
690 R = R+1:PRINT @ P,"▽▽▽▽▽▽▽▽▽CORRECT.":GOTO 770
700 REM
710 REM  WRONG ANSWER
720 REM
730 W = W + 1:PRINT @ P,"▽▽▽▽▽▽▽▽▽▽▽WRONG.":
       PRINT
      @ P+32," ▽▽▽▽▽THE ANSWER IS▽";N3:GOTO 770
740 REM
750 REM  CHECK FOR REPEAT & CLEAR LINES
760 REM
770 FOR D=1 TO 600:NEXT D
780 PRINT @ P," ▽":PRINT @ P+32," ▽":PRINT @ Q,"   "
790 PRINT @ P,"DO YOU WANT TO TRY ANOTHER?(Y/N)"
800 A$=INKEY$:IF A$="" THEN 800
810 IF A$="Y" THEN 130
820 REM
830 REM  GIVE RESULTS AND FINISH
840 REM
850 CLS: PRINT @ 128,"YOU GOT";R;"CORRECT
       AND";W;"WRONG"
860 END
```

DO IT AGAIN, AND AGAIN, DRAGON

Often a program needs to repeat a sequence of lines a number of times. The next type of branching statement allows us to do this. Enter this small program and RUN it.

```
10   CLS
20   FOR I=1 TO 50
30   PRINT @ 198,"COUNTER I="
40   PRINT @ 214,I
50   NEXT I
60   PRINT @ 396,"LOOP ENDED"
```

It is far too fast to see what is going on, so add the line,

```
45  FOR  J = 1 to 100: NEXTJ
```

As you can see, the program counts from 1 to 50. The lines that are being repeated are lines 30,40 and 45. A repeat sequence like this is called a 'loop', because the program loops back, in this case to line 20. The statements which control the loop are at line 20, (the top of the loop) and at line 50 (the bottom of the loop). Line 20 translates as 'FOR all values between 1 and 50 in steps of 1, do all the following statements until the NEXT statement is reached'. The variable 1 is acting as the counter and has values 1,2,3,. . . .50. We added the line at 45, because it was counting so fast. This loop has no statements to perform, so the counter (variable J this time) just counts from 1 to 100. The pause is sufficient to slow down the other loop, to enable us to read the numbers on the screen. If we now change line 20 to,

```
20 FOR 1 = 1 TO 50 STEP 2
```

the counter will now count on from 1 in twos (1,3,5,7, ... 49). The addition of the STEP allows us to decide on the counter increment, or how much to count on by. If the STEP word is left off, the counter assumes you want to count on in ones. Change the example by typing in the following lines,

```
15 INPUT "START,FINISH,STEP";A,B,C
20 FOR I = A TO B STEP C
70 PRINT @ 428,"AND I = ";I
```

RUN the program with various values for start, finish and step. Try some of the following values

START	FINISH	STEP
50	1	1
50	50	5
1	10	0.5
2.6	3.9	0.1
1	2	1

You will notice that you can count forwards or backwards in any step you like. The only rule is if the step is positive the start must be less than the finish, or if the step is negative the start must be larger than the finish. From the last example given above (1, 2, 1) you can see that even if you break the rule, (it is not possible to count from 1 to 2 in steps of + l) the loop will still be performed once.

Loops can be 'nested' inside another loop (the loop at line 45 is nested inside the other loop starting at line 20). You must make sure that the loops are closed in the correct order.

```
20   FOR I = 1 TO 10
30   FOR J =    2 TO 4 STEP 0.6
40   FOR K = 1 TO 0 STEP    0.1
.
.
.
.
.
.
100 NEXT K
110 NEXT J
120 NEXT I
```

Every FOR statement must have its corresponding NEXT statement, the variable after the NEXT indicating which FOR it belongs to. The loops should be closed in the opposite order to that which they were opened in. If the last three lines above were,

```
100 NEXT J
110 NEXT K
120 NEXT I
```

the program will stop and give an error message, because loop J and loop K overlap.

If all the loops end in the same place, the NEXT statements can be combined as,

```
100 NEXT K,J,I
```

but the order of the variables must still be as before.

The variables used to set up a loop, (in the example I,A,B, and C), can be used inside the loop. But you will not be allowed to alter the start, finish or step size. The counter can be changed by appearing on the left hand side of an assignment statement, but this is not a good practice.

Other branching statements, like GOTO or IF ... THEN, can be used inside a loop to leave it before it is finished. You cannot jump into the middle of a loop, however, you *must* start a loop with a FOR statement.

The following program demonstrates the use of nested loops. It simulates a digital timer. To make it really accurate you may have to adjust the delay loop at line 220. Note the use of INKEY$ to stop and start the clock at lines 90, 150 and 170.

```
10 CLS0: PRINT @ 10,"DIGITAL TIMER";
20 ' PRINT POSITIONS
```

FOR…NEXT…STEP

The FOR ... NEXT command consists of two statements,

FOR *numeric variable* = expression TO expression
STEP expression

and,

NEXT *numeric variable*

The FOR ... NEXT statements work together to control the number of times a section of program is executed. The technique is called *looping*.

The expressions are evaluated and the loop counts from the value of the first expression TO the value of the second expression with an increment given by the third expression. The current value of the counter is held in the *numeric* variable. The loop will be performed at least once, even if the range and increment are not possible. If the STEP part of the statement is omitted + 1 is assumed.

Loops can be nested inside each other, but must be in the correct order.

You may branch out of a FOR ... NEXT using GOTO, IF ... THEN or similar statement, you cannot however branch into the middle of a loop.

```
10 CLS: CLEAR: DIM L(1000)
20 INPUT"ENTER A NUMBER"; N:IF N > 1000 OR N < 2 THEN 20
30 CLS: PRINT "PRIME NUMBERS BETWEEN 2 AND"; N
40 FOR I=2 TO N:IF L(I)<0 THEN 80
50 PRINT I;
60 IF I>SQR(N)THEN 80
70 FOR J = I TO N STEP I: L(J)=   1:NEXT J
80 NEXT I
```

```
30    P = 261: Q = 196
40    ' INSTRUCTION BLOCK
50    PRINT @ 386"PRESS SPACEBAR TO STOP & START";
60    PRINT @ 424,"AND 'R' TO RESTART2;
70    ' PRINT DISPLAY AND WAIT FOR START
80    PRINT @ Q,"HOURS";:PRINT @ Q+10,"MINS";:PRINT @
      Q+22,"SECS";
90    A$ = INKEY$:IF A$<>"▽" THEN 90
100 ' START LOOP FOR HOURS MIN & SECS
110 FORH=0 TO 23:FOR M = 0 TO 59:FOR S = 0 TO 59
120 SOUND 220,1
130 ' TENTHS OF SEC LOOP
140 FOR T = 0 TO 9
150 A$ = INKEY$:IF A$<>"▽" THEN 200
160 ' CHECK FOR KEYPRESS
170 A$ = INKEY$:IF A$="R" THEN 110
180 IF A$<>"▽" THEN 170
190 ' PRINT TIME
200 PRINT @ P,H;:PRINT @ P+10,M;:PRINT @ P+19,S;
    ".";T;
210 ' TIMER ADJUSTMENT LOOP
220 FOR D=1 TO 13:NEXT D
230 NEXT T
240 ' END OF TENTHS LOOP
250 NEXT S,M,H
260 ' END OF SECS, MINS & HOURS LOOPS
270 PRINT @ 448,"STOPPED"
280 END
```

WHEELS WITH WHEELS

By now you should be aware that programs have an overall structure and are assembled from smaller blocks. Some of these blocks may be required many times in different places in the overall program. The structure of the program can often be simplified by treating these situations as *subroutines.* A subroutine, as the name implies, is a subsidiary part of a program, or a program within a program. The main feature of a *subroutine* is that when *called* it carries out its sequence of lines and then returns to the place it was cared from. To call a subroutine, you use the statement,

GOSUB *line number*

where *line number* is the number of the program line where the subroutine starts. As a subroutine must come back, each subroutine ends with the statement,

```
RETURN
```

The GOSUB behaves in a similar way to the GOTO statement. The difference being with GOTO the program branches to a line and continues from there, it does not come back unless it meets another GOTO statement. Type in the following example and RUN it,

```
10   CLS: PRINT "IN MAIN PROGRAM"
20   GOSUB 50
30   PRINT "BACK IN MAIN PROGRAM"
40   END
50   PRINT "IN FIRST SUBROUTINE"
60   GOSUB 90
70   PRINT "BACK IN FIRST SUBROUTINE"
80   RETURN
90   PRINT "IN SECOND SUBROUTINE"
100  RETURN
```

The program line number sequence is as follows:

10,20,50,60,90,100,70,80,30,40

Note how the first subroutine (lines 50   80) calls the second subroutine (lines 90,100). The END statement we have slipped in means exactly what it says. It is used to indicate where the program actually finishes. Try taking out line 40 and running the example again. You should get ? RG ERROR IN 80. This is because the program encountered a RETURN statement without being told to go to a subroutine. The program 'fell' through the bottom into the first subroutine. You must always protect subroutines and ensure the only way in is by a GOSUB statement, and the only way out is by a RETURN. You can use GOTO, IF ... THEN, and similar branching statements inside a subroutine, but they must not cause a branch to a line number outside the routine.

As with the ON ... GOTO statement, a multiple branch is also available for subroutines, and has a similar form,

ON *expression* GOSUB *list of line numbers*

Most experienced programmers keep a library of subroutines, so many new programs can be constructed from stock, so to speak. It is usually a good idea to number subroutines with large line numbers, 10000   , so that they can fit into a program without renumbering.

Many examples of subroutines will appear in following chapters. so we will not give specific examples now.

GOSUB
RETURN
ON ... GOSUB

The GOSUB command transfers program control to the beginning of a program subroutine. The RETURN transfers control back to the line following the GOSUB statement.

The GOSUB is followed by a line number, which is the first line of the subroutine.

```
GOSUB 1600
```

A subroutine must contain at least one RETURN statement.

The ON ... GOSUB command allows a multiple branch to subroutines in a similar way to the ON ... GOTO command.

ON *expression* GOSUB *list of line numbers*

lf the *expression* is negative the program will stop with an error message. If the *expression* is zero or greater than the number of items in the *list of line numbers*, the statement will be ignored and the program will continue on the next line.

```
10   CLS:INPUT"ENTER ANY TWO NUMBERS";A,B
20   INPUT"NOW ENTER A NUMBER FROM 1 TO 4";C
30   ON C GOSUB 100,200,300,400
50   PRINT C;"IS NOT BETWEEN 1 AND 4":GOTO 20
100 PRINT"ADDITION";A;"PLUS";B;"IS";A+B
110 RETURN
200 PRINT "SUBTRACTION.";A;"MINUS";B;"IS";A   B
210 RETURN
300 PRINT"MULTIPLICATION.";A;"TIMES";B;"IS";A*B
310 RETURN
400 PRINT"DIVISION";A;"DIVIDED BY";B;"IS";A/B
410 RETURN
```

CHAPTER SIX

NEW DIMENSIONS

In chapter 2 when we were discussing variable types, we said that variables came in two sizes, *simple* and *array*. Up to now we have only used *simple* variables.

If you decided to use your computer to keep an index of all your books, or records, it would not take long to run out of variables to store them in. It would also be a very difficult program to write, keeping track of all those different names! Your computer comes to the rescue with *array* variables.

LISTS AND TABLES

Array variables are especially useful for dealing with lists of items, so we could set up a list of books as follows:

1. Title 1
2. Title 2
3. Title 3

.
.
.

To refer to our books now, we could ask for number 8 on the list. In our program we give a name to this sequence of variables (the titles), and refer to a single value in the list by giving the index number. So first we must give our list a name.

An *array* variable name follows the same rules as for *simple* variables. The computer recongnises the difference between the two, because array variables are *always* followed by brackets containing the index number.

A(5)	refers to the 5th item in a numeric array (list called A)
D7$(28)	refers to the 28th item in a string array called D7$

The index number is called a *subscript*. To set up an array you have to tell the computer what it is called and what size it is. This is done with the DIM statement,

DIM *arrayname* (number). *arrayname* (number, number)

DIM stands for dimension, the statement not only names the array, but sets the maximum size it may take. The number can be any positive number, or a simple variable, provided the simple variable has already been given a value. Only set up the size you need as very large arrays take up a lot of the memory available..

```
10 DIM A(22), NA$(40)
```

The above line tells the computer to set up an array called A of length 22 and a string array called NA$ of length 40. (Actually the lengths are 23 and 41, because the index numbers start with 0).

To refer to an array element in an expression, you just include the index number, or subscript, in brackets after the name.

```
25 A(4) = 7.0
32 A(M) = B*C/A
```

Line 25 will set item 4 in array A to 7.0. Line 32 will evaluate the expression and put the result into item M of array A, (note that the A on the right hand side of line 32 is a *simple* variable called A, and has nothing to do with the *array* A on the left hand side).

Arrays can also have two dimensions, the line,

```
10 DIM T(10,5), TB$(12,4)
```

will create a numeric array, T, which will be a table with 10 rows and 5 columns. The string array, TB$, will have 12 rows and 4 columns. For instance, a teacher may wish to enter the examination scores for 25 students on 6 different subjects. An array EXAM(25,6) would do for this purpose. To refer to an element in the table you would need two subscripts.

```
25 PRINT EXAM(10,3)
```

would display the mark for the loth student on the 3rd subject. Of course, the subscripts must refer to an element which exists. Trying to use EXAM(30, 7) will cause an error because you did not set up an array of that size.

Time for an example using arrays. The following program is short, but will require a little thought on your part as to how it works. lt is used for shuffling a pack of cards. Each card is given a number from 1 to 52, in array X. Items are selected at random from X, and put into array Y. When the program ends, Y contains the numbers 1 to 52, but in random order (i.e. Shuffled). You cannot use the random number generator to produce the shuffled pack directly, because you can have only one number per card. The RND function could draw the same number more than once. Line 50 is printing out the shuffled array, note how it uses an expression as an array subscript.

DIM

The DIM command is used to dimension arrays. Arrays may be one or two dimensional and he either numeric or string. Array names follow the same rules as for simple variables.

DIM *arrayname* (n), *arrayname* (n,n)

If the maximum size fo the array does not exceed lo the DIM statement is not necessary.

To refer to an array element in an expression, the name must be followed by the *subscript* in brackets.

A(14,K) N9(B) L$(14,4)

```
10 DIM X(52), Y(52):CLS
20 FOR I = 1 TO 52: X(I) = 1:NEXT I
30 FOR I = 52 TO 1 STEP     1
40 J = RND(I): Y(I) = X(J): X(J) = X(I):NEXT i
50 FOR I = 1 TO 13: FOR J = 1 TO 4: PRINT Y(4*(I   1)+J);: NEXT
    J,I:END
```

The elements of a string array can be moved about in the same way. One of the most common applications to strings is sorting a list into alphabetical order. In the following example a list of words is sorted by the subroutine starting at line 200. Many books have been written about sorting on computers and the method used here is called the exchange sort. It is not necessarily the best method, but it is the simplest. If you are not familiar with the method, work through by hand just using the list D, B, A, E, C.

