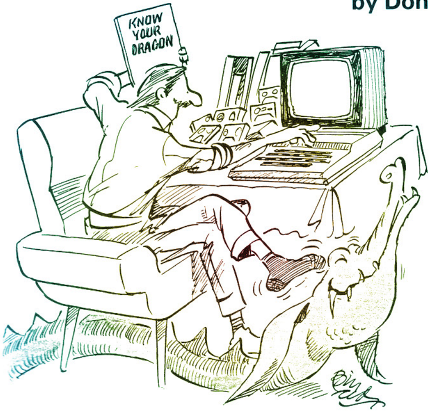
Know your DRAGON

a friendly guide to a friendly computer

by Don Monro



Illustrated by Bill Tidy

To obtain the cassette called '95 Programs from Know your DRAGON' which is designed to accompany this book, please send £ 6.50 to include postage and packing to: The Tiny Publishing Company, PO Box 120, Haywards Heath, West Sussex, RH16 2TG, England. Please allow 28 days for delivery.

The Tiny Publishing Company

DRAGON DATA LTD.

Published under licence
by The Tiny Publishing Company Ltd.,
PO Box 120, Haywards Heath,
West Sussex, RH16 2TG

ISBN 0 907909 02 7

To Woodstock
and all who sailed in her 1979-1981

Printed and bound in Great Britain
by Billing and Sons Limited
Worcester

Preface

I believe that this book fills a need for a serious but friendly learning aid for people who wish to master computer programming in BASIC and are lucky enough to have access to a DRAGON 32 Computer. This new machine is very generous with its memory and offers outstanding features at an affordable price. I think that my book offers a 'structured' introduction to the DRAGON 32 and also gives you lots and lots of programs to help you learn. In fact when I sat down to compile the cassette tape that you can get to go with the book, I was staggered to find that I had to put 95 programs on it. That's got to be pretty good value also, and you can order it — see the title page for details. I have also tried to make my book a good reference guide.

I wrote this book in Verbier, Switzerland while I was laid up with numerous injuries and the rest of my family skied. I have an optimistic theory that behind every misfortune lurks a hidden opportunity, and I hope you like the results. Charlotte Crewe typed the manuscript for me, and Malcolm Clarke exhausted the North Sea's reserves of midnight oil putting it all into its final form. Thanks, kids.

Bill Tidy is the worlds greatest illustrator, and has produced some side splitting observations which do so much to enliven the text. Once again, thank you Bill Tidy.

And so Dear Reader, over to you .

Contents

One

WELCOME

1 More than just a toy

All around us the computer revolution is taking place. It is the most important social and cultural change since the industrial revolution 150 years ago. Some people think that this revolution is taking mankind into a new historical age. It all began about 40 years ago with bulky, expensive monster computers that could do very little compared to modern machines. Since then the size and cost of computers has been falling faster and faster, recently because of the silicon chip which is at the heart of today's computers. The DRAGON 32 is a real computer which is capable of all the tasks that bigger, more expensive machines used to do. It is one of the very first real computers that most households can afford. Very soon the ability to use and program a computer is going to be a basic skill as important as reading and writing. If you have a DRAGON, you're a kind of pioneer; you and your family have a great educational opportunity.

It would be a great pity to miss out on this. It is all too easy to buy a DRAGON, play with it for a while using a few programs or games that someone else has written, and then let the whole thing be discarded like some forgotten toy. The DRAGON is not a toy, it is a real computer. Since computer skills are already in great demand, you owe it to yourself to make the most of it. This book is intended to help unleash the computer experts hiding inside real people, by helping them learn to program a real computer (the DRAGON) using a real programming language (BASIC).

2 What's so special about computers?

Nothing much. A computer is a machine that can do a few quite ordinary things, but it is fast and does not make mistakes. It does arithmetic, and always gets the answers right. It remembers things perfectly, and this means that it can remember the orders you give it and repeat them perfectly forever, or at least until you stop it. It can compare things and decide what to do as a result. These are very simple things. All the fuss about computers is because they are new. There is nothing difficult about them, only different.

3 What is BASIC?

Computers obey instructions. A computer program is a list of instructions given to a computer. These instructions are given in a language that people can understand, but which is adapted to the requirements of computers. Almost anyone can learn to program a computer, and BASIC is an excellent language for this purpose. The word BASIC stands for 'Beginners All-purpose Symbolic Instruction Code' - which means that it was originally intended for learners. It is a simple, even friendly language as languages go, and this is a good thing for you and for the people who make computers because BASIC goes well with small machines. Because of the rapid growth of 'microcomputers' like the DRAGON, it has become a very important computer language.

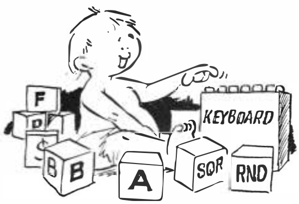
4 This book is for you

This book is intended to help you learn. It is not a library of programs, either useful or useless. I want to show you how easy it is to do your own BASIC programming, and to help you teach yourself. I won't deny that there are a few hurdles along the way, but together we can take them in our stride. All you need is your DRAGON computer. You don't need a cassette unit, although round about Chapter 9 you would begin to find one useful for saving programs. To make the typing easier, a cassette is available with all the programs from this book that are 8 lines or longer. Although I try not to write very long programs, some of my most entertaining and informative ones are long enough that you probably would not enjoy typing them. If you can't get the cassette from your DRAGON dealer or bookshop, you can order it. Details are given on the title page at the beginning of the book.

You already know how to connect the DRAGON to your television and get it running. Probably you have run a few programs on it - but this is not essential. All you need now is to start on the next page and follow it through. When you reach the end you will know all about BASIC as it applies to the DRAGON 32 Computer without extras. Good luck!

Two

FIRST THINGS FIRST



1 A real little program

Let's start by trying to be useful. Here in Switzerland the cuckoo clocks are famous. You and Sonia went to the local tick tock shop to buy some for your ranch. You actually bought 23, but 9 of them kept poor time and so you took those back. On the second trip you got a bit carried away and came back with 17 new ones.

How many do you have now? Of course you could work this out yourself, but here is a little BASIC program on two lines that you could use to make your DRAGON do the hard arithmetic for you:

```
10 PRINT 23-9+17
20 END
```

From these small beginnings great programmers grow. You and the computer have to understand each other, and that is the purpose of the BASIC language. The great thing about BASIC compared to other languages is that you need only a few facts to start using it.

In BASIC you write a series of instructions telling the DRAGON what to do, and in what order. The one above tells it first of all to work out and 'print' the sum 23−9+17 and secondly to stop.

You can see that each of the lines begins with a line number:

`10 PRINT 23-9+17`	is line number 10
`20 END`	is line number 20

The line numbers tell you and the computer in what order to obey the instructions given by the program. Therefore every line of a BASIC program has to begin with a line number.

Each numbered line of a BASIC program is called a statement. There are two statements in this simple example, the PRINT statement and the END statement. The PRINT statement asks the computer to work out the sum 23 − 9+17 and print the answer. The END statement indicates that the program is finished. On the DRAGON, the END statement is not essential (on some computers every program must end with an END, but not on the DRAGON).

2 Take this, DRAGON!

To use a computer program, you first have to get it into the computer. This is what the keyboard is all about. It is laid out very much like a typewriter, but some of the special symbols such as '+' are not where a typist would expect them and there are some extra keys. There is a special key called ENTER near the right hand end of the keyboard. This is a very important key as you are about to discover.

All you have to do to get the program into the computer is type it, beginning each line with its line number and ending each line with the ENTER key. The number of blank spaces inserted between things (or left out) during typing is unimportant. For example,

```
10 PRINT 2 + 3
```

and

```
10PRINT2+3
```

and

```
10 PR  INT 2+3
```

mean the same thing.

The order in which lines are entered is not important either, because the line numbers say what the real order should be. It is usual to separate line numbers by 10, since a programmer may later wish to insert extra lines, although any separation is allowed.

EXERCISE:

Type the little program into your DRAGON. Be careful to begin each line with the line number. At the end of each line type ENTER. If you make a real mess of it, switch the computer off and on again and start over.

3 Let's look — the command LIST

It is very likely that you will make mistakes when you type BASIC programs at the keyboard. It is important to be able to examine a program as the computer sees it, because it is not always all there on the screen for you to look at. So we should get used to using the LIST command right from the beginning. With it, the DRAGON can be asked to display the latest version of your program.

EXERCISE:
Type in LIST and push ENTER to obtain a listing of the program.

How did the computer know that LIST was a command? Simple: statements of a BASIC program always begin with a line number. Anything that does not begin with a number is a command. If you type in a statement of BASIC and forget the line number, your DRAGON will think it is a command.

Actually, there is a bit more to the LIST command, which you won't find useful just yet. You can list any part of a program. When you need this, look it up in the Appendix.

4 Now correct our errors — line replacement

Because everyone will make typing errors, it is necessary to be able to correct them. In BASIC it is always possible to change lines, insert new lines, and delete unwanted ones. This is easy because all lines of a BASIC program begin with line numbers, and whenever a new line is typed it becomes part of the program. Here's how:

(i) To replace or correct a line: just type it again and push ENTER.

EXAMPLE:
Your program is

```
10 PRONG 23-9+17
20 END
```

and you type in

```
          1Ø PRINT 23-9+17          and push ENTER
```

The program is now correct, and is

```
          1Ø PRINT 23-9+17
          2Ø END
```

which you can check by the LIST command.

(ii) To insert a new line: type the new line and give it a line number that gets it in the right place. As an example, some new lines could be added to the same little program.

EXAMPLE:
With the same program already in the computer, you type

```
14 PRINT 64+5
16 PRINT 73-2
```

and the program then reads

```
1Ø PRINT 23-9+17
14 PRINT 64+5
16 PRINT 73-2
2Ø END
```

The DRAGON has looked at the line numbers and put the new lines in the correct place. This is why the original line numbers were spaced by 10. This is also why the lines of the program could have been typed in any order. The DRAGON will arrange your program by its line numbers.

(iii) To get rid of a line: type in only the line number and push ENTER.

EXAMPLE:
To delete those extra lines again, type

```
14          (just the line number and ENTER)
16          (ditto)
```

The program again reads

```
1Ø PRINT 23-9+17
2Ø END
```

This method of correcting a program is called 'line editing'. You can also edit on the screen as will be described soon.

EXERCISE:

Experiment with the line editing facilities. Be sure that you know how to add, change, and delete lines. You can check out each change you have made with the LIST command. Finally put the program back to its given form and list it to be sure.

5 All systems go — the command RUN

I hope you are patient enough to have gone through all of this very carefully. If so, you have made a good start. But I expect you and Sonia are anxiously waiting for the answer.

So far we have created a little program and learned to list and edit it. Now we want to try it. At last! The command RUN launches your program. When the RUN command is given, the DRAGON begins to obey the instructions given by the program.

Once your computer has taken over it is not possible to edit the program or use other commands until it is finished. The little example won't hold you up for too long! It will give up at the END statement. If you have made an error in the grammar of the program, the DRAGON will tell you by saying

```
?SN ERROR IN 1Ø
```

if the error is in line number 10. SN stands for 'syntax' which is just a difficult word for 'grammar'.

EXERCISE:

Type in the command RUN. If it doesn't work it is not typed correctly. Fix it if that happens. Where are the answers printed. Isn't your ranch going to be a bit noisy particularly at noon?

6 Screen editing

While you have been using your DRAGON, you will have noticed a small rectangle on your screen. It always seems to be just to the right of whatever you have just typed or on top of the thing you are about to type. This is called the 'cursor'. This can do two useful things for you while you are entering a computer program. Everyone makes mistakes in typing, and very often you will spot an error just after you make it. It is possible to move the cursor backwards in the line that you are typing and this helps you to correct mistakes. To move it backwards, you use the 'back arrow" or ← key.

EXAMPLE:

What you really want is

```
        10 PRINT 2+2
```

but you notice that you have just typed

```
        10 PROMY█
```

with the cursor just to the right of the Y. Press← three times and you will wipe out the three letters that are wrong. Then carry on typing to get it right. Try this.

You may have made such a mess of a line that you want to chuck it away altogether before you reach the end of it. You could ENTER it and then type it again. Or you could hold down the SHIFT key and press←: This will throw away the line.

EXAMPLE:

You have typed

```
        10 SPRONG█
```

and the cursor is still next to the G because you have not pressed ENTER. Hold down SHIFT and press←: Magic! Try this.

7 A Very Flashy Editor

Your clever DRAGON has an additional editing facility which lets you change, delete, and insert characters in a program that already exists.

(i) Getting the Editor

Suppose you have a program

```
        10 PRANG 23-9+17
        20 EGG
```

This isn't going to work too well if you try to RUN it. Let's edit line 10. To do that, type in EDIT 10 and press ENTER (of course) The computer will display line 10 on the screen and repeat the line number underneath, like this:

```
        10 PRANG 23-9+17
        10 █
```

Now type a series of spaces. Whoops! Line 10 is copied one character at a time. What you are doing is editing line 10 without changing anything. When you get to the end, it won't let you go on until you press ENTER. You could have pressed ENTER earlier, while in the middle of the line. Again, nothing would have changed.

So try this: get line 10 with the EDIT command, space to the middle of it and press ENTER.

(ii) Changing some characters — nCxxx

In line 10 we really want to change 'ANG' to 'INT'. To do this, get line 10 on the screen with

```
EDIT 1Ø
```

and space the cursor until it's under the A:

```
1Ø PRANG 23-9+17
1Ø PR█
```

Now type the characters '3CINT'. Aha! The 'edit command' 3C tells the editor that you want to change 3 characters, and 'INT' are the three replacement characters. Try it. After you have changed the characters, press ENTER. LIST the program to make sure. Now change EGG to END.

So now you know that under the editor, the nC command will change n characters — n is a number. And there's more!

(iii) Delete some characters — nD

With the same program, you could put in

```
EDIT 1Ø
```

again, and get

```
1Ø PRINT 23-9+17
1Ø█
```

on the screen — at least you would if you had used the character change command above. Now we want to remove the term −9 just for fun. Space along until the cursor is under the minus sign:

```
1Ø PRINT 23-9+17
1Ø PRINT 23█
```

And type 2D. That deletes 2 characters. You won't see anything happen until you space along a bit, or press ENTER. Wow! So now you also know that the nD edit command will delete n characters. What more could you ask? Well, read on.

(iv) Insert something — I

If you did the last section, the program is now:

```
1Ø PRINT 23+17
2Ø END
```

Now let's put the −9 term back into line 10. Type

```
EDIT 1Ø
```

and as usual you will see

```
1Ø PRINT 23+17
1Ø █
```

Space along until the cursor is under the + sign:

```
1Ø PRINT 23+17
1Ø PRINT 23█
```

Now type the letter I. Nothing will happen on the screen, but anything else you now type will be inserted before the + sign. So all you want to do is type −9 and press ENTER. Fantastic! The I command starts inserting. Press ENTER when youv'e finished inserting, or if you want to carry on editing the same line press SHIFT and ↑ as explained in (x) a bit later. These are the basic essentials, but there are some more editing commands that can sometimes be convenient. If you find that this is getting too complicated, come back to it later.

(v) Splice something on the end — X

Type in EDIT 20, and you see

```
2Ø END
2Ø █
```

Now type in X, meaning 'extend'. The cursor jumps to the end of line 20 and anything else you now type is grafted on to the end of line 20. Make it say

```
2Ø END OF THE WORLD
```

If you're not supersititious, try running the program:

```
10 PRINT 23-9+17
20 END OF THE WORLD
```

Did the world end? Computers can't do every thing. Fortunately.

(vi) Another insertion — H

The H command lets you delete unwanted things from the end of a line and splice new things on in their place. You can change

```
20 END OF THE WORLD
```

to

```
20 END OF THE LINE
```

as follows: First type in

```
EDIT 20
```

and you see:

```
20 END OF THE WORLD
20 █
```

Space along until the cursor is under the W in WORLD:

```
20 END OF THE WORLD
20 END OF THE
```

Now type the letter H and the word LINE. Press ENTER and it's done. This can be useful.

You can also delete nonsense from the end of a line this way. To correct line 20, type

```
EDIT 20
```

and get

```
20 END OF THE LINE
20 █
```

Now space along until you have

```
          20 END OF THE LINE
          20 END
```

and type only H and press ENTER. You have ripped the rubbish off the end of the line — you could have inserted something but you haven't bothered.

(vii) Searching along — nSx

When you are editing a line you can ask the DRAGON to scan along the line looking for the nth occurrence of a particular character. Suppose you enter

```
     10 ABABABABAB
```

and then put in

```
     EDIT 10
```

which gives on the screen

```
     10 ABABABABAB
     10 █
```

Enter 3SB and the editor will find the third B along from where the cursor is now:

```
     10 ABABABABAB
     10 ABABA█
```

and you can now do anything you want with one of the other commands.

(viii) What have I done? — L

In the middle of a complicated editing operation, you may want to see exactly what you've got. The L command makes the computer display your line again and you can carry on editing it. If you have, as above

```
     10 ABABABABAB
     10 ABABA█
```

you can enter, say 3DL and you will see:

```
     10 ABABABABAB
     10 ABABAAB
     10 █
```

you have taken three letters out of line 10 and are still editing it.

(ix) Zip the cursor around in the line

With the cursor in the middle of a line, you can zap it back and forth. Consider this fairly long line:

```
10 THIS IS A FAIRLY LONG LINE
```

Put in EDIT 10 and get the cursor under the F of FAIRLY by either spacing or 1SF:

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A █
```

Now put in 5 and SPACE. Youch! You have

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A FAIRL█
```

Because you zapped the cursor along by 5 spaces. Now enter 10←. Zounds ! You have zipped it back 10 spaces.

```
10 THIS IS A FAIRLY LONG LINE
10 THIS █
```

Now try 30 SPACE and 50 . What happens?

(x) Escaping from an edit command

Usually you would press ENTER when you are finished with a line. However, if you are inserting using I and want to stop inserting, you may find it useful to enter SHIFT and ↑. This enables you to escape from inserting and carry on editing the same line. For example, edit the line

```
10 THIS IS A FAIRLY LONG LINE
```

by entering EDIT 10. Space forward to get the cursor under the F in FAIRLY:

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A █
```

Now enter I and type in QUITE with a space after it:

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A QUITE █
```

To escape from inserting, press SHIFT and ↑, then enter 12 and SPACE. You should have

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A QUITE FAIRLY LONG █
```

As you can see, we have the cursor at the right hand edge of the screen. Don't panic! Type in I to get you into insertion mode again, and type the word SILLY with a space after it.

```
10 THIS IS A FAIRLY LONG LINE
10 THIS IS A QUITE FAIRLY LONG S
ILLY █
```

Now press ENTER. You can see that the DRAGON doesn't care if you go past the end of the screen, your line carries on. Neither BASIC nor the editor have any trouble with lines that 'wrap around' the end of the screen.

EXERCISE:

Use screen editing to practise replacing characters, deleting characters, and inserting them. This is a very powerful editing facility. Have a bash at it before going on, but remember that you can always refer back to this section if you need it.

8 And for your last trick — look in the Appendix

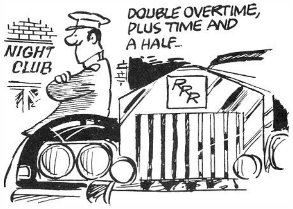
This is a good time to introduce you to the summary of BASIC at the end of this book. If you ever need reminding of the rules for any part of BASIC, look them up in the back. Occasionally you will find extra information there. For example, I haven't told you everything about the LIST command! Using it you can LIST any part of a program. There are also commands for DELeting and RENUMbering the lines of a program. The editing information is repeated there also. From time to time you will get cryptic error messages on the screen — like

```
? SN ERROR IN 10
```

Look those up in the Appendix also.

Three

THE THIRD R



1 It's 'Rithmetic of course

One of the things computers can help us with is arithmetic — they can be made to do the hard work in adding up our accounts and working out (shudder) our taxes. That is one reason why many people want to learn BASIC — so they can program these things themselves. So we need to know what we can expect. It's very easy, actually, although there's a little sting in the tail.

2 Adding and subtracting

In the previous chapter, you and Sonia did a sum:

```
10 PRINT 23-9+17
20 END
```

Actually, the DRAGON doesn't need the END statement although it is a good idea to know about it because some computers demand that BASIC programs must have an END statement as the last line. You could have only

```
10 PRINT 23-9+17
```

Numbers like 1, 2, 23, 9, 17, and so on, are called constants. Our program had the expression 23−9+17 which added and subtracted the constants 23, 9 and 17. Everyone knows that the order in which you add numbers is unimportant: 5+1 is the same as 1+5. This is not true for subtraction, as 5−1 is not the same as 1−5. Therefore the order in which you write numbers and operations like addition and subtraction is important. In BASIC, operations are done from left to right, just as we have all learned to do arithmetic. The meaning of

```
10 PRINT 23-9+17
```

is obvious to you, me, and the computer. The symbols + and− also have a meaning if they come before a number, as in

```
      10 PRINT +99    or    10 PRINT -73
```

and on the DRAGON you can do something like

```
          10 PRINT 6+-+-5
```

What does it mean? Try it to make sure.

3 Multiply and divide

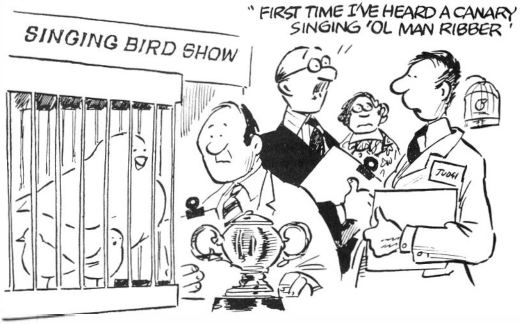
In BASIC the symbol * is used for multiplication. This is written between the numbers to be multiplied. Like addition, multiplication is not order-dependent.

EXAMPLE:

There are 2.204 pounds to the kilogram. How many pounds does a nine kilogram canary weigh?

This will do it: `10 PRINT 9*2.204`

Here there are constants with decimal points in them. You can always do this in BASIC.



EXERCISE:

In 1974, my publisher sold 318 copies of my book. For each one I received a royalty of 0.24 interplanetary credits. How much did I get? Will I soon be rich?

Numbers are divided when the symbol / is used. The order of the numbers does matter, since

15/3 is 5   and 3/15 is 0.2

EXAMPLE:

How many kilograms does a nine pound canary weigh? Do this:

```
10 PRINT 9/2.204
```

Do you believe in nine pound canaries?

EXAMPLE:

It's a long long way to San Jose; 434 miles to be exact. At 56 miles per hour the highway patrol will probably not stop me. How long will it take to get there? This is it:

```
10 PRINT 434/56
```

EXERCISE:

You can do 29.2 miles to the gallon and you have 14 gallons left. Can you make it to San Jose?

4 Raise a number to a power

There is one more operation in the arithmetic of BASIC, the raising of a number to a power. The symbol for this is the vertical arrow, ↑.

5↑3 means	5*5*5 or 125
3↑5 means	3*3*3*3*3 or 243

There are also fractional powers; for example

2↑0.5 is the square root of 2, or 1.4142...

EXAMPLE:

You bet one interplanetary credit and throw the dice 12 times. Each time you double your money. How much have you won? Ask the computer:

```
10 PRINT 2↑12
```

On the 13th throw you lose it all. Tough.

EXERCISE:

As you were leaving old St Ives, you met a man with seven wives. Every wife had seven brats and every brat had seven cats. Every cat had seven kits and every kit had seven lives. How many kit's lives were going to St Ives? The answer is 16807. Can you get it?

5 'Rithmetic stew — expressions

Now is the time to throw all the operations into the pot at once. This has been delayed because there is a complication.

After a successful seven-a-side football match, your team is awarded 3 pints of grog each plus another 11 to share between you. How much grog does each player get? The computer will tell you:

```
10 PRINT 3+11/7
```

Is it really evaluated from left to right? If so, it would mean

`10 PRINT(3+11)/7` with the result 2. You've been cheated!

Or does it mean instead

`10 PRINT 3+(11/7)` with the result about 4.57?

I think that you can see that the result you want is 4.57. Check that this is what the DRAGON gives you. You can probably see that expressions are not evaluated from left to right — it is a bit more complicated.

It is necessary to have an exact set of rules about this arithmetic so that both you and the computer know what should happen. In BASIC some operations are given a higher priority than others. The order is:

( )	things in brackets	highest
↑	raising to a power	
* /	multiplication and division	
+ −	addition and subtraction	lowest

The computer will always look at an expression and do the things you have put in brackets first. Of course inside the brackets could be all kinds of things including more brackets, which have to be done first!

Among the actual arithmetic operations, powers are done first. Then the multiplications and divisions are done from left to right. Finally, the additions and subtractions are carried out, again from left to right. You can always force the machine to do things in the order you want by using round brackets. If you do this, the brackets must always be in pairs — one right bracket for every left bracket, as

```
        10 PRINT 1+2*(3+4*(5+6))
```

You can write a sequence of plus and minus signs, and BASIC will work out what you want — the expression 1 + − + −2 is allowed and has the value 3; similarly 1 +(−2) is permitted because anything in brackets is a self-contained expression and can have + or − in front of it. However you cannot write the * or / operations on their own. Therefore −3 is a correct expression on its own but *3 is not. Similarly 2/*3 is nonsense.

This business of priority in arithmetic is the sting in the tail I referred to earlier. People have a tendency to get caught, particularly by division. Notice that

7/3+4 is not 1 and 7/(3+4) is 1.

EXAMPLE:

Stevastian Covett runs the mile in 3 minutes and 40 seconds. What is his average speed in miles per hour? This is slightly more complicated because the time is in minutes and seconds. But we can convert this to hours and get the average speed:

```
        10 PRINT 1/((3+40/60)/60)
```

EXAMPLE:

An Acapulco diver jumped off a cliff and hit the water after 7.72 seconds. How high was the cliff?

The formula for falling under gravity, ignoring wind resistance, is

$$\text{distance} = \frac{1}{2} \text{ x acceleration x time}^2$$

We have to use the right units. Working in seconds and metres, the acceleration under gravity is 9.81 metres/second/second, and so the answer is:

```
        10 PRINT 0.5*9.81*7.72↑2
```

Do you see how I managed to square the time?

EXERCISE:

You do this one. Fernando deposited one interplanetary credit with the Interstellar Bank just over 7 years ago. Four times a year, Interstellar credited him with 4% interest. How much has he now? If you can't do it, compound interest is used as an example in the next chapter.

6 All this and a calculator too — direct expressions

This is not part of the normal BASIC language, but many small computers will do it, including your DRAGON. The difference between a line of a program and a command is the line number. You know that

```
        10 PRINT 5↑3
```

is a line of a BASIC program. What happens if you type just

```
        PRINT 5↑3
```

Oh dear! I suppose you think I should have told you about this earlier. It turns out that your DRAGON will obey many statements of BASIC immediately if you type them in as a command — they call this 'direct mode'. It is useful, but not powerful because you would have to get your entire program on one line — which in theory you can — but it gives you no opportunity to correct it. Use it to get quick results, as from a calculator, but don't use it to do more serious programming.

As a shorthand you don't have to type the word PRINT — you can use a question mark. Try this:

```
        ?2↑8
```

So try this:

```
        10 ?2↑8
```

When you LIST this you will get a big surprise. This is not a normal BASIC feature, but a handy little thing that the DRAGON does for you.

So you can actually use your DRAGON like a calculator. The only statements that are not allowed as direct statements are INPUT, which is introduced in Chapter 4, and DEF FN from Chapter 20. You know, the first hand-held calculators were nearly as expensive. And your friendly neighbourhood DRAGON 32 computer can do much, much more. After all, this is only Chapter 3!

Four

TALKING TO BASIC

1 Send yourself a message

By 'talking to BASIC' I do not mean that the machine can literally speak to you, or you to it — although that isn't so many years off. What I do mean is that you can easily include messages in your BASIC program that appear on the screen so that when you run a program, it can tell you what to do. Also, when the program is running, you can give it values to work on.

The programs of the previous chapter showed their results in a somewhat unhelpful way. In a complicated program a lot of numbers appearing on the screen with no explanation could be very confusing. In a PRINT statement, a message can very easily be displayed on the screen. This greatly improves almost any program.

EXERCISE:
Try this program.

```
        1Ø PRINT"HELLO YOU"
```

As can be seen from this example, a message can be produced by a PRINT statement simply by enclosing it in quotation marks. Be sure to use double quotes as shown. Whatever you put inside the quotation marks is displayed on the screen when you run the program. Earlier, you ran programs like

```
        1Ø PRINT 3+11/7
```

to help you divide up the grog. The program

```
        1Ø PRINT "3+11/7"
```

is quite different. Try it.

EXERCISE:
Now try this one.

```
        10 PRINT "YOUR SHARE"3+11/7
```

Can you see what this does?

Normally several items to be printed are separated by commas, as in

```
        10 PRINT "CATCH",2*11,2+2
```

but it is not necessary to include commas before or after messages in quotation marks. Here is a longer program which still doesn't do much. This one has two PRINT statements. You will see when you run this that each PRINT statement starts a new line on the screen.

```
        10 PRINT "TWO AND TWO"
        20 PRINT "ANSWER ="2+2
```

2 Inserting remarks — the REM statement

You can include statements in your BASIC programs which help to explain the meaning of the program to you, but which do not do anything when you run the program. The REM (for REMark) statements do this. It is always a good idea to use them, and in complicated programs they are essential.

The REM statement has the form

line number REM any remark or comment

and added to the simple program this could be

```
        10 REM A DEMONSTRATION
        20 PRINT"TWO AND TWO"
        30 PRINT"2+2 ="2+2
```

Are we going too fast? Run this program. Be sure you understand the difference between what a REM statement does and the message facility that goes with the PRINT statement.

REM statements help to explain programs. If you later add a cassette recorder or disk unit to your DRAGON, then you will save lots of programs that you have written. When you later return to them, the REM statements will help you to recall the details of your program in case you want to change it, or add to it.

3 Chucking old programs away — the command NEW

In the previous chapters, you learned how to create and edit programs, and to run them. Now you are going to want to make new programs that are more than one line long, and you don't want old programs hanging around to mess them up. There is a command that throws away your old program and lets you begin a new one. If you enter the command NEW (and push ENTER of course — you should be used to that by now) your old program is lost and you can start a new one. You can also clear the screen at any time — without losing your program — by pushing the CLEAR key.

Remember that the commands are summarized in the Appendix. So far you can LIST a program or any part of it. You can RUN a program. You can also DELete or RENUMber program lines. And now you can create NEW programs.

4 Giving values to a running program — the INPUT statement

You have seen how you can use BASIC to print numbers and messages. It is equally important to be able to give values to a running BASIC program yourself. This enables the usefulness of the computer to leap beyond what a mere calculator can do. In BASIC, a program is written using names for the values to be used, much as in algebra, and an INPUT statement in the program will ask you for the actual values when you run the program.

EXAMPLE:

You sell sea shells by the sea shore. Whenever a sea shell is sold you have to add on 15% sea shore tax (SST) which the SS tax police have ways of making you collect. So every time you sell a sea shell you ask your DRAGON how much to charge your customers. The price with the tax added on is 1.15 times the price before tax. Here is a program:

```
10 INPUT S
20 PRINT 1.15*S
```

When you run this program, the computer will ask for the price of the sea shells you are selling. The program calls this S. your DRAGON prompts you for the value of S by putting a '?' on the screen. When this happens, you should type in the sea shell price, push ENTER, and the DRAGON will tell you what the sea shore sea shell selling price is.

EXERCISE:

Run this program. When you are prompted, type in 100 and push RETURN. Did you get the answer 115? Good. Now every time you RUN this program it works out the sea shell sea shore selling price for you.

The INPUT statement, introduced here, is one of the two statements of BASIC on the DRAGON 32 that cannot be used as a 'direct' statement. It can only exist as a statement with a line number in a program. If you try to use it when you are pretending that the DRAGON is a calculator in direct mode, you will get the message 'ID ERROR'. If you look at the list of error messages in the Appendix you will see that it means exactly what would be expected — INPUT cannot be used in direct mode.

EXAMPLE:

Actually the sea shore tax program is lousy. Since we know about REM statements and how to print messages, we can make this a whole lot better:

```
10 REM TAX ADDER ONER
20 REM EXPLAIN PROGRAM
30 PRINT"HELLO BOSS, IF YOU ENTER"
40 PRINT"THE SEA SHELL PRICE, I'LL"
50 PRINT"ADD ON THE SS TAX FOR YOU"
60 INPUT S
70 REM WORK IT OUT
80 PRINT"WITH TAX THAT'S"1.15*S
```

When you type this in, you will see that some lines are a bit too long for the screen. Just keep typing and see what happens. Don't forget to push ENTER when you get to the end of the statement. Is your program OK? Will it RUN? From now on we don't have to be too bothered about the lengths of our BASIC statements.

This is also the first program that is long enough to go on the cassette that you can get to go with this book. No one minds typing short programs, I hope, but some of the best ones are longer. It is called '95 Programs from Know your DRAGON'. If your dealer or bookshop doesn't have it, you can order it. There are details on the title page at the beginning of the book.

EXAMPLE:

Your two kids have different numbers of marbles (the glass things, not brains). This is because the big one keeps winning. You're a socialist. The DRAGON will help you redistribute the wealth. You are going to tell it how many marbles each of the two has, and your computer is going to tell you how many they should have. Put another way, it will find the average of two numbers:

```
10 REM SOAK THE RICH
20 PRINT"HOW MANY MARBLES HAS EACH BOY"
30 INPUT X,Y
40 PRINT"RUPERT HAS"X"OLIVER HAS"Y
50 PRINT"TOTAL MARBLES ="X+Y
60 PRINT"GIVE THEM EACH"(X+Y)/2
```

OK, Solomon. If the total is odd, are you really going to cut one in half?

There is another good thing added to this program at line 40. The program 'echoes' the input. Do you see why brackets are used in line 60?

EXERCISE:

> Run this program several times. The INPUT statement at line 30 will prompt you with '?'. When you see this, you type in the values for X and Y with a comma between them and push RETURN. The answer should then appear on the screen. However, there are several things that you might do wrong. If you give too many values, that doesn't matter — the computer takes the first two and says 'EXTRA IGNORED'. But if you give only one value, the DRAGON will make you give more by prompting you again — this time with '??'.
>
> You get the message 'REDO' if what you type is not a number. Make these errors deliberately this time so that you can see what happens.

A good programmer gives a lot of attention to the dialogue between the program and the person who runs it. The 'personality' of the program depends on the care that has been taken in making it friendly — this is up to you.

5 So now you know about variables in BASIC!

In the previous section, values were given names like S, X and Y. These are 'variables' to which you can give names when you write the program, and set the values later when you run the program. There are lots of names for ordinary variables allowed by BASIC. These are the single letters A to Z, any combination of one letter followed by a number, such as A0, A1, ..., A9, B0, B1, ..., B9, and any combination of two letters such as XY, BQ, and so on. There are a few names that you cannot use because they mean something special to BASIC: ON, OR, GO, TO, FN, and IF.

You may notice that the letter O (oh!) and the number 0 (zero) are easily confused. This is why the number 0 appears with a line through it on the keyboard, and in all the programs in this book. On the screen, zero looks narrow and O (oh!) looks square.

6 Another way of giving values to variables — the assignment statement

To begin with, we could only do arithmetic with constants in a PRINT statement. Then we discovered how variables could be used instead, with their values given manually in response to an INPUT statement. Now we will see that there is a special statement for giving values to variables — the assignment statement.

Here is an example:

```
80 TX=1.15
```

This gives the value 1.15 to the variable TX, and we could add this to our tax collecting program:

```
80 TX=1.15
90 PRINT"WITH TAX THAT'S"TX*S
```

An assigment statement has the form:

line number variable name = expression

The expression can be more complicated, for example

```
80 C2=C1*(1+I/100)↑N
```

You will wish to know exactly what happens when an assignment statement is obeyed. First of all, the expression on the right hand side is worked out. The result then replaces the value of the variable on the left hand side. Because of the way it works, a statement like

```
66 I=I+1
```

is not a nonsensestatement, but something very useful as we shall see a bit later.

On many computers, the assignment statement includes the word LET, which helps to make it more self-explanatory:

line number LET variable = expression

such as

```
44 LET Z=X↑3
```

The DRAGON 32 computer will accept the word LET, although it is never necessary to have it.

EXAMPLE:

Let's convert pounds and ounces to kilograms. The number of pounds in a kilogram is 2.204. We'll make our program really friendly so that it introduces itself and explains that it wants to be given pounds and ounces so that it can convert them to kilograms:

```
10 REM A PROGRAM WHICH WILL
20 REM CONVERT POUNDS, OUNCES
30 REM TO SENSIBLE KILOGRAMS
40 REM GREET THE USER WARMLY
50 PRINT"HELLO YOU, I'M THE"
60 PRINT"FRIENDLY METRIC THING"
70 PRINT"AT YOUR SERVICE"
80 PRINT"GIVE ME A WEIGHT IN"
90 PRINT"POUNDS AND OUNCES"
100 PRINT"AND I'LL TELL YOU"
110 PRINT"WHAT IT IS IN KILOS"
120 INPUT LB,OZ
130 PRINT"THANK YOU SO MUCH"
140 PW=LB+OZ/16
150 KG=PW/2.204
160 PRINT"THAT MAKES"KG"KILOS"
170 PRINT"IT WAS TERRIFIC TO"
180 PRINT"SERVE YOU. PLEASE RUN"
190 PRINT"ME AGAIN ONE DAY"
```

The program first of all converts the given weight to pounds (there are 16 ounces in a pound):

```
140 PW=LB+OZ/16
```

and then works out the result in kilograms. You could use this program to work out how much your canaries weigh. If you try it, as you should, you will find that the 19 lines of this program more than fill the screen. So how can you LIST it? Look in the Appendix, at the LIST command. You can list whatever bits of it you want.

EXAMPLE:

Here is a money program. The computer will tell you how much money you get back from your investments.

```
10 REM COMPOUND INTEREST PROGRAM
20 PRINT"HOW MUCH DO YOU INVEST"
30 INPUT C1
40 PRINT"WHAT IS INTEREST RATE"
50 INPUT I
60 PRINT"HOW MANY PERIODS"
70 INPUT N
80 C2=C1*(1+I/100)^N
90 PRINT"RETURN IS"C2
```

This is based on the formula

$$r = c\,(1 + i/100)^n$$

where

c = capital invested
i = interest rate in percent for each investment period
n = number of investment periods, or the number of times the interest has been compounded
r = the return

EXERCISE:

Using this program, find out the number of periods taken to at least double an investment at a rate of 8%. Is it worth it with inflation like it is?

Five

OVER AND OVER AND OVER AGAIN

"...JR, NOW 169 YEARS OLD IS FRANTICALLY TRYING TO COVER UP THE FACT THAT...

1 Forever

We will see here how to make a program go on and on forever — and we will have to learn how to stop it!

The GO TO statement is the hero (or villain!) of the piece. This enables you to change the order in which the statements of a program are obeyed. Normally they are taken one after another, in the order of their line numbers. However, if you put in a GO TO statement, it jumps to somewhere else. Try this:

```
10 REM CHEEKY PROGRAM
20 PRINT "SO JUST YOU TRY AND"
30 PRINT "MAKE ME STOP ";
40 GO TO 30
```

Notice the semicolon at the end of line 30. This makes printing continue on on the same line. You should RUN this program, and when you do you will see that it seems to go on mocking you forever. This is because the GO TO statement at line 40 sends you back, to line 30 — again and again and again. Forever. Do not despair — nil illegitimati carborundum! stop it with the BREAK key. That's better! If you want it to resume, type in the command CONT. If you make some changes to the program between stopping it and trying to CONT, you can't CONT. Try it and see.

So far so good. A repeating program can be stopped by pushing BREAK. What if there's an INPUT statement? Let's see!

EXAMPLE:

A program in the previous chapter showed you how to add tax on to the price of sea shells. I'll let you in on a secret — it doesn't have to be sea shells, it will work for anything. More complicated is the problem of trying to deduce the price without tax if you know the price with tax.

Suppose a banana warmer costs 100 interplanetary credits with banana warmer tax at 15%. What did it cost before tax? If you think the answer is 85, you're wrong. Here is a program to tell you. Notice that because it has a GO TO statement in it it will keep on running forever, taking away banana warmer tax as many times as you want:

```
10 REM TAX TAKEAWAY
20 PRINT "THIS PROGRAM TELLS"
30 PRINT "WHAT YOUR BANANA"
40 PRINT "WARMER COST BEFORE"
50 PRINT "BANANA WARMER TAX"
60 PRINT "WAS ADDED ON."
70 PRINT "TYPE IN THE PRICE"
80 INPUT BW
90 PRINT "IT WAS" BW/1.15
100 PRINT "BEFORE TAX"
110 GO TO 70
```

Run this. If you type in 115, you should get 100 as the price before tax. Agreed? So what is the pre-tax price if the taxed price is 100? It's not even close to 85!

To stop this, push BREAK just as you did before. Try it.

I think you can see that a GO TO statement looks like this:

line number GO TO another line number

and it forces the program to jump to 'another line number' instead of carrying on in the normal order of line numbers.

2 Something tricky — self-replacement

Something cool happens if the same variable is used in both sides of a LET statement, such as

```
60 CT=CT+1
```

Here, CT has been used in calculating its own replacement value. You can use this for counting as in this program:

```
10 REM LEARN TO COUNT      50 REM REPLACE IT
20 REM SET FIRST VALUE     60 CT=CT+1
30 CT=1                    70 REM AND GO BACK
40 PRINT CT                80 GO TO 40
```

Notice what is required for counting. First a starting value is set at line 30. Then the counter CT has one added to it over and over again.

EXERCISE:
Run this program. Remember to push BREAK to stop it.

3 Adding up — your bank account

By using self-replacement, a number of useful things can be done. One of the most useful is to be able to add things up. To do this you set aside a variable to hold the sum and give it a starting value. Inside a loop we add to the sum each time around the loop.

Here is a program for adding up. It prints the count J, and the sum of all the numbers up to J. An INPUT statement stops it each time around.

```
10 REM COUNT AND SUM
20 REM SET STARTING VALUES
30 J=1
40 S=1
50 PRINT "COUNT IS NOW" J
60 PRINT "LATEST SUM IS" S
70 PRINT "PRESS RETURN TO GO ON"
80 INPUT X
90 REM GET NEXT VALUES
100 J=J+1
110 S=S+J
120 GO TO 50
```

EXERCISE:
You might like to add to this program

```
85 CLS 1
```

Useful! Remember this for future use. CLS means 'clear the screen' and 1 is for green.

EXAMPLE:
The computer can check your latest bank statement. This program asks you for the starting balance, and then you enter each transaction and get your new balance. It uses BB for your bank balance, and TR for each transaction.

```
30 PRINT "GIVE STARTING BALANCE"
40 INPUT BB
50 PRINT "NOW THE TRANSACTIONS"
60 PRINT "MINUS FOR WITHDRAWLS"
```

```
70 PRINT "PLUS FOR DEPOSITS"
80 INPUT TR
90 BB=BB+TR
100 PRINT "NEW BALANCE IS" BB
110 PRINT "NEXT"
120 GO TO 80
```

4 A better class of prompt

You may find it convenient to put a message in the INPUT statement, such as

```
80 INPUT "NEXT";TR
```

in the bank balance checker. This will include the message 'NEXT' in your computer's prompt for the value of TR. This would then make line 110 unnecessary, and the prompt would look better because the '?' will appear right after the word 'NEXT'. You can always have one message in an INPUT statement:

line number INPUT message ; variable list

You need the semicolon after the message.

5 Today's new word — recurrence

So what is this thing called recurrence? It is what happens whenever a variable is used for self-replacement. Therefore both counting and summing make a kind of recurrence. There are other kinds of recurrence.

For example, suppose an ageing snail is running out of energy so that it goes one metre today, half a metre tomorrow, a quarter of a metre the next day, and so on. If the snail lives forever, how far does it get? To work this out you need to do this sum:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

(Do you know what the answer is?) You could write

```
10 S=1                              40 N=N+1
20 N=0                              50 S=S+0.5↑N
30 PRINT "LATEST SUM" S             60 GO TO 30
```

If you want to slow this down, put in

```
35 INPUT X
```

so that you get one stop each time you press ENTER. However, if you were clever you would notice that the bit added on is half the bit added on before, i.e.

```
10 S=1
20 T=1
30 PRINT "LATEST SUM" S
40 T=T/2
50 S=S+T
60 GO TO 30
```

What we have done is notice a different recurrence which gives each new term by dividing.

```
40 T=T/2
```

Do you understand? The second version is clever, but is it any better? Well, it will be a bit faster because raising to a power is slower than dividing.

EXERCISE:

Run this latest snail program. Evidently the DRAGON thinks that the snail arrives. This is because eventually the new bit called T gets too small for it.

6 Rabbits certainly can breed!

Here is another fascinating recurrence. A long time ago a man nicknamed Fibonacci (1202 AD) was interested in population growth. He asked the question 'How many rabbits would be produced from a single pair in n generations?' He assumed that every month a pair of rabbits produces another pair, and that rabbits begin to bear young when they are two months old. Honi Soit Qui Mal Y Pense! This led to the famous Fibonacci series, usually expressed as a recurrence:

F_n is the number of pairs after n months. We use $F_1 = 1$ and $F_2 = 1$ also.

Here is a program to do this:

```
10 REM RABBITS RAMPANT
20 F1=1
30 F2=1
40 REM MAKE NEXT TERM
50 F=F1+F2
60 PRINT "BREED" F
70 INPUT X
80 REM REVISE OLD TERMS
90 F1=F2
100 F2=F
110 GO TO 50
```

OK, IF IT'S TO DO WITH POPULATION GROWTH WHY THE GREASY RAINCOAT?

When you RUN this program, you breed a new generation of rabbits every time you push ENTER.

All these fascinating things arise from the ability of a BASIC program to loop back on itself using the GO TO statement, and from the fact that a variable can be used to calculate a new value for itself. This is the meaning of recurrence. Do you see what is done in statements 90 and 100?

If you run this program, you are going to see some large numbers begin to appear after several generations. Soon, the numbers get too big for to put on the screen. When this happens, the display will switch to 'scientific notation' in which you see something like

```
1.13490317E+09
```

This means that the population is now 1.13490317 multiplied by 10^9. The 'E+09' means to multiply by 10 raised to the power 9, which is the same as moving the decimal place 9 places. The answer then is approximately 1134903170 — approximately because we do not actually know what the last digit is. The computer doesn't know either, because it can only store nine digits.

If you remove line 70 and let the program run on even further, you will get to a point where the answer becomes too large for your DRAGON. How large is that? You can see that it can handle quite an impressive range of numbers.

Six

DECISIONS, DECISIONS

1 Comparing things — relational expressions

I hope that you have found everything that we have done so far to be fairly easy. Because now is the time to take a bit of a jump. So far it has been easy to follow programs from line to line because they have either been taken in order or formed a loop. Now is the time to let a program make a decision.

To do this, a new kind of expression gives us the answer TRUE or FALSE when we make a comparison. For example, you may have a variable BA which is the balance of your bank account. You want to know if you can afford that new interstellar scooter which costs 10000 credits. The expression

BA>10000

is either true or false because the symbol > means 'greater than'. You would then say in a BASIC program something like

```
70 IF BA>10000 THEN 30
```

which would jump to line 30 if you have more than enough.

In more general terms, to make a comparison, you use what is called a relational expression, which is

something	compared to	something

or, to be more exact,

arithmetic expression	relational operator	arithmetic expression

The relational operators that can be used to compare things are:

=	equal to, i.e. A=B is TRUE if A=B, otherwise FALSE
>	greater than, i.e. 10>5 is TRUE
<	less than, i.e. 10<5 is FALSE

and the combinations

>=	or	=>	greater than or equal to, i.e. 5>=6 is FALSE
<=	or	=<	less thanor equal to, i.e. 5<=6 is TRUE
<>	or	><	not equal to, i.e. 5<>5 is FALSE

2 Be decisive — the IF... THEN statement

The IF... THEN statement uses a relational expression to make a decision about whether to jump to a chosen line number. This allows programs to decide what to do next.´ It looks like this:

```
line number  IF  relational  THEN  destination
                 expression
```

If the relational expression is TRUE, the program jumps to the destination, which must be a line number that really exists.

EXAMPLE:

Now we can stop counting whenever we want, as in

```
10 REM COUNT TO FIVE
20 J=1
30 PRINT J
40 J=J+1
50 IF J<=5 THEN 30
```

Try it. Later on we will find an even easier way to do this.

EXAMPLE:

It is unhealthy to let the recurrence programs of the previous chapter go on forever. Here is how to stop the program for doing

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} +$$

when the next term is less than 0.00001. This is when the snail is within a whisker of its destination. Change line 60 to

```
60 IF T>0.00001 THEN 30
```

3 A more powerful IF — the IF. . . THEN. . . ELSE statement

The TRS80 has a more versatile version of the IF. . . THEN statement than most kinds of BASIC. First of all, in the IF. . . THEN form, you can have a statement of BASIC as the TRUE condition, as in this example:

```
40 IF BB>=10000 THEN PRINT"YES YOU CAN AFFORD IT"
```

Furthermore, a program often encounters a situation where it wants to act differently on TRUE and FALSE conditions. In this case, if what you want to do is simple enough, you can add an ELSE condition to the IF statement, which is again any statement of BASIC:

```
40 IF BB>=10000 THEN PRINT"YES YOU CAN AFFORD IT"
                  ELSE PRINT"NO YOU CAN'T"
```

In both cases the IF. . . THEN or IF. . . THEN. . . ELSE statement has to go entirely on one line of BASIC. You will find that you get some pretty long lines if you use this, but it doesn't matter — a long line of BASIC just continues on several lines on the screen until you push ENTER. No problem.

It might be even more useful if you could have several lines of BASIC for the TRUE and FALSE conditions, but you can't. However you can squeeze several statements of BASIC into either the TRUE or FALSE conditions by using colons. As long as you keep the entire IF. . . THEN. . . ELSE on one long line of BASIC, you can have lots of statements, for example

```
10 INPUT I
20 IF I=0 THEN PRINT"TRUE":PRINT"IT WORKS"
       ELSE PRINT"FALSE":PRINT"BUT IT STILL WORKS"
```

EXAMPLE:

Those who market toothpaste are very sly. On the supermarket shelf you might find a tube of CRASP containing 25 grams of blue guck for 1.25 interplanetary credits and next to it a bigger 4 ounce tube selling for 5.49 credits. Which do you buy? There are 28.35 grams in an ounce. The best value is the one whose price per gram is the lowest.

EXERCISE:

Run this program:

```
10 REM BRUSH DAILY WITH GUCK
20 REM PRICE PER GRAM OF SMALL TUBE
30 LET SM=1.25/25
40 REM PRICE PER GRAM OF BIG TUBE
50 LET BG=5.49/(4*28.35)
60 REM NOW REPORT ON THE RESULT
70 IF SM<=BG THEN 100
80 PRINT "BUY THE BIG ONE"
90 END
100 PRINT "BUY THE SMALL ONE"
```

Why do we buy the small one if the costs per gram are the same? Because more blue guck will stick to the inside of the larger one and be wasted. If you wanted to take this factor into account properly, you would have to carry out some experiments.

You will notice when you enter this program that line 70 is too long for the screen. This doesn't matter. Long lines just carry on, and that applies to editing as well.

4 More complicated decisions — NOT, AND, OR

This is a feature of BASIC on the DRAGON which not all computers have. You can use special relational expressions with NOT, AND, OR in them to make more complex decisions. They are called logical operators. You can have

relational expression	logical operator	relational expression

and again get a result TRUE or FALSE; for example in

```
30 IF I>10 OR J<20 THEN 66
```

Here is what they mean:

something OR something
is TRUE if either or both of the somethings are TRUE

thingie AND thingie
is TRUE only if both thingies are TRUE

NOT something has the result TRUE
if the something is FALSE and vice versa

If you want to make a really complicated expression, you have to know about the priority of these operations. Here it is:

AND	highest
OR	
NOT	lowest

and you can use brackets to get what you want.

EXAMPLE:

Your two kids each get an allowance. The eldest has an allowance of A1 per week. The youngest gets A2 per month — this is a result of complex negotiations. Because there are not four weeks in a month, you think you might be able to fool them about who gets the most but the DRAGON is on their side. Here is a program to check that A1*52 is more than A2*12, so that in a year the eldest gets more than the youngest, unless both are zero. As the kids have learned not to trust you, it also checks that neither allowance is negative. So what are the rules?

> Both A1 and A2 must be greater than or equal to zero and A1*52 must be more than A2*12 unless both A1 and A2 are zero.

IT REALLY HURTS A FATHER WHEN HIS KIDS DON'T TRUST HIM OVER MONEY

Here is the program they will use to check up on you:

```
10 REM DOWN WITH CHEAP DADS
20 PRINT "ENTER A1 AND A2"
30 PRINT "WITH BOTH >=0 AND"
40 PRINT "A1*52 > A2*12 UNLESS"
50 PRINT "BOTH A1 AND A2 ARE 0"
60 INPUT A1,A2
70 IF NOT((A1*52>A2*12 AND A2>=0)OR(A1=0 AND A2=0)) THEN 20
120 PRINT "DAD GO IT RIGHT"
```

5 Try not to make ugly programs

Look at the IF statement from the above example. It is complicated:

```
70 IF NOT((A1*52>A2*12 AND A2>=0)OR(A1=0 AND A2=0)) THEN 20
```

There are a number of ways of getting the same result. You could have written

```
70 IF (A1*52>A2*12 AND A2>=0) OR (A1=0 AND A2=0) THEN 120
80 GO TO 20
```

Why not? Well, quite honestly to an experienced programmer it is repulsive. If you write an IF statement followed by a GO TO statement, you are creating an ugly structure in which the flow lines of your program cross. This is never necessary. If you are tempted to write

```
70 IF something THEN 120
80 GO TO somewhere
```

then write instead

```
70 IF opposite THEN somewhere
```

Sometimes the opposite would be achieved by the NOT as in the example above, or sometimes you would use a different comparison. These two are the same:

```
50 IF J>5 THEN 70
60 GO TO 30                50 IF J<=5 THEN 30
70 END
```

Do you recognise this? There are no prizes for guessing which is best.

If you want to, you can make some really horrible things. Look at this:

```
70 IF A1*52>A2*12 THEN 110
80 IF A2<>0 THEN 20
90 IF A1=0 THEN 120
100 GO TO 20
110 IF A2<0 THEN 20
120 PRINT"DAD GOT IT RIGHT"
```

It is the same as the allowance program, but it is truly awful. Don't do things like this. Not only is a GO TO never necessary after an IF statement, a cluster of IF statements is never necessary. The NOT, AND, and OR operations are there to help you. Use them!

6 Another way of making decisions — the ON. . . GO TO statement

This one is not used as often as the IF statement. It is sometimes useful to be able to jump to one of several places in a program. The ON. . . GO TO statement does this. It looks like

```
line number  ON  expression  GO TO  line     ,  line    , . . .
                                    number      number
                                      a           b
```

When the DRAGON sees an ON. . . GO TO statement, it works out the expression and jumps to the line number a if the integer part of the result is 1. We will find out about integer parts in the next chapter. It means that if the expression gives anything between 1 and slightly less than 2, the program jumps to line number a. Similarly, if the integer part of the result is 2, it jumps to line number b, and so on for the number of destinations you have given. If the expression gives a result less than 1, or one whose integer part is greater than the number of destinations, then the program just continues with the next line in order.

EXAMPLE:

```
60 ON I+J GOTO 10,20,30
```

If I+J is a number between 1 and just under 2, the program jumps to line 10.
If I+J is between 2 and just under 3, the jump is to line 20.
If I+J is between 3 and just under 4, the jump is to line 30.
If I+J is 4 or greater, the program carries on to the next line after line 60.

Seven

SOME FUNCTIONS

1 Little helpers

There are a number of operations with numbers that you can't do very easily by writing out a formula in BASIC, but which are needed quite often. BASIC provides a number of little functions to help you out. Here is what there is:

Name	Meaning
SQR(expression)	Square root of expression
ABS(expression)	Absolute value of expression
SGN(expression)	Sign of expression: 1 if >0, 0 if 0, −1 if <0
INT(expression)	Integer part — the largest integer not greater than expression
FIX(expression)	Truncate to an integer
EXP(expression)	These are specialised mathematical functions, covered in Chapter 19
LOG(expression)	
SIN(expression)	
COS(expression)	
TAN(expression)	
ATN(expression)	
RND(expression)	Randy, the random number generator — has its own chapter, Chapter 16

To use a function, simply write it where it is desired, with its expression in brackets immediately after it. For example,

```
30 PRINT ABS(X)
```

prints the 'absolute value' of X, which is X with a positive sign. Similarly you can use functions inside functions as

```
        50 PRINT INT(ABS(X-Y))
```

Note the double brackets at the end.

2 Trying them out

Here we consider SQR, ABS, and SGN in turn. INT is super-useful and will be looked at in detail in the next section, along with FIX.

(a) SQR

You could find a square root by an involved mathematical procedure every time you need it — some people need them quite often. This would mean spending a lot of time programming square roots — very boring. The SQR function saves us the trouble. The only thing to remember is that a negative number does not have a square root.

EXAMPLE:

If you walk 100 metres north and 300 metres east, how far are you from where you started?

Do you remember your hypotenuse formula? If you have a right-angled triangle, then the hypotenuse is

$$h = \sqrt{a^2 + b^2}$$

where h is the length of the hypotenuse and a and b are the lengths of the other two sides. Your little walk is an hypotenuse:

```
10 PRINT "GOOD MORNING, CAN I "
20 PRINT "FIND YOU AN HYPOTENUSE?"
30 PRINT "ENTER TWO SIDES"
40 INPUT N,E
50 PRINT "HYPONENUSE ="SQR(N*N+E*E)
60 GO TO 30
```

EXERCISE:

Try it out. It won't give any trouble about negative square roots. Do you know why I wrote N*N+E*E instead of N↑2+E↑2? Find out what happens if you try to SQR with a negative number.

(b) ABS

This function forces the sign of an expression to be positive. It can be used to keep out of trouble with SQR, as for example in

```
        90 QR=SQR(ABS(T))
```

Very often a program is interested in the size of something regardless of its sign, and this is when ABS is used.

(c) SGN

Sometimes it is important to know when a number is positive, zero, or negative, without caring about its actual value. In a program to manage your bank account, you might want to act differently if your balance were negative, zero, or positive:

```
50 ON SGN(BB)+2 GO TO 90,30,10
```

This jumps to line 90 if you are in the red, line 30 if you're exactly broke, and 10 if you have some money left. Why was 2 added to SGN(BB)?

Here is a nifty little statement to transfer the sign of one variable to another; Z is supposed to have the same size as X but with the same sign as Y.

```
90 Z=SGN(Y)*ABS(X)
```

It will not work if Y is zero.

3 Cutting away decimal places — the INT function

The INT function is probably the most useful of the lot, which is why it is discussed here separately. By its definition, INT(X) gives an answer which is the largest integer which is not greater than X. This means, in simpler terms, that a positive number has its decimal places stripped away:

INT(73.7) is 73

but negative numbers move down:

INT(−73.7) is −74

Look carefully at this last example. INT does not just take away the decimal places — it moves down.

(a) Rounding

You may want to round a result to the nearest whole number. Although INT does not do this on its own, it can be very easily forced to. To round to the nearest whole number, simply add 0.5 before using INT:

```
400 RN=INT(XN+0.5)
```

Nearly everyone knows how to do that, but here is a nifty trick. You can round a number to any coarseness you want, which can be pretty useful with money for example. All you do is scale the values so that the required coarseness is represented by integers, round it, and then undo the scaling. Sounds complicated? Here are some examples.

(i) After calculations of interest on an investment, your friendly Interstellar Savings Corp rounds your interplanetary credits to the nearest centeroonie (there are 100 centeroonies to the credit). Without rounding, one credit invested for 7 epochs with interest at 8% per epoch would give

```
30 PRINT 1.Ø8↑7
```

But rounding to the next centeroonie could be done like this:

```
30 PY=1.Ø8↑7
40 PZ=INT(PY*1ØØ+Ø.5)/1ØØ
50 PRINT PZ
```

See how the payoff PY was scaled up by 100 before the integer part was taken, and then down again.

(ii) Banana warmers come in cases of 50. A customer orders WB banana warmers. You have to push this up (good for business, that) to the next 50 to get the number of cases to send, called CS.

```
80 CS=INT((WB+49)/5Ø)
```

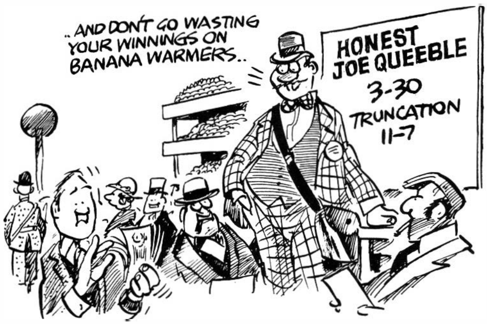
This is not quite the opposite of the centeroonie procedure. I have added 49 to be sure that the customer who orders 51 banana warmers is sent 2 cases.

(iii) Actually inflation has spoiled the value of the interplanetary credit to the extent that the smallest coin issued by the Pan Galactic Council is 5 centeroonies. If you bet a quarp (25 centeroonies) on a geegee at 11-7, the bookies will cut your prize to the next lowest 5 centeroonies. This is truncation again, with our clever scaling applied. So you get

```
140 PRINT INT((25*11/7)/5)*5
```

centeroonies. The scaling is the other way — reduced by five times before the integer part is taken. A more gererous bookie would round:

```
140 PRINT INT((25*11/7)/5+Ø.5)*5
```



(b) Truncation — your actual chop!

When the decimal places are stripped away from a number, it is called 'truncation'. This is what the bookies did to you above. The INT function is not quite a truncation because of its operation with negative numbers. However, the expression

```
SGN(X)*INT(ABS(X))
```

is a truncation for any number X.

The FIX function on the DRAGON does the same thing.

```
FIX(X) is the same as SGN(X)+INT(ABS(X))
```

(c) Converting units

EXAMPLE:

We can use INT very nicely to break numbers into parts. If we know the distance to San Jose in kilometres (unlikely, I know) we may wish to convert this to whole miles plus yards, feet, and inches, rounding to the nearest 0.1 inch. We need to know that

1 mile = 1.60934 kilometres

and also that

1 mile = 1760 yards
1 yard = 3 feet
1 foot = 12 inches.

The easy part is converting the kilometres to miles:

```
10 REM DO YOU KNOW THE
20 REM WAY TO SAN JOSE?
30 PRINT "HOW MANY KILOMETRES"
40 INPUT "TO SAN JOSE";KM
50 REM MAKE IT MILES
60 ML=KM/1.60934
70 PRINT "THAT'S" ML "MILES"
```

Now we start using INT to rip off decimal places. First of all, whole miles:

```
80 REM NOW BREAK IT DOWN
90 REM FIRST WHOLE MILES
100 WM=INT(ML)
```

and the bit that is left over can be converted to yards:

```
110 REM YARDS LEFT OVER
120 YD=1760*(ML-WM)
```

If you see how that works, the rest is easy:

```
130 REM WHOLE YARDS
140 WY=INT(YD)
150 REM FEET LEFT OVER
160 FT=3*(YD-WY)
170 REM WHOLE FEET
180 WF=INT(FT)
190 REM INCHES LEFT OVER
200 IN=12*(FT-WF)
210 REM TO THE NEAREST 0.1
220 IN=INT(IN*10+0.5)/10
230 REM PRINT THE ANSWERS
240 PRINT WM "MILES" WY "YARDS"
250 PRINT WF "FEET" IN "INCHES"
```

Now this program is a bit long, and if you type slowly it could take you quite a while to enter. Most people with a DRAGON will also have a cassette recorder. All the examples in this book which are 8 lines or longer are available on a cassette, called '95 Programs from Know your DRAGON'. If your computer dealer or bookshop don't have it, please see the title page at the beginning of the book for information on how to order one.

(d) Remainders are useful too

When you divide 22 by 7 the integer part of the answer is 3 and the remainder is 1. Suppose we were dividing N by D and both N and D are positive. The integer part of the answer would be

INT(N/D)

and the remainder would be

N −INT(N/D)*D

if N and D were positive. Do you see how this works?

EXAMPLE:

If you don't want to upset your customers in the hot banana trade, perhaps it would be better not to send them all those extra banana warmers. There are 50 to a case, remember. This program tells you how many full cases to send and how many odd ones are needed in addition. See the remainder?

```
10 REM ARE HOT BANANAS MUSHY?
20 PRINT "HOW MANY BANANA"
30 INPUT "WARMERS ARE ORDERED";WB
40 REM FIRST THE QUOTIENT
50 CS=INT(WB/50)
60 PRINT "SEND" CS "CASES"
70 REM THEN THE REMAINDER
80 RM=WB-CS*50
90 PRINT "AND" RM "ODD ONES"
```

EIGHT

ROUND AND ROUND WE GO

1 Loop the loop

From modest beginnings, we have come a long way in learning to drive the DRAGON 32 in BASIC. One of the really useful things learned is making loops. First we found out how to make a loop repeat forever, and how to use a variable to count. Then we learned how to stop the counting at a particular value with an IF statement.

2 The hard way

Actually we have learned to count the hard way. Before discovering the easy way, let's be sure we know what we did. First of all, a starting value was set up. Each time around the inside of the loop the counter was incremented and tested to see if the loop should be repeated again. It is easy to see the importance of these steps: initialise, increment, and test. In the BASIC that we know so far, to repeat something 10 times, we would need to write something like:

```
50 C=1                  (initialise)
60 :
   :
80 C=C+1                (increment)
90 IF C<=10 THEN 60     (test)
```

3 The easy way

In BASIC there is a special pair of statements for loop control. These are the FOR and NEXT statements. All you have to do is write

```
        50 FOR C=1 TO 10
              :
              :
        90 NEXT C
```

The FOR statement is responsible for initialising some variable — C in this case. The FOR is matched by a NEXT statement naming the same variable.

EXAMPLE:

Type this program as one line, i.e. without pushing RETURN until the very end. The colons squeeze several statements on one line:

```
        10 FOR J=1 TO 5: PRINT J: NEXT J
```

Now try it without the line number. All these statements work in 'direct mode'. Using the colon you can make a complete little program.

You will remember that the IF... THEN... ELSE statement had to be all on one line. Using colons you can make the TRUE and FALSE procedures as long as you want:

```
        30 INPUT I
        40 IF I>0 THEN PROD=1:FOR K=1 TO I:
                PROD=PROD*K:NEXT K:PRINT PR:
                ELSE PRINT "I IS WRONG":GO TO 30
```

EXAMPLE:

If you want to, you can use a FOR... NEXT loop deliberately to waste time. This program counts forever. A FOR... NEXT loop at line 50 wastes a bit of time and has no other purpose.

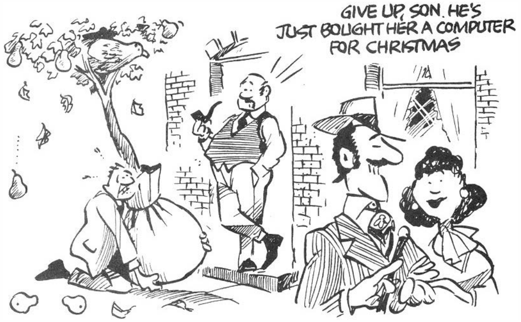
```
        10 REM COUNT FOREVER
        20 I=1
        30 PRINT I
        40 REM WASTE TIME
        50 FOR J=1 TO 500: NEXT J
        60 I=I+1
        70 GO TO 30
```

What would happen if I had been used as the FOR... NEXT variable?

4 Adding up again

You know the song about the Twelve Days of Christmas. On the first day your truelove gives you a partridge in a pear tree. On the second day you get another partridge plus two new gifts (turtle doves), and so on. How many gifts do you receive on day 12? Here is a program to add the numbers from 1 to 12. Notice that the variable used for summing, GF, is given a starting value before the FOR. . . NEXT loop.

```
10 REM ADD NUMBERS TO TO 12
20 GF=0
30 FOR I=1 TO 12
40 GF=GF+I
50 NEXT I
60 PRINT "SUM IS" GF
```



5 Mind your STEP

There is another useful facility in the FOR. . . NEXT loop. Everything up until now has counted forward by ones. You may want to do something different. Using the word STEP, you can have such things as

```
10 FOR J=1 TO 21 STEP 3
```

or

```
10 FOR X=0.25 TO 1.50 STEP 0.25
```

or

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

Don't forget the NEXT in each case. There are a number of rules about FOR. . . NEXT loops — mostly common sense — which will be given a bit later, after some examples.

6 Multiplying up — factorials

Here is something else interesting. The product of all the whole numbers from 1 to n is called the factorial of n. Mathematicians write this as n!.

$$n! = 1*2*3. . . *n$$

Now n! is pretty useful. Think of how many ways you have of arranging five different coloured balls in a row. You can choose any one as the first one. You then have a choice of four as the second. Furthermore, for each of these you can choose any of the three remaining colours for the third one. You have 5*4*3*2*1 possible ways of arranging the colours, which is 5!. These statements would find N! for you by multiplying:

```
70 F=1
80 FOR I=1 TO N
90 F=F*I
100 NEXT I
```

This is a bit like adding up, only we're multiplying up! Here is a super-duper N! program:

```
10 REM MAKE FACTORIALS
20 PRINT "HI THERE"
30 PRINT "SLIP ME A NUMBER"
40 PRINT "AND I'LL FIND ITS"
50 PRINT "FACTORIAL IF I CAN"
60 INPUT N
70 F=1
80 FOR I=1 TO N
90 F=F*I
100 NEXT I
110 PRINT "FACTORIAL" F
120 GO TO 30
```

EXERCISE:

Factorials get pretty big. Run this program and find out how big N can be before the computer gets N! wrong. Do you remember how it represents large numbers?

7 Nesting

You've seen those little Russian dolls — one goes inside another inside another and so on. Nested loops are the same. You can put one loop inside another as long as it uses a different variable to count and as long as the loops do not cross. Here are correct and incorrect examples:

Correct

The loops are 'nested'

```
20 FOR X=5 TO 15
       :
50 FOR Y=7 TO 1 STEP -2
       :
80 NEXT Y
       :
120 NEXT X
```

Incorrect

The loops cross

```
5 FOR P=1 TO 5
        :
45 FOR Q=100 TO 101 STEP 0.1
        :
95 NEXT P
        :
200 NEXT Q
```

EXAMPLE:

After the twelve days of Christmas, how many gifts have you received in total? This would work it out and tells you the sum at the end of each day.