```
10   CLS: DIM W$(50)
20   INPUT "HOW MANY WORDS";N
30   CLS: PRINT"ORIGINAL"
40   FOR I = 1 TO N: PRINT I;".▽▽";
50   INPUT W$(I): NEXT I
60   GOSUB 200
70   PRINT @ 18,"SORTED"
80   FOR I = 1 TO N
90   PRINT @ 18+32*I,I;".▽▽";W$(I)
100 NEXT I: END
200 M = N
210 F = 0:FOR I = 1 TO M   1
220 IF W$(I)<=W$(I+1) THEN 240
230 T$=W$(I): W$(I) = W$(I+1): W$(I+1) = T$:F = 1
240 NEXT I: IF F = 1 THEN M = M     1:GOTO 210
250 RETURN
```

WHAT'S ITS FUNCTION

Remember RND, and how we keep calling it a *function?* Well, it is not alone. A *function*, in the computing sense, is a special subprogram which when given a set of *arguments*, *returns* a single value. A *function* in the BASIC language has the form,

Function name(*arguments*)

and can be used in an expression in the same way as other arithmetic operators (, *, /, +,   ). Functions, however, take priority over all other operators, except brackets.

The *arguments* of a function are the values to be given to the function, which then *returns* the result. Arguments may be constants, variables or expressions.

RND(10), RND(X) or RND(A*2+F)

are all acceptable arguments. Note that the argument is *always* enclosed in brackets.

Your computer supplies you with a number of functions, like RND, which are part of the BASIC language. The available 'built-in' functions can be considered as belonging to one of five different classes. We will take each class in turn, give a list of the functions, a short explanation, and an example line. They will appear in programs from now on, so we will not give example programs for each one, there are far too many.

CLASS I functions

These are numeric functions, mostly for mathematical use. They have a *numeric* argument and return a *numeric* value. Class 1 functions can only be used in numeric expressions. For those of you not familiar with trigonometric functions see Appendix D.

Function Name	Operation	Example
ABS(X)	The absolute of X	100 A = ABS(D*2 C)
ATN(X)	Arctangent of X, in *radians*. The inverse of TAN (X).	110 PRINT "ANGLE="; ATN(R3)
COS(X)	Cosine of X, where X is an angle measured in *radians*.	510 F7 = COS(X+4)
EXP(X)	Raise the base e, (natural logarithms) to the power x, (e^x). The inverse of LOG (X)	215 Q=EXP(A*A)
FIX(X)	Returns the integer part of X, (i.e. truncates all digits after the decimal point).	172 N = FIX(Z*.05)
INT(X)	Truncates if X is positive, as for FIX. If X is negative, it rounds downwards (i.e. INT 12.001 is 13)	280 P = INT(100*X)

Function Name	Operation	Example

JOYSTK(X)	Returns the current horizontal, or vertical position of the left or right joystick as follows:
	X = 0, horizontal left joystick
	X = 1, vertical left joystick
	X = 2, horizontal right joystick
	X = 3, vertical right joystick
		1040 A = JOYSTK(0): B=JOYSTK(1)

LOG(X)	Natural logarithm of X. The value of the argument must he greater than zero. The inverse of EXP (X).	617 L1 = 5.2*LOG(W4)
PEEK(X)	Returns the contents of the memory location whose address is X.	55 P = PEEK (65280)
POINT (X,Y)	Test whether a low resolution graphics cell is on or off. X must be in the range 0 63 (horizontal), Y in the range 0 31 (vertical). Returns the value 0 if cell is off 1 if in text mode 1 to 8 if on, the number of colour.	50 IF POINTS (5,A)=C THEN 210
POS(X)	Returns the current print position. The only arguments are, 0 for screen display 1 for printer.	168 IF POS (0) > 30 THEN PRINT A$
PPOINT (X,Y)	Tests high resolution graphic cell. Returns 0 if cell is off, otherwise returns colour code of cell.	115 C = PPOINT(A1,A2)

Function Name	Operation	Example
	(X in range 0 255, Y from 0 191)	
RND(X)	Returns random whole number between 1 and X. X equal zero (0) returns random number between 0 and 1.	220 PRINT @ RND(510); "*";
SGN(X)	Returns the sign of the argument X negative returns 1 X zero returns 0 X positive returns + 1	412 Y = RND (ABS (N))*SGN(N)
SIN(X)	Sine of X, where X is an angle in *radians*.	205 S=SIN(K*PI/180)
SQR(X)	Square root of argument ($\sqrt{x}$), X should not be negative. If X is negative then function returns $\sqrt{\text{ABS(X)}}$	330 C=SQR(A*A + B*B)
TAN(X)	Tangent of X, where X is an angle in radians. The inverse of ATN (X)	840 R5=B/TAN(EQ 5)

Class II functions

The class II functions have a *numeric* argument, but return a *string* value. They can only be used in string expressions.

Function Name	Operation	Example
CHR$(X)	Returns the character for the code given by X. X must be from 0 to 255. See Appendix A for list of codes.	20 M$=CHR$ (143) + CHR$ (128)

Function Name	Operation	Example
HEX$(X)	Computes the hexadecimal value for a decimal number X	42 PRINT HEX$(30)
STR$(X)	Converts a numeric expression into the string equivalent	175 A$ = STR$(12,49)

Class III functions

Class III functions are *string* functions. The arguments (there are usually a least two), are a string and a number. They all return a *string* value and therefore must be part of a string expression.

Function Name	Operation	Example
LEFT$ (X$,N)	Returns the first N characters of string X$	114 A$ = LEFT$(B$,7)
MID$ (X$,M,N)	Returns the N characters of string X$, starting at position M. If N is omitted the entire string to the right of M is returned. M must be greater than 0.	760 K$ = MID$(W$,I,4)
RIGHT$ (X$,N)	Returns the last N characters of string X$	340 T$ = RIGHT$(Q$,B+7)
STRING$ (N,C)	Returns a string of length N consisting of the character defined by C. The argument C can be, either a number, (the ASCII code for the character) or, the character itself, enclosed in quotes.	400 A$ = STRING$(5,67) 410 PRINT STRING$(32,"*")

Class IV functions

These are a mixed function similar to class II. They have a *string* argument and return a *numeric* value, and so only appear in numeric expressions.

Function Name	Operation	Example
ASC(X$)	Returns the ASCII code number of the first character of the string argument.	715 P = ASC(F$) 64
INSTR (P,S$,T$)	Searches the string S$ for the target string T$, starting from position P of the search string. Return 0 if not found, otherwise the position of the target string.	212 F = INSTR(N,X$,"AB")
LEN(X$)	Returns the length of the string X$. All characters are counted including spaces. If the string is empty returns 0.	845 N = LEN(N$)
VAL(X$)	Converts the character representation of figures into a number. If the string starts with alphabetic character it returns 0	92 Z = VAL(AB$)

Class V functions

Class V are system functions. They have no arguments

Function Name	Operation	Example
INKEY$	Checks the keyboard and returns key being pressed any). Returns a string so must be used in a string expression.	146 P3$ = INKEY$

Function Name	Operation	Example
MEM	Computes the hexadecimal value for a decimal number X	PRINT MEM
TIMER	Returns contents of the timer, a value in the range 0 65535. To reset use TIMER	62 T1 = TIMER T 65 TIMER = 0

D.I.Y FUNCTIONS

Apart from the functions supplied hy the system, it is possible for you to create up to another 26 numeric functions of your own. The form of the statement is

DEF FN *letter*(*dummy variable*) = *formula*

The *letter* is any letter from A to Z. The *dummy variable* is a letter, which will be replaced by the function argument when the function is called. The *formula* is a BASIC expression written in terms of the *dummy variable* and/or other variables. Other functions, both built in or user-defined, may be present in the expression, but a function *cannot* call itself.

The equation $y = ((x \quad 3)^{2} + (x \quad 4)^{4})/x^{3}$, will translate directly into a defined function as,

```
25 DEF FN Y(X) = ((X    3)    2 + (X    4)    4) X    3
```

The X here is the *dummy variable*, not a variable name. When the function is called later in the program it will be replaced by the argument. To use the function, just include it in an expression in the same way as a 'built-in' function.

```
150 Y(I) = FN Y(X) + FN Y(W)
```

Functions can also be used to supply 'service routines', (commonly used operations). You may have noticed that the numbers output to your screen as results, tend to be a bit untidy. The computer is trying to be helpful by printing the number to the maximum accuracy. Sometimes this accuracy is not necessary and at other times a nuisance, printing cash amounts for instance. The following function can be used to print to the required number of decimal places, (D)

```
10 DEF FN D(X) = INT(X * 10    D + 0.5)/10    D
```

Note that the X is a dummy variable, the D is not a dummy variable, the value for D must be supplied from outside the function, possibly by an INPUT statement. To use the function.

`205 PRINT FN D(A)` etc.

As all trignometric functions require the argument to be in radians, a function to convert degrees to radians could be useful.

```
10 DEF FN R(X) = X/57.295779
```

will do this. A more accurate result can be obtained by using instead,

```
10 DEF FN R(X) = X * ATN(1.0)/45
```

The constant PI can be created by,

```
20 DEF FN P(X) = 4.0 * ATN(1.0)
```

Note that in this case the dummy variable has no effect at all, it is only there because an argument is required.

As you cannot call a function that has not yet been defined, it is a good practice to place your function definitions at the start of the program.

ALTERNATIVES TO INPUT

The only way we have been able to get values into variables has been by using an INPUT statement. This is very convenient, but you may have noticed that the INPUT statement will not accept certain characters. If you start a string with spaces they are lost, and if you type a comma you lose everything after it. There is an alternative, the LINE INPUT statement,

LINE INPUT "*prompt*"; string variable

The LINE INPUT behaves in a similar way to INPUT, except that it will accept everything, including spaces and commas. The *prompt* is the same as for INPUT, and *string variables* can be any string variable. You can only have one variable in each LINE INPUT statement.

```
25 LINE INPUT "TYPE IN A LINE OF TEXT";L$
```

Often in a program it is necessary to set up a number of constants before the program really starts work. You could, of course, enter these every time the

DEF FN

The DEF FN command is used to define a user numeric function

DEF FN *name(dummy variable) = formula*

The *name* may be any letter from A to Z.

The *dummy variable* may be any letter, it is replaced by the argument when the function is used. Only one *dummy variable* may be used.

The *formula* describes the operation in terms of the dummy variable and/or other variables.

User defined functions must be contained in one program line. A defined function may use other functions, (either defined or 'built in') in the formula, but must not call itself.

A function must be defined before it is used, it should therefore appear at the beginning of a program.

Other mathematical functions can be defined as user functions as follows:

```
10 DEF FN S(X) = 1/COS(X):' SECANT
20 DEF FN I(X) =    ATN(X/SQR(   X*X+1))+1.5708
30 DEF FN H(X) =    EXP(X)/(EXP(X)+EXP(   X))*2+1
40 DEF FN M(A) = INT((A/B   INT(A/B))*B+0.5)*SGN(A/B)
50 B = 8:PRINT FN M(13)
```

LINE INPUT

The LINE INPUT command enters an entire line into a string variable, including commas and leading spaces not accepted by the INPUT command

LINE INPUT *"prompt"*; *string variable*

The *prompt* is any prompt message included in quotes. lt is optional and if included must be separated from the *string variable* by the semi   colon (;). The *string variable* may be any string variable. only one variable may appear in the LINE INPUT statement. The maximum length of the line stored by a LINE INPUT command is 255 characters.

```
10 CLEAR 500:CLS
20 LINE INPUT"ENTER YOUR FULL NAME";N$
30 LINE INPUT"AND ADDRESS";A$
```

program is run with INPUT statements. There is, however, a more convenient method using the READ and DATA statements. They are always used together and have the form,

READ *list of variables*
DATA *list of values*

The READ statement behaves in the same way as the INPUT statement, except instead of halting the program and waiting for you to enter a value, it
looks for the value in a DATA statement which is part of the program. The DATA statement may be included anywhere in the program. If there is more than one DATA statement, the READ starts with the lowest numbered statement
and works through in order.

```
10 DATA 1,2,3,4,5
20 FOR I = 1 TO 3
30 READ  A: PRINT A: NEXT  I
40 READ  D,G: PRINT  D,G
```

The above example will read the first item in the DATA list (1), print it out, read the second (2), and so on. As each item is read the pointer moves to the next item. Line 40 will read the last two items. If you now add the line,

```
50 READ X: PRINT X
```

and run the program again, you will get ?OD ERROR IN 50. This means the READ has no more items in the DATA list and is therefore out of data (OD). Add another line,

```
45 RESTORE
```

When you run the program this time it is alright, and X has the value 1. The RESTORE statement sets the pointer back to the beginning of the first DATA statement. Strings can also be included in DATA statements. You must take care that any mixture of variables in a READ statement are matched in order by values of the same type in the DATA list.

If you have written a program using a lot of strings, you may already have had an ? OS ERROR. This means you have run out of memory reserved for string storage. To allocate memory for strings use the CLEAR statement.

```
10 CLEAR 1000
```

This statements reserves 1000 bytes of memory to store strings. As CLEAR also sets *all* variables to zero, only use at the beginning of the program.

READ
DATA
RESTORE

The READ command reads the next item in a DATA line and assigns to the specified variable in the list.

READ *list of variable names*

The DATA line stores data within the program and may be a numbered line anywhere in the program.

DATA *list of values*

Both string and numeric variables may be used in READ and DATA provided the sequence is correct. A string must be assigned to a string variable, etc.

The RESTORE command sets the data pointer back to the first item in the lowest numbered DATA line.

RESTORE

```
10 CLS: PRINT: PRINT: PRINT
20 READ A,B: IF A =    9999 THEN RESTORE:GOTO 2O
30 PRINT A;" ▽+▽5   ▽IS▽";:INPUT C
40 IF C = B THEN PRINT"CORRECT" ELSE PRINT "WRONG"
50 FOR D = 1 TO 600: NEXT D: GOTO 10
60 DATA 8,13,12,17,5,10,27,32,14,19,3,8
70 DATA 7,12,6,11,1,6,   9990,9999
```

CLEAR

The CLEAR command erases all variables and reserves space for string storage.

```
CLEAR 500
```

will reserve 500 bytes of storage for string variables.

The CLEAR command can also be used to set the highest BASIC address in memory to reserve space for machine language routines.

```
CLEAR 200,14000
```

will reserve 200 bytes for string storage and set the highest address for BASIC to 14000. Machine language routines may now be stored from 14001 onwards.

If CLEAR is not used 200 bytes of string space are automatically reserved.

PAUSE FOR REFLECTION

The contents of this chapter, together with those of Chapters 1, 2, 3 and 5, constitute the core of the BASIC language. Though there are still more statements to come, they are to some extent icing on the cake.

All the material we have covered so far will be used frequently in the following chapters, as they are essential parts of any program. While you may be eager to get into drawing pictures with your computer, some time spent at this point ensuring you understand exactly what is going on, will make using graphics so much easier.

Check back through the examples, try to adapt them to suit your own ideas.

We finish this section with two more examples. The first extends the shuffling example of the last chapter. The program 'deals' a hand of cards. Note the following points,

a) the shuffle now appears as a subroutine at line 90,
b) the use of READ and DATA to set up the arrays at the start,
c) lines 130 and 140 to find the suit and which card within the suit.

```
10   DIM X(52), PACK(52), CARD$(13), SUIT$(3)
20   FOR I = 1 TO 3: READ SUIT$(I): NEXT I
30   DATA SPADES, DIAMONDS, CLUBS, HEARTS
40   FOR 1 = 1 TO 13: READ CARDS(I): NEXT 1
50   DATA ACE, TWO, TREE, FOUR, FIVE,SIX, SEVEN
60   DATA EIGHT, NINE,THEN, JACK, QUEEN, KING
70   CLS: INPUT"HOW MANY CARDS TO DEAL";N
80   GOSUB 190
90   ST = 1
100 EN = ST + N      1:IF EN>52 THEN GOTO 80
110 CLS:PRINT @ 10,"YOUR HAND":PRINT:PRINT
120 FOR 1 = ST TO EN
130 S = INT((PACK(I)   1)/13)
140 C = PACK(I)      S*13
150 PRINT TAB(8);CARD$(C);"OF";SUITS(S)
160 NEXT I:ST = ST + N
170 PRINT @ 448,"ANOTHER HAND. YES OR NO";:INPUT A$
180 IF A$ = "YES" THEN 100 ELSE END
190 FOR I9 = 1 TO 52: X(I9) = I9:NEXT I9
200 FOR I9 = 52 TO 1 STEP    1
210 J9 = RND(I9):PACK(I9) = X(J9):X(J9) = X(I9)
220 NEXT I9: RETURN
```

The second example uses nearly all the available string functions. The program checks through the entered text and reports the number of occurrences of each letter. This type of program is often used for deciphering coded messages. With little effort it can be adapted to search for words or a sequence of characters.