```
10 REM PURE GREED
20 GF=0
30 FOR I=1 TO 12
40 FOR J=1 TO I
50 GF=GF+J
60 NEXT J
70 PRINT "DAY" I "TOTAL" GF
80 NEXT I
```

Notice that the upper limit of the inner loop is I. Very sly, that!

8 A note on program debugging — software probes and traces.

Perhaps you have a program that doesn't work, and you don't know why. People would say your program had a 'bug' in it. Exterminating program errors is called 'debugging'. Suppose the program to add up the twelve days of Christmas had an error in it which you couldn't see; line 40 was

```
40 FOR J=1 TO 12
```

The usual way of finding errors is to put PRINT statements everywhere and use your brain to spot things going wrong. You might add

```
35 PRINT"SUMMING ON DAY" I

55 PRINT J "MORE GIFTS NOW HAVE" GF
56 INPUT"PRESS ENTER TO CONTINUE";X
```

If you run this you should spot the error immediately — on day I you are only supposed to get I gifts and so line 40 should be

```
4Ø FOR J=1 TO I
```

If you are interested in a good cocktail party line, you could say that you had 'located the bug by inserting software probes'. A PRINT statement used this way is a software probe.

The DRAGON has a special facility for making debugging even easier if the errors are in the logic of your loops or IF statements. If you put

line number TRON

then when your program is RUN it will tell you the line number of every statement you execute until you hit

line number TROFF

this is called a 'program trace' — our vocabulary is getting very rich isn't it. On the twelve days of Christmas you would want

```
5 TRON
```

right at the beginning. If you forget to put TROFF somewhere, it stays on forever and ever even when you run a new program.

```
85 TROFF
```

EXERCISE:
Try both those methods on the greedy program. Put the deliberate error in first. Which is best? Do you see the need for the INPUT statement? Use one in the trace method. Now which is best?

9 Is yours prime?

A number which cannot be broken down into the product of smaller ones except 1 is called a prime number. For example, 21 is 3x7 so it is not prime, but both 3 and 7 are primes; the first few are 1, 2, 3, 5, 7, 11. . . How can we tell if a number N is prime? We can try all integers I from 2 to SQR(N) to see if they are factors. How will we know? Because if I is a factor, the remainder is zero when we divide N by I. We did remainders in the last chapter. If we find any zero remainder, then N is not a prime. Here is the bit that does this:

```
100 REM IS N A PRIME?
110 FOR I=2 TO ABS(N)↑0.5
120 R=N-INT(N/I)*I
130 IF R=0 THEN 170
140 NEXT I
150 REM IF GET TO HERE N IS PRIME
160 PRINT N "IS A PRIME NUMBER"
```

Now let's use this to find all the primes less than M, a number that you type in:

```
10 PRINT "SUPPORT YOUR LOCAL"
20 PRINT "PRIME SEARCHER."
30 PRINT "GIVE ME ANY NUMBER"
40 PRINT "GREATER THAN 2 AND"
50 PRINT "I'LL TELL YOU EVERY"
60 PRINT " PRIME FROM 3 TO THE"
70 PRINT "THE NUMBER YOU ENTER"
80 INPUT MX
90 FOR N=3 TO MX
```

the rest of the program as above

```
   :
   :
170 NEXT N
```

This program has a nested loop.

EXERCISE:
Run this program. Make it count the primes, and find out how many there are less than 500 — note that 1 and 2 are not found.

10 More uses for factorials

The factorial n! was the number of ways of arranging n different things in a row. There are some more complicated things that factorials can do. The number of ways of arranging n different things where you can only take r of them at a time is called the number of 'permutations' of n things taken r at a time, or

$$
\begin{aligned}
{}_nP_r &= n(n-1)\ldots(n-r+1) \\
&= \frac{n!}{(n-r)!}
\end{aligned}
$$

To find this you could take n! and divide it by r! Better still, you could take the product of all numbers from $n-r+1$ to n:

```
10 REM PERMUTATION CALCULATION   60 PR=1
20 PRINT "I'LL CALCULATE PERMS"  70 FOR I=N-R+1 TO N
30 PRINT "FOR YOU, BOSS!"        80 PR=PR*I
40 INPUT "HOW MANY OBJECTS"; N   90 NEXT I
50 INPUT "HOW MANY CHOSEN"; R    100 PRINT "NO. OF PERMS IS" PR
```

EXERCISE:

Try it. What is nPn? What is nP_0? What does this program do if $r>n$? Is that right? What does this program do if $r<1$? Is that right?

Finally, you may wish to know how many combinations there are if you can select r things from a choice of n, but you are not interested in the order. This is the same as taking the r! reorderings out of nPr. This is called nCr:

$$
{}_nC_r = \frac{n!}{r!(n-r)!}
$$

EXERCISE:

Make a program to do nCr.

11 The rules

line number FOR variable = expression TO expression STEP expression

(i) The FOR statement gives the initial, final, and step values. These can be any expressions.

(ii) A variable name must be used as the counter. There is no need to actually use it, as in:

```
10 FOR K=1 TO 5
20 PRINT "OVER AND"
30 NEXT K
```

(iii) The STEP part is optional. If you leave it out, the step size is 1. You can make the step size fractional or negative.

(iv) Regardless of the initial, final, and step values, the loop will be done at least once, as in this silly loop:

```
40 FOR BB=5 TO 1 STEP 3
60 PRINT "DID IT?"
80 NEXT BB
```

which is done once.

(v) The initial, final, and step values are considered only when the loop is first entered. They cannot be altered within the loop. For example, in

```
50 FOR I=J TO K STEP L
60 L=L*2
70 K=K-1
80 NEXT I
```

all that messing around with L or K has no effect on the number of repetitions of the loop. If J, K, and L were 1, 10, and 1 the loop would be repeated 10 times.

(vi) The counter itself can be changed and this will affect the loop. For example

```
50 FOR I=1 TO 10
60 I=I+1
70 PRINT I
80 NEXT I
```

is repeated only 5 times. Be sure you understand the difference between this and rule (v).

(vii) You must match every FOR statement with a NEXT statement which names the same counter.

line number NEXT variable

Actually on the DRAGON you can just have

line number NEXT

and it is obvious to the computer which loop should be closed. However, it is not always so obvious to you. You could save yourself a lot of trouble if you always use the full version.

(viii) Loops must be nested properly. Nested loops cannot use the same counter.

Nine

GET IT TAPED

1 You might like to know...

Nowhere else in this book is it assumed that you have anything to go with your DRAGON 32 except a television set, probably colour. However you will have noticed that the examples are getting longer and longer, with some quite large ones to come. You may find it convenient to save programs on tape so that you can use them again later. Some of the music and graphics programs that are coming up soon might be of particular interest. Indeed, many video games are available on cassette. Also, all of the programs that are eight lines or longer in this book have been recorded for you on a cassette which you can order — look on the title page at the beginning of the book for details.

2 Fire it up

A remote controlled recorder is best. You really shouldn't use anything else. The Radio Shack computer cassette recorder from TANDY is inexpensive and is ideal for use with the DRAGON 32 because it has all the neccesary remote controls and works with the DRAGON's connecting cables. Do this:

(i) Connect the 'AC in' socket on the cassette recorder to the electricity supply using the power cable — of course! Now if you want to you can use the recorder like any portable audio recorder for music and so on. You could also use it with batteries, although there doesn't seem much point in that when you're using it with the computer.

(ii) Connect the recorder to the DRAGON. The cable plugs easily into the socket marked 'cassette' in the side of the computer. The smaller plug on the end of the blue wire goes into 'MIC' on the side of the recorder. The plug on the red wire goes into 'AUX', and the white one goes into 'EAR'. Turn the volume control to 5, which isn't there, actually. Halfway between 4 and 6 is ideal.

3 Load a Tape

Is there anyone left in the world who doesn't know how to put a cassette in? Oh, sorry there. Well all you do is pop it in with the tape towards you and close the lid. Push REWIND to get the tape back to the beginning, and then if you push STOP and then PLAY you're ready to go. If you want to get the tape out, push STOP and then EJECT. Whoops! Mine practically throws the cassette at me.

4 How to load a program — CLOAD

If there is only one program on the tape, enter the command CLOAD. This will load the first program found. If that program has a name, you will see it on the screen while the program is loading.

If there are several programs on the tape, you can load them one after another using CLOAD over and over again. The command CLOAD always loads the first program it finds.

If you know the name of the program you want, enter CLOAD giving the name in double quotes, like

```
CLOAD "PRIMES"
```

which will search for and load the prime searching program from Chapter 8 of this book. While it's searching, you will see the names of all the programs that are found appear on the screen while the computer is going past them. The code S that you see means 'Searching' and F means 'Found'.

Therefore, if you don't know what is on your tape, type in something like

```
CLOAD "JUNK"
```

and as long as JUNK isn't the name of a real program, you will find out the names of everything on your tape.

You may get an error message 'I/O ERROR' if the computer reads something it doesn't understand like a bit of Beethoven or a fragment of a old program. You can rewind and try again, or go on by entering the command again. If it searches for hours without finding anything, then either

(i) There isn't anything there
(ii) The volume control is wrong — perhaps the dog changed it, or
(iii) The cable isn't connected.

Knowing the proven high intelligence of DRAGON owners, this isn't going to give anyone much trouble.

If you have the tape '95 Programs from Know your DRAGON', you will find that a list of the page numbers and names of the 95 programs with the approximate counter settings to use with the Radio Shack recorder is printed on the insert card that comes with the cassette. There is also provision for you to write in your own counter settings if you have some other recorder.

5 Save a Program

Now this is a bit more tricky. Before you try, remember that when you say you want to save a program on tape, it gets saved on the tape wherever the tape happens to be. So you have to be careful. You also have to remember to press the RECORD button as well as the PLAY button on the recorder. Otherwise the computer will pretend to save the program but it won't be there. The Radio Shack recorder won't let you press record if you are trying to use a protected tape.

(i) On a new tape — CSAVE

Make sure the tape is rewound. If the tape has a 'leader' on it, which is a bit of tape that isn't brown at the beginning, then do a tiny bit of 'FAST-F' to get into the tape itself. Leaderless tapes are less troublesome than ordinary ones with leaders. Press PLAY and RECORD and enter the command

```
CSAVE "name"
```

to save your program. You can put in CSAVE without a name but then you could have trouble finding your program later on. As soon as you have done this. it isn't a new tape any more.

(ii) On an old tape — CSAVE

Usually to save a program on an old tape which already has programs on it, you will want to put it on the end. Remember that a program is saved wherever the tape happens to be at the time. When you know you're at the right place, press PLAY and RECORD and enter

```
CSAVE "name"
```

just as you did before — but choose a new 'name'. You can safely CSAVE one program after another.

(iii) Finding the right place — SKIPF

Most of the time your cassette won't be in the right place. Probably you know the name of the last program on the tape. If you do, enter

```
SKIPF "name of last program"
```

and the cassette will search until it has passed the last program on your tape. You can then safely CLOAD.

You will notice that SKIPF tells you the names of the programs it is skipping, just like CLOAD. Therefore, you can use it to find what is on a tape, just like CSAVE:

> SKIPF skips over the next program.
>
> SKIPF "name" skips over programs until it has passed the program called 'name'.

I recommend using something like

```
SKIPF "JUNK"
```

to get past the last program on a tape, relying on an I/O error to stop you at the right place. This doesn't work on new blank tapes because the error never occurs. Anyway, be careful.

6 Double check everything

It is a very good idea to make sure that you can reload a program as soon as you have saved it — because there are a lot of little things that could go wrong. After you do

```
CSAVE "MYPROG"
```

to save your program called 'MYPROG', rewind the tape — perhaps not all the way but far enough to get back over MYPROG. Then put in

```
CLOAD "MYPROG"
```

and make sure it works by listing the program again. If it doesn't work check the following:

(i) Was volume control at 5?
(ii) Is the cable connected?
(iii) Did you push PLAY and RECORD?
(iv) Is yours a protected tape? Some tapes cannot be written on. There are little plastic tabs that can be removed to make it impossible to record on a tape. If you do have a valuable tape, you can do this to protect it. Most prerecorded tapes you buy will be protected.

Ten

SOUNDS INTERESTING

1 Resources

The DRAGON 32 has a sound generator inside it which can be used to create a variety of sounds. It produces a single tone whose pitch and duration can be controlled, and as we will see in Chapter 23, the volume can be turned up and down. The most obvious use for this is in making music, but quite a few useful sound effects are also possible. Later, when we are putting fancy pictures on the screen, we will see that it is usually possible to make a sound that fits the picture.

2 Beep Beep

Here, to start us off, is a blast from a car horn. Push enter each time you want it to honk:

```
10 REM GET OUTTA MY WAY
20 SOUND 128,8
30 SOUND 128,8
40 INPUT X
50 GO TO 20
```

If you can't hear it, turn up the volume and try again. If you still can't hear it, perhaps you need to tune your set more accurately. Clearly the SOUND statement is the noisy bit. The line

```
20 SOUND 128,8
```

means that a tone is to be produced with pitch 128 and duration 8. Generally, you write

line number SOUND pitch, duration

where

the pitch is a number between 1 and 255

and

the duration is a number between 1 and 255

In the above example, you can hear a slight break between the two SOUND statements.

3 Low to High

This little program will give you an idea of the range of pitch that is available:

```
10 FOR I=1 TO 255 STEP 16
20 SOUND I,1
30 NEXT I
40 GO TO 10
```

Zounds! Can't stand that one for long.

There are quite a few things you can do by changing the pitch of a sound. Does this sound like a police car?

```
10 SOUND 164,6
20 SOUND 128,6
30 GO TO 10
```

4 Long and Short

This little program will drive them wild:

```
10 FOR I=1 TO 8 STEP 2
20 SOUND 128,I
30 SOUND 164,I
40 NEXT I
50 GO TO 10
```

By listening to this for a little while, you can get a feeling for the length of a note. Clearly

```
SOUND 128,1
```

is fairly brief, while

```
          SOUND 128,8
```

which is eight times longer lasts for quite a while.

```
          SOUND 128,255
```

is very boring. Don't try it — you can't BREAK into a SOUND, but you can push the RESET button on the back of the computer without losing the program.

Naturally, SOUND can be used in direct mode. Try this:

```
          FOR I=1 TO 255: SOUND I,1: NEXT I
```

All systems go. We have ignition. We have lift off. Bye!

So now we can try something more ambitious. With a bit of imagination this sounds like a motorcycle coming towards us:

```
          10 SOUND 9,1
          20 SOUND 8,1
          30 GO TO 10
```

and this sounds like one going away:

```
          90 SOUND 2,1
          100 SOUND 1,1
          110 GO TO 90
```

So we can have the motorcycle approach us at high speed, pass by, and disappear. Because of the "Doppler Effect" we can make it a bit realistic by having the pitch of the engine drop at the moment of passing. Hang on:

```
          10 REM HELL'S ANGELS
          20 FOR I=1 TO 12
          30 SOUND 9,1
          40 SOUND 8,1
          50 NEXT I
          60 SOUND 6,1
          70 SOUND 4,1
          80 FOR I=1 TO 12
          90 SOUND 2,1
          100 SOUND 1,1
          110 NEXT I
```

5 Music be the food of life

If we know what pitch in the SOUND statement to use for each note of the musical scale, we can make tunes. Actually the computer's notes are not exact. They are best near the bottom and quite bad in some cases near the top. Here is a table:

Note	Bottom Octave	Middle Octave	High Octave	Higher Octave	Top Notes
C		89	176	218	239
C#		99	180	221	241
D		108	185	223	242
D#		117	189	225	243
E		125	193	227	244
F	5	133	197	229	
F#	19	140	200	231	
G	32	147	204	232	
G#	45	153	207	234	
A	58	159	210	236	
A#	69	165	213	237	
B	78	170	216	238	

We can try to make a tune. Do you know this one?

```
10 REM WOOLY MUSIC
20 SOUND 89,4
30 SOUND 89,4
40 SOUND 147,4
50 SOUND 147,4
60 SOUND 159,2
70 SOUND 170,2
80 SOUND 176,2
90 SOUND 159,2
100 SOUND 147,8
110 GO TO 20
```

Of course. Now you get the idea. You can see what has been done with the durations to get the tune right. Actually it can be very tedious to write such a long list of SOUND statements. We are next going to learn how to set up the data for the tune in a sort of table and then we can write a program which will play any tune.

6 Setting up DATA inside the computer

It can be crashingly boring to have to type a lot of values in through the keyboard or to have a long list of assignment statements. So BASIC has a method of defining a list of values in advance. You do this with a DATA statement, and you get at the values using a READ statement. This will save the effort of writing a lot of SOUND statements in our music programs.

The DATA statement is

line number DATA constant, constant, . . .

as for example

```
30 DATA 159,170,176,159
```

The READ statement is used to transfer values from the DATA list to variables:

line number READ variable, variable, . . .

The values are taken from the DATA list and assigned to the variables one by one for example, in

```
10 DATA 38.2,10.5,-9.6
20 READ A,B,C
30 PRINT A,B,C
```

A will be assigned the value 38.2, B the value 10.5, and C will be −9.6. In this example the number of items was exactly right.

If there is data left over after a READ statement, another READ will continue through the data list. It is as if a pointer moves through the DATA list. Look at this program:

```
10 DATA 38.2,10.5,-9.6
20 FOR I=1 TO 3
30 READ Z
40 PRINT Z
50 NEXT I
```

At the beginning, the pointer is at the beginning:

```
next value
↓
38.2              10.5              −9.6
```

Then with I=1, the statement READ Z assigns 38.2 to Z and moves the pointer along:

```
                  next value
                  ↓
38.2              10.5              −9.6
```

so that when I=2, 10.5 is assigned to Z and the pointer moves again:

```
                                        next value
                                            ↓
          38.2              10.5          −9.6
```

so that the final READ gives Z the value −9.6.

You can also have several DATA statements, and this extends the list, still in the order that the data originally appears as in

```
10 DATA 89,89,89,89
20 DATA 147,147,147,147
30 DATA 159,170,176,159
40 DATA 147,147,147,147
50 FOR I=1 TO 16
60 READ N1
70 SOUND N1,2
80 NEXT I
```

If you run this, it will play a tune for you. This is beginning to look promising? The correct lengths of the notes are obtained by repeating them. If you don't like this, it will be improved a bit later on. It is wrong in READ statements to ask for more values than are available in all the DATA statements.

The RESTORE statement returns the pointer to the beginning of all the DATA statements, so that all the values can be used again. There is no way of getting back to the middle of the list, except for going to the beginning using RESTORE and reading through the list, for example in a FOR. . . NEXT loop.

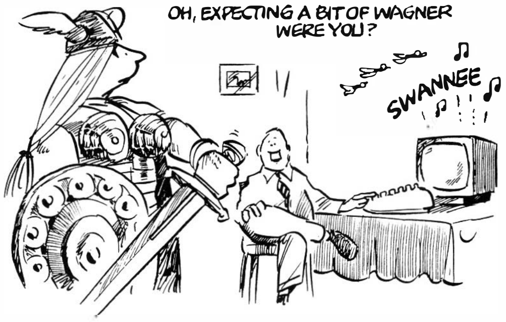
EXAMPLE:

Add the following lines to the little tune, and try it:

```
90 FOR J=1 TO 500: NEXT J
100 RESTORE
110 GO TO 50
```

Do you see the purpose of line 90?

We can make the program slightly more sophisticated by having the first item in the DATA statements tell us the tempo. As before, we read the notes one at a time and play them using a SOUND statement. So that we can make tunes of any length, we signal the end by an impossible note.



```
10 REM PLAY ANY TUNE
20 REM FIRST READ TEMPO
30 READ TM
40 REM NOW READ A NOTE
50 READ N1
60 REM QUIT IF IMPOSSIBLE
70 IF N1<1 OR N1>255 THEN 110
80 SOUND N1,TM
90 GO TO 50
100 REM THE QUIT BIT
110 END
```

Here is a little hornpipe:

```
200 REM A SAILOR'S HORNPIPE
210 REM THE TEMPO
220 DATA 2
230 REM THE TUNE-ONE BAR PER LINE
240 DATA 147
250 DATA 125,89,89
260 DATA 32,89,89
270 DATA 125,89,125
280 DATA 147,133,125
290 DATA 133,108,108
300 DATA 58,108,108
310 DATA 133,108,133
320 DATA 159,147,133
330 DATA 125,89,89
340 DATA 32,89,89
350 DATA 125,89,125
360 DATA 147,133,125
370 DATA 133,125,133
380 DATA 159,147,133
390 DATA 125,89,89
400 DATA 89,0
```

EXERCISE:

Try it. Here is a program that is quite a bit too long for your screen. Make sure you know enough about the LIST command to be able to look at any part of the program you want.

The sailor's hornpipe was a tune in which all the notes had the same length. It would be very boring to have to give the duration of every note in a DATA statement, although it could be done quite easily. All you have to do is delete line 30 in the previous program and change line 50 to

```
50 READ N1,TM
```

With this change, the data must include the notes and their durations in pairs. Do you know this tune?

```
200 REM SWING WITH 'TONIO
210 REM EACH LINE IS A BEAT
220 FOUR LINES MAKE A BAR
230 DATA 193,2
240 DATA 207,2,207,2
250 DATA 207,2,200,1,193,1
260 DATA 216,6
270 DATA 216,1,210,1
280 DATA 207,2,207,2
290 DATA 207,2,200,1,193,1
300 DATA 216,6
310 DATA 216,1,210,1
320 DATA 207,2,210,1,216,1
330 DATA 210,2,207,2
340 DATA 200,2,189,2
350 DATA 170,2,0,0
```

Why not put in

```
110 RESTORE
120 GO TO 50
```

to make it go on forever.

We can make the amount of data less if we agree that any DATA value of 31 or less is not a note, but a new duration, and that any value of 32 or greater is a pitch. This way we lose F and F# at the bottom of the scale, but we can shorten most pieces. Here is the program, rewritten again to do this.

```
10 REM PLAY ANY TUNE
20 REM READ A SOMETHING
30 READ N1
40 REM QUIT IF IMPOSSIBLE
50 IF N1<1 OR N1>255 THEN 140
60 REM IS IT A NOTE?
70 IF N1>31 THEN 110
80 REM NO, IT'S A TEMPO
90 TM=N1
100 GO TO 30
110 SOUND N1,TM
120 GO TO 30
130 REM THE QUIT BIT
140 END
200 REM SWING WITH 'TONIO
210 REM EACH LINE IS A BEAT
220 FOUR LINES MAKE A BAR
230 DATA 2,193
240 DATA 207,207
250 DATA 207,1,200,193
260 DATA 6,216
270 DATA 1,216,210
280 DATA 2,207,207
290 DATA 207,1,200,193
300 DATA 6,216
310 DATA 1,216,210
320 DATA 2,207,1,210,216
330 DATA 2,210,207
340 DATA 2,200,189
350 DATA 170,0
```

The durations now come before the pitches, but only have to be given when a rate has a different duration from the one before it.

Here is a little thingie by Bach. Fantastic!

```
200 REM BOP WITH J.S.
210 REM EACH LINE IS A BEAT
220 REM TWO LINES MAKE A BAR
230 DATA 2,216,1,223,216
240 DATA 2,200,1,216,200
250 DATA 2,185,1,200,185
260 DATA 4,170
270 DATA 1,140,170,185,170
280 DATA 180,170,180,170
290 DATA 165,180,193,180
300 DATA 2,185,170
310 DATA 216,1,223,216
320 DATA 2,200,1,216,200
330 DATA 2,185,1,200,185
340 DATA 4,170
350 DATA 2,185,185
360 DATA 185,185
370 DATA 216,185
380 DATA 185,180
390 DATA 200,200
400 DATA 200,200
410 DATA 223,200
420 DATA 200,197
430 DATA 1,180,200,210,200
440 DATA 207,200,207,200
450 DATA 197,207,216,207
460 DATA 210,207,210,207
470 DATA 200,210,200,197
480 DATA 200,216,200,197
490 DATA 200,221,200,197
500 DATA 200,223,200,197
510 DATA 200,223,221,216
520 DATA 221,210,207,200
530 DATA 2,210,207
540 DATA 4,200
550 DATA 0
```

It's the flute solo in the last movement of the Suite in B minor. Why don't you put in the rest of it? You will find some more music in Chapters 17 and 21.

Eleven

A PICTURE IS WORTH A THOUSAND WHAT?

1 Or was it something to do with Fiona of Troy?

Anyway, one of the delightful features of the DRAGON 32 Computer is its ablity to launch ships, sorry, I mean make pictures. There are several ways of doing this. In this chapter we are going to operate through PRINT statements, but later it will be done in other ways. In Extended Colour BASIC there are a number of special statements to enable you to use the high resolution graphics of the DRAGON 32 Computer in some quite advanced ways.

2 All about PRINT

You have probably noticed that when you use the PRINT statement to put values on the screen, they mostly appear in two 'fields'. Your text is 32 columns wide, and the first field used by numbers is made up of columns 1-16 and the second by columns 17-32. If you don't believe it, use this example to convince yourself:

```
10 INPUT A,B
20 PRINT A,B
30 GO TO 10
```

If it is short enough, you can also get a message to appear in one column and numbers in the other:

```
10 I=10
20 PRINT "A MESSAGE! ",I
30 I=I*10
40 FOR J=1 TO 500
50 NEXT J
60 GO TO 20
```

When you try this example you will see that the columns continue to line up even after the display switches to exponential form for large numbers.

You can force BASIC not to use these two fields. The secret is the semicolon. Try these:

```
10 PRINT "ONE","TWO"          10 PRINT "ONE";"TWO"
20 GO TO 10                   20 GO TO 10
```

If there is a semicolon in a PRINT statement, the output is squashed together.

Another great semicolon feature allows you to continue printing on the same line. Normally a PRINT statement starts a new line on your text screen. But if you leave a semicolon dangling at the end of a PRINT statement, the next PRINT continues on the same line. You can do something similar with commas, but it isn't as useful. Try these and see the difference. The effect is shattering.

```
10 FOR I=1 TO 5     10 FOR I=1 TO 5     10 FOR I=1 TO 5
20 PRINT I          20 PRINT I,         20 PRINT I;
30 NEXT I           30 NEXT I           30 NEXT I
```

This has many great possibilities. In Chapter 16 I will show the geniuses how to make graphs of mathematical functions. Here is a sneak preview which everyone should enjoy:

```
10 FOR I=1 TO 14
20 FOR J=1 TO I
30 PRINT"*";
40 NEXT J
50 PRINT
60 NEXT I
```

I call that one 'Climb Every Mountain'. See how the clever inner loop prints the right number of stars! See how mild mannered line 50 puts us on a new line just in the nick of time! Overwhelming.

One more little thing you may need to know. When you PRINT a message and follow it with a value, or another message in the same PRINT statement, you don't need any punctuation. The computer will assume you mean a semicolon:

```
10 FOR I=1 TO 10
20 PRINT "S" "E" "E" I "!"
30 NEXT I
```

It even knows when to squash two letters together, and when to leave a space before (and after) a number.

3 Two Fantastic Functions — TAB and POS

Well, TAB is fantastic anyway. TAB can only be used in a PRINT statement, and makes your output jump immediately to the column you want. This is very useful in making pretty pictures. Here's an example:

```
10 FOR I=0 TO 13
20 PRINT TAB(I);"*"
30 NEXT I
```

Notice the semicolon to squash the star into the column we want.

In Extended Colour BASIC, if you want to know where your PRINT statement is on the screen at the moment, use POS(X) like any ordinary function to tell you. The X can have any value and has to be there, but it isn't used by the POS function. POS tells you the column number you are about to use. Can you predict the value of POS in this example? If you can, your understanding is profound.

```
10 PRINT "FIRST"
20 FOR I=1 TO 5
30 FOR J=1 TO I
40 PRINT "#";
50 NEXT J
60 NEXT I
70 PRINT POS(X)
```

4 Roll over, Rembrandt

Now we know quite a lot about the PRINT statement, and can begin to find out how to use it to make pictures. The simplest way to produce a shape on the text screen is to make it up using keyboard symbols and put it there using PRINT statements. We will see how to put it exactly where it is wanted shortly. But first, we create a shape. Referring back to our discussion of the Trojan Wars, here is how to draw the famous paddle steamer, Queen Fiona.

```
10 PRINT
20 PRINT TAB(3);"#"
30 PRINT TAB(5);"#"
40 PRINT TAB(6);"M"
50 PRINT TAB(6);"M"
60 PRINT TAB(3);"ITT TTI".
70 PRINT "ITT   O   TTTT/"
80 PRINT "I    OXO     /"
90 PRINT "IWWWWWOWWWWW/"
100 PRINT
```

But it isn't all that good. We need some other shapes that are better for drawing pictures. These are available in the DRAGON, but are not on the keyboard. You have to ask for them with a special function. Try this:

```
10 FOR I=Ø TO 255
20 PRINT CHR$(I);
30 NEXT I
```

This produces different shaped blobs on the text screen. The function CHR$(I) produces a character or symbol, and you get a different one for each I from 0 to 255. All the letters and other keyboard symbols are included. Try this:

```
1Ø FOR I=Ø TO 15
2Ø PRINT CHR$(192+I);" ";
3Ø NEXT I
```

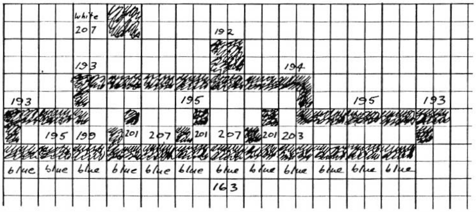
Now that was a bit unexpected, wasn't it! A number of pretty clever (and pretty pretty) things are going to come out of this. The blobs are called "graphics characters", and here is a list of them.

Graphics Shape No	Graphics Character	Graphics Shape No	Graphics Character
0		8	
1		9	
2		10	
3		11	
4		12	
5		13	
6		14	
7		15	

For the black and white ones, add 192 to the shape number and use CHR$:

```
1Ø FOR I=192 TO 2Ø7
2Ø PRINT CHR$(I);" ";
3Ø NEXT I
```

Now we can create a better boat — the QF II! On a bit of squared paper, sketch the drawing you want, work out which graphics symbols go where, and we have



```
20 REM THE SMOKE
30 PRINT TAB(3);CHR$(207)
40 REM THE FUNNEL
50 PRINT TAB(6);CHR$(192)
60 REM THE TOP DECK
70 PRINT TAB(2);CHR$(193);
80 PRINT CHR$(195);CHR$(195);
9 0 PRINT CHR$(195);CHR$(195);
100 PRINT CHR$(195);CHR$(194)
110 REM THE MIDDLE DECK
120 PRINT CHR$(193);CHR$(195);
130 PRINT CHR$(199);CHR$(201);
140 PRINT CHR$(207);CHR$(201);
150 PRINT CHR$(207);CHR$(201);
160 PRINT CHR$(203);CHR$(195);
170 PRINT CHR$(195);CHR$(195);
180 PRINT CHR$(193)
190 REM WATERLINE
200 FOR I=1 TO 12
210 PRINT CHR$(163);
220 NEXT I
230 PRINT
240 PRINT
```

It is not possible to see what the picture is like without rerunning the program. So run it. When you do, you will immediately notice that I have cheated a bit — where did that blue water come from? All will be revealed in a moment.

5 Pretty, pretty

When you ran this little program a few pages back, you saw for the first time (in this book) the range of colours on the DRAGON 32.

```
1Ø FOR I=Ø TO 255
2Ø PRINT CHR$(I);
3Ø NEXT I
```

The 9 available colours all have a number:

Colour 0 is black
Colour 1 is green
Colour 2 is yellow
Colour 3 is blue
Colour 4 is red
Colour 5 is 'buff' (looks white to me)
Colour 6 is cyan (a greeny blue)
Colour 7 is magenta (a purply blue)
Colour 8 is orange

All the characters with numbers between 128 and 255 are coloured graphics symbols. You take 128, add the symbol number as defined a few pages back, and add (colour number − 1) * 16.

Character number = 128 + symbol number + (color number − 1) * 16

For example, the waterline of the QF II is character number 163, which is a black and blue graphics symbol:

128 + symbol no 3 + (3 − 1) * 16

blue

To make this simpler, you could call this

symbol number + thingie

where for the various colours you can look up 'thingie' here:

Thingie 128 is green
Thingie 144 is yellow
Thingie 160 is blue
Thingie 176 is red
Thingie 192 is 'buff'
Thingie 208 is cyan
Thingie 224 is magenta
Thingie 240 is orange

Twelve

YET MORE COLOURFUL

1 Introduction

In this Chapter we learn a second method of making pictures. Here we treat the text screen as a map, and we can then arrange to put any symbol we want anywhere we want, in any colour. A new version of the PRINT statement is used, and we also find out how to blank out the entire text screen. You will notice that I have started to call your screen the 'textscreen'. The Extended Colour BASIC on your fantastic DRAGON 32 gives you yet another screen that we haven't seen yet — the 'graphics screen' which we'll find in the next Chapter.

2 Your text screen is a map — the PRINT @ statement

There are 16 rows and 32 columns on the text screen. Up until now the cursor has moved across the screen as you PRINT, and we know how to make it stay on one line with a semicolon, and how to get it to move to the next one.

Actually every row and column on the text screen has a number or 'address' that you can refer to. The top left corner is address number 0, for example. There is a special version of the PRINT statement that lets you say exactly where on the screen you want something to go. Type this in

```
        PRINT @Ø,"*"
```

and you will see the * appear in the top left corner of the screen. In general, you put

line number PRINT @ address, something to print

0 to 511

in a BASIC program to get something to appear exactly where you want it on the text screen. The addresses run across the screen. Address 0 is the top left corner, address 1 is next to it, and so on. When you run out of room in a row, you drop down into the next one. Here is a program to let you explore the screen addresses. There are 512 different ones — you can give the program any address from 0 to 511 and it will put a blob there:

```
10 REM POKE AROUND THE SCREEN
20 PRINT "GIVE ME A SCREEN ADDRESS"
30 PRINT "AND I'LL PUT A BLOB THERE"
40 INPUT X
50 PRINT@ X,CHR$(207)
70 GO TO 40
```

Here is a formula to help you work out the screen address if you know the row and column numbers. If you call the 32 columns by numbers from 0 to 31 and the 16 rows from 0 to 15, then you can use this formula:

$$\text{Text Screen Address} = \underset{0 \text{ to } 31}{\text{Column Number}} + 32 * \underset{0 \text{ to } 15}{\text{Row Number}}$$

Try this:

```
10 REM SCOTLAND THE BRAVE
20 FOR I=0 TO 510
30 PRINT CHR$(175);
40 NEXT I
50 FOR I=0 TO 15
60 IS=I+32*I
70 PRINT@ IS,CHR$(207)
80 JS=(16-I)+32*I
90 PRINT@ JS,CHR$(207)
100 NEXT I
```

I suppose it's pretty, but it isn't what I want. I'm trying to draw the flag of St Andrew, which is a white X on a blue background. Before I started, I drew enough blue blobs to make the whole screen blue, but then some seem to have been wiped out by the familiar green. Why?

It all has to do with semicolons. Whenever there is a PRINT statement without a semicolon, the cursor goes to the next line but you will find that the line you have just PRINTED gets filled out with green blobs. Look at the previous program. The 16 colours of the flag are each called I. On line I, I want a white blob in row I, column I and also another one in row I, column 15 − I. This worked for the top half of the flag because I made the blob on the left before the one on the right. But on the bottom half my blob was wiped out because I made the blob on the left after the one on the right, so the right hand one was wiped out. We can cure most of it with semicolons:

```
10 REM SCOTLAND THE BRAVE
20 FOR I=0 TO 510
30 PRINT CHR$(175);
40 NEXT I
50 FOR I=0 TO 15
60 IS=I+32*I
70 PRINT@ IS,CHR$(207);
80 JS=(16-I)+32*I
90 PRINT@ JS,CHR$(207);
100 NEXT I
```

I take the trouble to show this to you because it is the one thing that gives most trouble when you are trying to make up graphics on the text screen. Always remember the semicolon to prevent the rest of the line from being wiped out.

Now final improvements. Add the line

```
110 GO TO 110
```

so that the program never finishes. To stop it you will have to push BREAK. Also try this:

```
15 CLS 3
```

which is something new. Nice? You can now take out lines 20, 30 and 40. The CLS statement clears the text screen to whatever colour you specify:

line number CLS colour number

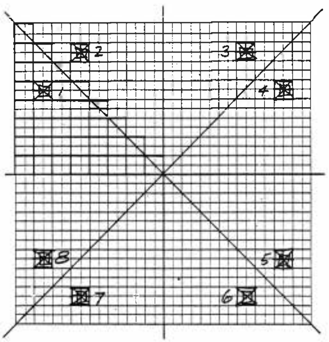
The colour numbers are the same as before:

Colour 0 is black	Colour 5 is 'buff'
Colour 1 is green	Colour 6 is cyan
Colour 2 is yellow	Colour 7 is magenta
Colour 3 is blue	Colour 8 is orange
Colour 4 is red	

Try these out in direct mode. You can get the whole text screen cleared to any colour, but then the 'OK' message spoils it.

3 Your own kaleidoscope

A kaleidoscope is a toy full of little coloured shapes. When you look into it, you see these reflected in mirrors which repeat the pattern of shapes around several lines of symmetry. Let us take the left half of the text screen and divide it into eight segments. If we have a shape in row I, column J, then you can see in the drawing that we want it repeated in the following places:



Place in Drawing	Row No.	Column No.	Address
1	I	J	J + 32*I
2	J	I	I + 32*J
3	J	15−I	(15−I) + 32*J
4	I	15−J	(15−J) + 32*I
5	15−I	15−J	(15−J) + 32*(15−I)
6	15−J	15−I	(15−I) + 32*(15−J)
7	15−J	I	I + 32*(15−J)
8	15−I	J	J + 32*(15−I)

To make the first segment, the area that includes place number 1 in the drawing, we use a series of shapes from the graphics symbols. We need 36 of these for a segment — taking them spaced by 5 from the available shapes seems fair. Here goes:

```
10 REM PHANTASMAGORICAL!
20 CLS Ø
3Ø CH=128
4Ø FOR I=Ø TO 7
5Ø FOR J=Ø TO I
6Ø PRINT@ J+32*I,CHR$(CH);
7Ø PRINT@ I+32*J,CHR$(CH);
8Ø PRINT@ (15-I)+32*J,CHR$(CH);
9Ø PRINT@ (15-J)+32*I,CHR$(CH);
1ØØ PRINT@ (15-J)+32*(15-I),CHR$(CH);
11Ø PRINT@ (15-I)+32*(15-J),CHR$(CH);
12Ø PRINT@ I+32*(15-J),CHR$(CH);
13Ø PRINT@ J+32*(15-I),CHR$(CH);
14Ø CH=CH+3
15Ø NEXT J
16Ø NEXT I
17Ø GO TO 17Ø
```

Furthermore you can make the colours shift and dance forever.

Add

```
3Ø FOR MH=128 TO 143 STEP 5
35 CH=MH
```

which replaces line 30, and also

```
165 NEXT MH
17Ø GO TO 3Ø
```

and run it again.

4 Sailing, sailing, over the ocean blue — animation

For animation, you draw something on the screen and then make it move. You have to be careful about semicolons and things to avoid the nasty green lines. Now is the time to make Queen Fiona II steam across the text screen in stately (well, lurching) fashion by changing the program from page 75.

First of all, make the whole text screen the colour of the sea by adding

```
10 CLS 3
```

and colour the sky cyan — which isn't a bad sky colour if your set is adjusted properly, using

```
12 FOR I=0 TO 319
14 PRINT @I,CHR$(223);
16 NEXT I
```

You could try the program with these few changes, and you will see why we have to go to a lot of trouble to avoid the green. TAB makes green shapes, and you get one at the end of every PRINT statement unless there is a semicolon.

Now we put the drawing of the QF II in a loop with the variable NO taking values from 1 to 19 to represent which column her stern is in as she moves.

```
25 FOR NO=1 TO 19
    :
230 NEXT NO
```

Each time around the loop we deposit a blob of cyan — character number 223 — which rubs out the stern from the last time. Every line has to be started by PRINT @ instead of TAB, and every line has to end with a semicolon to avoid green stripes. At the end, put

```
240 GO TO 240
```

so it will halt without spoiling the picture. Here is the modified program so far:

```
10 CLS 3
12 FOR I=0 TO 319
14 PRINT @I,CHR$(223);
16 NEXT I
20 REM THE SMOKE
25 FOR NO=1 TO 19
30 PRINT @195+NO,CHR$(207);
40 REM THE FUNNEL
50 PRINT @229+NO,CHR$(223);CHR$(192);
60 REM THE TOP DECK
70 PRINT @257+NO,CHR$(223);CHR$(193);
```

```
80 PRINT CHR$(195);CHR$(195);
90 PRINT CHR$(195);CHR$(195);
100 PRINT CHR$(195);CHR$(194);
110 REM THE MIDDLE DECK
120 PRINT @287+NO,CHR$(223);CHR$(193);CHR$(195);
130 PRINT CHR$(199);CHR$(201);
140 PRINT CHR$(207);CHR$(201);
150 PRINT CHR$(207);CHR$(201);
160 PRINT CHR$(203);CHR$(195);
170 PRINT CHR$(195);CHR$(195);
180 PRINT CHR$(193);
190 REM WATERLINE
195 PRINT @319+NO,CHR$(175);
200 FOR I=1 TO 12
210 PRINT CHR$(163);
220 NEXT I
230 NEXT NO
240 GO TO 240
```

Try it! Great. You have to push BREAK to stop it.

A bit more fun is added if the paddle wheels seem to turn. The paddle wheels were originally drawn using CHR$(201). Now if character number 198 was used instead whenever NO was even, the wheels would appear to spin. Look at this change:

```
125 NC=198+3*(NO-INT(NO/2)*2)
130 PRINT CHR$(199);CHR$(NC);
140 PRINT CHR$(207);CHR$(NC);
150 PRINT CHR$(207);CHR$(NC);
```

Try it. Fantastic! Do you see how it works? I am using the remainder when NO is divided by 2 to give me CHR$(198) when NO is even and CHR$(201) when it is odd.

You might also like the QF II to give a blast on her whistle before she starts moving:

```
224 IF NO>1 THEN 230
226 SOUND 1,20
```

The whole program with these last changes is called BOATING on the cassette.

5 Another way to make symbols — SET and RESET

We have seen how to use the PRINT @ statement of Colour BASIC, which gives complete control of where symbols are printed on the text screen. On the DRAGON, this is the most common way of making pictures. Sometimes, however, it may be more convenient to use the SET and RESET statements introduced here. SET can turn on a blob of colour anywhere on the text screen, and RESET turns it off. Although it may be more convenient, it is a bit slower than PRINT @, as will be demonstrated.

You will have noticed that each graphics symbol that can be used with the CHR$ function has four little blobs that make it up. The SET command lets you turn on any little blob in (almost) any colour you want. With SET, the text screen has 32 rows of 64 blobs — four times as many as were used with PRINT @. You write

```
        SET(Column number,  row number,  colour number)

             0 to 63          0 to 31         0 to 8
```

The colour numbers are the same as before:

Colour 0 is black
Colour 1 is green
Colour 2 is yellow
Colour 3 is blue
Colour 4 is red
Colour 5 is 'buff'
Colour 6 is cyan
Colour 7 is magenta
Colour 8 is orange

Try this as a direct command. Push CLEAR to clear your screen first, then enter

```
SET (30,14,3)
```

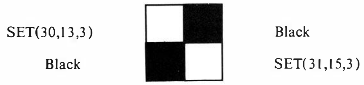
Now enter also

```
SET (31,15,3)
```

Very handy. This looks the same as putting

```
PRINT @ 239, CHR$(137)
```

But there are limitations. You will notice that it isn't red on green — there are black dots as well. SET will always give a colour and black. You are working on the same text screen position as with PRINT @, but are setting on as many little blobs as you need — but you have to put up with the rest of the screen position being black. What you have just done is illustrated by this drawing:



Screen position 239
(Row 7 Column 15)

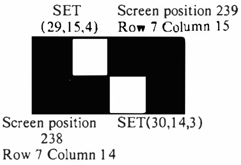
Another limitation is that you can't make the little blobs in one screen position different colours. Clear the screen (by pushing CLEAR) and try this:

```
SET (30,14,3)
SET(31,15,4)
```

Oh dear. The first one was blue, but the second one turned it all red. So why does this next one work?

```
SET (30,14,3)
SET(29,15,4)
```

It works because the text screen positions referred to are next to each other. Confused? Look at this little drawing:



So you can only have black and a colour in any text screen position. This means that you can build all the same things that you could do with the graphics symbols and PRINT @, and nothing more. So why use it? Sometimes it just happens to be easier. But it is a bit slower.

This program takes 32 seconds to draw a row of thingies 50 times:

```
1Ø FOR T=1 TO 5Ø
2Ø FOR K=Ø TO 63 STEP 2
3Ø SET(K+1,2Ø,3)
4Ø SET(K,21,3)
5Ø NEXT K
6Ø NEXT T
```

This one does the same in 18 seconds

```
1Ø FOR T=1 TO 5Ø
2Ø FOR K=32Ø TO 351
3Ø PRINT @K,CHR$(166)
4Ø NEXT K
5Ø NEXT T
```

Can you see why this one is slightly faster, in 15 seconds?

```
10 FOR T=1 TO 50
20 PRINT @320,CHR$(166);
30 FOR K=321 TO 351
40 PRINT CHR$(166);
50 NEXT K
60 NEXT T
```

As an example, we will make a big red spiral on your text screen. We start from a red dot in the middle, and spiral outwards: right one place, up two places, left three places, down four places, right five places and so on. To set the first blob in the middle, put

```
20 CLS 0
30 REM RED BLOB IN MIDDLE
40 X=32
50 Y=15
60 SET (X,Y,4)
```

In making the spiral, we simply adjust the values of X and Y.

To move right put	X=X+1
To move up put	Y=Y−1
To move left put	X=X−1
To move down put	Y=Y+1

We'll do this in a loop for each round of the spiral with four sections inside it: right, up, left, down. Here it is

```
10 REM DON'T FALL IN
20 CLS 0
30 REM RED BLOB IN MIDDLE
40 X=32
50 Y=15
60 SET (X,Y,4)
70 REM NOW SPIRAL
80 FOR RO=1 TO 25 STEP 4
90 REM GO RIGHT, YOUNG PERSON
100 FOR I=1 TO RO
110 X=X+1
120 SET (X,Y,4)
130 NEXT I
140 REM UP, UP AND AWAY
150 FOR I=1 TO RO+1
160 Y=Y-1
170 SET (X,Y,4)
180 NEXT I
190 REM NOW TO THE LEFT
200 FOR I=1 TO RO+2
210 X=X-1
220 SET (X,Y,4)
230 NEXT I
240 REM AND FINALLY DOWN
250 FOR I=1 TO RO+3
260 Y=Y+1
270 SET (X,Y,4)
280 NEXT I
290 NEXT RO
300 GO TO 300
```

The program deliberately sticks at line 300 to preserve the display. Push BREAK to stop it.

You might like to try some variations on this yourself. Make different arms of the spiral different colours. What happens at the corners? Try making every dot a different colour — you can't, can you? Do you remember why? If you tried doing all of this using PRINT @, you would find that you could, but I think you would also find it more difficult. SET is easier to use when you are pushing a dot around the text screen because the (X,Y) position is more convenient to work with than the text screen address that you have to work out with PRINT @. Now add this to the program:

```
105 RESET (X,Y)          205 RESET (X,Y)

155 RESET (X,Y)          255 RESET (X,Y)
```

Well! Now it looks as if you have an insect fluttering around on your screen, and that's exactly what we'll use it for a bit later on. The statement

line number RESET (column number, row number)

works exactly like SET except that it removes the blob from X, Y — and knows what colour to leave behind. This clearly has a lot of possibilities for animation.

6 What's the POINT?

You might want your program to look at the screen to find out what is in a particular text screen position already. You can do this using a function called POINT. Using the same row and column numbers as with SET, you can ask the computer to tell you what is there. You use the function

```
POINT (column number, row number)
           0 to 63       0 to 31
```

which gives you a value

−1 if it's a character
0 if it's blank
the colour code if it's a blob of colour

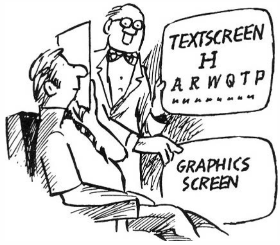
Remember that this is a function, so that it is not a complete statement on its own. You could do this:

```
10 CLS
20 PRINT@239,CHR$(137)
30 PRINT POINT(30,14);POINT(31,14)
40 PRINT POINT(30,15);POINT(31,15)
```

This is a little program to check the truthfulness of what I told you a little while ago about the relationship between SET and PRINT @. Was it correct? Don't be lazy — check it out.

Thirteen

TWO SCREENS FOR THE PRICE OF TWO



1 What's that again?

One of the things that makes the DRAGON 32 such a good value is the Extended Colour BASIC. With it you get a whole new screen, the 'graphics screen' and a lot of advanced facilities in BASIC to help you use it. So the next two chapters are all about the graphics screen, and how to use it.

2 The two screens — the SCREEN statement

Until now, you have used the 'text screen'. The PRINT statement always uses it, and the PRINT @ statement lets you print wherever you want in the 16 rows and 32 columns of the text screen. With the SET and RESET statements you discovered that you could use the text screen as if it had 32 rows and 64 columns although it turned out that this really was the same old16 rows and 32 columns in disguise.

Now you will find that there is a 'graphics screen' as well — hiding inside your computer waiting to be let loose. The graphics screen has up to 192 rows and 256 columns, but there is a choice of resolutions and colours. Using the graphics screen, you can make pictures in much more detail. Here's what you really have:

Two Screens	— Text and Graphics
On each Screen	— Two Colour Sets
On the Graphics Screen	— Four different resolutions

So let's have a look at them. Normally you are looking at the text screen, although the graphics screen always lurks invisibly inside. The SCREEN statement allows you to choose:

```
line number  SCREEN  screen number,  colour set

                       0 or 1          0 or 1
```

You can do this in direct mode but it won't help you much — it will all happen so fast that you can't see it. Here are some little demonstrations.

Normally you are looking at SCREEN 0,0 which is the text screen with black and green display. You could get SCREEN 0,1 which is the text screen again, with a black and orange display. Try this:

```
10 SCREEN Ø,1
2Ø GO TO 2Ø
```

Youch! There it is. Now push BREAK. You will see that you can only hold colour set 1 on the text screen when a program is running. You get forced back to SCREEN 0,0 whenever a program ends, or whenever a PRINT or PRINT @ statement is executed. Try this:

```
1Ø SCREEN Ø,1
2Ø PRINT "WHAT COLOR?"
3Ø GO TO 3Ø
```

You will see that, despite the SCREEN statement, the PRINT has forced you back to colour set 0. If you put SCREEN after PRINT, you get what you want:

```
1Ø PRINT "WHAT COLOR?"
2Ø SCREEN Ø,1
3Ø GO TO 3Ø
```

So what if the thing you printed was a coloured graphics symbol? Well, it doesn't matter, it will stay there. Do this:

```
1Ø SCREEN Ø,1
2Ø FOR I=1 TO 5ØØ:NEXT I
3Ø FOR J=128 TO 255
4Ø PRINT CHR$(J);
5Ø NEXT J
6Ø SCREEN Ø, 1
7Ø GO TO 7Ø
```

Isn't that nice. The graphics stay untouched while the colour set for text is switched back and forth. The PRINT @ statement behaves the same way — it switches the colour set back to green but any coloured graphics will not be touched.

The SET and RESET statements do not affect the colour set in use:

```
10 SCREEN 0,1
20 FOR X=0 TO 31
30 FOR Y=0 TO X
40 SET (X,Y,3)
50 NEXT Y
60 NEXT X
70 FOR X=32 TO 63
80 FOR Y=0 TO 63-X
90 SET (X,Y,5)
100 NEXT Y
110 NEXT X
120 GO TO 120
```

Remember that all these demonstrations have been on the text screen:

line number SCREEN 0, colour set
0 or 1

selects the text screen with

colour set 0 — black on green
colour set 1 — black on orange

3 Now the graphics screen

Try this:

```
10 SCREEN 1,1
20 GO TO 20
```

Say! This is something new — is it a bird, or a plane — no, it's the graphics screen! Push break to stop it. You could also try

```
10 SCREEN 1,0
20 GO TO 20
```

When you use

line number SCREEN 1, colour set
0 or 1

you turn on the graphics screen. Actually there are several ways this might turn out, because you have a choice of resolutions and it affects the colour set. We'll worry about that a bit later. First lets put something on the graphics screen. We use PSET:

line number PSET (column, row, colour)
0 to 255 0 to 191 0 to 8

Notice how this resembles the SET we used on the TEXT screen. The big difference is in the resolution; 0-255 is a lot of columns! Your graphics screen is divided into a lot of little cells which the smarties call 'pixels' — I guess that is short for 'picture cells'. There are actually 256x192 of these — or 49,152. If you're using the highest resolution, this is 4 times as many columns as you could get blocks on the text screen using SET.

You will not be surprised that the statement

```
        PCLS  colour number
```

clears the whole graphics screen to the colour you ask for. Let's try something

```
10 SCREEN 1,1             50 PSET (X,Y,1)
20 PCLS 0                 60 NEXT X
30 FOR Y=1 TO 63          70 NEXT Y
40 FOR X=Y TO 63          80 GO TO 80
```

It takes a while, but you will get a diagonal flag in two colours in the corner of the screen. You can see that this will make better pictures than the text screen because the zigzags are smaller.

Now explore the colour sets. This program makes stripes:

```
10 SCREEN 1,1             70 FOR X=0 TO 255
20 PCLS 1                 80 PSET (X,Y,C)
30 Y=0                    90 NEXT X
40 FOR C=0 TO 8           100 NEXT YY
50 FOR YY=1 TO 4          110 NEXT C
60 Y=Y+1                  120 GO TO 120
```

Oh! Do you see how this works? Expect 8 stripes of depth 4 pixels each, and each a different colour. But you didn't get 8 colours, did you? You will get either 4 or 2 depending on the resolution. Now change line 10 to

```
10 SCREEN 1,0
```

and run it. A different colour set, but still not 8 of them. Let's explain. There is a statement to let you select the resolution, called PMODE. Once you have selected a resolution, your colour sets are fixed. PMODE is like this:

```
line number  PMODE  resolution,  page

                     0 to 4     1 to 8
```

Forget about the 'page' for now and use page 1. Here is what you can have

```
        PMODE resolution,1
```

Resolution code	Resolution on graphics screen columns x rows	Colours set 0	Colours set 1
0	128x96	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff
1	128x96	0,4,8=red 1,5=green 2,6=yellow 3,7=blue	0,4,8=orange 1,5=buff 2,6=cyan 3,7=magenta
2	128x192	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff
3	128x192	0,4,8=red 1,5=green 2,6=yellow 3,7=blue	0,4,8=orange 1,5=buff 2,6=cyan 3,7=magenta
4	256x192	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff

When you first switch on the DRAGON, you get PMODE 0,2. Now try the program that makes stripes with

```
          5 PMODE 3,1
```

and look at both colour sets by changing SCREEN. Refer to the table and try other PMODE and SCREEN combinations. Go over this section until you understand the interaction of PMODE and SCREEN. You always use PSET with column numbers from 0-255 and row numbers from 0-192 even though you can only get at all the pixels individually using PMODE 4,1. Using PMODE 1,1 program STRIPES could have had

```
          5Ø FOR YY= 1 TO 2
          6Ø Y=Y+2
```

because it can only set the pixels in groups of 4. This is a bit like the difference between SET and PRINT CHR$() on the text screen, except that on the graphics screen any pixel you address can be any available colour.

Now with lines 50 and 60 changed as above, try

```
          5 PMODE 3,1
          1Ø SCREEN 1,1
```

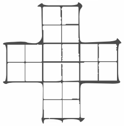
Do you see why the lines aren't solid any more? Try

```
          1Ø SCREEN 1,Ø
```

like this. Red doesn't show up very well on green, does it? Notice that the colour numbers that you are used to will all give you the right colours if you are using a colour set that they belong to.

4 Jose, can you see? — PRESET

We would like to draw Old Glory — red, white and blue. But these colours aren't in the same colour set. I'll use cyan, buff, and orange. In the flag, I want the stars to be a group of buff pixels surrounded by cyan, 6 pixels high:



I can use 128x96 resolution using 4 colours:

```
          1Ø REM JOSE CAN YOU SEE?
          2Ø PMODE 1,1
          3Ø SCREEN 1,1
```

First I'll draw 13 stripes representing the 13 colonies, then a solid block of cyan, and finally the stars — there are 5 rows of 6 stars, and between them 4 rows of 5 stars.

Here's the program, first the stripes. I make sure there isn't anything on the screen first

```
   4Ø REM DEFINE COLORS
   5Ø R=4
   6Ø B=2
   7Ø W=1
   8Ø REM CLEAR THE SCREEN
   9Ø PCLS W
   1ØØ REM DRAW RED STRIPES
   11Ø REM FIRST 3 NARROWER
   12Ø FOR ST=Ø TO 48 STEP 24
   13Ø FOR RO=Ø TO 1Ø STEP 2
   14Ø FOR CO=1Ø8 TO 254 STEP 2
   15Ø PSET(CO,ST+RO,R)
   16Ø NEXT CO
   17Ø NEXT RO
   18Ø NEXT ST
   19Ø REM REMAINING 4 FULL WIDTH
   2ØØ FOR ST=72 TO 152 STEP 24
   21Ø FOR RO=Ø TO 1Ø STEP 2
   22Ø FOR CO=Ø TO 254 STEP 2
   23Ø PSET(CO,ST+RO,R)
   24Ø NEXT CO
   25Ø NEXT RO
   26Ø NEXT ST
```

Now make a solid block of blue:

```
270 REM THE BLUE SKY
280 FOR RO=0 TO 72 STEP 2
290 FOR CO=0 TO 106 STEP 2
300 PSET(CO,RO,B)
310 NEXT CO
320 NEXT RO
```

To put in the stars, we use a new statement

line number PRESET (column, row)

0 to 255 0 to 191

and I think you can guess what it does — it is just like RESET on the text screen. PRESET clears the pixel at (column, row) to the background colour, which at the moment is buff — just what we want. The centre of the stars is going to be at (CO,RO) in the loops, and the four pixels next to this are also reset. First the 5 rows of 6:

```
330 REM 5 ROWS OF 6 STARS
340 FOR RO=12 TO 60 STEP 12
350 FOR CO=12 TO 92 STEP 16
360 PRESET(CO,RO)
370 PRESET(CO+2,RO)
380 PRESET(CO-2,RO)
390 PRESET(CO,RO+2)
400 PRESET(CO,RO-2)
410 NEXT CO
420 NEXT RO
```

and then the ones in between:

```
430 REM AND 4 ROWS OF 5 STARS
440 FOR RO=18 TO 54 STEP 12
450 FOR CO=20 TO 84 STEP 16
460 PRESET(CO,RO)
470 PRESET(CO+2,RO)
480 PRESET(CO-2,RO)
490 PRESET(CO,RO+2)
500 PRESET(CO,RO-2)
510 NEXT CO
520 NEXT RO
530 GO TO 530
```

See how the stars are born? Wow! Hand on heart, and repeat after me:

'I pledge allegiance . . .'

Now try the other colour set. In the program, I called R, W, and B by colour numbers, so you don't have to change much. Whoops! All that green. Try this:

```
35 COLOR 2,2
```

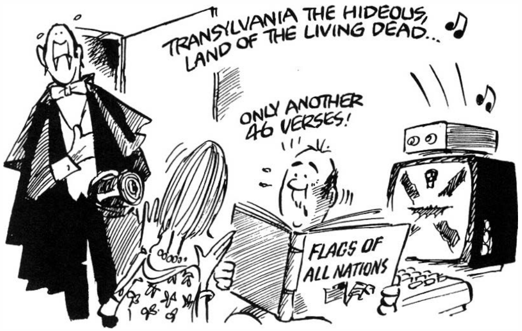
this sets the 'foreground' and 'background' colours both to yellow in colour set 0 with PMODE resolution 1 or 3. We'll see what 'foreground' means in the next chapter. The statement

line number COLOR foreground, background

can be useful. On the graphics screen you always have the same as

```
line number  COLOR 1,8
```

if your program doesn't change this with a COLOR statement.

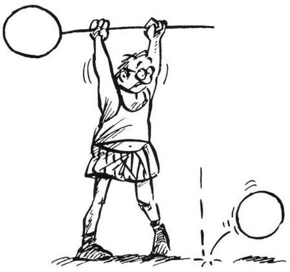


EXERCISE:

If you live somewhere else, make your own flag. If you live in the USA, make the British flag — now that's really a tough one! Add music. That's right — you can splice on the music program from page 68 and put your national anthem in DATA statements.

Fourteen

LINES AND CIRCLES



1 Straight and Narrow

We can already make lines. A magenta horizontal line on the buff graphics screen could be:

```
10 REM HORIZONTAL LINE
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 FOR X=64 TO 191
60 PSET(X,32,3)
70 NEXT X
80 GO TO 80
```

Let's remind ourselves of how it works. The PMODE statement selects the resolution (128x96 with 4 colours) and the page which we're not worried about until later in this chapter. The SCREEN statement turns on the graphics screen and selects colour set 1 (orange, buff, cyan, magenta). PCLS clears everything on the screen to buff, and then the FOR. . . NEXT loop makes a line running along most of the screen. We could make a similar program draw a square magenta box with a cyan cross in it:

```
10 REM X MARKS THE SPOT
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 FOR X=64 TO 191
60 PSET(X,32,3)
70 PSET(X,159,3)
80 PSET(64,X-32,3)
90 PSET(191,X-32,3)
100 NEXT X
110 FOR X=64 TO 191
120 PSET(X,X-32,2)
125 PSET(255-X,X-32,2)
130 NEXT X
140 GO TO 140
```

Notice in this program that the programmer has really had to think about the drawing of each point on the line. If he hadn't been clear, six FOR. . . NEXT loops would have been needed. You can see that to draw any line which is not either horizontal, vertical, or diagonal is going to be even more difficult this way. And notice that it isn't very fast.

Extended Colour BASIC has a special statement for drawing lines on the graphics screen which makes it easy for you. You can write

line number LINE(column 1, row 1) −(column 2, row 2), PSET

and the line will get drawn. As usual you can choose any of columns 0-255 and rows 0-191 of the graphics screen. Try this:

```
10 REM HORIZONTAL LINE
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 LINE(64,32)-(191,32),PSET
60 GO TO 60
```

Fast, isn't it! But the line is orange. Remember the COLOR statement:

line number COLOR foreground, background

Now we can see what the foreground colour is for. The LINE statement with PSET in it draws the line in the foreground colour. You can also have PRESET:

line number LINE(column 1, row 1) −(column 2, row 2), PRESET

which will draw a line in the background colour. So try this:

```
10 REM BUILDING UP
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 COLOR 3,2
60 LINE(64,32)-(191,32),PSET
70 LINE(64,32)-(191,159),PRESET
80 GO TO 80
```

If you run this, you will see magenta and cyan lines drawn on a buff background — so fast you can't see it happen. I think you know what we're building up to. Here is the X in the box drawn using LINE:

```
10 REM X MARKS THE SPOT
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 COLOR 3,2
60 LINE(64,32)-(191,32),PSET
70 LINE(191,32)-(191,159),PSET
80 LINE(191,159)-(64,159),PSET
90 LINE(64,159)-(64,32),PSET
100 LINE(64,32)-(191,159),PRESET
110 LINE(191,32)-(64,159),PRESET
120 GO TO 120
```

But we can be even fancier. If you use

line number LINE(column 1, row 1) −(column 2, row 2), PSET, B instead of a line between the points, you will get a box using those points as corners. So the whole program is:

```
10 REM X MARKS THE SPOT        60 LINE(64,32)-(191,159),PSET,B
20 PMODE 1,1                   100 LINE(64,32)-(191,159),PRESET
30 SCREEN 1,1                  110 LINE(191,32)-(64,159),PRESET
40 PCLS 1                      120 GO TO 120
50 COLOR 3,2
```

Of course you can use PRESET with B. You can also get a solid box by putting BF instead of B; change line 60 to:

```
60 LINE(64,32)-(191,159),PSET,BF
```

Great! So here's a summary of LINE:

```
                                                                  nothing
                                                        PSET        or
line number  LINE (column 1, row 1)—(column 2,row 2),    or   ,     B
                                                       PRESET       or
                                                                    BF
```

This joins the screen points given by (column 1, row 1) and (column 2, row 2). Where

PSET	— uses foreground colour
PRESET	— uses background colour
nothing	— draws a line
B	— draws a box in outline with points at opposite corners
BF	— draws a box and fills it with colour

The foreground and background colours are set by

```
line number  COLOR foreground,  background
                     colour        colour
```

where the colours have to be chosen by the colour set in use.

It would be fun to see the effect of PMODE on the lines. Experiment with line 20 of the program that draws the box. You will see that with higher resolution you get better lines — of course. They are particularly good with PMODE 4,1.

2 Are you still there, Jose

Remember how slowly the star spangled banner was drawn? We can do it all with LINE. Each orange stripe is a box full of colour, and so is the blue part. It certainly makes the program shorter:

```
1Ø REM JOSE CAN YOU SEE?
2Ø PMODE 1,1
3Ø SCREEN 1,1
4Ø REM DEFINE COLORS
5Ø R=4
6Ø B=2
7Ø W=1
75 COLOR R,W
8Ø REM CLEAR THE SCREEN
9Ø PCLS
1ØØ REM DRAW RED STRIPES
11Ø FOR ST=Ø TO 144 STEP 24
12Ø LINE(Ø,ST)-(254,ST+12),PSET,BF
13Ø NEXT ST
27Ø REM THE BLUE SKY
28Ø COLOR B,W
29Ø LINE(Ø,Ø)-(1Ø6,72),PSET,BF
33Ø REM 5 ROWS OF 6 STARS
34Ø FOR RO=12 TO 6Ø STEP 12
35Ø FOR CO=12 TO 92 STEP 16
36Ø PRESET(CO,RO)
37Ø PRESET(CO+2,RO)
38Ø PRESET(CO-2,RO)
39Ø PRESET(CO,RO+2)
4ØØ PRESET(CO,RO-2)
41Ø NEXT CO
42Ø NEXT RO
43Ø REM AND 4 ROWS OF 5 STARS
44Ø FOR RO=18 TO 54 STEP 12
45Ø FOR CO=2Ø TO 84 STEP 16
46Ø PRESET(CO,RO)
47Ø PRESET(CO+2,RO)
48Ø PRESET(CO-2,RO)
49Ø PRESET(CO,RO+2)
5ØØ PRESET(CO,RO-2)
51Ø NEXT CO
52Ø NEXT RO
53Ø LINE(Ø,Ø)-(254,156),PSET,B
54Ø GO TO 54Ø
```

And much faster! The stars are still drawn the old way. You will see that there are two COLOR statements here, one near the beginning which makes the foreground orange and the background white, and the one at line 280 which makes the foreground cyan and the background white. The PCLS statement at line 90 has no colour:

line number PCLS colour number

clears the screen to the given colour number but

line number PCLS

clears the screen to the background colour, which in this case is white.

You will also see that at line 530, a box is drawn around the flag to improve it slightly.

3 Rolling, Bouncing, and Heavy Breathing

Extended Colour BASIC has a fast and convenient statement for drawing circles on the graphics screen — very fast and convenient:

```
line number  CIRCLE (column,    row   ), radius, colour

                     0 to 255  0 to 191
```

The circle is drawn with its centre in the column and row specified, with the radius asked for and in a choice of colours. Try this little program and see:

```
1Ø REM JUST ONE CIRCLE
2Ø PMODE 1,1
3Ø SCREEN 1,1
4Ø PCLS
```

```
        50 CIRCLE(127,95),64,3,1,0,1
        60 GO TO 60
```

Quite a good circle, really. If you think it's a bit jagged, it's because of the resolution — the computer does the best it can with the screen resolution you ask for in PMODE. In this case, we have a resolution of 128 by 192 pixels. As you know, different PMODES will give higher or lower resolution. The highest would be

```
        20 PMODE 4,1
```

but then only two colours are available. If you change line 20 in the above example, you will get a very nice black circle on a green background.

Here is a screen full of circles:

```
10 REM TAKE ONE BEFORE MEALS   50 FOR Y=15 TO 175 STEP 32
20 PMODE 3,1                   60 FOR X=15 TO 239 STEP 32
30 SCREEN 1,1                  70 CIRCLE(X,Y),12,4
40 PCLS                        80 NEXT X
                               90 NEXT Y
                               100 GO TO 100
```

and if you make the circles overlap, it comes out like a golden tapsetry of finest silk:

```
        70 CIRCLE(X,Y),24,4
```

Also try it with PMODE 4,1 — but change the colour of the circles to an odd number.

CIRCLE is fast enough to try some animation. Here a circle rolls across your screen:

```
10 REM ROLLERBALL
20 PMODE 4,1
30 SCREEN 1,1
40 PCLS
50 FOR X=1 TO 255 STEP 16
60 CIRCLE(X,83),12,1
70 NEXT X
```

Well, actually you get a trail of circles. If you want the impression of one circle moving, you'll have to wipe out each old one. If you put this statement in, the old ones are wiped before the new ones are drawn:

```
55 PCLS
```

or, instead you could use

```
65 CIRCLE(X-16,83),12,2
```

which removes the old circle after the new one is drawn by going over it again in the background colour. Which do you think is best?

And now the bouncing. Do you remember old Newton? No, not the one who played Long John Silver — the scientist Isaac Newton. Quite a smartie, he was. One day, the story goes, an apple fell on his head which made him recognize the gravity of the situation. Ugh! Anyway, a bouncing ball obeys laws of motion identified by Newton. If you drop an object, it falls with constant acceleration (ignoring wind resistance), and the distance it travels in metres, called d, is given by a formula

$$d = 4.9\,t^2$$

where t is the time in seconds since you dropped it. If we're smart and drop a ball at the top of the graphics screen and pretend that the screen is about 313 metres high, then the ball falls as shown by this little table.

After this no. of seconds	The ball has reached this row
1	2
2	11
3	26
4	47
5	74
6	107
7	146
8	191

Too cunning. I've manipulated things so that

$$\text{Row Number} = 3t^2 - 1$$

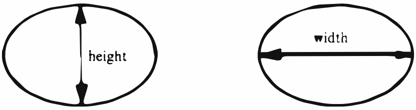
The ball hits the bottom of the screen at 8 seconds. Because of another of Newton's laws it will bounce back again — a really good superball will bounce right back to the top of the screen, getting there after 16 seconds. Lets have the ball moving slowly to the right at the same time — Newton says that this motion will go on forever. We'll have it cross the screen in 16 seconds. Here is this highly scientific demonstration — all three of Newton's laws in one little program:

```
1Ø REM AN APPLE A DAY
2Ø PMODE 4,1
3Ø SCREEN 1,1
4Ø PCLS
5Ø FOR T=1 TO 8
6Ø RO=3*T*T-1
7Ø CO=16*T-9
8Ø CIRCLE(CO,RO),12,1
9Ø NEXT T
1ØØ FOR T=9 TO 16
11Ø TT=17-T
12Ø RO=3*TT*TT-1
13Ø CO=16*T-9
14Ø CIRCLE(CO,RO),12,1
15Ø NEXT T
16Ø GO TO 16Ø
```

Have you been waiting for the heavy breathing? I hope you are not disappointed. First of all, the CIRCLE statement can draw circles which are squashed flat from above or from the side. Actually they're 'ellipses'. You can make the ratio of the height to width of the 'circle' anything from 0.0 to 4.0:

line number CIRCLE (column, row), radius, colour, shape

where 'shape' is the height to width ratio as in the drawing:



So run this next little program. You get a screen with 12 circles, not unlike program CIRCLES, but each is a different shape. The height to width ratios are 0 (a straight line), 0.25, 0.50, 0.75, 1.0 (a circle) and so on.