```
10   CLEAR 100,0: CLS: READ A$
20   DATA ABCDEFGHIJKLNlNOPQRSTUVWXYZ
30   PRINT "TYPE IN ANY LINE OF TEXT":PRINT
40   LINE INPUT L$
50   FOR I = 1 TO LEN (A$): CLS
60   T$ = MID$(A$,I,1): C = 0: P = 1: P$ = L$
70   F = INSTR(P,L$,T$)
80   IF F>0 THEN C=C+1 ELSE 140
90   P$ = LEFT$(P$,F   1) + STRING$(LEN(T$),CHR$(128))
100 IF F>LEN(L$) THEN 120
110 P$ = P$ + RIGT$(L$,LEN(L$)   F)
120 P = F + LEN(T$)
130 IF P<= LEN(L$)     LEN(T$) + 1 THEN 70
140 PRINT P$
150 PRINT @ 354,"FOUND";C;"OCCURRENCES OF";T$
160 PRINT @ 416,"PRESS SPACEBAR TO CONTINUE, N TO STOP"
170 Z$ = INKEY$:IF Z$ = "" THEN 170
180 IF Z$ = "N" THEN 200
190 NEXT I
200 CLS:END
```

CHAPTER SEVEN

GETTING THE POINT ACROSS

When your computer displays anything on a TV screen, what it is doing is setting points in the TV tube either on or off, to build up the image. If the point is set on it appears as a coloured dot, if off as black. All the letters we have been printing are made up from these dots of light. The size of the dot you can control determines the *resolution* you are working in. A large area is a *low resolution*, a small dot area is a *high resolution*, (because the smaller the dot, the higher the number of points available on the screen).

Your computer has the ability to work in five different resolutions, ranging from 512 points on the screen up to 49152 points. This gives you a large amount of flexibility in the amount of detail you can put into your pictures. We will start by creating images in the lowest resolution and working upwards. The methods used to put drawings and movement onto the screen are much the same, irrespective of the resolution you are working in.

PRINTING PICTURES

You remember from chapter 3 when we introduced the PRINT @ statement, we described how the screen was divided into a 16 x 32 grid. This allowed us to print a character anywhere on the screen by giving the appropriate position. Using the CHR$ function (see chapter 6 for functions), we can generate special graphic characters. The following program will display all the characters available from CHR$.

```
10 FOR I = 1 TO 255:CLS 0
20 PRINT @ 100,"CHR$(";I;")";
30 PRINT @ 120,CHR$(I)
40 FOR D = 1 TO 600: NEXT D,I:CLS
```

The numbers from 1 to 31 are used for control characters, so not much happens. From 32 to 127, the keyboard characters appear. The codes from 128 to 255 are the special graphics characters. (A complete list of the available characters is given in Appendix A). These graphics characters are patterns of colour blocks which can be assembled into simple shapes. The simplest pattern is a rectangle of colour. For instance, CHR$(143) gives a rectangle of green, which is colour 1. Add 16, and CHR$ (159) gives a rectangle of yellow, colour 2, and so on. There are sixteen patterns from CHR$ (128) to CHR$ (143), these are made up from green and black. To obtain the same pattern, but in a different colour, just add the appropriate number of 16's to the code.

+16 yellow	+32 blue	+48 red
+64 buff	+80 cyan	+96 magenta
	+112 orange	

The following program shows the effect of increasing the code by sixteen (it can also be used for adjusting the colour balance on your TV set, use C = 143).

```
10 CLS0: INPUT"ENTER CODE FROM 128 TO 143";C
20 FOR I=1 TO 14: FOR J = C TO 255 STEP 16
30 FOR K=1 TO 4: PRINT CHR$(J);: NEXT K,J,I
40 GOTO 40
```

As you can see from the above, the CHR$ characters can be printed directly onto the screen. But as they are characters they can also be placed in string variables. This is more convenient, as they can be manipulated much more easily.

We will now start to construct a picture. We will draw a castle one, because it is a simple shape and two, because it shows how starting from a simple base, you can add increasing detail. Enter and run each section as we give it, so you can see the steps the picture goes through.

First we need to build a wall across the screen, so it will be 32 blocks wide, and we will make it 6 blocks high.

```
10 CLEAR 500: CLS0
20 FOR I=1 TO 6:FOR J=1 TO 32
30 WALL$ = WALL$ + CHR$(207): NEXT J,I
40 PRINT @ 256, WALL$
200 GOTO 200
```

The first line reserves space for the strings we are going to use. Lines 20 and 30 'build' the wall out of buff coloured blocks, CHR$ (207), and store it in a string called WALL$. Line 40 prints WALL$ which appears on the screen as a solid block of colour. The last line (200) is just to hold the picture on the screen.

Next we add the battlements, this is done with alternative blocks of buff and black. We only need one row this time.

```
50 FOR I = 1 TO 16: B$ = B$ + CHRS(128) + CHRS(207): NEXT I
60 PRINT @ 224,B$;
```

Line 50 constructs the battlements and 60 prints them on top of the wall.

Now we need a tower. The tower is built from the same material as the wall, so let us take some bricks from WALL$.

```
70 P = 11
80 FOR I = 1 TO 3:PRINT @ 128+32*I+P,LEFT$(WALL$,10);
    : NEXT I
```

Line 80 takes 10 bricks from WALL$ and builds 3 rows in the middle of the wall. The value of P places the tower in the middle. If you want to try the tower in another place, change P. The tower is 10 blocks wide, so P can be any number from 0 to 22.

We now do the same with the battlements for the tower.

```
90 PRINT @ 128+P,LEFT$(B$,10)
```

That completes our basic, (BASIC?) castle. We can add arrow slits by using a different character and overprinting.

```
100 FOR I = 2 TO 32 STEP 4:PRINT @ 288+I,CHR$(206);: NEXT I
```

We also need a gate, so we print black blocks in the appropriate place.

```
110 G$ = CHR$(128) + CHR$(128) + CHR$(128)
120 FOR I=353 TO 417 STEP 32:PRINT @ I+P+5,G$;:NEXT I
```

The gate is stored in G$ and printed in line 120 (using the P in the PRINT @ expression to keep the gate in line with the tower).

A castle with an ever open gate is not much use, so now we need a portcullis, using another character, (142+112) we can construct something similar.

```
130 P$ = CHR$(254) + CHR$(254) + CHR$(254)
140 FOR I = 353 TO 417 STEP 32:PRINT @ I+P+5, P$;
150 FOR K = 1 TO 300: NEXT K,I
```

Line 130 constructs an orange (!) portcullis which is printed from the top down in line 140. The delay loop K causing it to be 'lowered' slowly.

We will leave the castle now, you might like to add further details, like a blue moat and a flag on the tower, and so on.

MOVING PICTURES

In the castle example, lines 140 and 150 showed us how to get some movement into the picture, by printing parts in succession. This type of movement is limited, largely to opening and closing doors. A much better method is to print the image in full, then blank it out and print it again in a slightly different position. As you are constantly redrawing the figure; the drawing part should be a subroutine. But first the figure, we will construct it in a 3 x 4 block. The top 'line' will be the head, the next the body, and the last the legs.

```
10 CLEAR 500: CLS0
20 M1$ = CHR$(128) + CHR$(193) + CHR$(194) + CHR$(128)
30 M2$ = CHR$(196) + CHR$(207) + CHR$(207) + CHR$(200)
40 M3$ = CHR$(128) + CHR$(202) + CHR$(197) + CHR$(128)
```

The figure is now stored in three string variables M1$, M2$ and M3$. Now the subroutine to print the strings in the correct order.

```
500 P = 32*Y+X
510 PRINT @ P,M1$;:PRINT @ P+32,M2$
520 PRINT @ P+64,M3$;:RETURN
```

This wall print the figure as three lines directly under each other, starting at a point decided by X and Y. (Remember the X,Y grid? X is 0 to 31 horizontally, and Y, 0 to 15 vertically). You will have to enter all the lines up to line 160 below before you will be able to run the program.

Now to move the character we need to change the print position, that is to change X or Y. This can be done from the keyboard. We wall use INKEY$ to read the keyboard, and the obvious characters to use are the arrow keys. These keys have codes as well, just as the letters do.

[   ] CHR$(8)
[   ] CHR$(9)
[   ] CHR$(10)
[   ] CHR$(94)

We shall use X to hold the horizontal position of the figure, and Y to hold the vertical position. If the background arrow key [   ], is pressed, we want to move to the left, so take one off the current X value. But we must make sure that we do not go off the edge of the screen.

```
900 GOSUB 500
100 A$ = INKEY$:IF A$ ="" THEN 100
120 IF A$ = CHR$(8) THEN X = X   1: IF X < 0 THEN X = 0
```

Line 100 reads the keyboard until a key is pressed. If it is [   ] then line 120 takes one˜off X, checks to see if we are off the screen, and if we are, then stops the movement at the left hand side. To move right, up and down the pattern will be similar.

```
130 IF A$ = CHR$(9) THEN X = X+1:IF X>28 THEN X=28
140 IF A$ = CHR$(94) THEN Y = Y   1:IF Y<0 THEN Y=0
150 IF A$ = CHR$(10) THEN Y = Y+1:IF Y>13 THEN Y = 13
160 GOSUB 500: GOTO 100
```

In line 130 we set the maximum permissible value of X to be 28. It cannot be greater than 31, anyway, and remember our figure is four blocks wide. The same applies to Y, we must leave room to print the three lines. Line 160 goes to the print subroutine, then back to check for another keypress. If you run the program now, you should be able to move the figure around on the screen, but it makes a mess because we are not removing the figure from the old position. To do this we need a blanking string and a subroutine to print it.

```
50 BL$ = CHR$(128) + CHR$(128) + CHR$(128) + CHR$(128)
```

```
600 P = 32*Y+X: PRINT @ P,BL$
610 PRINT @ P+32,BL$;:PRINT @ P+64,BL$
620 RETURN
```

The new subroutine (600) does exactly the same as the other, except this time prints a block of black squares. All we need to do now is to erase the figure just before moving it.

```
110 GOSUB 600
```

You should now be able to move the figure anywhere on the screen. In its current form this program is just an example, but moving figures like this can be incorporated into games and junior educational programs.

A NEW RESOLUTION

We now move to the next level of resolution. This has a 32 x 64 grid, which gives 2048 points on the screen. This level and the previous 16 x 32 screen are the low resolution screens, and can be used together, if wanted. To switch on, or off, the points on this screen, there are two commands

SET(X,Y,C) and RESET(X,Y)

The SET command switches on the point X,Y in the colour C. The X, (from 0 to 63) and the Y, (from 0 to 31), are the horizontal and vertical positions as before. The C is a number from 0 to 8 representing the number of the colour you want the dot to be.

The RESET command switches off the point X,Y. Using these commands movement can be suggested by switching on and off in turn. Try the following program.

```
10 CLS0:X1 = 0:Y1 = 0:XI = 2:YI = 2
20 X2 = X1 + XI: IF X2>63 OR X2<0 THEN XI  =     XI:
    SOUND 180,1: GOTO 20
30 Y2 = Y1 + YI:IF Y2>31 OR Y2<0 THEN YI  =     YI:
    SOUND 180,1: GOTO 30
40 SET(X2,Y2,8): RESET(X1,Y1): X1 = X2:Y1 = Y2: GOTO 20
```

You can see what it does, how does it do it? Starting with a point X1, Y1 the program increases X1 by a small amount XI and Y1 by YI, to create a new point X2,Y2. The new point is switched on and the old one (X1,Y1) off in line 40. The X2,Y2 point becomes the old point and the program goes back to line 20 to create another X2,Y2. This moves the 'ball' across the screen. When the 'ball' reaches the edge of the screen, the sign of the increment is changed. This means that X (for instance) now starts to decrease, causing the direction to change. This causes the 'bounce' off the edges. If you change the size of the increment, (XI and YI in line 10) you can make the 'ball' move at different speeds. This type of program is the basis of most computer ball games, but these are usually written in machine language, not BASIC.

Another use for moving points is in shooting type games. These sort of games require movement over the screen, and the ability to 'fire' a weapon. We could use the arrow keys as before, but a much better method is to use the *joysticks.*

The joysticks plug into the sockets on the side of your computer, and allow much finer control over movement than the arrow keys. The joystick position is read by the function JOYSTK. JOYSTK(0) returns the horizontal position of the left joystick, and JOYSTK(1) the vertical position. JOYSTK(2) and JOYSTK(3) do the same for the right joystick. As the value returned by the function in each case is between 0 and 63 it will be necessary to scale the value to fit the resolution of the screen you are working in.

SET

The SET command is used to set a specified point on the low resolution screen to a specified colour

SET(*x*,*y*,*c*)

x,*y* are the co-ordinates of the screen point. X must be in the range 0 to 63, and Y in the range 0 to 31.

c is the colour code of the desired colour. It must be a number between 0 and 8.

```
10 CLS0:SET(5,27,8): SET(6,27,8)
20 FOR X=0 TO 6: FOR Y=28 TO 30
30 SET(X,Y,8): NEXT Y,X
40 FOR X = 7 TO 63: FOR D = 1 TO 200: NEXT D
50 FOR Y = 27 TO 30: IF Y = 27 THEN RESET(X   2,Y)
60 SET(X,Y,8): RESET(X   7,Y): NEXT Y,X
70 GOTO 70
```

RESET

The RESET command is used to erase a point switched on by the SET command. It is used in low resolution graphics mode.

RESET (*x*,*y*)

X,Y are the co-ordinates of the point to be switched off. X must be from 0 to 63, and Y from 0 to 31.

The point is set to the background colour, causing it to be 'erased'. See SET for an example.

```
10 CLS0:FOR I=0 TO 3
20 PRINT @ 74+32*I,"JOYSTK(";I;")▽▽";JOYSTK(I);
30 NEXT I: FOR D=1 TO 400:NEXT D: GOTO 10
```

Run the program above and move the joysticks. You wall see the values change with the joystick position. You can also use the button on the joystick. Add the line.

```
25 P = PEEK(65280):PRINT @ 202,"BUTTON VALUE▽ ▽";P
```

The PEEK function tells the computer to look at a specified part of its memory. The memory address 65280 contains the result of checking the button. It will be either 127 or 255 at the moment. If you press the *left* button it wall change to 125 or 253, press the *right* and it will change to 126 or 254. (If both are pressed together, the number wall be 124 or 252).

So let us start work on a game program   a battle between two ships in space. We can use the joysticks to move the ships and the button to fire the weapon.

First the ships; we will use a similar method to that used to construct the figure in the last example. Each ship will he a 2 x 3 block, one yellow, the other blue, stored in arrays S$ and S2$.

```
10 CLEAR 500:FOR I=0 TO 5:READ S(I):NEXT I
20 DATA 128,131,128,134,140,137
30 FOR Y=0 TO 1:C = (Y+1)*16
40 S$(Y) = CHR$(S(0) + C) + CHR$(S(1) + C) + CHR$(S(2) + C)
50 S2$(Y) = CHR$(S(3) + C) + CHR$(S(4) + C) + CHR$(S(5) + C)
60 NEXT  Y
```

If you do not like the shape of the ships, then design your own and change the data in line 20.

Next we must read the joysticks, check we are still on the screen, and work out where to print the ships.

```
70   FOR Y = 0 TO 1: A(Y) = JOYSTK(Y*2)
80   B(Y) = INT(JOYSTK(1+Y*2)/2)
85   IF A(Y)>58 THEN A(Y) = 58
90   IF A(Y)<2 THEN A(Y) = 2
100 IF B(Y)>27 THEN B(Y) = 27
110 L(Y) = INT(B(Y)/2*32 + INT(A(Y)/2): NEXT Y
```

The joystick positions are read in turn by line 70. The limits are set in lines

80   100 (remember the ship size). The final result is then converted into a value for the PRINT @ command. (This is an example of mixing the two low resolution screens   the joysticks work in one, the PRINT @ command in the other).

We now need to print the ships and return to see if the joysticks have been moved.