```
1Ø REM LOTS OF DIFFERENT SHAPES
2Ø PMODE 4,1
3Ø SCREEN 1,1
4Ø PCLS
5Ø SH=Ø.Ø
6Ø FOR Y=31 TO 159 STEP 64
7Ø FOR X=31 TO 223 STEP 64
8Ø CIRCLE(X,Y),12,1,SH
9Ø SH=SH+Ø.25
1ØØ NEXT X
11Ø NEXT Y
12Ø GO TO 12Ø
```

And here's the panting part. You can make a shape that breathes. How's it done? A series of circles is drawn with the height to width ratio gradually changing. Then they're taken away by drawing over the top in black. Inhale, exhale, inhale, exhale . . . .

```
10 REM HEAVY BREATHING
20 PMODE 2,1
30 SCREEN 1,1
40 PCLS
50 FOR J=0 TO 10
60 SH=J/10
70 CIRCLE(127,96),64,1,SH
80 NEXT J
90 FOR J=10 TO 0 STEP -1
100 SH=J/10
110 CIRCLE(127,96),64,2,SH
120 NEXT J
130 GO TO 50
```

4 And there's more, too

There is still more to the circle statement. You learned how to make a circle:

line number CIRCLE (column, row), radius, colour

and also to draw a 'squashed circle' or ellipse:

line number CIRCLE (column, row), radius, colour, shape

You can also draw any part of a circle:

line number CIRCLE (column, row), radius colour, shape, start, end

A whole circle has start=0 and end=1, like line 50 in

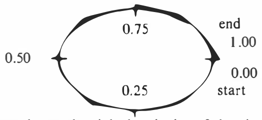
```
10 REM JUST ONE CIRCLE
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS
50 CIRCLE(127,96),64,3,1,0,1
60 GO TO 60
```

You can draw the bottom half of a circle by putting instead:

```
50 CIRCLE(127,95),64,3,1,0,.5
```

or the top half:

```
50 CIRCLE(127,95),64,3,1,.5,1
```



As the little diagram shows, the right hand edge of the circle is 0 and also 1. When you draw a circle, the circle starts at 'start' and goes clockwise to 'end'. Therefore the bottom is 1/4 of the way around, the left side is 1/2, the top is 3/4 and at 1 the circle returns again to the right hand edge.

The left half of a circle would be drawn by:

```
          50 CIRCLE(127,95),64,3,1,.25,.75
```

To draw the right half, the DRAGON manual said this wouldn't work:

```
          50 CIRCLE(127,95),64,3,1,.75,1.25
```

but it did! The manual says that start and end should both be between 0 and 1. I have found that values between 0 and 4 are accepted. However the same half circle can be drawn by

```
          50 CIRCLE(127,95),64,3,1,.75,.25
```

because the computer starts at 'start' and continues on to 'end'. So you can get any part of a circle you want using the full-blown CIRCLE statement.

Fifteen

GET OUT YOUR PAINT BRUSH

1 Slap it on

The PAINT statement in Extended Color BASIC is very useful and clever. You can paint the inside of any shape on the graphics screen with any available color, and the computer works out exactly how far to go:

line number PAINT(column, row), color, border color

When you use this statement, your DRAGON starts rolling on the paint at the column and row you ask for, using the specified color, and fills an area surrounded by a border whose color you give. As a simple example, we draw a white circle on a black background, and then paint it all white.

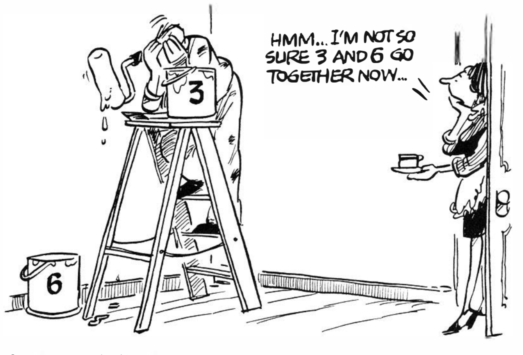
```
10 REM IS THIS SOMEONE'S FLAG?
20 REM WHITE IS FOR PURITY AND
30 REM BLACK IS FOR ....
40 PMODE 4,1
50 SCREEN 1,1
60 PCLS
70 REM DRAW A WHITE CIRCLE
80 CIRCLE(127,96),64,1
90 REM NOW PAINT IT WHITE
100 PAINT(127,96),1,1
110 GO TO 110
```

This next little program is based on an overlapping circle program from the previous chapter, but paints each circle a different color. The result is like a nice old fashioned patchwork quilt. The original circles are orange, and each new one is painted to an orange border. Do you see how the remainder is used at statement 110 to keep the color number CL inside the allowed range. This is set up so that it repeats 0,1,2,3,4,0,1,2,3,4,0 and so on — orange is both 0 and 4 in this color set.

```
10 REM KEEPS YOU WARM IN WINTER
20 PMODE 3,1
30 SCREEN 1,1
40 PCLS 3
50 CL=0
60 FOR Y=15 TO 175 STEP 32
70 FOR X=15 TO 239 STEP 32
80 CIRCLE(X,Y),24,4
90 PAINT(X,Y),CL,4
100 CL=CL+1
110 CL=CL-INT(CL/5)*5
120 NEXT X
130 NEXT Y
140 GO TO 140
```

And now the Union Jack. Its easier now with PAINT. First a cyan background is drawn. Then St Andrew's flag (for Scotland) is drawn in white. The red St Patrick's flag (for Ireland) goes on top but it is drawn a bit off centre. St George's cross of England is last — red on a white background as represented by a white border round it. We do it all with **LINE** to give the outlines, and **PAINT** to fill it in. You can see that **PAINT** never makes a mistake even in these complicated crosses.

```
10 REM BRITTANIA WAIVES THE RULES
20 PMODE 1,1
30 SCREEN 1,1
40 R=4
50 B=2
60 W=1
70 PCLS 1
80 REM THE DEEP CYAN SEA
90 COLOR W,B
100 LINE(0,0)-(255,159),PRESET,BF
110 REM ST ANDREW'S WHITE CROSS
120 LINE(23,0)-(255,145),PSET
130 LINE(0,14)-(232,159),PSET
140 PAINT(0,0),W,W
150 LINE(0,145)-(232,0),PSET
160 LINE(23,159)-(255,14),PSET
170 PAINT(255,0),W,W
180 PAINT(0,156),W,W
190 COLOR R,W
200 REM ST PATRICK IS TRICKY
210 REM FIRST MAKE A DIAGONAL
220 LINE(0,0)-(255,159),PSET
230 LINE(0,159)-(255,0),PSET
240 REM NOW DRAW AND PAINT ARMS
250 LINE(0,9)-(127,88),PSET
260 LINE(127,88)-(127,79),PSET
270 PAINT(0,2),R,R
280 LINE(255,150)-(128,70),PSET
290 LINE(128,70)-(128,79),PSET
300 PAINT(255,157),R,R
310 LINE(15,159)-(127,88),PSET
320 LINE(0,159)-(15,159),PSET
330 PAINT(13,157),R,R
340 LINE(240,0)-(128,70),PSET
350 PAINT(242,0),R,R
360 REM BACKGROUND FOR ST GEORGE
370 LINE(108,0)-(147,159),PRESET,BF
380 LINE(0,64)-(255,95),PRESET,BF
390 REM AND THE CROSS ITSELF
400 LINE(112,0)-(143,159),PSET,BF
410 LINE(0,68)-(255,91),PSET,BF
420 GO TO 420
```



●K Somebody — do Canada!

2 So what's the PPOINT?

The POINT function, in section 6 of Chapter 12, allowed you to look at a place on the text screen to find out what was there. On the graphics screen, you can do the same thing with the function PPOINT. If you use the function

```
          PPOINT (column number, row number)

                       0 to 255     0 to 191
```

you are given the color code of the graphics position specified. As an example, if you change line 420 to

```
42Ø PRINT PPOINT(127,88)
```

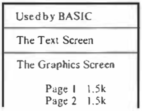
in the British Flag program, it tells you that the graphics cell (127,88) had color code 8 in it, which is red in that particular PMODE. Note that the DRAGON switches to the text screen to print this. If you're really observant you will notice that the program originally used color 4 for point (127,88). The table of colors for each PMODE resolution in section 3 of chapter 13 shows that colors 0, 4, and 8 are equivalent in PMODE resolution 1. You will also notice that you can enter

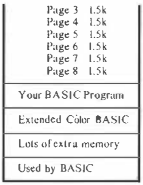
```
PRINT PPOINT (127,88)
```

as a direct command. If you do this after running the program, you will again get color 8. The British flag is still there on the graphics screen after the program terminates. It will stay there until you put something else on top of it, or wipe it out with something like a PCLS command.

3 Thanks for the Memory

Your DRAGON 32 has a lot of memory—that is one reason why it is such a good value. The maximum amount of memory that is available for you to use for the graphics screen is 12K bytes. A byte is a piece of memory that can remember one character or a few pixels. This extra 12K (1K is 1024 bytes) can be used for large BASIC programs, but its main use is for the graphics screen. This 12K of memory is divided into 8 'pages' of 1.5K each. The graphics screen that you see on your television is a picture of part of this memory. Here is a little map of the memory:





With the PMODE statement you are given a choice of different resolutions and numbers of colors. Different PMODE choices use different amounts of memory. You can see from this table that a higher resolution or a larger choice of colors takes more memory — from 1 to 4 pages depending on the combination.

PMODE resolution,1

Resolution chosen	Resolution of screen columns x rows	No. of colors	No. of pages of memory used
0	128x96	2	1
1	128x96	4	2
2	128x192	2	2
3	128x192	4	4
4	256x192	2	4

You will be able to see from the memory map that it should be possible for a large program to take memory away from the graphics pages if it needs to. In fact the DRAGON normally assumes that you want only pages 1-4 for graphics and it can use pages 5-8 for your program. However a program has to be fairly large before you really need to borrow any of these pages no program in this book does this.

So it seems that there is a lot of memory available for the graphics screen that is not being used. What can we do with it? Well, we can actually make as many different pictures as we can fit into this part of memory, and then switch back and forth between them. This is done using the other parameter in the PMODE statement:

PM ODE mode, page

The 'page' parameter tells the DRAGON which page you want your graphics screen to use. Mode 0 requires only one page to make a screen, and so there could be 8 different pictures using mode 0 in memory at the same time. Mode 1 uses 2 pages, so a statement like

PMODE 1,1

uses up pages 1 and 2 of memory. There can be four different pictures in mode 1 or mode 2. Similarly you can have two different pictures in modes 3 and 4.

So lets use them. First of all a new statement:

line number PCLEAR number of pages

This statement tells the DRAGON how many pages you want to use for graphics, starting from page no. 1. When you turn on a DRAGON, the first four pages are automatically reserved for graphics. So the DRAGON normally assumes that you are using pages 1-4 for graphics screens and that pages 5-8 can be used for larger BASIC programs. This is why we have not needed a PCLEAR statement before, because we have always used

PMODE mode, 1

which starts the graphics screen at page 1 and can never use more than the four graphics pages already reserved. But if you switch the computer on and then try

PMODE 3,3

you will get an error message. This is because mode 3 starting at page 3 wants to use pages 3, 4, 5, and 6. But pages 5 and 6 aren't available because you haven't reserved them. Remember that you need to use PCLEAR only if you want more than 4 pages of memory for graphics (or less, if your program is huge).

4 Rolling again and Bouncing again

Using the different graphics pages we can make much better animated graphics than are possible just by drawing and redrawing pictures in the same page. First of all we will see a ball rolling back and forth across the screen, hitting the edges as it does. Using resolution 0, the ball can be represented in 8 different pictures. We then switch the pictures at a constant rate to give the impression of movement.

```
1Ø PCLEAR 8
2Ø REM MAKE 8 DIFFERENT PICTURES
3Ø FOR PG = 1 TO 8
4Ø PMODE Ø,PG
5Ø SCREEN 1,1
6Ø PCLS 1
7Ø REM DRAW AN OUTLINE
8Ø LINE(Ø,Ø)-(255,191),PRESET,B
9Ø REM MAKE THE FLOOR
1ØØ LINE (Ø,176)-(255,191),PRESET,BF
11Ø REM AND DRAW THE BALL
12Ø X=PG*32-17
13Ø CIRCLE (X,144),32,Ø
14Ø PAINT (X,144),Ø,Ø
15Ø NEXT PG
16Ø REM NOW START ROLLING
17Ø REM FIRST TO THE RIGHT
18Ø FOR PG=1 TO 8
19Ø PMODE Ø,PG
2ØØ SCREEN 1,1
21Ø FOR T=1 TO 2Ø
22Ø NEXT T
23Ø NEXT PG
24Ø REM AND THEN TO THE LEFT
25Ø FOR PG=7 TO 2 STEP -1
26Ø PMODE Ø,PG
27Ø SCREEN 1,1
28Ø FOR T=1 TO 2Ø
29Ø NEXT T
3ØØ NEXT PG
31Ø GO TO 18Ø
```

You will notice the FOR... NEXT loops at 210-220 and 280-290 which do not appear to do anything. What they do is control the rates at which the pictures are switched by taking up a certain amount of time. If you were to decrease the top limit of the loop, the ball would move faster; if you increased it, the ball would be slower.

You will see that after each PMODE statement there is a SCREEN statement. If it wasn't there you would not see the pictures. You could make the pictures in secret and look at them afterwards. Try changing line 50 to

```
50 PRINT"NOW I'M MAKING PICTURE";PG
```

tricky!

After you have run a program, the pictures are still there in the graphics pages until you run another program or turn the machine off. This means that you could have one program to make the pictures and another to display them. This would save you the bother of waiting for the pictures to be made if you wanted to look at them again. Here's a program to make the rolling balls:

```
10 PCLEAR 8
20 REM MAKE 8 DIFFERENT PICTURES
30 FOR PG = 1 TO 8
40 PMODE 0,PG
50 PRINT"NOW I'M MAKING PICTURE";PG
60 PCLS 1
70 REM DRAW AN OUTLINE
80 LINE(0,0)-(255,191),PRESET,B
90 REM MAKE THE FLOOR
100 LINE (0,176)-(255,191),PRESET,BF
110 REM AND DRAW THE BALL
120 X=PG*32-17
130 CIRCLE (X,144),32,0
140 PAINT (X,144),0,0
150 NEXT PG
```

and another one to look at them if they're already there:

```
10 REM DISPLAY 8 PICTURES
20 PCLEAR 8
30 REM SHOW THEM IN ORDER
40 FOR PG=1 TO 8
50 PMODE 0,PG
60 SCREEN 1,1
70 FOR T=1 TO 20
80 NEXT T
90 NEXT PG
100 REM AND THEN IN REVERSE
110 FOR PG=7 TO 2 STEP -1
120 PMODE 0,PG
130 SCREEN 1,1
140 FOR T=1 TO 20
150 NEXT T
160 NEXT PG
170 GO TO 40
```

This last program could be used to look at any animation in resolution 0. Here's another program to make balls that really bounce — and properly too, moving more slowly at the top of the bounce and faster at the bottom as under the influence of gravity. Remember this from before?

```
10 REM BOUNCY, BOUNCY
20 PCLEAR 8
30 REM DRAW 8 PICTURES OF BALL
40 REM FALLING UNDER GRAVITY
50 FOR T=1 TO 8
60 PRINT "DRAWING IN POSITION";T
70 PMODE 0,T
80 PCLS 1
90 RO=3*T*T-1
100 CIRCLE (127,RO),12,0
110 NEXT T
```

Why don't you make an animated picture and use my program to look at it? How about a running figure, or a face changing its expression or wiggling its ears.

5 And finally

There is a PCOPY statement which can also be used for animation. It is

line number PCOPY page number TO page number

Obviously you can only copy to a page which has been reserved by PCLEAR. You can use this for animation by copying different pictures into the page which is being shown on the screen. Sometimes it can make programs shorter if you're drawing complicated pictures which are almost the same. Then you PCOPY the part that doesn't change — the face without the ears — and then just draw the ears.

There are two more graphics statements, DRAW and PUT. DRAW uses character string variables and is covered in Chapter 23. PUT uses arrays and is covered in Chapter 24.

Sixteen

PURELY BY CHANCE

1 Introducing Randy Random

Randy Random is a silly fellow. He never knows where he is going next. But he can be useful, and fun. He is actually a function, called RND. If you write something like

```
10 FOR I=1 TO 200
20 PRINT RND(10);
30 NEXT I
```

your screen will fill up with apparently unrelated numbers. This is because our friend RND (for short) is a 'random number generator'. The result of this is always a number between 1 and 10 that you can't predict. You can use this for all kinds of interesting things.

2 Scaling our friend Randy

RND is a function like INT or ABS; it gives you a value when you use it. In the case of RND, using

RND(X)

gives you a 'random number' between 1 and X each time you use it. The random numbers appear to be unrelated and so can be used for all kinds of interesting things. For example, generating numbers between 1 and 2 is like tossing a coin. Suppose we write a program to use 1 = heads and 2 = tails, like this:

```
10 REM TOSS A COIN
20 PRINT "WHEN YOU PUSH RETURN"
30 PRINT "I'LL TOSS A COIN"
40 INPUT X
50 RN=RND(2)
60 IF RN=1 THEN 90
70 PRINT "IT WAS A TAIL"
80 GO TO 20
90 PRINT "IT WAS A HEAD"
100 GO TO 20
```

The chances are that you don't want a number between I and 2 or 1 and 10. You may want one between some minimum value and some maximum value. Call the minimum MN (Minnie Minimum?) and the maximum MX (Maximillian Maximum?). To get a random integer RN which is always between Minnie and Maximillian, write

```
line number RN = MN−1 + RND(MX−MN + I)
```

This is quite easy to do, as examples in the next two sections illustrate.

3 Do you like modern music? Neither do I.

I heard an appalling piece on the radio yesterday, so here's my effort at producing a similarly horrible noise — roll over Rachmaninov! We can do even better (worse?) than Hans Weiner Henze because our orchestra is not limited to the notes of a conventional scale. I want a tone in the range 5-244, which I can get from

```
30 TN=4+RND(240)
```

and a duration from 1-10, as in

```
40 DR=RND(10)
```

Here it is. Play on. Give me excess of it that, surfeiting, the appetite may sicken and so die . . .

```
10 REM WELCOME TO THE
20 REM DRAGON CONCERT HALL
30 TN=4+RND(240)
40 DR=RND(10)
50 REM EARWASH
60 SOUND TN,DR
70 GO TO 30
```

4 Do you like modern art? At least it's quiet.

As another example, let's write a program to fill the text screen with random graphics shapes in random screen positions. The coloured graphics shapes have character numbers between 128 and 255. So to get a random one, use

```
CH = 127 + RND(I28)
```

The screen positions in the PRINT @ statement run from 0 to 511, so we use

```
SC = RND(512)−1
```

Do you understand the − 1? Here is the program:

```
10 REM WELCOME TO THE DRAGON
20 REM GALLERY OF MODERN ART
30 CLS
40 CH=127+RND(128)
50 SC=RND(512)-1
60 REM EYEWASH
70 PRINT@SC,CHR$(CH)
80 GO TO 40
```

EXERCISE:

I think these last two programs would go rather well together. Combine them. The critics would say: 'the brutal juxtaposition of aural and visual textures indicate the obdurate iconoclasticism of author and composer as artist.'

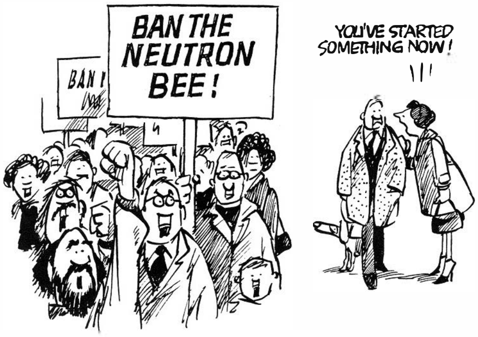
5 Shifting colours

When I wrote the kaleidoscope program in Chapter 12, I did not like the obvious order in which the kaleidoscope produced its blocks of colour. Here, in much the same program, I use a random graphics symbol at a random place instead of a predictable one. I suppose you could add some sound if you wanted to:

```
10 REM PHANTASMAGORICAL!
20 CLS 0
30 CH=128
40 I=RND(8)-1
50 J=RND(I+1)-1
60 CH=127+RND(128)
70 PRINT@ J+32*I,CHR$(CH);
80 PRINT@ I+32*J,CHR$(CH);
90 PRINT@ (15-I)+32*J,CHR$(CH);
100 PRINT@ (15-J)+32*I,CHR$(CH);
110 PRINT@ (15-J)+32*(15-I),CHR$(CH);
120 PRINT@ (15-I)+32*(15-J),CHR$(CH);
130 PRINT@ I+32*(15-J),CHR$(CH);
140 PRINT@ J+32*(15-I),CHR$(CH);
150 GO TO 40
```

6 The fable of the Bomb and the Buzzing Bee

There is a neutron bomb at column 15 of row 7 on your text screen. A bee is flying around on your screen. Eventually . . .



In this program, I use two random numbers to tell me where the bee is going next. The column number is XB, and it can move one place to the left or right each time (or not at all). This is done by

```
90 XB=XB-2+RND(3)
```

we have to keep the bee on the screen:

```
100 REM MUST KEEP IT ON THE SCREEN
110 IF XB>-1 THEN 130
120 XB=0
130 IF XB<32 THEN 160
140 XB=31
```

The row number, YB, is treated similarly and the address of the bee on the screen for the PRINT @ statement is calculated from XB,YB as:

```
220 SC=XB+32*YB
```

the bee is shown in yellow:

```
210 REM FLASH THE YELLOW BEE
220 SC=XB+32*YB
230 IF SC=511 THEN 25Ø
24Ø PRINT@SC,CHR$(159);
25Ø FOR T=1 TO 2Ø
26Ø NEXT T
```

I'm going to leave a row of dots where the bee has been. There is an explosion when it eventually gets there.

```
1Ø REM UNFAIR TO BUGS
2Ø CLS 5
3Ø REM PUT IN THE BOMB
4Ø PRINT@239,CHR$(191);
5Ø REM BEE STARTS AT RANDOM
6Ø XB=RND(32)-1
7Ø YB=RND(16)-1
8Ø REM BEE MOVES IN X BY -1, Ø OR 1
9Ø XB=XB-2+RND(3)
1ØØ REM MUST KEEP IT ON THE SCREEN
11Ø IF XB>-1 THEN 13Ø
12Ø XB=Ø
13Ø IF XB<32 THEN 16Ø
14Ø XB=31
15Ø REM BEE MOVES SIMILARLY IN Y
16Ø YB=YB-2+RND(3)
17Ø IF YB>-1 THEN 19Ø
18Ø YB=Ø
19Ø IF YB<16 THEN 22Ø
2ØØ YB=15
21Ø REM FLASH THE YELLOW BEE
22Ø SC=XB+32*YB
23Ø IF SC=511 THEN 25Ø
24Ø PRINT@SC,CHR$(159);
25Ø FOR T=1 TO 2Ø
26Ø NEXT T
27Ø REM AND LEAVE A DOT BEHIND
28Ø PRINT@SC,CHR$(2Ø6);
29Ø REM HAS THE BEE HIT BOMB ?
3ØØ IF SC><239 THEN 9Ø
31Ø REM NOW THE EXPLOSION
32Ø CLS Ø
33Ø REM RED BLOB IN MIDDLE
34Ø X=32
35Ø Y=15
36Ø SET (X,Y,4)
37Ø REM NOW SPIRAL
38Ø FOR RO=1 TO 25 STEP 4
39Ø REM GO RIGHT, YOUNG PERSON
4ØØ FOR I=1 TO RO
41Ø X=X+1
42Ø SET (X,Y,4)
43Ø NEXT I
44Ø REM UP, UP AND AWAY
45Ø FOR I=1 TO RO+1
46Ø Y=Y-1
47Ø SET (X,Y,4)
48Ø NEXT I
49Ø REM NOW TO THE LEFT
5ØØ FOR I=1 TO RO+2
51Ø X=X-1
52Ø SET (X,Y,4)
53Ø NEXT I
54Ø REM AND FINALLY DOWN
55Ø FOR I=1 TO RO+3
56Ø Y=Y+1
57Ø SET (X,Y,4)
58Ø NEXT I
59Ø NEXT RO
6ØØ GO TO 6ØØ
```

The explosion is the red spiral from Chapter 12. Well, if all the great composers re-used material, why shouldn't we?

Seventeen

MAKE A LIST

1 Lists and subscripts

The variable names we have used so far represent a single value. We already know what is meant by

```
        120 XB=0
```

Here, the value 0 is assigned to the variable **XB** for further use in our program, and it holds this value until we change it.

BASIC also allows you to assign a list of values to a variable name. When you do this, you have what is called an 'array'. Try this:

```
10 REM FIRST MAKE A LIST
20 FOR I=0 TO 10
30 RN(I)=RND(100)
40 NEXT I
50 REM AND THEN PRINT IT
60 FOR I=0 TO 10
70 PRINT RN(I);
80 NEXT I
```

How interesting. On the screen you will see 11 random numbers between 1 and 100, but if you look at the program you will see that the PRINT statement was not in the same loop as the RND function. The key is in the statement

```
        30 RN(I)=RND(100)
```

The variable RN(I) in this program represents a list of values and the first FOR... NEXT loop assigns random numbers to item number I of the list with I running from 0 to 10. BASIC recognizes that RN(I) is a list because we have used the subscript (I) with it. Therefore

RN(3) refers to the third random number in the above program
X(9) refers to the ninth entry in the list called X

AB(I+J) refers to the (I+J)th value in the list called AB

A subscript can be any expression, but it is obvious that it has to be interpreted as a positive integer. In BASIC the lowest subscript is 0. If the value of a subscript is not an integer, it is chopped off to make one. This means that

PQ(4.66) refers to the fourth entry in the list called PQ

The DRAGON will check every subscript as it is used, and if it is less than zero or too large, the program will not be allowed to continue. You can use the same variable name to represent both an ordinary value and a list in a program on the DRAGON but this could cause you trouble if you try to run a program on another machine. It's better not to. We'll come to the question of how large a list can be a bit later.

2 Using a list

Suppose Narrowdale Prison has 4 cells — a bit small, but this one is for long stay VIPs such as heads of state who inflict savage monetarist policies on the people. Each prisoner is given a number by which we know them. A list could be used to represent the prisoner numbers. This list could be described as

The prisoner in cell 1 is number 728
The prisoner in cell 2 is number 213
The prisoner in cell 3 is number 900
The prisoner in cell 4 is number 463

Let's call the list of prisoner numbers PN. There are four cells, and so we need to define PN(1), PN(2), PN(3), and PN(4). Remember the DATA statement from Chapter 10? Here is a little program which reads the prisoner numbers from a DATA list, and prints them out again. In a FOR. . NEXT loop the READ statement at line 50 gets the prisoner number for cell number I:

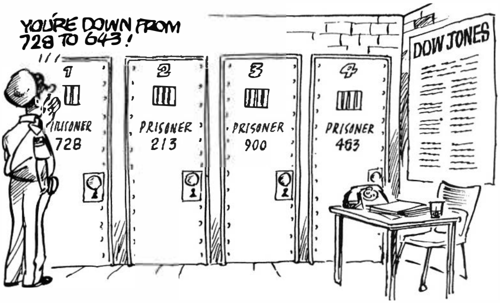
```
10 REM JAILHOUSE ROCK
20 DATA 728,213,900,463
30 REM FIRST DEFINE 'EM
40 FOR I=1 TO 4
50 READ PN(I)
60 NEXT I
70 REM NOW PRINT 'EM
80 FOR J=1 TO 4
90 PRINT PN(J)
100 NEXT J
```

If you wanted to type the numbers in instead, you could have put

```
50 INPUT PN(I)
```

Now let's look for the highest prisoner number. Searching a list is a very common thing in computing — and there are some special ways of doing it. There is nothing special, however, about the way I'm doing it here. It is called a linear search. To do the search, we need to set aside a variable for the answer — HI in this case. We could give it as an initial value a lower prisoner value than we ever expect to find. It would be clearer, however, to use PN(1) as the first value of HI. We then search along the list for a higher number, and put into HI if we find one. Graft this onto the end of the previous program:

```
110 REM FIND HIGHEST NUMBER
120 HI=PN(1)
130 FOR J=2 TO 4
140 IF PN(J)<=HI THEN 160
150 HI=PN(J)
160 NEXT J
170 PRINT "HIGHEST IS"HI
```



Easy? You should be able to see how this works. Finding the largest or smallest value from a list is something you often have to do.

3 99 years is almost for life

It is common for programs to deal with several lists at once, which are related; for example we might also wish to know the age and remaining sentence of each prisoner:

Cell	1	2	3	4
Prisoner Number	728	213	900	463
Remaining Sentence	37	99	61	12
Age	28	41	16	24

Call the age AG, and the sentence SN. In this little fragment of program, one data statement is used for each prisoner:

```
10 REM PRISON RECORDS
20 REM DATA IS IN CELL ORDER
30 REM NUMBER, SENTENCE, AGE
40 DATA 728,37,28
50 DATA 213,99,41
60 DATA 900,61,16
70 DATA 463,12,24
80 REM READ THE DATA
90 FOR I=1 TO 4
100 READ PN(I),SN(I),AG(I)
110 NEXT I
120 REM PRINT DATA AS A TABLE
130 PRINT" NO. SENT. AGE"
140 FOR J=1 TO 4
150 PRINT PN(J);SN(J);AG(J)
160 NEXT J
```

EXERCISE:

Now you do some work. Search this data here to find the prisoner who will be the youngest when released. Show on the screen the cell number, prisoner number, present age, remaining sentence, and age when released.

4 Are the dice loaded? — a historygram

An array can be used to keep track of how many times various events occur — it is often all just counting. For example, when you throw a die you expect a result from 1 to 6. If we do a lot of throws, we can use an array DI to count the number of 1's that occur in DI(1), the number of 2's in DI(2), and so on. As well as printing the result at the end, let's draw the result as a bar graph — you could call this a 'histogram'.

First of all the throwing. We generate a random number from 1-6 called TH, and all we have to do to count the occurences is add 1 to DI(TH) each time. Quite easy if you think about it:

```
10 REM COMPULSIVE GAMBLERS
20 CLS
30 REM FIRST THROW THE DICE
40 PRINT "THROWING NOW"
50 FOR I=1 TO 36
60 TH=RND(6)
70 DI(TH)=DI(TH)+1
80 NEXT I
90 FOR I=1 TO 6
100 PRINT DI(I);
110 NEXT I
```

Now please do try this program, because there is an interesting point to be made about random numbers. If you throw the die 36 times, you might expect about 6 occurences of each number. But not exactly. If it was exactly 6, your random numbers wouldn't be random, would they?

Now the histogram. In the lower part of the text screen I draw little columns containing the number of blobs to show how many occurrences.

```
12Ø REM MAKE THE HISTORYGRAM
13Ø FOR I=1 TO 6
14Ø FOR RP=1 TO 2
15Ø CO=3*(I-1)+RP
16Ø FOR RO=16-DI(I) TO 15
17Ø SC=CO+32*RO
18Ø PRINT@SC,CHR$(2Ø7);
19Ø NEXT RO
2ØØ NEXT RP
21Ø NEXT I
22Ø GO TO 22Ø
```

Very nice. We'll return to the dice again in a moment.

5 So how long is a list? — DIM

We have ignored so far the space taken up by a list. BASIC assumes that any list you mention contains 11 values, with subscripts from 0 to 10, and this is quite often suitable as you are unlikely to miss the wasted space if your list is shorter. However, you can make a list longer or shorter than 11 by using the DIM statement. No, it's not an insult, it stands for DIMension. You simply put

line number DIM name(size), name(size), . . .

to tell BASIC the size of any number of lists. You can have any number of DIM statements as long as you define the size of a given list before you first use it — otherwise you're stuck with 11 and BASIC will object anyway. It is usual to put the DIM statements for a program at the beginning. If you mention a particular name more than once in your DIM statements, the computer won't like it.

EXAMPLES:

```
5 DIM PN(4),AG(4),SN(4)
```

reserves exactly four spaces for the lists that make up the Narrowdale prison database. You could put this in the previous example to save a bit of memory space.