```
120 CLS0:FOR Y=0 TO 1:PRINT @ 0,Z(0);:PRINT @ 26,Z(1);
130 PRINT @ L(Y),S$(Y);:PRINT @ L(Y)+32,S2$(Y);:NEXT Y

170 A$ = INKEY$:IF A$ = "" THEN 70
180 CLS:END
```

Line 120 also prints the score, (but we haven't done that yet). To end the game, press any key, otherwise go to 70 and read the joysticks again.

Run the program so far and check that the ships will move anywhere on the screen.

The next step is to fire the guns and display the 'plasma bolt' This is the awkward part, because we have to read the joystick buttons and decide who is firing. Also as the ships are able to move anywhere on the screen we have to know the direction of the bolt. To keep it simple, we will only allow the bolt to move from the ship firing, along a horizontal line towards the target ship. It will travel at the vertical height of the
firing ship.

```
140 P = PEEK(65280)
150 IF P=125 OR P=253 THEN F=0:T=1:GOSUB 200
160 IF P=126 OR P=254 THEN F=1:T=0:GOSUB 200
```

These lines read the buttons and decide which ship is firing. The bolt will be displayed by subroutine 200

```
200 V1 = B(F):H1 = A(F):H2 = A(T):ST = 1
210 IF H1>H2 THEN ST =    1
220 FOR H=H1+ST*5 TO H2+2 STEP ST

240 SET(H,V1,4): SOUND 200,1: RESET(H   2*ST,V1)
250 NEXT H: RETURN
```

Note the switch of the step in line 210, if the left and right positions are reversed. The movement of the bolt is done in line 240

All that is left now is to check if you have made a 'hit' If so, make the appropriate noises and record the score.

The function POINT is used to check for a hit. The form is POINT (X,Y), where X,Y is the point you want to check. The function returns 0 if the point is off, and the number of the colour code, if on.

As the screen is black, and we are firing in the correct direction (we hope), we only have to know whether any coloured point in the line of fire is on. If we add the following line to our subroutine,

```
230 IF POINT(H,V1)>0 THEN GOSUB 300: RETURN
```

then if a 'hit' takes place it will go to subroutine 300, (where we keep score etc.), when it comes back, there is no point in firing the bolt any further so we leave the subroutine and start again.

```
300 Z(F) = Z(F) + 1
310 FOR K = 1 TO 15: I = RND(5)   2:J = RNI)(4)   2
320 SET(H+I*ST,V1+J,8): SOUND(RND(95)),1
330 NEXT K: RETURN
```

This subroutine keeps the score, draws and sounds the 'explosion'.

Though only 28 lines long, this program is sufficient to produce a game involving considerable movement. We leave it to you as an exercise to develop refinements that bring it to the arcade level.

This has been a long and involved chapter, but it contains most of the elements needed for graphics on your computer, whatever level of resolution you may be using.

CHAPTER EIGHT

MOVING TO A HIGHER PLACE

We now move to the high resolution screens which are completely separate from the low resolution screens. The two low resolution screens can be used together, and are displayed on what is known as the 'text screen'. The high resolution screens cannot be mixed with the text screen. You may switch from one to the other, but cannot write text onto the high resolution screen, or draw high resolution graphics on the text screen.

When anything is drawn in high resolution the Computer writes the instructions on how to display the information to a special part of its memory called the 'video RAM'. The video RAM is then read to the TV and converted into pictures. A number of 'pages' are reserved in the video RAM for this purpose, normally four. As the amount of detail you are using increases, so do the number of instructions required to display the result. More instructions need more room and so you have to reserve more pages. This is done with the PCLEAR command, followed by the number of pages you want to reserve, (up to a maximum of 8),

```
PCLEAR 8
```

As each page takes up 1536 memory locations, only reserve what is actually needed. The amount of available memory is fixed, so the more you assign for graphics pages, the less is available for program space. PCLEAR behaves in a similar way to CLEAR and should be used at the beginning of a program.

IN THE MODE

The amount of space you need to reserve is dependant upon the level of resolution you want to use. One disadvantage of the increased resolution is that it is not possible to use the full range of colours available in low resolution. The available colours and the resolution are determined by the mode you are working in. The mode is set with the PMODE command,

PMODE *mode, starpage*

where mode is a number from 0 to 4, and *startpage* is the 'page' in video RAM you wish to start writing to. As before the screens are divided into grids. This time, however, it is only necessary to remember one size (256 x 192). Even though the resolution changes with different modes, you still refer to points

PCLEAR

PCLEAR is used to reserve graphics pages in the high resolution modes.

PCLEAR *n*

N must be a number between 1 and 8. If the PCLEAR statement is omitted

PCLEAR 4 is the default.

As each graphics page requires 1536 bytes of memory only reserve what is needed.

PMODE NUMBER	GRID SIZE	POINT SIZE	PAGES USED	COLOUR SET AVAILABLE	
				SCREEN 1,0	SCREEN 1,1
0	128 x 96	■■ ■■	1	1 Black(0), Green(1)	Black(0), Buff(5)
1	128 x 96	■■ ■■	2	Green(1), Yellow(2) Blue(3), Red(4)	Buff(5),Cyan(6) Magenta(7),Orange(8)
2	192 x 128	■■	2	1 Black(0), Green(1)	Black(0), Buff(5)
3	192 x 128	■■	4	Green(1), Yellow(2) Blue(3), Red(4)	Buff(5),Cyan(6) Magenta(7),Orange(8)
4	256 x 192	■	4	1 Black(0), Green(1)	Black(0), Buff(5)

SCREEN

The SCREEN command is used to switch the display between the graphics and text modes.

SCREEN *type, colour set*

type is either 0 for text and low resolution graphics
or 1 for high resolution graphics

colour set is either 0 or 1. The colour set for the text screen is 0, black on green, or 1, black on orange. For the high resolution graphics the *colour* set available depends on the working mode as follows:

PMODE	SCREEN 1,0	SCREEN 1,1
0	Black, Green	Black, Buff
1	Green, Yellow	Buff, Green
	Blue, Red	Magenta, Orange
2	Black, Green	Black, Buff
3	Green, Yellow	Buff , Cyan
	Blue, Red	Magenta, Orange
4	Black, Green	Black, Buff

on the screen by using the 256 x 192 grid. The difference is in the size of the point that is drawn. The mode selected also decides which colours you can use. Each mode has two colour sets available. The colour set is selected with the SCREEN command, which also selects the screen type.

SCREEN *type, colour set*

Type is 0 for the text screen and 1 for the high resolution screen. The *colour set* is also either 0 or 1. The default, which we have been using up to this point, is SCREEN 0, 0. This sets the text screen with the black on green colour set. (It is possible to use SCREEN 0,1, which gives black text on an orange background, but every time the computer prints it will revert to black on green). To display the high resolution screen you have to set *type* to 1. The table opposite gives available modes and colour sets.

As you can see from the table, the resolution and colour set are closely related. You will also notice that as the mode increased from 0 to 4, the number of pages required also increases.

So to display one screen of graphics PMODEO only needs one page of memory, while PMODE 3 and PMODE 4 require four pages. When the colour set has been selected, the computer chooses the lowest numbered colour from the set as the background colour. The highest numbered colour in the set is used as the foreground colour. For instance, with PMODE 3 and SCREEN 1,0 set, the computer will draw in red on a green background. You can change the foreground and background colours with the COLOR command,

COLOR *foreground, background*

where foreground and background are the required colour codes from the set for that mode.

FAMILIAR FRIENDS

You will remember from the low resolution screens the commands CLS, SET, RESET and POINT. Their high resolution equivalents are also available. They are called PCLS, PSET, PRESET and PPOINT to indicate their new status. Their job is the same as before, PCLS clears the high resolution screen, and if followed by a colour code will set the background to that colour. PSET switches on a point and PRESET switches it off. PPOINT tests whether the point is on or off. The following example works through each available mode and colour set in turn. It sets dots in a random colour on the screen, the dots should be in a rectangular grid. The blank places in the grid are caused by the random colour being the same as the background, or not available in the colour set.

```
10 FOR P = 0 TO 4: PMODE P,1
20 FOR S = 0 TO 1: SCREEN 1,S
30 PCLS:FOR I = 50 TO 150 STEP 20
40 FOR J = 50 TO 150 STEP 20
50 C = RND(8): PSET(I,J,C): NEXT I,I
60 FOR D = 1 TO 1000:NEXT D,S,P
```

Look closely at the size of the dot, this is the resolution available in that mode.

DRAW THE LINE, SOMEONE

So we can put dots on the screen, what next? The most obvious thing to do with two dots is to join them with a line. Fortunately there is a command to do just that, LINE. Delete lines 40 and 50 from the last example and change line 30 to read,

```
30 PCLS:LINE(10,180)    (245,10),PSET,BF
```

and run the program. A line is drawn across the screen from bottom left to top right. The statement means draw a line from the start point, (10, 180) to the finish point (245,10) in the foreground colour (PSET). If you change PSET to PRESET the line will be drawn in the background colour. Drawing in the background means it cannot be seen, it can aiso be used for erasing previously drawn line. The PSET and PRESET are essential parts of the LINE command, and here nothing to do with the commands to switch points on or off.

It is not always necessary to specify the start point of LINE. With no start point the line will begin at the most recent end point. (If the LINE statement has not yet been used in the program, the last end point is taken to be 128,96, screen centre). Add a further line to the current example.

```
40 LINE    (130,180),PSET
```

A line is now drawn from the last end point (245,10) to a point at the bottom of the screen (130,180).

COLOR

The COLOR command is used to change the default settings of the foreground and background colours in the high resolution graphics modes.

COLOR *foreground, background*

Both foreground and background are numbers between 0 and 8, representing the colour code. Both colours must be in the available colour set for the current mode

PCLS

The PCLS command is used to clear the screen to a given background colour in the high resolution mode.

PCLS c

c is the colour code of the background required. It must be one of the available colour set for the working mode. If the colour is not available or *c* is omitted the default background colour is used.

See the box for CLS for the colour codes.

PSET

The high resolution version of the SET command

PSET (*x*,*y*,*c*)

switches on the point (*x*,*y*) to the colour *c*, *x* must be in the range 0 to 255, and *y* in the range 0 to 191.

c is the colour code 0 to 8 and must be one of the available colour set.

PRESET

The high resolution version of the RESET command.

PRESET (*x*,*y*)

switches the point (*x*,*y*) off, (sets it to the background colour). *x* must be from 0 to 255, y from 0 to 191.

To draw a square or rectangle, you could use four lines, but there is an extension to the LINE command which takes care of this. Use the EDITOR to add B to line 30, which should now read,

```
 30 PCLS:LINE(10,180)    (245,10),PSET,B
```

Instead of a diagonal line you now have a rectangle. To draw a rectangle all you need do is specify the position of two opposite corners and add B to the LINE statement. Get back into the EDITOR and add F to the end of line 30.

```
 30 PCLS:LINE(10,180)    (245,10),PSET,BF
```

The added F means fill the box, (rectangle) with the foreground colour. Such a flexible command must have some use, so let us draw a picture.

We will start as before, in the construction business, but this time a house. Run the program after each section so that you can see how it builds up.

First we set the resolution and draw the body of the house.

```
 10 PMODE 3,1:SCREEN 1,9: PCLS
 20 LINE(60,48)    (200,144),PSET,B

 260 GOTO 260
```

Next we add the roof.

```
 40 LINE(60,48)    (130,20),PSET
 50 LINE    (200,48),PSET
```

and a garage, with a door.

```
 70 LINE(200,144)    (255,94),PSET,B
 90 LINE(210,144)    (245,104),PSET,BF
```

We can use the same technique to put a door into the house.

```
 100 LINE(160,144)    (188,105),PSET,BF
```

To draw windows we need a rectangle with two lines as cross pieces.

```
 110  LINE(85,132)    (135,108),PSET,B
 120 LINE(110,108)    (110,132),PSET
 130 LINE(85,120)    (135,120),PSET
```

The upstairs windows use the same approach,

```
 140 LINE(90,84)    (125,64),PSET,B
 150 LINE(90,74)    (125,74),PSET
 160 LINE(110,84)    (110,64),PSET
```

LINE

The LINE command is used to draw lines and rectangles in the high resolution graphics modes.

LINE (*x*1,*y*1)   (*x*2,*y*2),*a*,*b*

*x*1,*y*1 are the co-ordinates of the line's start point.
*x*2,*y*2 are the co-ordinates of the line's end point.
a is either PSET or PRESET. If PSET is used the line is drawn in the current foreground colour. If PRESET, the line is drawn in the background colour.
b is an optional parameter. If used it is either B or BF. If B, a rectangle is drawn instead of a line, the upper corner of the rectangle will be *x*1, *y*1, and the lower right corner *x*2, *y*2. If BF is used, the rectangle is drawn and filled with the current foreground colour.

```
10 PMODE 4,1: SCREEN 1,1: PCLS 5: COLOR 0,5
20 FOR I = 1 TO 1000
30 X =X+L*SIN(R): Y = Y+L*COS(R)
40 IF X<   128 OR X>128 THEN 90
50 IF Y<   96 OR Y>95 THEN 90
60 LINE    (X + 128,Y + 96),PSET
70 R1 = R1+60: R = R1/57.29578: L = L+0.5
80 NEXT I
90 GOTO 90
```

```
170 LINE(155,64)      (175,84),PSET,B
180 LINE(165,84)      (165,64),PSET
190 LINE(155,74)      (175,74),PSET
```

and to complete the structure a chimney,

```
200 LINE(150,40)      (160,15),PSET,BF
```

This little program shows how quickly a picture can be drawn, and all with just one command. This, of course, assumes you know where to draw the lines. The easiest way to find these points is to take a copy of the graphic screen grid in Appendix B, sketch the picture on it and read off the points.

A SPLASH OF COLOUR

Our house looks a little drab, what it needs is a coat of paint to brighten it up. So we will tell DRAGON to get the brushes out and start work. The PAINT command allows you to paint any shape with any available colour. All you have to do is to tell it where to start, what colour to paint with, and the colour of the border where the painting is to stop.

PAINT (*x*,*y*),*a*,*b*

where x,y are the co   ordinates of the start, and a and b are the colour codes of the paint and the border. Add the following line to the house example,

```
30 PAINT (90,90),2,4
```

This means starting at point (90,90), paint in yellow, (colour 2) until you meet a red (4) border. Run the program to see what it does. Now delete the line and re-enter as line 195 and run the program again.

```
195 PAINT (90,90),2,4
```

Note how it now stops at the window borders which were not there before. Paint the garage the same way,

```
80 PAINT (210,140),2,4
```

Now the roof. If you omit to put in a colour or a border in a PAINT statement, the current foreground colour is selected for both.

```
60 PAINT (130,25)
```

We finish off by drawing the skyline and the sky.

```
210 LINE(0,64)     (60,64),PSET
220 LINE(200,64)     (255,64),PSET
230 PAINT(0,54),3,4 no,
```

PAINT

The PAINT command is used in high resolution graphics modes, to fill a shape with a specified colour.

PAINT (*x*,*y*),*c*,*b*

x,y are the co   ordinates of the point where the painting is to start.

c is the colour code of the colours to be used to paint. lt must be between 0 and 8 and be one of the available colour set for the working mode. If omitted, the current foreground colour is used.

b is the colour code of the border at which painting is to stop. It must also be between 0 and 8, the painting will continue over a border of any other colour. If omitted, the current foreground colour is used.

See CIRCLE for example of usage.

The place looks a little brighter now. You may like to continue with improvements, add a path and a fence. Or you could take some time off to practice drawing your own shapes and painting them to see what happens.

GOING ROUND IN CIRCLES

We have got lines, squares and rectangles, and now circles. The CIRCLE statement will draw circles, ellipses and arcs.

CIRCLE (*x,y*), *radius, colour, hwratio, start, end*

The *x,y* point is the centre of the circle, the *radius* is the circle's radius measured in screen points. The *colour* is one of the available colours in the mode you are working in, (if omitted, the foreground colour is used). The other parameters are for drawing ellipses and arcs, we will deal with them later. First let us see what happens with circles.