```
69 DIM XR(2ØØ)
```

reserves 200 spaces for a biggish list called XR.

And now back to the dice. If we throw two dice at once, the sum of the spots can be any number from 2 (snakes eyes) to 12. To make the subscripts easy, we can use an array

```
DIM DI(12)
```

to make a histogram. This is a lot like the other histogram program. But please do run it because the shape of the histogram should surprise you.

```
1Ø REM SUM OF TWO DICE
2Ø DIM DI(12)
3Ø CLS
4Ø REM FIRST THROW THE DICE
5Ø PRINT "THROWING NOW"
6Ø FOR I=1 TO 36
7Ø D1=RND(6)
8Ø D2=RND(6)
9Ø TH=D1+D2
1ØØ DI(TH)=DI(TH)+1
11Ø NEXT I
12Ø FOR I=2 TO 12
13Ø PRINT DI(I);
14Ø NEXT I
15Ø REM MAKE THE HISTORYGRAM
16Ø FOR I=2 TO 12
17Ø FOR RP=-3 TO -2
18Ø CO=3*(I-1)+RP
19Ø FOR RO=15-DI(I) TO 15
2ØØ SC=CO+32*RO
21Ø PRINT@SC,CHR$(2Ø7);
22Ø NEXT RO
23Ø NEXT RP
24Ø NEXT I
25Ø PRINT@448
26Ø GO TO 26Ø
```

If you want an explanation, the chances of snakes eyes is 1 in 36, as is the chance of any particular combination. You can, however, get a 3 sum in two different ways: 1+2 or 2+1. The chance of this is therefore 1 in 18. And so on. The most likely sum is 7 which can be formed in 6 different ways and has a chance of 1 in 6 — it comes up about once in six throws. The shape of your histogram shows — roughly — these different probabilities. But as RND is random, you don't get exactly the expected number of each possible sum.

6 So how big is memory? — the MEM function

None of the programs in this book are particularly large, but in your later career as a BASIC expert you will one day write a program that runs out of space in the memory of the DRAGON 32 Computer. This will probably happen when you try to use a big array.

If I RUN this one-line program on my DRAGON, everything is OK:

```
1Ø DIM Z(4959)
```

But if I try

```
1Ø DIM Z(496Ø)
```

I am out of luck. I get a message

```
?OM ERROR IN 10
```

This is because I have run out of memory — OM means 'out of memory'. I can get more memory on the DRAGON by using the PCLEAR statement. You may recall from Chapter 15 that I can control how many pages of memory are used for the graphics screen. Normally 4 pages are set aside for graphics. If I didn't want a graphics screen at all, I could put

```
10 PCLEAR 1
20 DIM Z(5879)
```

and just get away with it. We can't put PCLEAR 0. At the other end of the scale

```
10 PCLEAR 8
20 DIM Z(3727)
```

is the most I can get if I need the maximum of 8 pages for graphics. As soon as I add additional statements to this program, the available space is decreased by the space taken up by the program. You can see this in the memory map in chapter 15.

At any time you can ask the DRAGON to tell you how much memory is available with the MEM function. As direct commands, put

```
PCLEAR 1
PRINT MEM
```

And you get the answer 29479 on my DRAGON, which means that 29479 Bytes of memory are free. Each value in a BASIC program — variable or member of array — uses 5 Bytes — so you can see that this approximates to the 5879 values I could put in the array in the earlier example. Similarly

```
PCLEAR 8
PRINT MEM
```

tells me I have 18727 Bytes. If I write a little program:

```
10 PCLEAR 8
20 DIM Z(3700)
30 PRINT MEM
```

it will tell me that 184 Bytes are left — not very many. Similarly

```
10 PCLEAR 1
20 DIM Z(2500)
30 PRINT MEM
```

tells me there are 436 Bytes free.

The MEM function will always tell us how many Bytes of memory are free. BASIC allocates spaces to arrays when it sees your DIM statement. Therefore if you try

```
10 PCLEAR 1
20 PRINT MEM
30 DIM Z(2500)
40 PRINT MEM
```

You will see that before the DIM statement BASIC thinks that there are 29439 Bytes free at line 20 but only 427 at line 40, after it has seen the DIM statement.

You can use MEM in the DIM statement. Here is a program that is asking for almost as much memory as it can get, using the knowledge that one value uses 5 Bytes of memory.

```
10 PCLEAR 1
20 PRINT MEM
30 DIM Z(MEM/5-100)
40 PRINT MEM
```

Wow! We still have 498 Bytes free for a bit of program which in a practical case we would want to add to this. This is all getting a bit advanced for a beginners' book, so I think we'll leave it there.

7 Westminster Chimes

Another musical program. The notes that make up the Westminster Chimes are defined in DATA statements and we read them into an array called WC. We then pick them out and play them one at a time. In Chapter 19 we're going to make the DRAGON into a chiming clock, called Huge Ben.

```
10 REM DING DONG
20 DATA 170,159,147,108
30 DATA 108,159,170,147
40 DATA 170,147,159,108
50 DATA 108,159,170,147
60 DIM WC(16)
70 REM READ DONGS
80 FOR I=1 TO 16
90 READ WC(I)
100 NEXT I
110 REM NOW DO DINGING
120 FOR PH=0 TO 3
130 FOR N=1 TO 4
140 SOUND WC(N+4*PH),12
150 NEXT N
160 FOR ST=1 TO 100
170 NEXT ST
180 NEXT PH
```

Eighteen

SORT IT OUT

1 Do the DRAGON shuffle

With all this music around, I suppose you think that I am referring to some kind of dance. Sorry. Once you have got data into lists you are probably going to want to move it around. You have to be careful how you do this.

Back in the jailhouse, suppose you decide you want to rearrange your prisoners, so that each moves down one cell number except for the prisoner in cell 1, who moves to cell 4. In a computer program, you can only move one value at a time. In the prison, this would be a bit like having only one guard to move the prisoners. Here is what you would have to do:

> The prisoner in cell 1 goes in temporary accommodation
> The former prisoner in cell 2 becomes the new prisoner in cell 1
> The former prisoner in cell 3 becomes the new prisoner in cell 2
> The former prisoner in cell 4 becomes the new prisoner in cell 3

and finally

> The former prisoner from cell 1 becomes the new prisoner in cell 4

If we have associated arrays, we have to shuffle them at the same time. Some prisoners would not be too happy if your prison computer caused them to inherit the previous inmate's sentence. Others, no doubt, would be delighted! This program does it properly; add it to the bit from the previous chapter which defines the lists and see what happens:

```
170 REM THE DRAGON SHUFFLE
180 REM MOVE CELL NO 1 OUT
190 P1=PN(1)
200 A1=AG(1)
210 S1=SN(1)
220 REM MOVE THE REST DOWN
230 FOR I=1 TO 3
240 PN(I)=PN(I+1)
250 AG(I)=AG(I+1)
260 SN(I)=SN(I+1)
270 NEXT I
280 REM BRING BACK OLD NO 1
290 PN(4)=P1
300 AG(4)=A1
310 SN(4)=S1
320 REM FINALLY SHOW RESULT
330 PRINT "NOW MOVED TO"
340 FOR J=1 TO 4
350 PRINT PN(J);SN(J);AG(J)
360 NEXT J
```

Notice how the prisoners have been moved down one place. This has been done between lines 190 and 230 in ascending order of cells. On the other hand, if you wanted to move them up, you should go backwards. Why?

```
To shuffle up
    save the top one
    move others in descending order

To shuffle down
    save the bottom one
    move others in ascending order
```

2 The great sorting problem

Arranging things in order is one of the great computer problems keeping computer geniuses occupied. It is very easy to do this, but it is hard to do it efficiently. I am going to show you two simple (but inefficient) methods.

Putting things in order is called sorting. The things to be sorted are called 'keys'. In Chapter 22 we'll put words in alphabetical order. Here we do it with numbers.

Bubble sorting is the simple one. We have a list LI of, say, 8 values, which are not in order, and we want LI(1) to be the smallest, LI(8) the largest. To do this we simply compare LI(1) with LI(2), and switch them if we need to. Now look at LI(2) and LI(3) and do the same, and continue along until we have dealt with LI(7) and LI(8). After we have done this, the numbers will be more ordered than they were, and we know that LI(8) has the largest value for sure — because it will have been carried along like the largest bubble rising more quickly. (Is this physically true? Falling objects go at the same speed in a vacuum. Do large bubbles rise faster?) Anyway, we now make another pass, until we know that LI(7) has the second largest number. And so on. Here it is; you can sort any number of keys from 2 to 10 using this program. For a larger number, put in a DIM statement.

```
10 REM BUBBLE POWER
20 INPUT"HOW MANY KEYS";N
30 PRINT"ENTER"N"KEYS ONE BY ONE"
40 FOR J=1 TO N
50 INPUT LI(J)
60 NEXT J
70 CLS
80 PRINT "HERE GOES"
90 FOR J=1 TO N
100 PRINT LI(J);
110 NEXT J
120 PRINT
130 FOR I=N-1 TO 1 STEP -1
140 FOR K=1 TO 500:NEXT K
150 FOR J=1 TO I
160 IF LI(J)<=LI(J+1) THEN 270
170 REM SWITCH THESE TWO
180 LT=LI(J)
190 LI(J)=LI(J+1)
200 LI(J+1)=LT
210 CLS
220 PRINT
230 FOR K=1 TO N
240 PRINT LI(K);
250 NEXT K
260 FOR K=1 TO 500:NEXT K
270 NEXT J
280 NEXT I
```

Do you see how the values are switched? Again a temporary store has been used. If you run it, you will see it all happen on the screen.

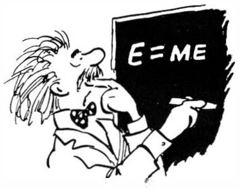
This next one is more efficient (slightly). We take each value LI(2), . . . LI(N) in turn and insert it in the right place. This works because when we get to a particular place in the list, all the ones before it are in order because of the sorting we did before. To do this we need our searching expertise from Chapter 17 and our shuffle knowhow from this chapter:

```
10 REM INSERTION SORT
20 INPUT"HOW MANY KEYS";N
30 PRINT"ENTER"N"KEYS ONE BY ONE"
40 FOR J=1 TO N
50 INPUT LI(J)
60 NEXT J
70 CLS
80 PRINT "HERE GOES"
90 FOR J=1 TO N
100 PRINT LI(J);
110 NEXT J
120 PRINT
130 REM INSERTION SORT
140 FOR I=2 TO N
150 REM PUT LI(I) IN RIGHT PLACE
160 FOR J=1 TO I-1
170 IF LI(I)>=LI(J) THEN 250
180 REM HERE COMES THE SHUFFLE
190 LT=LI(I)
200 FOR K=I TO J+1 STEP -1
210 LI(K)=LI(K-1)
220 NEXT K
230 LI(J)=LT
240 GO TO 260
250 NEXT J
260 NEXT I
270 PRINT"AND HERE IT IS"
280 FOR K=1 TO N
290 PRINT LI(K);
300 NEXT K
```

You may find this program a bit difficult; it certainly tests one's ability to think about subscripts. But that is why it is here. Work at it! And while you're at it, don't forget to try it. Put in lots of PRINT statements to help you follow it through. If you tried the bubble sort, then see if you can get a similar display of the sorting as it happens.

Nineteen

ANYONE FOR EINSTEIN?



1 You wouldn't want me to leave something out!

Perhaps this chapter caters for a minority interest, so skip it if you want to. But no mathematician can be without the mathematical functions of BASIC, and I wouldn't be earning my pitiful royalties if I didn't tell you how to use them. As I promised earlier, I'm going to show you how to plot graphs of them. Let's jump straight in.

First the graphs. In Chapter 11 I did this:

```
10 FOR I=1 TO 14
20 FOR J=1 TO I
30 PRINT "*";
40 NEXT J
50 PRINT
60 NEXT I
```

and this:

```
10 FOR I=0 TO 13
20 PRINT TAB(I);"*"
30 NEXT I
```

These are both graphs. Look at them sideways. The first one is a solid or bar graph and the second is an ordinary graph, both of a straight line. Here is a bar graph showing some random numbers:

```
10 REM MAKE A GRAPH - RANDOM NOS
20 FOR I=1 TO 14
30 X=RND(30)
40 FOR J=1 TO X
50 PRINT"*";
60 NEXT J
70 PRINT
80 NEXT I
```

You should remember how these work. The semicolon causes printing to continue on the same line and so we can print X consecutive stars. X is an output from the random number generator in the range 1-30. Now let's look at some functions.

2 EXP and LOG

These two mathematical functions make a pair and are based on e, or 2.71828182845, the magic number which is the base of natural logarithms. The EXP function provides e raised to a power, and LOG finds the logarithm to the base e. If you're good at that sort of thing, you will realise that

if Y = EXP(X) then X = LOG(Y)

These programs make graphs of these functions:

```
10 REM PLOT EXP(X)
20 FOR X=0 TO 3.25 STEP 0.25
30 PRINT TAB(EXP(X));"*"
40 NEXT X
```

```
10 REM PLOT LOG(Y)
20 FOR Y=1 TO EXP(3) STEP EXP(3)/14
30 PRINT TAB(10*LOG(Y));"*"
40 NEXT Y
```

The log function can easily be used to base 10 or any other base. You may know that

$$\log_a X = \log_e X \log_a e = \frac{\log_e X}{\log_e a}$$

so that you could find the log of X to base 10 by a statement like

```
110 L=LOG(X)/LOG(10)
```

or to base A by

```
110 L=LOG(X)/LOG(A)
```

3 SIN, COS, and TAN

These are the trigonometric functions of an angle, and are often used. In BASIC the angle is in radians and this may not always be convenient, although conversion is easy since π radians is the same as 180 . You can get π by a magic statement

```
          30 PI=4*ATN(1)
```

which is explained in the next section. Here is a program which will plot either SIN or COS for you:

```
          10 REM PLOT A TRIG THING
          20 CLS
          30 PI=4*ATN(1)
          40 FOR X=-PI TO PI STEP PI/7
          50 REM COULD BE EITHER COS OR SIN
          60 PRINT TAB(15.5+15*SIN(X));"*"
          70 NEXT X
```

TAN is not as pretty. Use

```
          60 PRINT TAB(15.5+2*TAN(X));"*"
```

If you ever want to use TAN in a real program, remember that it doesn't like to evaluate tan(90) — which is tan(π/2) — or any similar angle. Do you know why?

4 ATN

The ATN or 'arc tangent' function is one that a lot of people have trouble understanding. You tell this function the value of a tangent, and the result is the angle that goes with it. For example, the tangent of 45° is 1. Therefore the statement

```
          50 P4=ATN(1)
```

gives P4 the value of 45°; unfortunately in radians. However, 180° is π radians, and so 45° is π/4 radians.

You can use this fact to get π. If ATN(1) is π/4 radians, then you can find π using

```
          30 PI=4*ATN(1)
```

you can convert any angle given in degrees, called D, to one in radians, called **R**, by

```
          660 R=D*PI/180
```

Similarly you could calculate **D** in degrees from **R** in radians by

```
          700 D=R*180/PI
```

Sometimes all of this can be very useful.

The ATN function can only give results between −π/2 and π/2. This is because the TAN function, if you look at it, repeats over and over again and the computer does not know which repetition you want when you use ATN. So it always gives the result nearest zero.

Twenty

INVENT SOME FUNCTIONS

1 How to do it

We have been introduced to most of the built-in functions of BASIC already, although in Chapter 22 we will see that there are some more to come which deal with characters. Although the DRAGON provides functions to cover most important things we might wish to do, it is often useful to make up our own functions for special requirements. In BASIC you can define a function as long as you can pack it all into one line. You do this with the DEF FN statement:

line number DEF FNxx(variable) = expression

The name of your function is FNxx, where xx is a combination of letters and numbers very similar to the combinations allowed for variable names: one letter, two letters, or a letter plus a number. When you use it, you give it a value to work on, just like you do with most of the other functions. Normally you would use this in working out the result, although you don't need to. When the function is used, the expression on the right hand side is worked out, substituting the value you gave for the variable in the DEF FN statement. This variable is called the 'argument' of the function. I'm sorry if this sounds a bit complicated. Some examples will help to sort you out.

2 The apple and Mr. Newton

In Chapter 14, and again in Chapter 15, we used a ball falling under gravity and bouncing to produce animation. I pretended that the graphics screen was 313 metres high, and used a formula in which the position of the ball was

Row Number $= 3t^2 - 1$

where t was the number of seconds since it was dropped. I ran the formula backwards after the ball hit the bottom of the screen at 8 seconds.

I could use a function to calculate the row number from the time. For defining the function I call the time A:

```
          20 DEF FNRO(A)=3*A*A-1
```

and another one for the column numbers as the ball moves steadily across the screen

```
          30 DEF FNCO(A)=16*A-9
```

Here is the program:

```
10 REM AN APPLE A DAY
20 DEF FNRO(A)=3*A*A-1
30 DEF FNCO(A)=16*A-9
40 PMODE 4,1
50 SCREEN 1,1
60 PCLS
70 FOR T=1 TO 8
80 RO=FNRO(T)
90 CO=FNCO(T)
100 CIRCLE(CO,RO),12,1
110 NEXT T
120 FOR T=9 TO 16
130 RO=FNRO(17-T)
140 CO=FNCO(T)
150 CIRCLE(CO,RO),12,1
160 NEXT T
170 GO TO 170
```

You will notice that the DEF FN statements have been placed at the beginning of the program. Actually they could go anywhere before they are used. At line 80, the use of the function should be obvious; the value of T is used for A in the evaluation of 3*A*A−1. At line 130, it is a bit more subtle. To make it bounce back, 17−T is the argument of FNRO, so that 17−T gets substituted for A when the function is used.

'I DIDN'T KNOW ABOUT GRAVITY, BUT I COULD'VE TOLD HIM ABOUT EGGS!

3 Circles and Choppers

We can find the area of a circle of radius R. The area is R^2. This is the function:

```
          2Ø DEF FNCR(R)=4*ATN(1)*R*R
```

which uses the ATN function to get π. This smart idea was explained in Chapter 19. Here is a little program to print the area of a number of circles:

```
          1Ø REM CIRCLE AREAS
          2Ø DEF FNCR(R)=4*ATN(1)*R*R
          3Ø REM NOW DO IT
          4Ø FOR X=1 TO 2 STEP Ø.1
          5Ø PRINT "RADIUS" X
          6Ø PRINT "AREA   " FNCR(X)
          7Ø NEXT X
```

You can see how the actual value X is substituted for R when the program uses the function FNCR.

In Chapter 7, I talked about rounding and truncation or 'chopping' of numbers to get interger results. Recall that the INT function of BASIC takes the next lowest integer. Here is a function to chop or truncate to an integer value:

```
          6Ø DEF FNT(X)=SGN(X)*INT(ABS(X))
```

For comparison:

FNT(1.5) would be 1.0	INT(1.5) would be 1.0
FNT(-2.3) would be -2	INT(-2.3) would be -3

If you want to round instead to the nearest whole number, use this:

```
          7Ø DEF FNRN(X)=INT(X+Ø.5)
```

4 Still gambling?

As yet another example, we can write a function to give us the sum of the spots when two dice are thrown. Here it is:

```
          15 DEF FNTH(X)=RND(6)+RND(6)
```

Notice that the function argument X is not used on the right handside. It doesn't have to be. Here we find it used in our histogram program from Chapter 17.

```
10 REM SUM OF TWO DICE
15 DEF FNTH(X)=RND(6)+RND(6)
20 DIM DI(12)
30 CLS
40 REM FIRST THROW THE DICE
50 PRINT "THROWING NOW"
60 FOR I=1 TO 36
90 TH=FNTH(X)
100 DI(TH)=DI(TH)+1
110 NEXT I
120 FOR I=2 TO 12
130 PRINT DI(I);
140 NEXT I
150 REM MAKE THE HISTORYGRAM
160 FOR I=2 TO 12
170 FOR RP=-3 TO -2
180 CO=3*(I-1)+RP
190 FOR RO=15-DI(I) TO 15
200 SC=CO+32*RO
210 PRINT@SC,CHR$(207);
220 NEXT RO
230 NEXT RP
240 NEXT I
250 PRINT@448
260 GO TO 260
```

5 Something about bits

Normally we work and think in what are called decimal numbers. I'm sure you know that the number 343 is

3 hundreds
4 tens
3 ones

3	4	3
hundreds	tens	ones

We probably use the decimal system of numbers because we have ten fingers. Computers work in a different system, called binary. A computer can actually only store values which are on and off, like little switches. On and off have values 1 and 0 (or TRUE and FALSE). To make bigger numbers, computers have to count up numbers out of 1's and 0's. Here is a binary number:

100101

It means:
1 thirty two
0 sixteens
0 eights
1 four
0 twos
1 one

1	0	0	1	0	1
thirty-twos	sixteens	eights	fours	twos	ones

Each digit is called a 'bit' and the 'ones' bit is called the 'least significant bit'.

Any number which is odd has the ones bit turned on. Any number which is even has it off. We can use this information to make the computer give us the least significant bit on the screen. But we can do more! We can make it print the whole binary number.

If we divide a number by two and take the integer part, we shift its binary bits one position to the right. To demonstrate this, notice that the example above is the same as 37 decimal. Therefore

37 decimal is 100101 binary

INT(37/2) is 18 which is 10010 binary
INT(18/2) is 9 which is 1001 binary

and so on. So we can do the following:

(i) Get the least significant bit and print it
(ii) Shift the bits right
(iii) Go to (i) until the number becomes zero

Here is a function to get the least significant bit of a number I:

```
20 DEF FNLS(I)=I-INT(I/2)*2
```

You can see that this is just the old remainder again. If you take the remainder after dividing an integer by two, you have the least significant bit. We can shift the bits right by having

I = INT(I/2)

Here is the program. I have a very practical use for this in Chapter 24 — right at the end of the book.

```
10 REM CONVERT TO BINARY
20 DEF FNLS(I)=I-INT(I/2)*2
30 PRINT
40 PRINT"***THIS PROGRAM CHANGES LIVES***"
50 PRINT
60 PRINT"YOU TYPE IN A NUMBER AND"
70 PRINT"I CONVERT IT TO BINARY"
80 PRINT"THE ONLY PROBLEM IS .."
90 PRINT"YOU HAVE TO READ IT BACKWARDS"
100 INPUT"GIVE ME A DECIMAL NUMBER";D
110 PRINT "IN BINARY THAT IS"
120 REM GET THE LEAST SIGNIFICANT BIT
130 PRINT FNLS(D);
140 REM AND THEN SHIFT D RIGHT
150 D=INT(D/2)
```

```
            160 IF D>0 THEN 130
            170 PRINT
            180 GO TO 100
```

EXERCISE:

If you put the binary digits one by one into an array, then you can PRINT them in the correct order. Do this.

6 Involving other variables

The functions used as examples so far used only the value of their argument. You may want to use other variable names. If this is so, there is no way of having substitutions made for the extra variables — this is a slight limitation of BASIC. You can only have one dummy argument. Apart from the one name you use as a function argument, any other variables you mention in the function definition refer to their actual values.

For example, consider another kind of rounding as first discussed in Chapter 7. The function FNRS rounds its argument after using a scale factor S, so that the result of the function is rounded to the nearest S:

```
            20 DEF FNRS(X)=INT(X/S+0.5)*S
```

Here, the value that X is given when the function is used is substituted in the right hand side when it is evaluated, but the value used for S is the actual value of the variable S. Here is a program to try this out:

```
            10 REM A ROUNDING THING
            20 DEF FNRS(X)=INT(X/S+0.5)*S
            30 S=0.1
            40 PRINT"ENTER A NUMBER FOR"
            50 PRINT"ROUNDING TO NEAREST 0.1"
            70 INPUT N
            80 PRINT"ROUNDED ="FNRS(N)
            90 GO TO 40
```

Twenty One

SUBROUTINES

1 Pass the buck

A subroutine is a separate little program within a program. This can be very helpful. If you have an easily isolated little task inside a program that you wish to use over and over again to do the same thing but perhaps under slightly different conditions, you can make it a subroutine. This makes it easier to use in different programs (if you have a cassette tape or disk store), and if you have a printer you can trade subroutines in plain brown wrappers with other DRAGON owners who are similarly inclined.

All you do is write the bit of program out separately, with its own line numbers. It is usual to start a subroutine at a large line number, like 1000 or 2000. You end your subroutine with the RETURN statement

line number RETURN

When you want to use your subroutine, you write

line number GOSUB subroutine's line number

The program jumps to your subroutine, does what the subroutine says, and when it hits the RETURN statement it jumps back to the line after GOSUB.

2 Ring dem bells

Now we are going to make a subroutine to ring the bells. First of all, here is a subroutine simply to go 'DONG' with the note BL:

```
3000 REM DONG THE NOTE BL
3010 SOUND  BL,12
3020 FOR ST=1 TO 100
3030 NEXT ST
3040 RETURN
```

Alright, that subroutine doesn't do much. But it would be useful to be able to chime out any tune we want. Here is a subroutine which calls this subroutine — why not? We put any tune in the array WC of length NC notes. The main program will worry about that, and this subroutine will play it.

```
2000 REM PRIVATE CARILLON
2010 REM PLAYS NC NOTES FROM
2020 REM THE ARRAY WC
2030 FOR LB=1 TO NC
2040 BL=WC(LB)
2050 GOSUB 3000
2060 NEXT LB
2070 RETURN
```

Here's a program to play the good old Westminster chimes. We define the number of notes in a DATA statement and the tune follows in four more DATA statements.

```
10 REM BIG BEN STRIKES
20 DATA 16
30 DATA 170,159,147,108
40 DATA 108,159,170,147
50 DATA 170,147,159,108
60 DATA 108,159,170,147
70 REM READ THE NUMBER OF DONGS
80 READ NC
90 DIM WC(NC)
100 REM READ THE ACTUAL DONGS
110 FOR I=1 TO NC
120 READ WC(I)
130 NEXT I
140 REM NOW DO DINGING
150 GOSUB 2000
160 END
```

You will notice at line 160 there is an END statement. We have been leaving END statements out at the END of programs up until now. Here we need one to end the 'main program'. If it wasn't there, when you run the program it would go crashing on into the subroutine instead of finishing. What would happen? Try it and see.

You can terminate a program with the statement line number END

You always need one of these where subroutines are involved. Well, nearly always. In Chapter 23, we will use these ideas to make a fantastic chiming clock!

3 The royal fireworks

We used a spiral for an explosion a while back when the bee flew into the neutron bomb. So lets make it into a subroutine. At the same time, we will make it more general. The subroutine will centre the spiral at screen co-ordinates XE and YE. The size of the spiral will be SZ, and the colour will be CL.

We have to be careful to keep the spiral on the screen. If we are adjusting X, the column number of a blob, then we always have to check that it has not left the screen. You can see this in a statement like

```
2080 IF X=63 THEN 2310
```

which prevents us from pushing the spiral off the screen to the right. There are similar statements for each of the 4 arms of the spiral. Here is the subroutine.

```
2000 REM SET BLOB IN MIDDLE
2010 X=XE
2020 Y=YE
2030 SET (X,Y,CL)
2040 REM NOW SPIRAL
2050 FOR RO=1 TO SZ STEP 2
2060 REM GO RIGHT, YOUNG PERSON
2070 FOR I=1 TO RO
2080 IF X=63 THEN 2310
2090 X=X+1
2100 SET (X,Y,CL)
2110 NEXT I
2120 REM UP, UP AND AWAY
2130 FOR I=1 TO RO
2140 IF Y=Ø THEN 2310
2150 Y=Y-1
2160 SET (X,Y,CL)
2170 NEXT I
2180 REM NOW TO THE LEFT
2190 FOR I=1 TO RO+1
2200 IF X=Ø THEN 2310
2210 X=X-1
2220 SET (X,Y,CL)
2230 NEXT I
2240 REM AND FINALLY DOWN
2250 FOR I=1 TO RO+1
2260 IF Y=31 THEN 2310
2270 Y=Y+1
2280 SET (X,Y,CL)
2290 NEXT I
2300 NEXT RO
2310 RETURN
```

In the main program, we call up a fireworks display by playing our spirals at random on a black screen, in random colours and in random groups of upto 10 blasts with a short break between. Very artistic, although it may not look very much like fireworks.

```
1Ø REM FOURTH OF JULY
2Ø REM OR IS IT 5 NOV?
3Ø REM UP TO 1Ø BLASTS
4Ø CLS Ø
5Ø NB=RND(1Ø)
6Ø FOR B=1 TO NB
7Ø REM RANDOM COLOR AND POSITION
8Ø CL=RND(8)
9Ø XE=RND(64)-1
1ØØ YE=RND(32)-1
11Ø SZ=2*RND(4)+1
12Ø GOSUB 2ØØØ
13Ø NEXT B
14Ø REM SLIGHT PAUSE
15Ø FOR I=1 TO 5ØØ:NEXT I
16Ø GO TO 4Ø
```

Twenty Two

QUITE IN CHARACTER

1 String along with me

As we enter the final stages of our roundup of BASIC on the DRAGON Computer we find out here how letters and numbers and other symbols can be handled just as if they were variables. We have already met character string constants, without knowing it. Any series of symbols that you put into quotation marks is a character string constant:

```
10 PRINT"THIS IS A STRING CONSTANT"
20 PRINT"SO IS THIS"
```

Character string variables are given ordinary variable names with the symbol $ added on:

```
A$
N9$          are all names of string variables
CH$
```

A string variable can be subscripted, as for example

```
10 DIM TT$(60)
```

defines a string list or array with 60 entries, and

```
20 DIM X$(20,20)
```

defines a table of string variables. Tables or arrays with two or more subscripts are introduced in Chapter 24.

The maximum number of characters in any kind of string is 255.

2 What can you do with a string?

Many of the statements of BASIC can manipulate character strings. We have seen string constants in PRINT statements. Here are the other things you can do.

(a) Assignment
The assignment statement may contain a string variable on the left hand side and a string variable or constant on the right hand side. You cannot assign a character value to an ordinary integer variable, nor can you assign a numeric value to a string variable. Sorry about that. There are, however, some special functions to use with strings which are described a bit later. This is what you can do:

line number string variable = string variable or string constant

Of course this includes string variables that have subscripts.

(b) Concatenation — putting strings together
Con what? Another new word! When you push two strings together, it is called concatenation. You do it with a + sign, but it isn't quite like addition. Try this:

PRINT "WATER"+"GATE"

Aha! You could also have

```
1Ø X$="CASA"
2Ø Y$="BLANKA"
3Ø Z$=X$+Y$
4Ø PRINT Z$
```

This can be useful! As long as the result is less than 255 characters long, you can concatenate freely.

(c) PRINT or PRINT @
In a PRINT statement you can use string constants or variables, or the special string functions which we will see a bit later.