```
10 FOR P = 0 TO 4: PMODE P,1
20 SCREEN 1,1: PCLS
30 FOR R=120 TO 10 STEP    10
40 CIRCLE (128,96),R: NEXT R
50 FOR D = 1 TO 500: NEXT D,P
```

This will draw circles towards the centre of the screen. Circles are difficult to draw and for a very accurate one you will probably have to work with PMODE4.

You will notice that if a circle goes off the screen, there is no problem. If you try to draw a line to a point that is not on the screen, it may not be drawn at all, especially in the higher resolution modes. Try inserting

```
42 LINE     (300,40),PSET
```

and watch the result.

The PAINT command can also be used to fill in circles,

```
45 PAINT (128,96)
```

fills in the 'bullseye'.

By using the *hwratio* parameter you can change the circle into an ellipse. The *hwratio* means height to width ratio. The width in the CIRCLE command always remains the same, twice the radius. The height can be varied by the *hwratio*, if it is greater than 1 then the 'circle' will be higher than it is wide. A value less than 1 will squeeze the circle in the other direction, wider than it is

CIRCLE

The CIRCLE command will draw circles, ellipses and ares. It can only be used in the high resolution graphics modes.

CIRCLE(*x*,*y*),*r*,*c*,*hw*,*start*,*end*

x	is the X co-ordinate of the centre of the circle (from 0 to 255)
y	is the Y co-ordinate of the centre of the circle (from 0 to 191)
r	is the circle radius, measured in screen points
c	is a colour code (from 0 to 8), it must be one of the available set. If omitted, the foreground colour is used.
hw	is the height-width ratio (from 0 to 255). Used for drawing ellipses. If *hw* is omitted, 1 is used.
start	is the start of the arc of the circle (from 0 to 1). The 0 position represents 3 o'clock. If omitted, 0 is used.
end	is the end of the arc (from 0 to 1). The drawing proceeds clockwise from *start*. The .5 position represents 9 o'clock. If omitted, 1 is used.

```
10 PMODE 3,1: SCREEN 1,0: PCLS
20 CIRCLE (180,156),28,3: PAINT (180,156),3,3
30 CIRCLE (110,156),28,3: PAINT (110,156),3,3
40 CIRCLE (144,80),68,4,1,0,.5
50 LINE (212,80)    (76,80),PSET: LINE    (48,32),PSET
60 PAINT (144,82):CIRCLE(144,80),70,4,.8,.79,1
70 LINE (160,80)    (160,28),PSET: PAINT (210,75)
80 GOTO 80
```

high. So the width along the X axis (horizontal) is always the same, only the height on the Y axis (vertical) changes. When *hwratio* is 0 the 'circle' is a horizontal line, and with *hwratio* very large it approaches a vertical line (actually a long thin rectangle). The largest value allowed is 255. Change lines 30 and 40 in our current example to read.

```
30 FOR H = 0.5 TO 3 STEP 0.5
40 CIRCLE (128,96),40,,H: NEXT H
```

Note in line 40 the extra commas, these are because we have missed out the *colour* parameter.

The final extension to the CIRCLE command is the ability to draw arcs (part of a circle). To use this option you have to specify the start and finish of the arc. Both the start and finish values must be a number between 0 and 1. The starting point of the circle is equivalent to the 3 o'clock position on a clock. The drawing action then goes clockwise from the start. For instance, a start at 0.25 and end at 0.75 would draw 6 o'clock to 12 o'clock, the left half of the circle. Start at 0.5 and end at 1.0 to draw the top half of the circle. The following program uses arcs to draw a pattern.

```
10 PMODE 4,1:SCREEN 1,1:COLOR 0,5:PCLS
20 FOR R=15 TO 60 STEP 5
30 CIRCLE(128,96+R),R,,1,.5,1
40 CIRCLE(128,96   R),R,,1,0,.5
50 CIRCLE(128   R,96),R,,1,.75,.25
60 CIRCLE(128+R,96),R,,1,.25,.75
70 FOR D=1 TO 500:NEXT D
80 NEXT R
90 GOTO 90
```

TURNING THE PAGE

One way of introducing animation into drawings is to place a slightly different picture on each page and then 'flip' through the pages. Remember you set the number of pages with the PCLEAR command and the second parameter of PMODE decides the page you are writing to. Of course, you have to bear in mind the resolution in which you are working. In PMODE3 and PMODE4 each graphic screen requires 4 pages so it really only makes sense to flip between page 1 and page 5. In PMODE1 and PMODE2, which needs 2 pages, you would flip between 1, 3, 5 and 7. The following example shows how this is done, try entering all the PMODE values.

```
10   PCLEAR 8: PMODE 3,4: PCLS
20   INPUT"MODE";M: ON M GOTO 40,40,50,50
30   S = 1: GOTO 60
40   S = 2: GOTO 60
50   S = 4
60   FOR P = 1 TO 8 STEP S: PMODE M,P: PCLS
70   LINE(128,0)     (128,(P   1)*15),PSET
80   SCREEN 1,1:FOR I = 1 TO 1000:NEXT I,P
90   FOR P = 1 TO 8 STEP S: GOSUB 150:NEXT P
100 IF M>2 THEN D = 4: S1 = 3 ELSE D = 7: S1 = S
110 FOR P = D TO 1 STEP    S1:GOSUB 150:NEXT P
120 GOTO 90
150 PMODE M,P: SCREEN 1,1
160 FOR T=1 TO 20: NEXT T: RETURN
```

Lines 60 to 80 draw the changing figure onto the different pages. All this drawing takes place without being displayed, as no SCREEN command has yet been given. The remainder of the program displays each page in turn, flipping first forwards, then backwards to give the impression of movement. You can see that the more pages you use, the smoother the motion.

Another way of constructing displays is to use the command PCOPY

PCOPY *sourcepage* TO *destinationpage*

You may copy the contents of any page to any other page, provided the page has been previously reserved with PCLEAR. PCOPY can also be used to pack duplicates onto a PMODE3 or PMODE4 page. The following program shows how PCOPY is used for this purpose, note how you have to be careful where the figure is placed.

```
10 PCLEAR 8: PMODE 3,4: PCLS
20 LINE(100,20)     (140,40),PSET,BF
30 CIRCLE (50,25),20
40 CIRCLE (200,50),20
50 FOR D=3 TO 1 STEP    1
60 PCOPY 4 TO D: NEXT D
70 FOR P=4 TO 1 STEP    1: PMODE 3,P
80 SCREEN 1,1: FOR I=1 TO 1000: NEXT I,P
90 GOTO 90
```

In PMODE3, (and 4) the display consists of four pages, page 1 being the top quarter of the screen, page 2 the next, and so on. So by copying the contents of page 4 on to page 1 you have duplicated the top quarter of the display in the bottom quarter. For PMODE1 and PMODE2 the same effect can be obtained but this time the screen will be halved not quartered.

Those of you eager to continue with graphics may now turn to chapter 10, for the rest of us there will be a short musical interlude.

PCOPY

PCOPY is a high resolution graphics command used to copy the contents of a graphics page to another graphics page

PCOPY *source* TO *destination*

source and *destination* must be numbers between 1 and 8, and must refer to pages previously reserved with the PCLEAR COMMAND. The space required to
hold a display screen differs for each mode and should be considered when using PCOPY.

```
PCOPY 3 TO 5
```

CHAPTER NINE

SOUNDS ELECTRIC

ADDING A SOUND TRACK

Graphics, and other programs, can often be made more interesting by the addition of sound. We have already used the SOUND command for this, especially in the example in chapter seven. An easier way of adding sound is to provide it yourself. We do not mean sing along with your programs, or not quite. Your computer uses a cassette recorder to store programs, it can also run a tape on demand. The commands MOTOR ON and MOTOR OFF will do exactly that. Together, with the AUDIO ON and AUDIO OFF commands which connect, or disconnect, the cassette output to the TV loudspeaker. This means that by putting these statements into your program you could have background music to your graphics. Or, on a more serious note, a pre-prepared tape could deliver instructions and the questions in an educational program. The example following shows how easy it is. If you do not have a tape handy, use one of your program tapes. The strange noises you will hear are how computers talk to each other!

```
10 CLS: PRINT @ 135,"PRESS THE SPACEBAR"
20 PRINT @ 195,"TO STOP OR START RECORDER"
30 A$ = INKEY$: IF A$<>"▽" THEN 30
40 IF F = 0 THEN MOTOR ON: AUDIO ON: F = 1 ELSE
   MOTOR OFF:AUDIO OFF: F = 0
50 GOTO 30
```

Rewind the tape to the beginning and press the PLAY button, then run the program. By pressing the spacebar you will be able to stop or start the playback of the tape.

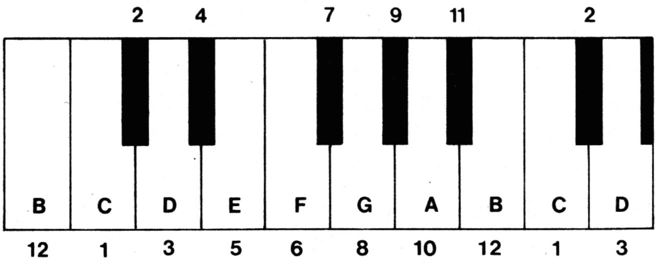
Using this method a general question and answer type educational program could be written to handle a number of different subjects, just by changing the question tape. Or your cartoon animations could be supplied with appropriate music and sound effects.

PLAY THAT THING!

Alternatively, you can make the computer play the music. The PLAY command converts the contents of a string into sounds

PLAY *string*

where *string* may be a string constant or a string variable. Not just any old string, however, but a music *string* made up from *note, octave, note, length, tempo* and *pauses*. The note is obviously the musical note you want to play. The easiest way to do this is to enter the letter representing one of the standard musical notes A,B,C,D,E,F,G. To indicate a sharp you use #, or + (F# or F+ for F sharp), and for a flat , (B for B flat). The computer will not recognise B# or C , as they do not exist in the music 12 tone scale. Another way of entering a note is to use the number representing its position in the 12 tone scale



The notes and their number equivalent are marked on the keyboard above.

The PLAY command can be used as a direct command, which is useful for checking a music string before incorporating it into a program. Like all good musicians we wall start by practising our scales.

```
PLAY "CDEFGABCCBAGFEDC" The scale in C
PLAY "GABCDEF#GGF#EDCBAG" The scale in G
```

Well the scale in C is nearly right, but the one in G is a mess. This is because the scales move into a different octave, and we must tell the computer this. To select the octave, use 0 followed by a number between 1 and 5, 02, (which includes middle C) is automatically set when the computer is turned on. The current setting for octave will be used until a change is made, so it is usually safer to always specify the octave you want to use at the start. Let us try the scales again.

```
PLAY "O3CDEFGABO4CCO3BAGFEDC"
PLAY "O3GABO4CDEF#GG#EDCO3BAG"
```

To play the scale in C using the numbers instead of letters

```
PLAY "O3;1;3;5;6;8;10;12;O4;1;1;O3;12;10;8;6;5;3;1"
```

Note the separator (;) used in the string. You can use the semi-colon anywhere you want but with numbers it is usually needed to avoid confusion.

AUDIO

The AUDIO command controls the connection of the sound output of the cassette recorder to the television set loudspeaker. AUDIO ON directs cassette output to the T.V. AUDIO OFF disconnects the link.

MOTOR

The command MOTOR controls the operation of the cassette recorder motor. MOTOR ON starts the motor, MOTOR OFF stops the motor.

The play button of the cassette recorder must be depressed for the command to be effective.

As the music string is still a string, it can be manipulated with all the usual string operations. The following example plays the scale in C over the entire range of the PLAY command.

```
10 A$ = "CDEFGAB": FOR I=1 TO 5
20 B$ = "O" + STR$(I) + A$
30 PRINT B$:PLAY B$:NEXT I
```

By using the same technique, and numbers instead of letters we can play the entire chromatic scale.

```
10 FOR I=1 TO 5:A$ = "O" + STR$(I) + ";"
20 FOR J=1 TO 12:PLAY A$ + STR$(J): NEXT J,I
```

In most tunes, the notes are rarely of identical length, so we need to set the duration of each note. This is done with the *note length* parameter of the music string (L). The letter L is followed by a number between 1 and 255. Usually, however, the number represents the lengths commonly used in music. As the size of the number increases so the length decreases, L1 is a whole note, L2 a half note, L4 a quarter note and so on. It is possible to have a 1/255th note, but not many composers use them. Those of you who read music will have heard of "dotted" notes. The dot tells you to increase the length of the note by one half of its normal value. To obtain that effect with the PLAY command, you put a dot, (or as many dots as you like) after the number in the L parameter.

L4. = 1/4 + 1/8 = a 3/8 note

We now have sufficient to play a simple tune. Carefully type in the following,

```
5 CLEAR 500
10 A$ = "O2L4GG;L2GDL4BB;L2BGL4GB;
          O3L2DDL4C02B;L1AL4AB;
          O3L2CCO2L4BA;L2BGL4GB;
          L2ADL4F#A;L1G;"
20 B$ = A$ + A$: PLAY B$
```

The separators are being used here to indicate the bar divisions, they are not actually needed. You should be able to recognise the tune, 'Clementine', but it is being played far too slowly. The *tempo* parameter takes care of this, the letter T followed by a number between 1 and 255. The higher the number the faster the tune is played. Try changing line 20 to read,

```
20 B$ = A$ + A$: PLAY "T6" + B$
```

PLAY

The PLAY command is used to generate a music sequence. The argument is a string expression, or string constant, or string variable. Its form is,

PLAY *music*

where *music* is constructed from the following elements:

note A letter from 'A' to 'G' or a numeral from 1 to 12.

octave 'O' followed by a number from 1 to 5. Default
O2. The default values are set by the computer when switched on.

note length 'L' followed by a number from 1 to 255.
Default L4.

tempo 'T' followed by a number from 1 to 155. Default T2.

volume 'V' followed by a number from 1 to 31. Default V15.

pause length 'P' followed by a number from 1 to 255.

execution of substrings 'X' followed by string variable and a semi   colon.

A sharp flat note can be indicated by '+' or '#' for a sharp or '   ' for a flat.

The *note length* parameter can be modified by the addition of a dot (.) after the number, (L2.) to represent a dotted note.

The octave, volume, tempo, and note length can be modified by using one of the following suffixes:-

+	Adds one to current value
	Subtracts one from current value
>	Multiplies current value by two
<	Divides current value by two

```
10 X$ = "O3L4EF#L4.EL8AAG#ABL4O+C#O   B"
20 A$ = "XX$;O4C#O   AF#O+DC#O   BL2AXX$;
    O+C#DEL8DO   BL<AG#L<AL4.BL8O+C#L4
    DO   BL4.O+C#L8DL<EC#L4.EL8EEEEEL1
    EL4.EL8DC#EDO   BL<AG#L<A"
30 PLAY"T2V20"+A$
```

Experiment by changing the 6, and find a value which suits your idea of what speed the tune should be played at.

Most music also requires the ability to insert pauses in between phrases and also to vary the loudness of certain passages. The *pause* parameter is the letter P followed by a number. It follows the pattern of the *note length* parameter (L), except you cannot use the dots after the number. To insert a pause the equivalent of an L4. note, P4P8 would have to be used. The *volume* parameter allows us to vary the loudness by inserting the letter V, followed by a number between 0 and 31, as the number increases the piece becomes louder. The example below uses the volume parameter to produce a crescendo.

```
10 A$ = "V10O2L4GG;L1GP4V14L4GGG;
         L1GP4V18L4GGG;L2BL4BBBV22L2BL4BBB;
         V26O3L2DL4DDDL2DL4DDD;
         V30L1GL2.F#L4C#;L2EDCO2A;
         L1GL2AL4.DL8A;L2B"
20 PLAY "T5"+A$
```

Often a piece of music will contain a passage which is repeated in a number of different places within the piece. Rather than type the passage more than once, it is usually better to put it into a separate string variable. The *execute* substring command X, allows this substring to appear as part of a normal sequence of play commands. The X must be followed by the name of the string variable *and a semi-colon*, as in,

```
10 X$ = "O3L2GBO4C;DL4CO3BAG"
20 Y$ = "L2ADD;L1.A"
30 Z$ = "L2ADD;L1.G"
40 A$ = "XX$;XY$;XX$;XZ$;L2BGG;O4CO3L4
         BAGF#;XY$;XX$;XZ$;"
50 PLAY "T8" + A$
```

We could have used a substring in our 'Clementine' example, just change line 20 to

```
20 PLAY "T6XA$;XA$;"
```

The semi-colon *must* follow the dollar sign ($), substrings and the use of numbers for the notes (instead of letters), are the only places where the semi-colon is essential.

There is one more option which can be used with the volume (V), octave (O), tempo (T) and note length (L) parameters. Instead of a number following the letter, you can use one of the following suffixes:

```
+    Adds one to the current value
      Subtracts one from the current value
>    Multiplies the current value by two
<    Divides the current value by two
```

Our final scale example could now be rewritten to include this extra option,

```
10 PLAY "O1C": FOR I = 1 TO 4:PLAY "DEFGABO+C": NEXT I
```

Where does one get the music from? You could, of course, compose your own, but for us lesser mortals, sheet music written for single line instruments such as flute, recorder and trumpet are useful sources.

For those of you that have no inclination to play music on the computer, do not ignore the PLAY command completely. Games fans can use it to produce some very useful effects. Just try any one of the examples in this chapter with the tempo parameter set at T255.

The final example uses ail the modern technology of the PLAY command on a 400 year old song. Would Henry be impressed?

```
10 A$ = "O3L2E;L1GL2AL2.BL4O+C#L2O   B;
    L1AL2F#L2.DL4EL2F#;L1GL2EL2.EL4DL2E;
    L1F#V10L2DV8L1O   BV6L2O+E;L1GL2AL2.B
    L4O+C#L2O   B;L1AL2F#L2.DL4EL2F#;
    L2.GL4F#L2EV8L2.D#V10L4C#V15L2D#;
    L1.EL1EP1;"
20 B$ = "O4L1.DL2.DL4C#O   L2B;L1AL2F#L2.D
    L4EL2F#;L1GL2EL2.EL4DL2E; L1F#L2D
    O   L1BO+L2B;O+L1DL2DL2.DL4C#O   L2B;
    L1AL2F#L2.DL4EL2F#;L2.GV10L4F#L2
    EV6L2.D#L4C#V4L2D#;V15L1.EL2EP1;"
30 PLAY "T10XA$;XB$;XA$:XB$;"
```

CHAPTER TEN

FURTHER GRAPHICS

In chapter eight, we showed how the LINE and CIRCLE commands could be used to produce regular shapes such as rectangles, circles, ellipses and arcs. While these commands are extremely useful, it can require considerable ingenuity to construct very detailed or irregular shapes using these commands. The easiest way to handle these sort of shapes is to draw them.

When you draw a figure onto a piece of paper, you start at a particular place and move the pencil up a certain amount, then to the right and so on. The DRAW command allows you to repeat this process on the screen. The form is,

DRAW *string*

where *string* is either a string constant, or a string variable, containing a set of the draw subcommands. The approach is very similar to the PLAY command of the last chapter.

Usually the first action of any drawing is to move to the start point.

M *x*,*y* means move to the co   ordinates given by *x*,*y*, as in M128,96, this will move to screen centre. When you move to a point it is usually a good idea to make a blank move, that is move without drawing or lifting the pencil off the paper. If you do not you may get unwanted lines on your drawing. A blank move is done by using the letter B, any drawing instruction following the B will be a blank line. BM128,96 means move to the screen centre without drawing.

Having decided the start point, you may now move up (U), down (D), right (R), or Left (L) by as many points as you like. The sequence U20R20D20L20 will cause a line to be drawn upwards 20 points for the start, then to the right 20, then down 20 and left 20, drawing a box. Time to start building an example,

```
10 PMODE 3,1: PCLS: SCREEN 1,1
20 DRAW"C8;"BM120,96:U26;R13;D26;L13"
80 GOTO 80
```

The semi-colon n a string is used as a separator. It is not actually required, we have just used it to make the string easier to read. The example draws a rectangle near the middle of the screen.

Apart from vertical and horizontal lines you can also draw diagonal lines. These use the subcommands E,F,G and H, for instance E12 will draw a diagonal line, 12 points long, at 45 degrees from the vertical. All the angles are measured from the vertical as follows;

E 45 degrees	F 135 degrees
G 225 degrees	H 315 degrees

This allows diagonal lines to be drawn in any of 4 directions. Add the line,

```
40 DRAW"L6;U6;E6;BR13;F6;D6;L6;BU26;H6;G6"
```

to our current program and the rectangle becomes a rocket! The computer remembers its last position so line 40 will continue drawing from that point. Work through the string in line 40 to see how it is done. The last position drawn is the bottom left corner of the rectangle and BR13 means move right 13 points without drawing.

The rocket has been drawn in the default foreground colour, but we can change that if we want by using C. The letter C is followed by a number from 0 to 8, representing the code for one of the available colours. Using the editor change line 40 to read,

```
40 DRAW"C7;L6;U6;E6;BR13;F6;D6;L6;H6;G6"
```

and we now have a two-tone rocket! The drawing can be painted exactly the same way as other shapes, but the C command changes the default foreground colour, so care is needed to avoid painting over everything.

The drawing is a bit small, so we wall scale it up with the S parameter. The S means scale, and allows a drawing, or parts of a drawing to be scaled up or down in units of ¼. So S1 reduces the drawing to ¼ scale, S2 to 2/4 (half) scale, S8 to 8/4 (twice) scale, and so on. The default setting for scale is 4/4 (i.e. 1 the original size). The S may be followed by any number from 1 to 62. Add the line

```
15 DRAW"S12"
```

and the rocket is now three times the original size.

Another option available is the angle parameter A. This allows us to rotate all or part of the drawing, as all lines after the A will be drawn with the displacement given by An. n is a number between 0 and 3, as follows,

0 0 degrees	1 90 degrees
2 180 degrees	3 270 degrees

Alter the current program with the following lines

```
18 FOR I  =  0 TO 3:DRAW"A"+STR$(I):PCLS
50 FOR D  =  1 TO 100:NEXT D,I
```

The rocket now turns, but it also changes colour! This is because the computer not only remembers the last position but also the last setting for C and A. This problem can be solved by putting C8 at the beginning of the string in line 30.

Line 20 shows that, as with the PLAY command, the strings used by DRAW can be used with the string functions. Also in a similar way to the PLAY command you can execute substrings with the X command followed by a string variable, XA$, and a semi-colon (;).

```
10 PMODE 3,1:SCREEN 1,1: PCLS
20 S$ = "L834F4"
30 D$ = "A0;XS$;A1;XS$;A2;XS$;A3;XS$;"
40 DRAW"S24"  +  D$
50 GOTO 50
```

A triangle is stored as a substring in S$, which is then used in line 30 to build a pattern. Note this is the only place the semi-colon is essential, after the dollar ($) sign.

The final parameter is N, meaning no update of drawing position. This is to draw a line as specified but do not use the end of the line as the new position, NU10L5, will draw a line 10 points up, *return to the start of the line*, and draw right 5 points (an L shape).

```
10 PMODE 3,1: SCREEN 1,1: PCLS
20 DRAW"BM128,96;NU25NR25NL25;NE17NF17NG17
    NH17"
30 GOTO 30
```

The above example will draw lines outwards from the centre, always returning to the centre for the start of the next line.

Often you will want to add another drawing near to the one you have just completed. You know where the new drawing is to be in relation to the old one, but do not wish to work out the co-ordinates. This can be done with *relative movement* such as 5 points to the right and lo up. The move command (M) allows this easily, ali you have to do is specify the distance as plus or minus the current point, i.e. M+5,   10. Remember to use the B to avoid unwanted lines.

```
25 DRAW"BM   25,   25;U10R25D10L25"
```

Add the above line to the last example and a rectangle will be drawn above the last drawing. The last position was 128,96, because of the N parameter. We have now moved 25 points to left and 25 points up (remember y = 0 is at the top of the screen), and the rectangle drawing starts at that point (103,71).

The results of a DRAW command can be combined with shapes from the LINE and CIRCLE commands, but remember that any subsequent scaling (S), colour (C), or angle (A) changes will effect only the contents of the DRAW part of the figure.

The PSET and PRESET commands can be used along with PAINT to block in extra detail and colour. Be careful, in particular with PAINT, as changes in the draw part of the figure may put the colour in all the wrong places.

GET THE PICTURE?

Having drawn your masterpiece using LINE, CIRCLE and DRAW etc, you now want to move it around the screen. We could, of course, do this by blanking out and redrawing cach time, as before. This could take quite a bit of time if the drawiiig was in any way complex, the next two commands take care of this. All you have to do is GET a copy of your picture and PUT it somewhere else. The GET command allows you to copy a rectangular area of the screen into an array, which can be PUT back onto the screen later.

GET(x_1,y_1)   (x_2,y_2),*arrayname*, G

The x_1,y_1; and x_2,y_2 are the co-ordinates of the upper left corner and the lower right corner of the rectangular area containing the picture you want to store. The *array name* is the name of a *previously dimensioned* array, in which the picture is to be stored. (If you have forgotten about arrays have another look at the beginning of chapter six). The size of the array must match the size of the display rectangle. The first array dimension is the *width* of the rectangle (x_2 x_1), the second the *length*, (y_2 y_1). The last parameter, G is optional and determines the amount of detail stored, it is necessary to include the G parameter in PMODEs 0, 1, or 3 otherwise horizontal moves via PUT may be inaccurate.

We will use our rocket to show how it is done. First, we must calculate the size of the array that will be needed. The drawing starts at 120,96, the left fin is 6 points across, the rocket and the right fin are 13 + 6, so the drawing is 25 points wide from 114,96 to 139,96. The height is 26 up, plus the nose cone, which is 5 at the point, so the height is 31. Add a few points either way and call it a 30 × 40 rectangle with the left top corner at 112,60 and bottom right at 142,100.

```
10 PMODE 3,1: SCREEN 1,1: PCLS: DIM R(29,39)
20 R$ = "C8BM120,96;U26R13D26L13;C7L6U6E6BR13
    F6D6L6BU26H6G6"
30 DRAW R$
40 GET(112,60)      (142,100),R,G

100 GOTO 100
```

The above example draws our rocket as before (all in one string now) and stores into the array R. Note that we only need a 29 x 39 array, because we can use the zero elements in the array.

Having stored the drawing we now need to PUT it back onto the screen. The PUT command has a form similar to GET

PUT (x_1,y_1)   (x_2,y_2),*arrayname*, *action*

The x_1,y_1 and x_2,y_2 are the co   ordinates of the rectangle as before, but this time refer to the area where you want to PUT the drawing, not where it came from. The *arrayname* is the array variable containing the stored drawing. The *action* parameter is optional and is only needed when the G parameter has been used with the GET commands. The *action* must be one of the following words, and decides how the result is displayed in its new position.

PSET	Set each point that is set in the source array. In other words, display it as you get it.
PRESET	Reset each point that is set in the source array. This will either blank out the picture or reverse the colours, depending on the foreground and background colour settings.
AND	Compares the points in the original with those at the destination. If both are set then the point will be set. If one or other is not set the point wili be reset. This means if one picture is placed on top of another only the points which coincide wall be shown.
OR	Compares the points as above. If either the source or destination point is set the screen point will be set. This has the effect of overlaying one drawing with another.
NOT	This reverses each point in the display area, thus displaying picture against foreground colour.

GET

The GET command may only be used in high resolution graphics modes. GET will copy the graphics contents of a specified rectangular area on the screen and store it into an array. The array must have been previously dimensioned to the correct size.

GET(x_1,y_1)   (x_2,y_2),*arrayname*, G

x_1,y_1 and x_2,y_2 are the upper left and lower right co   ordinate of the rectangle on the display.

arrayname is the name of the predimensioned array that will store the rectangle's contents.

G instructs full GRAPHIC detail to be stored, this command is optional.

See the PUT box for an example of GET usage.

You must always use PUT in the same mode as GET, otherwise strange results may occur. We can now return to our example and PUT our rocket in a different place. Add these extra lines.

```
50 Y = 150:FOR X=10 TO 210 STEP 40
60 PUT(X,Y)      (X + 30,Y + 40),R,PSET

80 NEXT X
```

There should now be a line of rockets along the bottom of the screen. To make the rocket move along the bottom just add the line,

```
70 FOR D =  1 TO 200:NEXT D:PCLS
```

By using the joysticks in conjunction with the GET and PUT commands, you can move your drawing at will,

```
10 PMODE 3,1:SCREEN 1,1:PCLS:DIM S(48,48)
20 DRAW"BM24,12,S8;C4;E2H2D4D8R8H8G8R2
    NR6F3R6E3"
30 GET(0,0)    (48,48),S
40 A$ = INKEY$:IF A$="" THEN 40
50 PCLS:A = JOYSTK(0)*3.25:B = JOYSTK(1)*2.25
60 PUT(A,B)    (A+48,B+48),S:GOTO 50
```

The above example draws a figure in the top left hand corner of the screen. When you press any key the screen is erased and the figure can be moved about using the left joystick.

This completes our coverage of the graphics facilities available. The examples we have offered are necessarily limited and do not in any way represent what is possible with a little thought and a lot of patience. Drawing pictures can be made much easier with a little forward planning and drawing the shapes onto the graphics worksheet rather than direct onto the screen.

DRAW

The DRAW command draws a line, or series of lines according to the instructions held in a string. It only operates in the high resolution graphics modes.

DRAW *string*

The string may be a string constant or a string variable and contain any of the following subcommands.

M*x;y*	Move to the draw position at *x,y*
U*n*	Up *n* points
D*n*	Down *n* points
L*n*	Left *n* points
R*n*	Right *n* points
E*n*	At 45 degrees for *n* points
F*n*	At 135 degrees for *n* points
G*n*	At 225 degrees for *n* points
H*n*	At 315 degrees for *n* points
X	Execute a substring and return
C	Set colour of line
A*k*	Displace next line by angle
	k = 0 0 degrees *k* = 1 90 degrees
	k = 2 180 degrees *k* = 3 270 degrees
S*k*	Scale drawing in units of ¼, *k* from 1 to 62
	k = 1 is quarter scale, *k* = 8 is double scale.
	Default *k* = 4
N	No update of draw position
B	Blank (do not draw, just move)

Relative movement can be specified with the 8 parameter in the form

M *x offset*, *y offset*

Where *x offset* and *y offset* are numbers specifying the distance to move from the current position. Both numbers *must* be preeceded by either a plus (+) or minus ( ) sign.

An example of the usage of DRAW appears in the PUT box.

PUT

The PUT command is used to display the contents of a graphic array stored by the GET command.

PUT must be used in the same mode that was used to create the array in the first place, otherwise the results may be unpredictable.

PUT(x_1,y_1)   (x_2,y_2),arrayname,action

x_1,y_1 is the co   ordinate of the top left hand corner of the display area. x_2,y_2 the bottom right hand corner. The *arrayname* refers to the predefined array containing the graphic detail. The *action* parameter is optional but *must* be used if the G parameter was present in the GET command.

PSET	Sets destination points as source array
PRESET	Resets each point that is set in source array
AND	Compares source array and destination. If both points set the point remains set, otherwise it is reset
OR	Compares points as above, if either point is set the screen point is set
NOT	Reverses the state of each point in destination area, regardless of source array.

The chosen display area must be the same size as the array or *'garbage'* will be drawn on the screen.