```
1Ø TH$="CENSORED"
2Ø PRINT "THE THOUGHT FOR"
3Ø PRINT "TODAY IS " TH$
```

(d) INPUT
An INPUT statement can ask for character strings. To respond, you simply type the message you want. You can put it in quotation marks if you want. You need to put it in quotes only if the message itself contains a comma or a colon. A bit later you will find out about the INKEY$ function for reading one symbol from the keyboard.

(e) IF. . . THEN

Two character strings can be compared, as in

```
line number IF string compared string THEN line number
                         to
```

as in

```
100 IF BS$(I)="STOP" THEN 66
```

This compares the Ith entry in the string array called BX$ with 'STOP' and jumps to line 66 if they are the same.

When you compare strings, you have to know what order the symbols take. The alphabet is always taken in order so that

```
'ABC' is greater than 'ABB'
                   or 'AB'
'ABC' is equal to 'ABC'
'ABC' is less than 'ABD'
                or 'ABCA'
                or 'ABC '   (note the blank)
```

You can prove this to yourself using this little program:

```
10 INPUT A$
20 IF "ABC">A$ THEN PRINT"GREATER"
30 IF "ABC"=A$ THEN PRINT"EQUAL"
40 IF "ABC"<A$ THEN PRINT"LESS"
50 GO TO 10
```

Because of this, you can sort lists into alphabetical order. If you want to know the order of the other symbols, look in the Appendix.

(f) DIM

As already mentioned, you can use string variables with subscripts, and therefore you can give them a size in the DIM statement. If you don't do this, then the subscripts are assumed to vary from 0 to 10.

(g) DATA and READ

You can put character string constants in your data list:

```
10 DATA ONE,TWO,THREE
20 FOR I=1 TO 3
30 READ ST$
40 NEXT I
50 PRINT ST$
```

This prints THREE. Be careful not to try to read a string constant into a numeric variable or vice versa. You won't like the results! If a string in a DATA statement contains either a comma or a colon, or has leading blanks, you have to put that string in quotation marks.

(h) MID$

MID$ is a statement for replacing part of a string. If you put

line number MID$ (oldstring, position, length) = newbit

then the first 'length' characters of 'newbit' replace part of 'oldstring' starting at character number 'position'. Complicated? Try this:

```
10 X$="OLDBIT"
20 MID$(X$,4,3)="NEWABC"
30 PRINT X$
```

You should get

```
OLDNEW
```

You can see that the value of 'length' was 3, which is less than the length of 'NEWABC', and so only the first three characters are used. 'oldstring' and 'newbit' have to be character strings — 'oldstring' would normally be a character string variable or a string constant. 'Position' and 'length' are ordinary values. If 'length' is greater than the length of 'newbit', then all of 'newbit' is used.

EXAMPLE:

```
10 X$="BIGLONGTHINGIE"
20 MID$(X$,4,20)="TINY"
30 PRINT X$
```

gives

```
BIGTINYTHINGIE
```

The result is always the same length as 'oldstring', so it may use less of 'newbit' than length asks for:

EXAMPLE:

```
10 X$="SILLY"
20 MID$(X$,2,6)="MARMITE"
30 PRINT X$
```

gives

```
            SMARM
```

You can leave out 'length'. If you do, all of 'newbit' is used, unless it is too long as above.

EXAMPLE:

```
        10 X$="STORM COMING"
        20 MID$(X$,7)="PENDING"
        30 PRINT X$
```

gives

```
            STORM PENDIN
```

Please note that there is also a character string function called MID$ that does not do the same as this MID$ statement.

(i) LINE INPUT
This is a variation of the INPUT statement that is occasionally useful because you can control the kind of prompt that you get. If you use

line number LINE INPUT prompt; string variable

your 'prompt' is printed without the usual ? and you can then enter one line of characters including commas and quotation marks if you want. This is the only way you can ever enter quotation marks. Your 'prompt' is a constant in quotation marks. Your line of input defines the 'string variable' — there can be only one.

EXAMPLE:

```
        40 LINE INPUT "TELL ME!";SC$
```

(j) PRINT USING
This is a variation of the PRINT statement which gives you total control over the way information is arranged on the line of output. It involves two lines — the PRINT USING statement and a separate line which gives an image of the layout you want. You put

line number PRINT USING image; things to print

The 'image' is a picture of your output line which will be given by some string expression — usually a constant or a variable.

The '#' specifies a digit. The '.' specifies the position of the decimal point. Here are the things that can go in an image:

#	—	specifies a digit
.	—	locates the decimal point
blank	—	separates values
%blanks%	—	specifies a character string longer than one character
!	—	specifies only one character to be used
messages	—	you can put messages in the image as long as they don't contain # . % ! + − or $.

EXAMPLES:

```
50 PRINT USING"## ## ##.#";1,2,3
```

prints 1 2 3.0

```
50 PRINT USING"! %   %";"DON","MONRO"
```

prints D MONRO

You can put little codes with the # specifiers to control how numbers are printed:

, at the end of the image puts commas after every three digits:

```
50 PRINT USING"#########,";1E6
```

prints a million with commas:
1,000,000

$ start the field with a $ sign

```
50 PRINT USING"$####";1
```

prints $ 1

$$ puts the $ sign before the first significant digit:

```
50 PRINT USING"$$###";1
```

prints $1

**$ fills the space before the $ sign with stars — you might do this in printing a cheque

```
50 PRINT USING"**$##.##";1
```

prints ***$1.00

** for ordinary numbers, fills up to the left with stars

```
50 PRINT USING"**#.##";3
```

prints **3.00

+ tells the computer to print a sign in front of every number

```
50 PRINT USING"+## +##";2,-2
```

prints +2−2

This can be combined with **

```
50 PRINT USING"+**#.#";3
```

prints **+3.0

and you can force the machine to use exponential or scientific notation by placing exactly 4 up-arrows at the end of a field:

```
50 PRINT USING"+##.##↑↑↑↑";1E6
```

prints +10.00E+05

(k) PRINT @ . . . USING
Another variation on PRINT. This is just like PRINT @ which was introduced in Chapter 12. You can use the @ to get you anywhere you want on the screen, and the USING to get exactly the image you want:

line number PRINT @ screen address, USING image; things to print

there is an example of this in the clock program of the next chapter.

(l) PLAY
A super sound generator which is covered in the next chapter.

(m) DRAW
A super duper graphics generator which is covered in the next chapter.

3 Sorting into alphabetical order

Here is a program to take a list of character strings and sort them into alphabetical order using an insertion sort. Each list item IN$(I) has another list item associated with it, PN(I) which is a number. I used a similar idea to make the Index for this book. In a character array I put the items I wanted indexed, and in an ordinary array the page numbers. Then I sorted the character array into alphabetical order, dragging the page number alongside. Some people think that it is very difficult to index a book. This one took me two hours. Only a computer can do this:

```
10 REM MAKE AN INDEX
20 REM DM IS THE MAXIMUM SIZE
30 DM=50
40 DIM IN$(DM),PN(DM)
50 PRINT"OK HEMMINGWAY, LET'S GO"
60 PRINT"ENTER ITEM,PAGE NUMBER"
70 PRINT"AFTER EACH PROMPT. KEEP"
80 PRINT"IT UP UNTIL FINISHED"
90 PRINT"THEN ENTER QUIT,Ø. YOU"
100 PRINT"CAN'T HAVE AN ITEM NAMED"
110 PRINT"'QUIT' ON PAGE Ø."
120 REM INPUT UP TO 100 ITEMS
130 IN=Ø
140 FOR I=1 TO DM
150 INPUT XN$,IP
160 IF XN$="QUIT" AND IP=Ø THEN 220
170 IN$(I)=XN$
180 PN(I)=IP
190 IN=IN+1
200 NEXT I
210 PRINT"NO MORE ROOM LEFT"
220 PRINT"SORTING NOW"
230 GOSUB 2000
240 PRINT"HERE'S YOUR INDEX"
250 PRINT"15 LINES AT A TIME"
260 PRINT"PRESS ENTER TO GO ON"
270 J=1
280 INPUT NL$
290 CLS
300 FOR I=1 TO 15
310 IF J>IN THEN END
320 PRINT IN$(J);PN(J)
330 J=J+1
340 NEXT I
350 GO TO 280
```

Here's the subroutine:

```
2000 REM INSERTION SORT STRINGS
2010 FOR IS=2 TO IN
2020 REM FIND PLACE FOR IN$(N)
2030 FOR JS=1 TO IS-1
2040 IF IN$(JS)>IN$(IS) THEN 2080
2050 NEXT JS
2060 REM IT GOES AT JS
2070 REM SHOVE OTHERS ALONG
2080 CS$=IN$(IS)
2090 PS=PN(IS)
2100 FOR KS=IS TO JS+1 STEP -1
2110 IN$(KS)=IN$(KS-1)
2120 PN(KS)=PN(KS-1)
2130 NEXT KS
2140 REM AND NOW POP IT IN
2150 IN$(JS)=CS$
2160 PN(JS)=PS
2170 NEXT IS
2180 RETURN
```

4 Those useful character functions

Here is a brief list of the functions that the DRAGON provides for you to use with character strings:

ASC(X$) gives you the ASCII code of the first character in X$. This is a number.

EXAMPLE: ASC("A") is 65

CHR$(X) changes a value X which is in the ASCII code into a string character.

EXAMPLE: CHR$(65) is "A"

The ASCII code is the number used inside most computers to represent a character. There is a list of ASCII codes in the Appendix.

LEFT$(X$,X)	—	picks off the leftmost X characters of X$
MID$(X$,S,X)	—	picks off X characters from the middle of X$, starting at charater number S. This does not do the samething as the MID$ statement.
RIGHT$(X$,X)	—	Surprise! This picks off the rightmost X characters in X$
LEN(X$)	—	gives the number of characters in X$. EXAMPLE: LEN("FIVE") is 4.
INSTR(X,X$,Y$)	—	searches X$ to see if it contains the shorter string Y$ inside it, starting the search from character number X. The value given is the character number at which the match is found, or 0 if it isn't.

For input there is a very special and useful function, INKEY$, which will be used in the example of the next section.

INKEY$	—	glances at the keyboard and gives the character presently being pressed on the keyboard. If no key is being pressed the value is "", i.e. nothing. When you press a key you only get its character once from INKEY$; holding it down doesn't cause it to repeat.

There are two functions which can convert a character string into a value and vice versa. Note that the character "1" is not the numeric value 1, ie

"1" is not equal to 1.

However

VAL("1") is equal to 1 and STR$(1) is equal to "1"

VAL(X$)	—	converts X$ into a number. X$ should be a string which makes up a number, eg '12.34'. If the DRAGON runs into a symbol that doesn't belong, it quits. This would give you a 0 result if the first character was wrong.

STR$(X) — converts a numeric value X into a character string that you could print.

There are two more functions for strings:

STRING$(length, code or string)
— gives a string of the same character repeated length times, whose ASCII code is given, or else it uses the first character of a given string
STRING$(4,) is " "
STRING$(7,"WOW") is "WWWWWWW"

HEX$(X) — gives a string of characters telling you the value of X in the hexadecimal number system.

5 Play it again, Sam

Just for the record, that isn't quite what Bogart said. Using the INKEY$ function, you can get just one character from the keyboard. Particularly when you are devising video games, you need to know what key your player is hitting (sorry, I mean gently pressing) at the moment. A statement like

```
        80 X$=INKEY$
```

This puts the key that is being pressed into the string variable. If there's nothing there, you get a null (not a blank). A null is like "". You can get any symbol, so your player doesn't hit ENTER every time he wants to dodge the Martian invaders. To show you how this works, we will use the keys 1 to 8 to play a tune on our bells.

In a data list, I put the note values of a G major scale, and then read them into an array. I then start looking at the keyboard. When I press the '1', what I get is the symbol for 1, not its value. To get its value, I borrow the VAL$ function from the next section, and pick out the note you are asking for. I use the DONG subroutine from Chapter 21:

```
10 REM DON'T FORGET THE CANDELABRA
20 DATA 147,159,170,176
30 DATA 185,193,200,204
40 FOR I=1 TO 8
50 READ NO(I)
60 NEXT I
70 KY$=INKEY$
80 IF KY$="" THEN 70
90 BL=NO(VAL(KY$))
100 GOSUB 3000
110 GO TO 70
```

Do you see why line number 90 is necessary? If you have not pressed a key, this makes the program wait until you do.

Twenty Three

HIGHLY STRUNG

1 Introducing BIG DRAGON — or is it DRAG BEN?

I promised you a chiming clock. Actually we have most of the bits and pieces already. In Chapter 21 I gave subroutines to play a tune on the bells. All we need now is to have a clock.

I have said very little so far about the clock ticking away inside the DRAGON 32 Computer. This is because we can't use it really well without knowing about character strings and we had more important things to learn about first.

There is a function called TIMER that reads the clock. You will see it ticking if you do this:

```
10 PRINT TIMER
20 GO TO 10
```

There it is, it ticking away merrily. What you are seeing is a number in the memory of the DRAGON which has one added to it 50 times a second if your electricity supply is 50 Hz as in Europe and many other places, or 60 times where it is 60Hz such as in the USA. You can set the timer by putting

line number TIMER = value

and it will start counting from 'value'. It counts all the way to 65535 and then starts again from 0. This takes about 15 minutes.

Using TIMER, we can do better than that. Assuming that the electricity supply is 50 Hz, them I am going to make the timer count to 3000, and then reset it, every minute. But while I am doing this I am going to count the minutes and hours so that the clock keeps time forever. We can always get the seconds by dividing the timer by 50. Here's a clock program. If your electricity supply is 60 Hz, change the value of Hz to 60. You will see here a nice use of the PRINT@. . . , USING statement.

```
10 REM A 24 HOUR CLOCK
20 F$="##:##:##.#"
30 HZ=50
40 PRINT"GIVE HOURS,MINUTES"
50 PRINT"PRESS ENTER TO START"
60 INPUT H,M
70 TIMER=0
80 CLS 0
90 S=TIMER/HZ
100 IF TIMER<HZ*60 THEN 170
110 TIMER =0
120 S=0
130 M=M+1
140 IF M<60 THEN 170
150 M=0
160 H=H+1
170 PRINT @0,USING F$;H,M,S;
180 IF H>23 THEN H=0
190 GO TO 90
```

Hey, that's really good, isn't it! You can use it like a stopwatch by entering 0,0 and pressing ENTER at the starting time. To stop it, press BREAK. Sorry, no split times although you could program them using INKEY$.

Now for the really cool bit. I insert in this program

```
175 GOSUB 1000
```

At 1000, I am going to put a subroutine which checks the time and chimes the Westminster Chimes, using the DONG subroutine from Chapter 21.

To do this I have to detect the quarter hours. These happen when S is zero and M is a multiple of 15. I use the good old remainder trick again — if the remainder of M divided by 15 is zero, it is time to chime. How much to chime? If the integer part of M/15 is 1, then four dongs, two is eight, three is twelve, and zero is the full treatment. Here it is:

```
1ØØØ REM THIS IS BIG BEN
1Ø1Ø REM IS S ZERO?
1Ø2Ø IF S<>Ø THEN RETURN
1Ø3Ø REM M MULTIPLE OF 15?
1Ø4Ø IF M-INT(M/15)*15<>Ø THEN RETURN
1Ø5Ø REM HOORAY, WE'RE GOING TO CHIME
1Ø6Ø DATA 17Ø,159,147,1Ø8
1Ø7Ø DATA 1Ø8,159,17Ø,147
1Ø8Ø DATA 17Ø,147,159,1Ø8
1Ø9Ø DATA 1Ø8,159,17Ø,147
11ØØ RESTORE
111Ø REM HOW MANY PHRASES?
112Ø NP=INT(M/15)
113Ø IF NP=Ø THEN NP=4
114Ø REM DO THE CHIMING
115Ø FOR I=1 TO NP
116Ø FOR J=1 TO 4
117Ø READ BL
118Ø GOSUB 3ØØØ
119Ø NEXT J
12ØØ REM A BIGGER GAP
121Ø FOR ST=1 TO 2ØØ
122Ø NEXT ST
123Ø NEXT I
124Ø REM AND THE HOUR?
125Ø IF M< >Ø THEN RETURN
126Ø REM AN EVEN BIGGER GAP
127Ø FOR ST=1 TO 5ØØ
128Ø NEXT ST
129Ø BL=32
13ØØ FOR DG=1 TO H
131Ø GOSUB 3ØØØ
132Ø REM ANOTHER GAP
133Ø FOR ST=1 TO 2ØØ
134Ø NEXT ST
135Ø NEXT DG
136Ø RETURN
```

2 Threerific noises — PLAY

The PLAY statement is a more versatile music maker than the SOUND statement. It uses character strings to define all the qualities of a particular note, and one big advantage of it is that you use the normal note names to make a tune. You write

line number PLAY tune

'tune' is a character string — a constant or a variable or an expression. Try these

PLAY "A"

PLAY "G" + "C"

They all work. Your 'tune' can include many bits of information separated by semicolons:

Notes 'A' to 'G' with sharps and flats, or semi-tones '1' to '12'

O — switches Octave. You can have O1 to O5.
Leave it out and O2 is used. Once you set it, it stays set.
L — change note length. You can have L1 to L255.
Leave it out and the last length you used stays set.
T — for Tempo. You can have T1 to T255.
Leave it out and T2 is used. Once you set it, it stays set.
V — for volume. You can have V1 to V31.

Leave it out and V15 is used. Once you set it, it stays set.

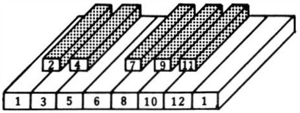
P — for Pause. You can have P1 to P255.
Leave it out and there is no pause.

X — for eXecute. Followed by the name of a string, which is PLAYed.

As you can see, there are quite a few facilities. We will illustrate them all.

(a) Notes A to G or '1' to '12'.
You can say F+ or F# for F sharp (or any other sharp) and B— for B flat (or any other flat). The numbers correspond to the twelve semitones:

Note	Number
C or B#	1
C# or D—	2
D	3
E— or D#	4
E or F—	5
F or E#	6
F# or G—	7
G	8
G# or A—	9
A	10
B— or A#	11
B or C—	12



This little program plays a chromatic scale:

```
10 FOR I=1 TO 12
20 PLAY STR$(I)
30 NEXT I
```

Notice the use of the STR$ function to convert a value of I to the corresponding character.

Right! Here's a little tune in F major by Thomas Tallis. I'm using it because all the notes have the same length and lie in the same octave, although I have cheated once.

```
10 REM A LITTLE CANON
20 PLAY"F;F;E;F;F;G;G;A"
30 PLAY"F;B-;B-;A;A;G;G;F"
40 PLAY"A;B-;G;A;A;G;G;F"
50 PLAY"C;D;E;F;A;G;G;F"
60 GO TO 20
```

This is a 'canon' or 'round' which means that the phrases will all harmonize with each other. Do you have a friend with a DRAGON?

Get them to start one phrase after you. How beautiful. Up to four people can do this! Here it is again using numbers:

```
10 REM ANOTHER WAY
20 PLAY"6;6;5;6;6;8;8;10"
30 PLAY"6;11;11;10;10;8;8;6"
40 PLAY"10;11;8;10;10;8;8;6"
50 PLAY"1;3;5;6;10;8;8;6"
```

(b) Tempo, Octave, and Pause

It is quite easy to vary the tempo and extend the range of notes to several octaves. We select the tempo with the letter T;

```
10 PLAY "T1"
```

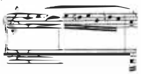
selects tempo 1, which will stay set until we change it. If a program doesn't set the tempo, it starts off at T2. The allowed tempos are between T1 (slow) and T255 (very, very fast).

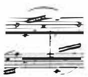
We can also select the octave we are using with the letter O. The normal octave set when you RUN a program is O2. O1 is lower, and O3, O4 and O5 are higher.

In this little program we play a fragment of a famous piano piece by Beethoven. We will complete it in a moment, but first of all see how the octaves and notes are defined in DATA statements, and read into the string variable X$

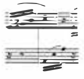
```
10 REM TAKE THIS, ELSIE
20 REM DEFINE THE TUNE
30 DATA"O3;E;D#;E;D#;E"
40 DATA "O2;B;O3;D;C;O2;A"
50 DATA"O1;E;A;O2;C;E;A;B"
60 DATA"O1;E;G#;O2;E;G#;B;O3;C"
70 DATA"O1;E;A;O2;E"
80 REM DEFINE THE TEMPO
90 PLAY"T8"
100 REM AND PLAY IT
110 RESTORE
120 FOR I=1 TO 5
130 READ X$
140 PLAY X$
150 NEXT I
160 GO TO 110
```

Now wait a minute, it doesn't go on forever, does it? The actual piece contains a lot of phrases repeated, and so we would be clever to try and make use of that, as follows:

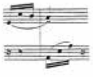
Suppose X1$ = 

X2$ = 

X3$ =



X4$ =



Then the piece goes (more or less) XI$;X2$;X3$;XI$;X2$;X4$ and then all over again. There is then a middle section which I break into 5 new bits called Y1$ to Y5$. Then various X and Y combinations conclude the piece. Here it is:

```
10 REM.LUDWIG'S MASTERPIECE
20 REM MAKE ALL THE PHRASES
30 X1$="O3;E;D#;E;D#;E;O2;B;O3;D;C;O2;A"
40 X2$="O1;E;A;O2;C;E;A;B;O1;E;G#"
50 X3$="O2;E;G#;B;O3;C;O1;E;A;O2;E"
60 X4$="O2;E;O3;C;O2;B;A;O1;E;A"
70 Y1$="O3;C;D;E;O1;G;O2;C;G;O3;F;E;D"
80 Y2$="O1;G;B;O2;F;O3;E;D;C"
90 Y3$="O1;E;A;O2;E;O3;D;C;O2;B"
100 Y4$="O1;E;O2;E;E;O3;E;O2;E;O3;E"
110 Y5$="E;O4;E;O3;D#;E;D#;E;D#;E;D#"
120 REM NOW PLAY IT
130 PLAY"T8"
140 PLAY X1$+X2$+X3$
150 PLAY X1$+X2$+X4$+"P8"
160 PLAY X1$+X2$+X3$
170 PLAY X1$+X2$+X4$+"O2;B"
180 PLAY Y1$+Y2$+Y3$
190 PLAY Y4$+Y5$
200 PLAY X1$+X2$+X3$
210 PLAY X1$+X2$+X4$+"O2;B"
220 PLAY Y1$+Y2$+Y3$
230 PLAY Y4$+Y5$
240 PLAY X1$+X2$+X3$
250 PLAY X1$+X2$+"O2;D;O3;C;O2;B;T2;A"
```

Can you count the notes? It's pretty good going for a 25 line program! You will see a few adjustments to get it just right. I tacked a pause on the end at line 50, and instead of that a B at line 160. If you study Beethoven's piece you will see why. The pause

'P1'

gives a break one note long at tempo T1. So P8 is the same length as a note at T8.

At the end of the piece, there is a note four times slower. We will shortly see another way of doing this.

(c) Volume and tempo — and leaving out semicolons

If we study Beethoven's score, we will see that we are asked to get louder and softer quite often, and also to slow down and speed up a few times.

The volume is set by the character V. If we say nothing about volume, it sets V15. You can have anything from V1 to V31. Here's a tone that gets louder:

```
10 REM CRESCENDO
20 PLAY"T16;V1"
30 FOR I=2 TO 31
40 PLAY"G;V"+STR$(I)
50 NEXT I
```

In the Beethoven piece, we could change the volume and the tempo using T and V strings. But there is a very clever way of making the computer get one step louder or softer, and also ways of speeding up and slowing down. You put

T+ to speed up by one step	V+ to get one step louder
T− to slow down by one step	V− to get one step quieter
T< to double the speed	V< to double the volume
T> to halve the speed	V> to halve the volume.

Try this:

```
10 REM ACCELERANDO
20 PLAY"T1"
30 FOR I=2 TO 100
40 PLAY"C;F;T+"
50 NEXT I
```

and this

```
10 REM CRESCENDO
20 PLAY"T8;V1"
30 FOR I=2 TO 31
40 PLAY"C;F;V+"
50 NEXT I
```

Now if you look at the first bar of the Beethoven piece that we used above, you can see that the volume is supposed to swell and diminish. So we will change X1$ accordingly:

```
30 X1$="O3;E;D#;T8;V+;E;V+;D#;V+;E;O2;V-;B;O3;V-;D;V-;C;O2;A"
```

Now this string is getting a bit long, so you may prefer it with the semicolons left out — actually with one exception that you haven't seen yet, the semicolons can be left out:

```
30 X1$="O3ED#T8V+EV+D#V+EO2V-BO3V-DV-CO2A"
```

Similarly there is a slowing down in several places, usually at the same time as the volume is changing. Here is the whole piece with the volume and speed adjusted throughout. I have kept some semicolons to make it more readable.

```
10 REM LUDWIG'S MASTERPIECE
20 REM MAKE ALL THE PHRASES
30 X1$="O3;E;D#;T8;V+;E;V+;D#;V+;E;O2;V-;B;O3;V-;D;V-;C;O2;A"
40 X2$="O1;E;A;O2;V+;C;V+;E;V+;A;B;O1;E;G#"
50 X3$="O2;V+;E;V+;G#;V+;B;O3;C;O1;E;A;O2;E"
60 X4$="O2;V-;V-;E;O3;V-;V-;C;O2;V-;V-;B;A;O1;E;A"
70 Y1$="O3;V+;V+;C;V+;V+;D;V+;V+;E;O1;G;O2;C;G;O3;F;E;D"
80 Y2$="O1;G;B;O2;F;O3;E;D;C"
90 Y3$="O1;E;A;O2;E;O3;D;C;O2;B"
100 Y4$="O1;E;O2;E;E;O3;E;O2;E;O3;E;V-"
110 Y5$="E;V-;O4;E;V-;O3;D#;V-;T-;E;V-;D#;V-;E;V-;D#;V-;T-;E;V-;D#;V-"
120 REM NOW PLAY IT
130 PLAY"T8;V2"
140 PLAY X1$+X2$
145 PLAY X3$+X1$
150 PLAY X2$+X4$+"P8"
160 PLAY X1$+X2$
165 PLAY X3$+X1$
170 PLAY X2$+X4$+"O2;B"
180 PLAY Y1$+Y2$+Y3$
190 PLAY Y4$+Y5$
200 PLAY X1$+X2$
205 PLAY X3$+X1$
210 PLAY X2$+X4$+"O2;B"
220 PLAY Y1$+Y2$+Y3$
230 PLAY Y4$+Y5$
240 PLAY X1$+X2$
245 PLAY X3$+X1$
250 PLAY X2$+"O2;V-;T-;D;O3;V-;T-;C;O2;V-;T-;B;T1;V-;A"
```

(d) Lengths, dots, and substrings.

The character L in the PLAY statement sets the lengths of notes. Previously, we had set the tempo and had all the notes at the same length — the length you get this way is the same as L4. So the tempo set by T is like one beat in 4/4 time. If you set a length with L, it stays set until you change it again.

L1	—	one whole note (semibreve)	𝅝
L2	—	one half note (minim)	𝅗𝅥
L4	—	one quarter note (crochet)	♩
L8	—	one eighth note (quaver)	♪
L16	—	one sixteenth note (semiquaver)	𝅘𝅥𝅯

and so on

EXAMPLE:

A short fragment of tune with quarter and eighth notes

```
10 REM COUNTING SHEEP
20 PLAY"T4"
30 PLAY"L4;C;C;F;F;L8;G;A;B-;G;L2;F"
```

Dotted notes in music have their lengths extended by half; they can also be double dotted or more:

L4	—	one quarter note	♩
L4.	—	1/4 + 1/8	♩.
L4..	—	1/4 + 1/8 + 1/16	♩..

EXAMPLE:
Contains a dotted note

```
10 REM TUM TE TUM
20 PLAY"T4"
30 PLAY"L4.;C;L8;F;L2;G"
```

And finally, there is a third way that the PLAY statement can give its string of music. This is called a substring. If you use the letter X followed by the name of a string variable, the computer executes the string variable as music. You always have to follow the $ in the substring name with a semicolon, even at the end of a PLAY statement. So we have three ways:

(i) String constant
PLAY "C;D;E;F;G"

(ii) String variable or expression

```
10 X$="C;D;E;F;G"
20 PLAY X$
```

or even

```
20 PLAY X$+X$
```

(iii) Substrings

```
10 X$="C;D;E;F;G"
20 PLAY"T8;XX$;T16;XX$;"
```

(note the semicolon at the end)

or even

```
10 X$="C;D;E;F;G"
20 Y$="T8;XX$;T16;XX$;"
30 PLAY"O1;XY$;O2;XY$;O3;XY$;"
```

in which the substring Y$ contains another substring, X$. The substring can be very useful because you can put long phrases together. Remember that strings can only be 255 characters long — when you use method (ii) you will be in trouble if X$+Y$ is longer than 255 characters, but not in method (iii).

EXAMPLE:

Brahms' lullaby contains changing lengths and dotted notes. I have done it with substrings.

```
10 REM GO TO SLEEP
20 REM DEFINE TUNE
30 X1$="O2L8AAO3L4.CO2L8AL4AO3CP4"
40 X2$="O2L8AO3CL4FL4.EL8DL4DC"
50 X3$="O2L8GAL4B-GL8GAB-P4"
60 X4$="L8GB-O3EDL4CEFP4"
70 X5$="O2L8FFO3L2FL8DO2B-O3L2C"
80 X6$="O2L8AFL4B-O3CDO2AO3C"
90 X7$="O2L8FFO3L2FL8DO2B-O3L2C"
100 X8$="O2L8AFL8.B-L32O3CO2B-L4AGL2F"
110 REM PATCH TOGETHER AS SUBSTRINGS
120 Y1$="XX1$;XX2$;XX3$;XX4$;"
130 Y2$="XX5$;XX6$;XX7$;XX8$;"
140 REM PLAY IT
150 PLAY"T2;XY1$;XY2$;"
```

3 Phantastic Graphics

Another string-controlled facility is provided by the DRAW statement. As with PLAY, you write something like

line number DRAW picture

and the computer will make line drawings for you on the graphics screen following the instructions given by 'picture' which is a string. It is as if you had a graphics pen which you can push around the screen, changing its colour and turning it on and off. It has the following controls

M x,y	—	move 'pen' to screen co-ordinate x,y. As with all operations on the graphics screen, x has the range 0-255 and y has the range 0-192.
U distance	—	move pen up distance 'distance'
D distance	—	move pen down distance 'distance'
L distance	—	move pen left distance 'distance'
R distance	—	move pen right distance 'distance'
E distance	—	move pen at angle 45 (up and right) distance 'distance'
F distance	—	move pen at angle 135 (up and left) distance 'distance'
G distance	—	move pen at angle 225 (down and left) distance 'distance'
H distance	—	move pen at angle 315 (down and right) distance 'distance'

X string name	—	executes a substring
C number	—	set colour of pen to colour number 'number'
A angle	—	start drawing at angle 'angle'
S scale	—	scale drawing by scale 'scale'
N	—	return to present position after drawing
B	—	don't draw — just move pen

You put these together, separated by semicolons which are mostly optional, just as with the PLAY statement.

(a) Simple shapes

We can use the DRAW statement to make simple shapes. This one makes a rectangle. First we move the 'pen' to 64,48. We use the B prefix to get it there without drawing a line. Then right, up, left, and down. Notice PMODE and all that because we are on the graphics screen.

```
10 REM DON'T BE SQUARE
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS
50 DRAW"BM64,194;R128;U96;L128;D96"
60 GO TO 60
```

We can also draw an 'isometric', or three dimensional view of a box quite easily. Can you see the order in which it's done? One of the lines is gone over twice — unless you use another M string (which is more bother) it is unavoidable.

```
10 REM BOXED RIGHT IN
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS
50 DRAW"BM64,48E32R128G32L128D96"
60 DRAW"R128E32U96G32D96"
70 GO TO 70
```

Notice that I left the semicolons out — just as with PLAY you can do this — except after the names of substrings as you will see.

(b) Colour, Angle, Scale

You can use the string

C colour number

to change the pen colour to any colour in the colour set being used, as determined by the PMODE and SCREEN statements — see Chapter 13.

You can cause the drawing to be rotated by an angle by using

A angle

The available angles are

A0 normal	A2 180° clockwise
A1 90° clockwise	A3 270° clockwise

You can alter the scale of the drawing, having it magnified or reduced by putting

S scale

where scale can be a whole number from 1 to 62 and indicates the scale factor in quarters, i.e.

S1	is	1/4 scale
S2	is	1/2 scale
S3	is	3/4 scale
S4	is	4/4, i.e. normal scale
S5	is	5/4, i.e. blown up to 125%
S8	is	double scale

and so on.

Here we use these all together. The box is drawn using colours 0 to 7, scales 1 to 8, and at different angles:

Scale	Angle	Colour
1	0	2
2	1	3
3	2	2
4	3	3

Actually, the program isn't much different. See how each of these new parameters is set by concatenating a STR$ function onto the character code.

```
10 REM SWINGING BOXES
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS 1
50 FOR I=1 TO 4
60 DRAW "S"+STR$(I)
70 DRAW "A"+STR$(I-1)
80 DRAW "C"+STR$(I+2-INT(I/2)*2)
90 DRAW"BM128,96E32R64G32L64D64"
100 DRAW"R64E32U64G32D64"
110 NEXT I
120 GO TO 120
```

(c) Relatively speaking

So far we have used the M string to move the 'pen' to the starting position of our drawings; it was preceded by B to blank it out. Without the B, the M string draws a 'vector' from the present position of the pen to its new place. Try this: you will see that the 'pen' zigzags in the screen corner

```
 1Ø REM ABSOLUTE VECTORS          7Ø REM START DRAWING
 2Ø PMODE 1,1                     8Ø FOR X=Ø TO 127 STEP 8
 3Ø SCREEN 1,1                    9Ø DRAW"M"+STR$(X+1)+",Ø"
 4Ø PCLS                          1ØØ DRAW"MØ,"+STR$(X)
 5Ø REM MOVE TO CORNER            11Ø NEXT X
 6Ø DRAW"BMØ,Ø"                   12Ø GO TO 12Ø
```

If you use a sign with the x and y co-ordinates, the pen will do a 'relative' movement — instead of drawing a line to the co-ordinate x,y, it will move by X and Y positions. Almost the same program:

```
 1Ø REM RELATIVE VECTORS          7Ø REM START DRAWING
 2Ø PMODE 1,1                     8Ø FOR X=Ø TO 127 STEP 8
 3Ø SCREEN 1,1                    9Ø DRAW"M+"+STR$(X+1)+",Ø"
 4Ø PCLS                          1ØØ DRAW"M+Ø,"+STR$(X)
 5Ø REM MOVE TO CORNER            11Ø NEXT X
 6Ø DRAW"BMØ,Ø"                   12Ø GO TO 12Ø
```

Do you understand the difference? The + after the M in the DRAW statements does this. You can have positive or negative steps, as long as there is some sign on the X co-ordinate. Here's a spiral:

```
1Ø REM SPIRALY VECTORS              7Ø DRAW"M+"+STR$(X)+",Ø"
2Ø PMODE 1,1                        8Ø DRAW"M+Ø,-"+STR$(X)
3Ø SCREEN 1,1                       9Ø DRAW"M-"+STR$(X+4)+",Ø"
4Ø PCLS                             1ØØ DRAW"M+Ø,"+STR$(X+4)
5Ø REM SPIRAL OUT FROM CENTER       11Ø NEXT X
6Ø FOR X=1 TO 128 STEP 8            12Ø GO TO 12Ø
```

You can see that I didn't move the pen before starting the spiral. The pen is always at (128,96) when you RUN a program.

(d) To update or not to update — that is the N

When you use any of the pen motion strings, the position of the pen is 'updated' after each move — it is put at the end of whatever line you have just drawn. We used this program to draw a box:

```
1Ø REM BOXED RIGHT IN     5Ø DRAW"BM64,48E32R128G32L128D96"
2Ø PMODE 1,1              6Ø DRAW"R128E32U96G32D96"
3Ø SCREEN 1,1             7Ø GO TO 7Ø
4Ø PCLS
```

Now if we prefix a move command with the letter N — for No update — the pen goes back to where it started. So if we take the same program but put N in front of each move, then all the lines are drawn from one place. The box program now produces a star:

```
10 REM NO UPDATES TODAY, THANKS
20 PMODE 1,1
30 SCREEN 1,1
40 PCLS
50 DRAW"NE32NR128NG32NL128ND96"
60 DRAW"NR128NE32NU96NG32ND96"
70 GO TO 70
```

The N prefix takes effect for only one move.

(e) Substrings and big explosions
As with PLAY, the DRAW statement can be told to execute a substring with the X string. The X is followed by the name of a string variable, and you always need a semicolon after that. You can have things like

DRAW "BM0,0R32D32XA$;"

and substrings can call substrings which call substrings . . . .

This program uses A$ as a substring that draws a star. It sets a random scale factor and chooses a random colour before drawing the star at a random location using the substring A$.