```
10   PCLEAR 4:PMODE 3,1:PCLS:SCREEN 1,1:DIM W(30,30)
20   DRAW"BM10,12;S8;R1U3R1D2R2U2R1D3R1D2L1
     D2R1D1L2U3L4D3L2U1R1U2L1U2R1U2BR1BD1D2R2
     U2NL2R2D2L2U2"
30   PAINT (11,13),6,5:GET(0,0)     (30,30),W
40   A$  =  INKEY$:IF A$="" THEN 40
50   PCLS:FOR C  =  0 TO 100 STEP 20
60   FOR A=0 TO 200 STEP 20
70   PUT(A,C)     (30+A,30+C),W
80   PUT(A,C+30)     (30+A,60+C),W
90   PUT(A,C+60)     (30+A,90+C),W
100  PLAY"T255;ABFGBA":PCLS:NEXT A,C
```

CHAPTER ELEVEN

THE FINISHING TOUCH

PRINT EXTRAS

While your control over the way results are shown on the screen is quite extensive, using the PRINT and PRINT @ commands, there is one more facility available. The PRINT USING command allows you to specify exactly how each line should be printed. It is especially useful for producing tables, forms and accounting type layouts.

PRINT USING *format; output list*

The *format* is a string constant or a string variable containing the instructions as to how the *output list* is to be printed. The *output list* is the usual list of constants and variables as appears in the ordinary PRINT command.

The instructions in the *format* are made up of 'field specifiers'. These are a set of characters which tell the computer exactly how many print postions to use to print a number or string.

The # specifier

This character is used to indicate the position of each digit in a number.

```
PRINT USING"#  #  #.#  #";A
```

The above statement will print the contents of A as 3 digits before the decimal point and 2 after. If there are more than 2 digits after the decimal point, the number will be rounded to fit. Any unused positions on the left hand side of the decimal point will be displayed as spaces if the number is too big to fit into the space allowed, the computer will do the best it can and print the number with a % sign in front to show that this has happened.

```
PRINT USING"###.##";13.4695
▽13.47
PRINT USING"###.##";1492.878
%1492.88
PRINT USING"###.##";146
146.00
PRINT USING "###";18.76
▽19
```

The * specifier

Most accountants do not like the idea of printing numbers with spaces in front, especially for cheques. This can be taken care of by using the asterisk specifier. If you place two asterisks at the beginning of your numeric field the unused positions will be filled with asterisks.

```
PRINT USING "**###.##";1.492
****1.49
```

The + specifier

When the + is placed at the beginning of a numeric field it forces the sign of the number to be printed,

```
PRINT USING "+###.##";14.7
▽+14.70
PRINT USING "+**###.##":   7.4
****   7.40
```

lf the plus sign (+) is placed after the numeric field it wall force the sign to be printed alter the number.

```
PRINT USING "###.##+";27.86
▽27.86+
PRINT USING "###.##+";   1.6
▽▽1.60
```

If a minus sign is placed after a number it will cause all negative numbers to appear with a following minus sign, positive numbers will be followed by a space,

```
PRINT USING "**###.##   ";   12.418
***12.4
PRINT USING "###.##   ";47.25
▽47.25▽
```

The   specifier

This field allows numbers to be printed in exponential form. The four upward arrows must follow the number field.

```
PRINT USING "###.##              ";123456
▽1.2346E+05
```

The ! specifier

This specifier is used with strings. It will print only the first string character that occurs.

```
PRINT USING "!";"CREDIT"
C
```

The % specifier

To print out strings it is necessary to specify the width of the field they are to appear in. This is done with two % signs separated by a number of spaces. The width of the field will be the number of spaces plus two. If the string is longer than the available field, only the first *n* characters will be printed, where *n* is the length of the field.

```
PRINT USING "%▽▽▽▽▽%";"DEBIT"
DEBIT▽▽
PRINT USING "%▽%";"BALANCE"
BAL
```

The $ specifier

The dollar sign is used to represent money. If placed in front of a numeric it will force a dollar sign onto the output.

```
PRINT USING "$# # #.# #";2.87
$▽▽2.87
```

If two dollar signs are used it will cause the $ to be printed just in front of the number.

```
PRINT USING "$$# # #.# #";2.87
▽▽▽$2.87
```

Used in conjunction with the two asterisks the dollar sign will produce the following result,

```
PRINT USING "**$#.# #";14.9
*$14.90
```

Spaces and other characters appearing in the format string will also appear in the output,

```
PRINT USING "MEAN▽▽# #.# #▽▽▽TOTAL▽▽# # #.# #";;3.4,40.8

MEAN▽▽▽3.40▽▽▽TOTAL▽▽▽40.80"
```

If the *output list* contains more items than the number of fields in a *format*, the *format* is restarted from the beginning.

```
PRINT USING "# # #.# #▽▽▽";7.84,142.5,.234
▽▽7.84▽▽▽142.50▽▽▽▽▽0.23
```

Of course, using the screen, the length of the line produced by the PRINT USING statement is still limited to 32. Anything over this will cause the line to 'wrap around', i.e. start on the next line. For those with a printer, however, the line length you can use will be much longer (at least 80 characters on most printers). The form for printer use is

```
PRINT #   2,USING format; output list
```

format and *output list* are as before, the   2 means send to the printer channel not to the screen. If you want it displayed and printed then you will have to use two PRINT USING statements.

CASETTE INPUT AND OUTPUT

So far all our programs have required us to enter any data we may need (or READ it from a DATA statement), and all the output has gone to the screen. You can, however, use your cassette to store data, as well as programs. This stored data can then be read back in at a later date. The cassette is connected and set up in exactly the same way as for storing programs. You then need to tell the computer it is working with data files. This is done with the OPEN command

OPEN *a*,#   1,*filename*

The a must be either "O" or "I". "O" means output, that is the data is going *out* from the computer to the tape. "I" means input, the data is coming from the tape *in* to the computer.

The #   1 tells the computer you are using the cassette recorder. The *filename* is the name you want to call the data file (any name, beginning with a letter and 8 or less characters long, will do).

The next step is to write the data to the tape. This is done with a PRINT command in the following way.

PRINT #   1, *output list*

The only difference from the PRINT command we have been using all this time is the #   1. This tells the computer to print the *output list* to the tape and not to the screen.

When you have finished writing out the data, you *must* close the file with the CLOSE command.

```
CLOSE #   1
```

PRINT USING

The PRINT USING command allows greater control over the layout of results output to the screen, (or printer).

PRINT USING *format; output list*

The *format* is a string constant or variable containing the 'field specifiers' indicating how the *output list* is to be printed. The *output list* is a list of string or numeric variables (or constants) separated by commas.

The 'field specifiers' are as follows;

CHARACTER	ACTION	EXAMPLE	RESULT
#	Formats numbers	"####";147.2	147
.	Decimal point	"##.##";34.678	34.68
,	Display comma to the left of every third character	"#####.#";123456	123,456
**	Fill leading spaces with asterisks	"**###.###";1.47	****1.470
$	Places dollar sign ahead of number	"$#####.##";12.689	$ 12.69
**$	Floating dollar sign	"**$#####.##";12.689	****$12.69
+	In first position causes sign to be printed in front, in last position prints after number	"##.##+"; 12.689	12.69
	Print in exponential format	"##.## ";12.689	1.27E+01
!	Prints only the first string character	"!";CREDIT	C
%spaces%	String field. Length of field is number of spaces plus 2	%▽▽▽▽▽%"; "BALANCE"	BALANCE

Each 'field specifier' may be separated by any number of spaces which will appears as spaces on the output line.

```
10   CLS:INPUT"ENTER LAST BALANCE";B:C=0:D=0
20   CLS:T$="▽▽▽▽%▽▽▽%▽▽▽%▽▽▽▽%▽▽▽%▽▽▽%"
30   L$="▽▽▽####.##▽▽####.##▽▽####.##+"
40   PRINT USING T$;"DEBIT","CREDIT","BALANCE"
50   PRINT USING L$;D,C,B
60   OPEN"I",#   1,"CHEQ"
70   IF EOF(   1)THEN 110
80   INPUT#   1,A:D=0:C=0
90   IF A<= THEN D=ABS(A)ELSE C=A
100 B=B+C   D:PRINT USING L$;D,C,B:GOTO 70
110 CLOSE#   1:END
```

OUTPUT TO PRINTER

For those with a printer connected to the parallel I/O port, there are variations in some commands which allow output to be directed to the printer and no to the screen.

PRINT#   2, *output list*
PRINT#   2, USING *format;output list*

The *format* and *output list* are the same as for use on the screen

POS(2) will return the current position of the print head

LLIST will list a program directly to the printer. Its use is as for the LIST command.

Using the [SHIFT][0] combination allows lower case to be output to the printer. The lower case option can only be used in strings or REM statements, as all commands to the computer must be in upper case letters.

To read the data back in you use the same steps except this time the file is opened for INPUT and instead of PRINT you will use,

INPUT# 1, *input list*

The CLOSE command is the same for both.

The examples below show how it's done. First set up the cassette recorder and wind the tape to the place you want the file to be, (use SKIPF). Now press the PLAY and RECORD buttons together.

```
10 CLS:PRINT"CREATE PHONE LIST"
20 OPEN"O",#   1,"PHONE":PRINT"ENTER XXX,XXX TO END"
30 PRINT @ 128,"";:INPUT"NAME   ",N$
40 INPUT"TELEPHONE NO.";T$:IF
N$="XXX" OR T$="XXX" THEN 60
50 PRINT #   1,N$,T$:PRINT @ 128,"":GOTO 30
60 CLOSE #   1:END
```

When you run the program the tape will come on and start the file on the tape. Each time you enter a name and number it is written to the file. (the PRINT @ 128 statement in line 30 just clears the line on the screen). This will continue until you enter XXX,XXX, at which point the file is closed and the program ends.

Now all we have to do is to read it back in again. The main difference between output and input, is with input you must not try to read past the end of the file. This is taken care of by the extra statement you will need for input, the EOF command, this checks to see if the end of the file you are reading has been reached.

Rewind the tape to the beginning and this time press only the PLAY button.

```
10 CLS:PRINT"READ PHONE LIST"
20 OPEN"I",#   1,"PHONE"
30 PRINT"NAME",NUMBER"
40 IF EOF(   1)THEN60
50 INPUT#   1,A$,B$:PRINT A$,B$:GOTO 40
60 CLOSE#   1:END
```

When you run the program this time the tape will start and look for the file "PHONE". (You may have to wait a short while if it is towards the end of the tape). It will then read in the name and number and display them on the screen. Note that you do not have to use the same variable name you used to write out the data. You must however use the same type of variable. When the end of the file is reached it is closed and the program ends.

The EOF command has to appear before the INPUT #   1 command, otherwise you wall get an IE error, (trying to read past the end of the file).

Do not forget to CLOSE a file either as this can cause problems, especially when writing to a file.

A BIT MORE

You are now on your way to becoming an expert BASIC programmer, and may wish to look ahead to the next step   machine language. This is the computers native language, and so far you have been talking to it through an interpreter which speaks BASIC.

Why would you want to bother? Well machine language instructions will work much faster, may use less memory, and even allow you to do some things that BASIC doesn't.

The best approach is to obtain a manual on machine language, with special reference to the 6800 series microprocessors. One such manual is

Basic Microprocessors and the 6800 by *Ron Bishop* and published by the *Hayden Book Co. Inc.*

Once you have the background, your computer has a number of routines which allow you to use machine language routines. Brief details of these are given below.

USR*n* this allows you to call up to ten (0 to 9) machine language routines. The form is,

USR*n* (*argument*)

Where argument may be a string or numeric expression. When a USR call is met in the program, control is transferred to the address given in the DEF USR*n* statement. The address specifies the entry point of the machine language routine.

DEF USR*n* is used to define the address of a USR*n* function. Its form is

DEF USR*n* = *address*

n is between 0 and 9 and matches the n in the USR. The *address* must be between 0 and 65535 and contain the entry address for USR*n*.

CLEAR *s,h.* The CLEAR statement should be used to reserve memory for USR functions. The *s* refers to the amount of string space reserved as before.

The *h* is the highest memory address that BASIC may use. From *h* + 1 onwards is now reserved for machine language routines.

POKE. The POKE command is used to place a value into a specific part of memory.

POKE *address, value*

The address is as above and value must be between 0 and 255.

VARPTR. A pointer to a BASIC variable can be used as an argument by a USR function. This would allow a USR function to access the contents of an array.

VARPTR (*variable name*)

Where variable name is the BASIC variable you wish to access. VARPTR is used as part of the USR argument as in.

USR0(VARPTR(X))

Machine language routines may be saved and loaded from cassette by using CSAVEM and CLOADM.

CSAVEM *name, start, end, entry*
CLOADM *name, offset*

name is the name for the file on tape, start is the starting address of the routine in memory, end the last address occupied by the routine and entry is the program entry point. The offvet in the CLOADM command allows you to reload the routine into memory it an address given by start + offset.

Once loaded, control can be transferred to the routine by the EXEC command,

EXEC *address*

The address is the start of the routine, if address is omitted the computer will use the start from the last CLOAD command.

APPENDIX A

A.S.C.I.I. CHARACTERS CODES
(Decimal)

KEY	WITHOUT SHIFT KEY	WITH SHIFT KEY
[BREAK]	3	3
[CLEAR]	12	92
[ENTER]	13	13
[SPACEBAR]	32	32
!	33	
"	34	
#	35	
$	36	
%	37	
&	38	
'	39	
(	40	
)	41	
*	42	
+	43	
,	44	
	45	
.	46	
/	47	
0	48	18
1	49	
2	50	
3	51	
4	52	
5	53	
6	54	
7	55	
8	56	
9	57	
:	58	
;	59	
<	60	
=	61	
>	62	

KEY	**WITHOUT SHIFT KEY**	**WITH SHIFT KEY**
?	63	
@	64	19
A	97	65
B	98	66
C	99	67
D	100	68
E	101	69
F	102	70
G	103	71
H	104	72
I	105	73
J	106	74
K	107	75
L	108	76
M	109	77
N	110	78
O	111	79
P	112	80
Q	113	81
R	114	82
S	115	83
T	116	84
U	117	85
V	118	86
W	119	87
X	120	88
Y	121	89
Z	122	90
	94	95
	10	91
	8	21
	9	93

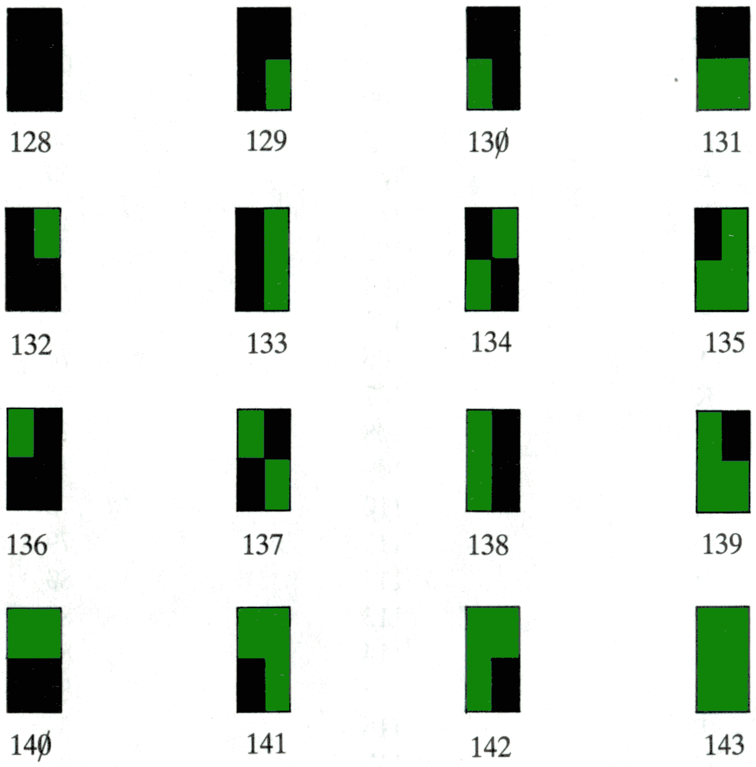
The without shift characters are obtained by using the [SHIFT][0] combination to move into lower case.

The following lower case characters are available with the CHR$ function:

[	CHR$(123)		CHR$(126)
/	CHR$(124)		CHR$(127)
]	CHR$(125)		

The characters from 128 to 255 are graphics characters as follows:

GRAPHICS CHARACTERS



To produce the above characters use CHR$ with the appropriate code. To obtain the other colours, add the appropriate number to the code. For example, PRINT CHR$(142+112) produces character 142 except the green area is orange.

+ 16 yellow	+ 32 blue	+ 48 red
+ 64 buff	+ 80 cyan	+ 96 magenta
	+ 112 orange	

APPENDIX B

PRINT AND GRAPHIC SCREENS

The following work sheets are useful for designing graphics and print layouts.

The first is used for the PRINT @ command

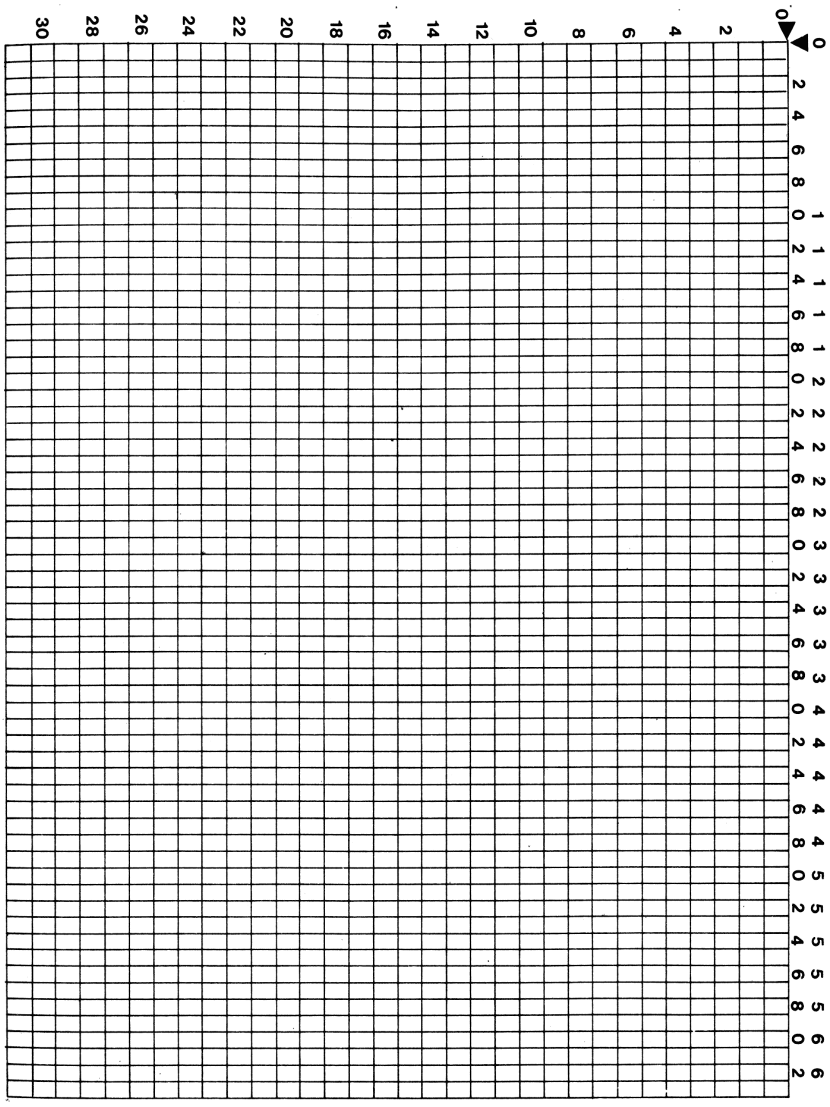
The second for low resolution graphics on the text screen, using the SET and RESET commands.

The third is for the high resolution screen and all the high resolution graphics commands.

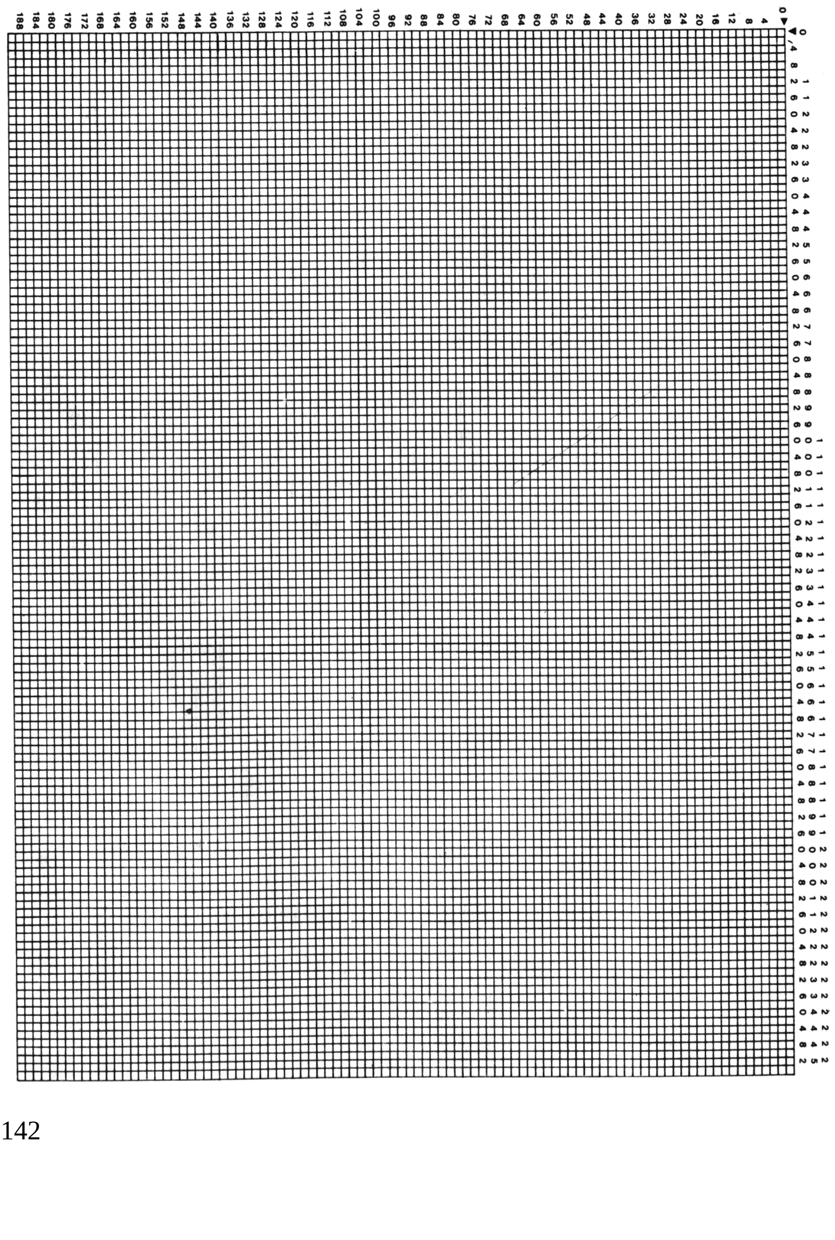
Print @ Grid

480	448	416	384	352	320	288	256	224	192	160	128	96	64	32	0	
																0
																1
																2
																3
																4
																5
																6
																7
																8
																9
																10
																11
																12
																13
																14
																15
																16
																17
																18
																19
																20
																21
																22
																23
																24
																25
																26
																27
																28
																29
																30
																31

Low Resolution Grid



High Resolution Grid



APPENDIX C

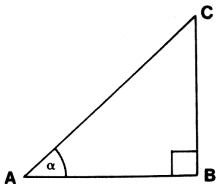
ERROR CODES

CODE	EXPLANATION
/0	Division by zero. Not possible.
AO	Attempt to open a file which is already open. Usually appears after pressing RESET to stop a program using files. Switch off and on again.
BS	Bad Subscript. Usually because the value of subscript is greater than the declared dimension of the array.
CN	Can't Continue. Trying to use CONT when at the END of a program.
DD	Attempt to redimension an array. Arrays can only be dimensioned once in a program.
DS	Direct Statement. Usually appears if you attempt to CLOAD a data file.
FC	Illegal Function Call. Usually parameter is out of range or the wrong variable type.
FD	Bad File Data. Caused by trying to read string data into string variable using cassette data files.
FM	Bad File Mode. Trying to INPUT from a file which is OPEN for output (O), or PRINT data to a file OPEN for input (I).
ID	Illegal Direct Statement. Attempt to use a statement which can only be used in a program e.g. INPUT, DEF FN.
IE	Attempt to input past the end of a file. Use IF EOF (1) to check this does not happen.
IO	Input/Output Error. Cassette not adjusted correctly or bad tape.
LS	String too long. Maximum is 255 characters.
NF	NEXT without FOR. Usually occurs when NEXT statements are reversed in a nested loop.
NO	File not open. Input and output to a data filo can only take place after OPEN.
OD	Out of data. A READ statement has read all the DATA statements.
OM	Out of Memory. All available memory is being used or has been reserved.
OS	Out of String Space. Use CLEAR to create more space if available.

CODE	EXPLANATION
OV	Overflow. The number is too large for the computer to handle (ABS(X)>1E38).
RG	RETURN without GOSUB. Program has most likely fallen through the end into the subroutine, (use END) or a branch has been made into the subroutine.
SN	Syntax Error. Usually caused by typing errors or incorrect punctuation.
ST	String formula too complex. Break the operation into smaller steps.
TM	Type Mismatch. Attempt to assign string data to numeric variable or vice versa.
UL	Undefined Line. A branching statement has been directed to a line that does not exist.

APPENDIX D

TRIGONOMETRIC FUNCTIONS



In a right angled triangle ABC,

AB is called the side *adjacent* to the angle á .
BC is called the side *opposite* to the angle á .
AC is called the *hypotenuse*.

The SINE, COSINE and TANGENT of the angle á are defined as follows:

$$\text{SIN á} = \frac{\text{opposite side}}{\text{hypotenuse}}$$

$$\text{COS á} = \frac{\text{opposite side}}{\text{hypotenuse}}$$

$$\text{TAN á} = \frac{\text{opposite side}}{\text{adjacent side}}$$

ALL BASIC trigonometric functions are assumed to be measured in *radians*. A *radian* is a measure of an angle in circular units. There are 360^{o} degrees, or 2 ð *radians*, in a circle. (ð is a Greek letter, pronounced "pie", representing a constant numbert 3.1415926). So convert from one to the other as follows:

degrees/(180/ ð) = radians
radians★(180/ ð) = degrees

The inverse of a function is the reverse of applying a function.
For instance, the tangent of angle of 1.5 radians is,

TAN(1.5) = 14.101419

The inverse of the function TAN is ATN. This is used to find the angle, if you already know the tangent,

ATN(14.101419) = 1.5

The inverse of the SIN and COS functions, are not available in BASIC, but can be found by using the ATN function in the following formulae:

```
   Inverse Sine  =  (ATN(X/SQR(   X*X*+1))
Inverse Cosine  =  (ATN(X)SQR(   X*X+1))+ 1.5708
```

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

Page No.

INDEX

INDEX

Page No.

INDEX

DRAGON - ADDITIONAL INFORMATION

This booklet contains some extra information which we hope will enable you to get the best out of your Dragon microcomputer. Particularly important is the paragraph concerning the cassette recorder lead, as this differs from the information given in your Dragon "Introduction to Basic Programming" manual.

information on the connection and adjustments of your television is also included with details of the printer connection and a copy of the Dragon memory map.

TELEVISON

Once you have connected your computer to a television anti switched on, select a spare channel control and tune in, as given in the "Introduction to Basic Programming" manual, lithe television picture is not stationary then it is necessary to adjust to vertical hold on the television set. If there is no external control for the vertical hold, contact your television service engineer for assistance.

To obtain a picture of clarity adjust the contrast, brightness and colour according to personal choice.

CASSETTE

To connect the cassette recorder to the computer using a Dragon Data cassette lead, put the DIN plug into the socket marked TAPE on the left side of the computer. The three plugs on the other end of the lead are connected to the cassette recorder as follows:

ii) The smallest jack plug with blue wire fits into the small jack socket usually marked REM and next to the microphone socket.
ii) The jack plug with the red wire fits into the socket usually marked AUX or MIC or LINE IN. If there is choice of socket between AUX and MIC, always use AUX.
iii) The _jack plug with the white wire fits into the socket marked EAR or MONIT, or L/S or SPKR.

When using the rewind or forward wind controls on the cassette recorder, it may be necessary to remove the jack plug from the REM socket jar these controls to operate.

Always ensure that this plug is reinserted afterwards.

Alternatively, type MOTOR ON and press ENTER before using the rewind or forward wind control. Afterwards type MOTOR OFF and press ENTER to continue using the cassette recorder in con junction with the computer.

If you wish to reuse a cassette tape it is recommended that the complete tape is erased before re-recording.

PRINTER

The printer point provided on the left hand side of the Dragon computer is for a printer using a parallel centronics type interface (socket 6 in illustration of "Introduction to Basic Programming" manual). The pin connections are as follows.

PIN 1	$\overline{\text{Print Strobe}}$	PIN 2	+5 volts
PIN 3	Data bit 0	PIN 4	+5 volts
PIN 5	Data bit 1	PIN 6	0 volts
PIN 7	Data bit 2	PIN 8	0 volts
PIN 9	Data bit 3	PIN 10	0 volts
PIN 11	Data bit 4	PIN 12	0 volts
PIN 13	Data bit 5	PIN 14	0 volts
PIN 15	Data bit 6	PIN 16	0 volts
PIN 17	Data bit 7	PIN 18	0 volts
PIN 19	$\overline{\text{ACK}}$	PIN 20	BUSY

The position of the odd numbered pins are on the top line of the connector part with PIN I situated on the right (viewed end on). The even numbered pins are on the bottom line with PIN 2 situated on the right.

CARTRIDGE

It is advisable to ensure that the power is switched off when inserting or removing a cartridge from the port on the right hand side of the computer.

DRAGON MEMORY MAP

Decimal Address	Contents Hex	Address
0 - 1023	System Use	0 - 3FF
255	Direct Page RAM	0FF
1023	Extended Page RAM	3FF
1024 - 1535	Text Screen Memory	400 - 5FF
	Graphic Screen Memory	
1536 - 3071	Page 1	600 - BFF
3072 - 4607	Page 2	C00 - 11FF
4608 - 6143	Page 3	1200 - 17FF
6144 - 7679	Page 4	1800 - IDFF
7680 - 9215	Page 5	1E00 - 23FF
9216 - 2559	Page 6	2400 - 29FF
2560 - 12287	Page 7	2A00 - 2FFF
12288 - 13823	Page 8	3600 - 7FFF
32768 - 49151	Basic Interpreter	8000 - BFFF
49152 - 65279	Cartridge Memory	C000 - FEFF
65280 - 65375	Input/Output	FF00 - FF5F
65376 - 65503	SAM Control bits	FF60 - FFDF
65504 - 65535	MPU vectors	FFE0 - FFFF

Release Notes

'An introduction to BASIC programming' PDF version 1.0 by R.Harding
Dragon Data Archive

Major Know Problems/Issues

- Zeros are not slashed.
- Vertical green bars missing from command description pages.

Acknowledgements

Original text version by Miguel Durán Uña biblioteca8bits