```
10 REM FILL UP WITH STARS
20 A$="NU8ND8NL8NR8NE8NF8NG8NH8"
30 PMODE 3,1
40 SCREEN 1,1
50 PCLS
60 FOR I=1 TO 40
70 REM RANDOM SCALING
80 DRAW"S"+STR$(RND(32))
90 REM RANDOM COLOR 2 3 OR 4
100 DRAW"C"+STR$(RND(3)+1)
110 REM RANDOM X AND Y
120 X=RND(256)-1
130 Y=RND(192)-1
140 REM HERE COMES SUBSTRING
150 DRAW"BM"+STR$(X)+","+STR$(Y)+"XA$;"
160 NEXT I
170 GO TO 50
```

When you run this program, you will see that the DRAGON very kindly lets you wander off the screen without getting into trouble.

Twenty Four

TURN THE TABLES

I A whole new dimension

We know how to use lists - they are arrays with one subscript. I expect we also know that working with a subscript often requires careful thought. Now we will find out that arrays with two or more subscripts are possible, although if you use a lot of subscripts you will run out of memory very quickly. You could think of a list, such as A, defined by

```
        10 DIM TH(3)
```

as being laid out in a column:

```
TH(0)
TH(1)
TH(2)
TH(3)
```

With two subscripts you have a table, like

```
        10 DIM TB(3,3)
```

which specifies

```
TB(0,0)   TB(0,1)   TB(0,2)   TB(0,3)
TB(1,0)   TB(1,1)   TB(1,2)   TB(1,3)
TB(2,0)   TB(2,1)   TB(2,2)   TB(2,3)
TB(3,0)   TB(3,1)   TB(3,2)   TB(3,3)
```

When you use a table, you write two subscripts separated by a comma; the first gives the row number and the second is the column number

TB(I,J) for Row number I, Column number J

If you use a variable name with two subscripts, the computer knows that you are talking about a table. But you have to be consistent. Once a variable is a list, you can't make it into a table and vice versa. As with a list, if you don't have a DIM statement for an array with two subscripts, BASIC will assume that each varies from 0 to 10, and it will therefore reserve 121 spaces in memory for it. As this is quite a lot of space, you should always remember to give a DIM statement if you want less. The normal array of three subscripts would use 1331 spaces, and so as the number of subscripts increases it becomes more and more important to have a DIM statement. The DIM statement can be used to specify several arrays:

```
line number  DIM  variable  sizes with  , variable  sizes with  , . . .
                            commas                  commas
```

The variables can be ordinary, integer, or string variables.

2 Do your budget

A lot of things in real life turn out to be best represented by tables. Here is a family budget:

The Fairwetherstonehaugh Family Budget

	The Stately Home	The Rolls	Banqueting	Couturier	Opera and Ballet
January	3000	540	900	1500	700
February	3000	560	900	1200	100
March	2000	580	1200	1650	0
April	1500	240	300	1300	300
May	1000	600	400	900	50
June	500	650	400	200	2300

(Glyndebourne and Bayreuth!)

The family name, by the way, is pronounced as if it were 'Fanshawe'. No kidding!

You could set this out in a DATA statement, and read it into a table:

```
10 REM FANSHAWE FAMILY BUDGET
20 REM THE STATELY HOME
30 DATA 3000,3000,2000,1500,1000,500
40 REM THE LIMOUSINE
50 DATA 540,560,580,240,600,650
60 REM THE CORDON BLEU
70 DATA 900,900,1200,300,400,400
80 REM THE STYLISH RAGS
90 DATA 1500,1200,1650,1300,900,200
100 REM THE ENTERTAINING
110 DATA 700,100,0,300,50,2300
120 DIM FB(6,5)
130 FOR J=1 TO 5
140 FOR I=1 TO 6
150 READ FB(I,J)
160 NEXT I
170 NEXT J
```

You can also put in captions for the rows and columns:

```
180 REM THE CAPTIONS
190 DATA JAN,FEB,MAR,APR,MAY,JUN
200 DATA HOUSE,MOTOR,FOOD,RAGS,GIGGLES
210 FOR I=1 TO 6
220 READ CR$(I)
230 NEXT I
240 FOR J=1 TO 5
250 READ CC$(J)
260 NEXT J
```

I have used I for column numbers and J for row numbers for no particular reason except that people usually do. After all of this the budget is in FB, and CR$ has the row captions and CC$ the column captions.

Now let's work on this table. First find out what the half-yearly budget is for each item:

```
270 CLS
280 PRINT"HERE IS THE FANSHAWE BUDGET"
290 PRINT
300 PRINT
310 REM DO HALF YEAR'S BUDGET
320 FOR J=1 TO 5
330 SM=0
340 FOR I=1 TO 6
350 SM=SM+FB(I,J)
360 NEXT I
370 PRINT"YOUR " CC$(J) " BUDGET IS" SM
380 PRINT
390 NEXT J
400 INPUT"PUSH ENTER TO GO ON";A$
```

and the monthly totals:

```
410 CLS
420 PRINT "HERE ARE YOUR MONTHLY TOTALS"
430 PRINT
440 FOR I=1 TO 6
450 SM=0
460 FOR J=1 TO 5
470 SM=SM+FB(I,J)
480 NEXT J
490 PRINT "IN " CR$(I) " YOU WILL SPEND" SM
500 PRINT
```

```
510 NEXT I
520 INPUT"PUSH ENTER TO GO ON";A$
```

and compute the total cash flow for the half year:

```
530 SM=0
540 FOR I=1 TO 6
550 FOR J=1 TO 5
560 SM=SM+FB(I,J)
570 NEXT J
580 NEXT I
590 CLS
600 PRINT "YOUR TOTAL EXPENDITURE THIS HALF"
610 PRINT"            IS "SM
620 PRINT
630 PRINT"BETTER SELL THE FAMILY SILVER !"
```

Do you see how the nested loops have been used with the subscripts to work out each sum?

You could use this program to do your own budget. Put your own expenses in the DATA statements. Extend the number of items and months if you want - you will have to reorganize your output if you do. When you finish your program will look quite different, but it will be the same, if you see what I mean.

3 Animation through arrays - GET and PUT

Now that we know about arrays of two dimensions, you will be delighted to find that you can use them to save little bits of the graphics screen, and put them back again. This is a nice way to organize animation, as you will see.

You can tuck away any little rectangle from the graphics screen into an array by writing

line number GET (left corner) — (right corner), array, G

In this statement,

'leftcorner'	is a pair of x,y co-ordinates giving the graphics screen co-ordinates of the upper left corner of what you want to save.
'rightcorner'	is a pair of x,y co-ordinates giving the graphics screen co-ordinates of the lower right corner of what you want to save.
'array'	is the name of the array to save it in
'G'	— don't ask questions, just use it.

When you want your rectangle back on the screen, write

line number PUT (leftcorner) — (rightcorner), array, action

In this statement,

'leftcorner', 'rightcorner' and 'array' are as above.
'action' specifies what you want done with the picture.

You can choose

PSET	—	set the screen rectangle to what is in the array
PRESET	—	any point set in the array is reset on the screen — you can make a picture disappear this way.
AND	—	clears every point in the screen rectangle unless it is set in the array.
OR	—	sets every new point in addition to what is already there. This adds the pictures together, with your new picture underneath.
NOT	—	'reverses' the picture in the screen rectangle. It doesn't actually use the picture in the array.

GET and PUT will work together properly if you use the same PMODE resolution for each. The size of the rectangle has to be the same as the size of the array. The next example illustrates this.

EXAMPLE:

That rolling ball again. First we define a circle on the screen. I centre it at 100, 100 and give it radius 12. Then I copy it into the array CR. The rectangle I copy has corners at 88,88 and 112,112. The size of it is therefore 25x25 which is (112-88+1) — remember that it includes both corners. Therefore CR has to be dimensioned

DIM CR(25,25)

This first program runs it across using PSET:

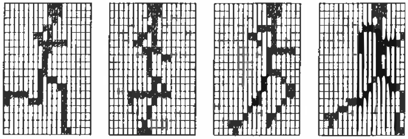
```
1Ø REM PUSHOVER                    1ØØ PCLS
2Ø PCLEAR 4                        11Ø LINE (Ø,175)-(255,191),PSET,BF
3Ø DIM CR(25,25)                   12Ø LINE (Ø,175)-(255,191),PRESET,B
4Ø PMODE 4,1                       13Ø SCREEN 1,1
5Ø REM SET UP CIRCLE               14Ø REM NOW START IT MOVING
6Ø PCLS                            15Ø FOR X=Ø TO 239 STEP 12
7Ø CIRCLE (1ØØ,1ØØ),12,1           16Ø PUT (X,15Ø)-(X+24,174),CR,PSET
8Ø GET (88,88)-(112,112),CR,G      17Ø NEXT X
9Ø REM SET UP SCREEN               18Ø GO TO 18Ø
```

Not terribly good. Do you see what is happening? To use OR, change line 160:

```
1Ø REM PUSHOVER                    1ØØ PCLS
2Ø PCLEAR 4                        11Ø LINE (Ø,175)-(255,191),PSET,BF
3Ø DIM CR(25,25)                   12Ø LINE (Ø,175)-(255,191),PRESET,B
4Ø PMODE 4,1                       13Ø SCREEN 1,1
5Ø REM SET UP CIRCLE               14Ø REM NOW START IT MOVING
6Ø PCLS                            15Ø FOR X=Ø TO 239 STEP 12
7Ø CIRCLE (1ØØ,1ØØ),12,1           16Ø PUT (X,15Ø)-(X+24,174),CR,OR
8Ø GET (88,88)-(112,112),CR,G      17Ø NEXT X
9Ø REM SET UP SCREEN               18Ø GO TO 18Ø
```

4 Quite the example — not only marathon runners but also (gasp) hexadecimal numbers!

Here I'm going to place a running figure on the screen. The use of GET and PUT to animate this is easy. Most of the complicated bit is in creating the four 'shots' of the runner that I use. This is how I designed the runner on graph paper:



Each shot is 18 screen locations wide and 18 high. Look at the top row of the first frame. All that is needed in this row is three blobs of the runner's head. If this figure uses only colours 0 and 1, I can define this row in a DATA statement:

```
1Ø REM THE HARD WAY
2Ø DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,1,1,1,Ø,Ø,Ø,Ø,Ø,Ø
```

To make all four frames this way would require 72 DATA statements with a total of 1296 zeros and ones. This is a bit excessive.

However, notice that the data looks a lot like a binary code. Surely I can give each row as an ordinary number, and 'crack' it into binary using the sort of decimal to binary subroutine that I told you about in Chapter 20. So I can give the whole top row of the first shot as one number which happens to be

448 whose binary code is 111000000

The only thing that is wrong with this is the difficulty of working out the code 448 in the first place. What we need is a code which is easier to construct from binary than a decimal number but less cumbersome than binary, which just has too many digits. The solution is to use the hexadecimal number system.

As it turns out, computers are generally organized around groups of four, eight, and sixteen binary bits. You may know that an 8 bit group is called a 'byte'. With 4 bits you can count from zero to 15, so if you use base 16, or 'hexadecimal' for counting everything works out conveniently. Here is a list of the possible digits in hexadecimal.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
8	1000	8	12	1100	C
9	1001	9	13	1101	D
10	1010	A	14	1110	E
11	1011	B	15	1111	F

You can see that decimal and hexadecimal are the same from 0 to 9. Then the decimal system runs out of digits, while hexadecimal goes on. We use A, B, C, D, E and F. Once you get over the shock of seeing letters used as digits, it's quite easy.

So how do you make big numbers? Think of your everyday decimal system:

4	4	8
hundreds	tens	units

In hexadecimal, instead of counting in units, tens, hundreds, etc., you count in units, sixteens, two hundred and fifty sixes, and so on:

Hexadecimal	1	C	0
	256's	16's	units
Binary	0001	1100	0000

Which is 256 + 12x16 + 0 = 448!!

You can see that 1CO hexadecimal is the top row of the first frame of the runner.

So, hey presto, I can now render the first shot of the runner as 18 hexadecimal numbers:

Binary	Hexadecimal
00 0000 0001 1100 0000	1C0
00 0000 0001 1100 0000	1C0
00 0000 0000 1000 0000	80
00 0000 0011 0000 0000	300
00 0000 1101 0000 0000	D00
00 0000 0111 1110 0000	7E0
00 0000 0011 0000 0000	300
00 0000 0010 0000 0000	200
00 0000 0010 0000 0000	200
00 0000 0110 0000 0000	600
00 0000 0101 1000 0000	580
00 0000 0100 0110 0000	460
11 1110 0100 0010 0000	3E420
10 0001 1100 0010 0000	21C20
00 0000 0000 0010 0000	20
00 0000 0000 0010 0000	20
00 0000 0000 0010 0000	20
00 0000 0000 0011 0000	30

In Extended Color BASIC, if you write &H in front of a number, you are using a hexadecimal constant. The snag is that you are allowed up to four digits only.

&H0 is 0 decimal
&H1C0 is our friend 448 decimal
&H1000 is 4096 decimal
&HFFFF is 65536 decimal, the largest hexadecimal constant allowed.

This is very boring because our runner occasionally needs 5 digits. However the fifth digit occurs only four times, so I can add these in later, as you can see in the program. Here it is:

```
10 REM LONG DISTANCE RUNNER
20 PCLEAR 4
30 REM FIRST FRAME
40 DATA &H1C0,&H1C0,&H80
50 DATA &H300,&HD00,&H7E0
60 DATA &H300,&H200,&H200
70 DATA &H600,&H580,&H460
80 DATA &HE420,&H1C20,&H20
90 DATA &H20,&H20,&H30
100 REM SECOND FRAME
110 DATA &H380,&H380,&H100
120 DATA &H300,&HD00,&H1900
130 DATA &H780,&H260,&H200
140 DATA &H200,&H180,&H140
150 DATA &H3FE0,&H2100,&H200
160 DATA &H200,&H400,&HE00
170 REM THIRD FRAME
180 DATA &H70,&H70,&H20
190 DATA &H1E0,&H620,&H120
200 DATA &H7E,&H40,&H40
210 DATA &H70,&H88,&H86
220 DATA &H106,&H308,&H410
230 DATA &H860,&H3840,&H2000
240 REM FOURTH FRAME
250 DATA &H70,&H70,&H20
260 DATA &H1E0,&H221,&H23E
270 DATA &H230,&H20,&H20
280 DATA &H30,&H48,&H86
290 DATA &H101,&HE01,&H7001
300 DATA &HC002,&H8002,&H3
310 DIM A(17,17),B(17,17),C(17,17),D(17,17)
320 REM MAKE THE ACTUAL PICTURES
330 PMODE 4,1
340 SCREEN 1,1
350 REM FRAME ONE
360 GOSUB 1000
370 PSET (0,12,1)
380 PSET (1,12,1)
390 PSET (0,13,1)
400 GET (0,0)-(17,17),A,G
410 REM FRAME TWO
420 GOSUB 1000
430 GET (0,0)-(17,17),B,G
440 REM FRAME THREE
450 GOSUB 1000
460 GET (0,0)-(17,17),C,G
470 REM FRAME FOUR
480 GOSUB 1000
490 PSET(1,15,1)
500 GET (0,0)-(17,17),D,G
510 REM AND START RUNNING
520 PCLS
530 PUT (118,86)-(135,103),A,AND
540 PUT (118,86)-(135,103),A,PSET
550 PUT (118,86)-(135,103),B,AND
560 PUT (118,86)-(135,103),B,PSET
570 PUT (118,86)-(135,103),C,AND
580 PUT (118,86)-(135,103),C,PSET
590 PUT (118,86)-(135,103),D,AND
600 PUT (118,86)-(135,102),D,PSET
610 GO TO 530
1000 REM SUBROUTINE TO DEFINE
1010 REM ONE FRAME FROM DATA
1020 PCLS
1030 REM DECODE 18 NUMBERS
1040 FOR ZR=0 TO 17
1050 READ ZN
1060 ZC=17
1070 GOSUB 2000
1080 NEXT ZR
1090 RETURN
2000 REM SUBROUTINE TO SET ONE
2010 REM ROW ON THE SCREEN BY
2020 REM DECODING THE NUMBER ZN
2030 REM ROW NO ZR IS SET
2040 REM STARTS AT COLUMN ZC
2050 REM WORKS BACK TILL ZN ZERO
2060 DEF FNLS(I)=I-INT(I/2)*2
2070 PSET(ZC,ZR,FNLS(ZN))
2080 ZN=INT(ZN/2)
2090 IF ZN=0 THEN RETURN
2100 ZC=ZC-1
2110 GO TO 2070
```

In the program, the four shots of the runner are coded into hexadecimal in the DATA statements. The first task of the program is to decode these one at a time and create the pictures, which you can see taking place in the corner of the screen. Frames one and four need a few extra bits turned on because the DATA statements contain only the rightmost 16 columns. As the pictures are created, they are tucked away in the arrays A, B, C, D using GET.

The runner is then animated in the centre of the screen, using PUT. To erase each frame, I use AND because that would only leave on bits that are common to two frames.

Appendix — A SUMMARY OF BASIC ON THE DRAGON 32 COMPUTER

I BASIC programs

A program has numbered lines, or statements, like

line number statement

When you use the command RUN, the BASIC program in the computer is obeyed in the order of line numbers unless the program itself decides otherwise. The DRAGON does not require an END statement, but many computers do. A program stops running when it runs out of lines or an END statement is encountered. It can be restarted by a CONT command as long as the program has not been changed.

A program can be made to pause by pressing SHIFT and @. It will then continue if any key is pressed.

A program can be stopped by pressing BREAK. You can then enter commands to the computer. You can restart the program by entering CONT as long as the program has not been changed.

The STOP statement has the same effect as pushing BREAK.

Several statements can be written on one line separated by colons, as

```
10 FOR I=1 TO 500:PRINT I:NEXT I
```

2 Commands

Anything you type without a line number is a command (except when a program is running). The following would normally be used as commands although there is nothing to stop you including them in a program.

CLOAD "name" Load a program 'name' from cassette. If no 'name' given, loads next program. See Chapter 9.
CLOADM "name", offset Load a machine language program.
CONT Resume execution of a program after STOP or END statements or after BREAK key has been pressed.
CSAVE "name" Save the current program on cassette with the name 'name' if given. See Chapter 9.
CSAVEM "name",start, end, transfer. Save a machine language program.
DEL Deletes lines from a program

DEL or DEL−	Kills it all
DEL 20 −	Delete 20 to end
DEL 50	Delete line 50
DEL −200	Delete up to line 200
DEL 10−50	Delete all lines from 10 to 50

DLOAD "name",rate Loads a BASIC program from strange device at specified baud rate.
'rate' = 0 for 300 baud. 'rate' = 1 for 1200 baud.
EDIT line number Edit specified program line. The line is copied to the bottom of the screen and you then edit it to make a new copy.

Push SPACE	copies one character
nSPACE	copies n characters
n<	jumps n spaces to left
nC	change n characters
nD	delete n characters
nSc	search forward for nth occurence of character c
H	hack line here and begin inserting
X	extend the line by jumping to end and inserting
L	list the line again and continue editing
SHIFT and	escape from current subcommand (usually used when inserting)
ENTER	finished editing this line

See Chapter 2 for a description of the editor with examples.

LIST Print the current program on the screen

LIST or LIST −	Print entire program
LIST 20 −	Print line 20 to end
LIST 50	Print line 50
LIST −200	Print up to line 200
LIST 10−50	Print all lines from 10 to 50

LLIST Print the current program on the 600 baud serial printer. Options identical to LIST.
NEW Throw current program away and start a new one.
RENUM new, start, difference. Change the line numbers of the current program starting at line number 'start' whose new number is 'new', and going up by 'difference' each line. It's clever — changes GOTO, GOSUB, IF. . . THEN statements as required.

Most of the statements of BASIC can be executed like a command; for example if you enter without a line number

```
CLS 3
```

your screen is cleared to blue. Statements used this way are called 'direct' statements. Using colons you can string several statements into a command:

```
FOR I=1 TO 1Ø:PRINT TAB(I);"*":NEXT I
```

You cannot use the statements INPUT or DEF FN as direct statements.

Everything you type that begins with a number is a line of BASIC, except when a program is running.

(a) Entering
Simply type in the program; each line begins with a line number and ends by pressing RETURN. You can put several statements on one line with colons between, as in

```
30 FOR I=1 TO 10:PRINT I*I:NEXT I
```

(b) Correcting
Replace a line by typing it in full, or use the editor summarized under commands above, also described in Chapter 2.

(c) Inserting
A new line is inserted by using a suitable line number.

(d) Deleting
A line is removed by typing the line number and pushing RETURN. Or use the DEL command described above.

4 Numbers in BASIC

Numbers which are constants in BASIC can take four forms:

(a) A number with a decimal point, such as 2.71828.
(b) As above with the letter E and an exponent, e.g. 3.61E23 meaning 3.61×10^{23}.
(c) A hexadecimal constant written &H number with 'number' in the range 0-FFFF hexadecimal. Example &H3E5C.
(d) An octal constant written &O number with 'number' in the range 0-177777 octal. Example &H377.

5 Variables in BASIC

(a) Ordinary variables
in BASIC represent numerical values. They can be given names which are

single letters: A,B,C, . . . Z
or
a letter plus a digit: A0, A1, . . . Z9
or
two letters: BL, ZC, . . .

You cannot use the following as variable names: FN, IF, OR, ON, TO, GO.

(b) String variables
represent values which are character strings and have ordinary variable names with a $ sign: A$, CH$, P7$, . . .

(c) Scalars
A variable is 'scalar' if it is written without a subscript. It represents a single value, e.g. X7.

(d) Arrays
A variable represents an array if it is written with subscripts, such as T(1,2,3), A$(7), WC(I). There is no limit on the number of subscripts. A particular array must always have the same number.

(e) Subscripts
A subscript may be any expression; however, the value it gives must be between 0 and the maximum for that subscript. Non-integer subscripts are truncated to integers.

(f) Array sizes
The maximum size of a subscript for an array is 10 unless a size is set up by a DIM statement.

6 Character strings

(a) A sequence of symbols in quotation marks is a character string constant, e.g. "WHOOPEE". They can be used in PRINT statements, DATA, IF. . . THEN, and on the right hand side of assignment statements. In the DATA statement the quotation marks can be omitted unless the string contains a comma or a colon or has leading or trailing blanks. String constants may be found in the statements MID$, PRINT USING, PRINT @ . . . USING, PLAY and DRAW.
(b) String variables have names ending with $, such as A$, CH$, P7$. They can be used in the statements INPUT, GET, PRINT, READ, IF. . . THEN, LET, DIM (and therefore they can be subscripted).
(c) In comparing strings in the IF. . . THEN statement, the order is that of the ASCII codes tabulated in Section 13 of this Appendix. Strings at the beginning of the alphabet are less than those near the end.
'QR' is less than 'QN'
'QR' is less than 'QRA'
'QR' is less than 'QR ' (note the blank)
(d) Character expressions can be written which concatenate character strings using the + sign:
"GOOD" + A$

7 Arithmetic expressions

Operations use the hierarchy:

( )	expressions in brackets	high priority
↑	exponentiation	
* /	multiplication and division	
+ −	addition and subtraction	low priority

Operations of equal priority are done left to right.

Expressions can be written using ordinary variables or constants, e.g. (A+3)/D(I). A single variable or constant is itself an expression. You cannot do arithmetic with character string variables.

The operators *, ↑ and / normally have to appear between values but the operators + and − can appear in front of any value.

A*/B is illegal	A+−B is legal
A*−B is legal	A+−+−+−B is legal

8 Relational expressions

(a)	arithmetic expression	relational operator	arithmetic expression	e.g. PQ < 3.
	character string	relational operator	character string	e.g. "NOW" > L$(K).

The result of a relational expression is TRUE or FALSE. Relational expressions are used only in the IF. . . THEN statement.

(b) The relational operators are

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

(c) The operators NOT, AND, OR can be used in relational expressions. These have hierarchy

	AND	highest	
(inclusive)	OR		
	NOT	lowest	e.g. (P=Q 2) OR (A$>="B")

9 **Library functions**

BASIC on the DRAGON Computer provides the following functions.

ABS(X)	The absolute value of X, i.e. made positive. (Chapter 7).
INT(X)	The integer next below X. INT(5.5)=5. INT(−3.5)=−4. (Chapter 7).
FIX(X)	X truncated to an integer. FIX(5.5)=5. FIX(−3.5)=−3. (Chapter 7).
SIN(X)	The sine of X where X is an angle in radians (Chapter 19).
COS(X)	The cosine of X where X is an angle in radians (Chapter 19).
TAN(X)	Tangent of X where X is an angle in radians (Chapter 19).
ATN(X)	Angle in range $-\pi/2$ to $\pi/2$ with tangent X (Chapter 19).
EXP(X)	The value e^x (Chapter 19).
LOG(X)	The natural logarithm $\log_e X$ (Chapter 19).
SQR(X)	The square root of X where X must be positive (Chapter 7).
SGN(X)	Sign of X: 1 if positive, 0 if zero, −1 if negative (Chapter 7).
PEEK(X)	The value contained in memory address X.
RND(X)	A random number in the range 1-x. (Chapter 16).
ASC(X$)	The ASCII code of the first character in X$ (Chapter 22).
CHR$(X)	A string character whose ASCII code is X (Chapter 22).
LEFT$(X$,X)	String leftmost X characters of X$ (Chapter 22).
MID$(X$,S,X)	A string of X characters starting from the Sth character in X$ (Chapter 22).
POS(X)	The present column number of the screen cursor. X=1 gives printer, X=2 gives screen.
RIGHT$(X$,X)	String using the rightmost X characters of X$ (Chapter 22).
VAL(X$)	Convert X$ to a number, e.g. "12.34" becomes 12.34 (Chapter 22).
STR$(X)	Convert X to a string, e.g. 12.34 becomes "12.34" (Chapter 22).
INSTR(start, string, target)	Searches 'string' for target string 'target' starting at character number 'start' (Chapter 22).
STRING$(len, code or string)	Gives a string of 'len' identical characters specified by the ASCII 'code' or by the first character of the 'string' (Chapter 22).
HEX$(X)	String of 4 Characters giving X in Hexadecimal (Chapter 22).
TIMER	Set or read internal clock (Chapter 23).
MEM	Number of bytes of free memory (Chapter 17).
POINT	Status of a point on text screen (Chapter 12).
PPOINT	Status of a point on graphics screen (Chapter 15).
EOF(X)	On file X, returns FALSE(0) if there is more data, TRUE(1) if there is no more data.
JOYSTK(X)	Co-ordinate of joystick on text screen. X = 0 horizontal position of left joystick = 1 vertical position of left joystick = 2 horizontal position of right joystick = 3 vertical position of right joystick

USR(X) Jump to machine language subroutine at memory address X.
VARPTR(variable) Gives memory address of the variable.

In the above, X is any expression, which could include references to other functions. X$ is a character string. The X or X$ are called the argument or parameter of the function.

The TAB function is used in PRINT statements and is described in Section 11 of this Appendix.

10 The statements of DRAGON Colour Basic

These are given here in alphabetical order, and include some not described in the main text. Most of the statements of BASIC can be executed as 'direct statements', like a command, by typing them without a line number. Those that cannot be treated that way are DEF FN, and INPUT or INPUT# .

line number AUDIO ON or AUDIO OFF Sound from cassette recorder is switched on or off to television set. Not described in the text.

line number CIRCLE (column,row), radius, colour, ratio, start, end (Chapter 14) Draw a circle.

line number CLEAR bytes, top Sets aside 'bytes' bytes of string storage space up to 'top' as highest address. Not described in the text.

line number CLOAD "name" (Chapter 9) Loads program called 'name' from cassette. If no 'name' loads first program found.

line number CLOADM "name", offset Loads machine language program 'name' from tape. Not described in text.

line number CLOSE number Closes file number 'number'. Not described in text.

line number CLR Sets all variables to zero but not your program. The command RUN does this anyway. Not described in the text.

line number CLS colour (Chapter 12) Clears text screen to colour number 'colour'.

line number CMD value Send output to file number value instead of to the screen. To get it back to the screen, CLOSE the file. Not described in text.

line number COLOR foreground, background (Chapter 14) Sets graphics screen foreground and background colours.

line number CONT (Chapter 5) Continues program execution after STOP or END statements or after BREAK is pressed.

line number CSAVE "name" (Chapter 9) Save the current program on cassette and call it 'name'.

line number CSAVEM Saves a machine language program on cassette. Not described in text.

line number DATA constant, constant, . . . (Chapter 10) The constants are stored in the computer in order. Successive DATA statements add to the DATA list. Information from the list can be assigned to variables by the READ statement.
If the constants are character strings, they can be given with or without quotation marks. The quotation marks are required if the string contains a comma or colon or has leading or trailing blanks.

line number DEF FNxx(variable)=expression (Chapter 20) Used to define your own functions. See Section 12 of this Appendix.

line number DEFUSRn. Define entry point for machine language function. Not described in the text.

line number DEL. Normally a command. See Appendix Section 2.

line number DIM name(sizes), name(sizes), . . . (Chapter 17) The DIM statement specifies the maximum size of arrays. The sizes must be integer numbers. If an array is not mentioned in a DIM statement, then each subscript can vary from 0 to 10. An array can be mentioned in at most one DIM statement, and must always have the same number of subscripts.

line number DLOAD name,rate Load program 'name' from strange cassette at specified baud 'rate'. 0=300 baud, 1=1200 baud.

line number DRAW string (Chapter 23) Sophisticated line drawing statement.

line number EDIT line number (Chapter 2) Normally used as a command. See Appendix Section 2.

line number END (Chapters 2 and 21) When an END statement is reached, the execution of a program terminates, as it does when it runs out of lines. It can be restarted with the CONT command.

line number EXEC (address) Go to machine language program at 'address', or last CLOADM address. Not described in text.

line number FOR variable=expression a TO expression b STEP expression c (Chapter 8)
This statement begins a FOR... NEXT loop, which is repeated with the variable starting at the value given by expression a and stepping by expression c until it reaches expression b. The STEP is optional and if it is not given the step used is 1. The variable can be adjusted during the loop but the initial, final, and step values are set when the loop first begins and cannot be altered from inside the loop. A loop ends on a NEXT statement which must be present. FOR... NEXT loops using different variables may be nested.

line number GET (leftcorner) — (rightcorner), destination, G (Chapter 23)
Saves rectangle of graphics screen in array.

line number GOSUB line number (Chapter 21) Used to call a subroutine. See Section 12 of this Appendix.

line number a GOTO line number b (Chapter 5) This causes an immediate jump to line number b.

linenumber a IF relational expression THEN linenumber b (Chapter 6)
When the IF statement occurs, the relational expression is evaluated and if it is TRUE the program jumps to line number b. If FALSE, the program carries on with the next line after line number a. Relational expressions are described in Section 8 of this Appendix.

line number IF relational expression THEN statements ELSE statements (Chapter 6)
An alternative form of IF... THEN in which several statements can be executed in either the TRUE or FALSE cases provided the whole construction is put on one line.

line number INPUT variable, variable, ... (Chapter 4) This causes the computer to request information from the keyboard. It will prompt with a '?' and wait for the information to be entered. The user should enter the correct number of values with commas between. If too many items are given, your DRAGON will ignore the extra and tell you so. If too few are given it will keep asking for more. You can write:

line number INPUT "message"; variable, variable, ...

to prompt yourself with the message. You have to use a semicolon in this case.

line number INPUT# value, variable, variable, ... The same as INPUT but takes input from the file given by value, which must have been OPENed. This is not described in the text.

line number LET variable=expression (Chapter 4) The expression on the right hand side is worked out, and its value replaces the value of the variable. The word LET can be used in DRAGON Colour BASIC. Without the key word LET this is the normal assignment statement.

variable = expression

line number LIST Normally a command. See Appendix Section 2.

line number LLIST Normally a command. See Appendix Section 2.

line number LINE(col 1,row 1)—(col 2,row 2),PSET or PRESET,B or BF (Chapter 14)
Draws a line or a box or a solid block of colour.

line number LINE INPUT "message"; string (Chapter 22) Nearly useless statement for input of the string variable 'string'. The message is optional and the semicolon goes with the message.

line number MID$ (oldstring, position, length)=newstring (Chapter 22) Replaces 'length' characters in the string 'oldstring' with the 'newstring' starting at 'position'.

line number MOTOR ON or MOTOR OFF. Turns cassette motor on and off. Not described in this text.

line number NEXT variable (Chapter 8) The NEXT statement indicates the end of a FOR... NEXT loop. You can leave out the variable name, and you can give several variable names separated by commas which are taken from left to right. This does not enable you to violate the proper nesting of loops.

line number NEW Normally a command — would make your program self-destruct. See Appendix Section 2.

line number a ON expression GOTO line number b, line number c, ... (Chapter 6)
or
line number a ON expression GOSUB line number b, line number c, ... (Chapter 19)
This allows a multiple choice of destinations. The expression is evaluated and truncated to an integer. If the result is 1, the branch is to linenumber b, if 2, to line number c, and so on. If the result is negative, zero, or too great for the number of destinations, the program continues with the next line after line number a. The statement is in two forms; one is a choice of destinations for an

ordinary jump and the other is a choice of subroutines.

line number OPEN name,# unit, file Opens the file whose value for INPUT# , CLOSE# and PRINT# is 'unit'. 'File' is 0 for keyboard/screen, 1 for cassette, 2 for printer. Not described in the text.

line number PAINT(col,row),colour,bound (Chapter 15) Colours in a shape, starting at graphics screen position (col,row) and painting to a boundary whose colour is 'bound'.

line number PCLEAR pages (Chapter 15) Reserves 'pages' pages of graphics memory. Normally 4 pages are cleared.

line number PCLS colour (Chapter 13) Clears graphics screen to desired 'colour', or background colour if 'colour' is omitted.

line number PCOPY source TO destination (Chapter 15) Copy graphics from source page to destination page.

line number PLAY string (Chapter 23) Sophisticated music making statement driven by strings.

line number PMODE mode, first page (Chapter 15) Selects resolution and memory used for graphics screen. See also Appendix Section 14 for modes.

line number POKE(address,value) Puts a 'value' in the range 0-255 into the memory at 'address'. Not described in the text.

line number PRESET(col,row) (Chapter 13) Reset a point on graphics screen to background colour.

line number PRINT value punctuation value . . . (Chapters 1 and 11) This produces printed output. See Section 11 of this Appendix.

line number PRINT# unit, value punctuation value . . . Produces output on the unit whose number is given, which must have been OPENed. It behaves like the PRINT statement described in Section 11 of this Appendix. Not described in this text.

line number PRINT USING image; output (Chapter 22) A form of PRINT statement with control over the layout.

line number PRINT @ screen address, output (Chapter 12) A variation on PRINT useful for graphics on the text screen.

line number PSET(column,row,colour) (Chapter 13) Colours a spot on the graphics screen.

line number PUT(leftcorner)–(rightcorner),source,options (Chapter 23) Puts a rectangle of graphics back on the screen.

line number READ variable, variable, . . . (Chapter 10) This assigns values to the variables in order from the list of values established by one or more DATA statements. Successive READ statements continue through the list. You have to be careful not to read numbers where character strings are expected, and vice versa.

line number REM any old comment (Chapter 4) This does nothing in a running BASIC program. It is provided to allow explanations to be inserted in programs.

line number RENUM new, start, difference Normally used as a command. See Appendix Section 2.

line number RESET(column,row) (Chapter 11) Resets a blob on the text screen when using graphics.

line number RESTORE (Chapter 10) Returns later READ statements to the beginning of the DATA list.

line number RETURN (Chapter 22) Used in subroutines. See Section 12 of this Appendix.

line number RUN Normally used as a command. See Appendix Section 2.

line number SCREEN screen, colours (Chapter 13) Selects graphics screen (1) or text screen (0) and colour set. Refer to Appendix Section 14 for available modes and colour sets.

line number SET(column,row,colour) (Chapter 12) Sets a blob of colour on the text screen.

line number SKIPF "name" (Chapter 9) Skips to end of program 'name' on cassette, or if no name skips one program.

line number SOUND tone, duration (Chapter 10) Makes the tone specified by 'tone' of duration 'duration' through television speakers. See Appendix Section 17 for musical note values.

line number STOP Your program will halt and tell you where it was broken into. You can carry on with the CONT command as long as you do not change the program.

line number TRON or TROFF (Chapter 8) Turns the program tracer on or off.

11 Printing

All output is done by the PRINT statement and its variations:

line number PRINT value punctuation value . . .
line number PRINT @ screen address, value puctuation value . . .

The values can be
- (a) expressions giving a numerical result
- (b) a string expression which is printed as a message
- (c) a character string in quotation marks
- (d) the TAB function.

The punctuation can be
- (a) a comma to jump to the next field on the screen. A row on the screen is divided into two columns, each 11 spaces wide.
- (b) a semicolon to squeeze the values together
- (c) omitted before or after a character string in quotation marks, in which case the effect is like a semicolon.

If you end a PRINT list with a comma or semicolon, the next PRINT statement continues on the same line. Otherwise all PRINT statements start a new line.

The TAB function

TAB(X) causes the output line to jump to column number X. It cannot, however, move backwards, and will use the next available space if the print position is already beyond column X. This is used only in a PRINT statement.

The PRINT @ statement

line number PRINT @ screen address, value, punctuation, value. . .

The screen address is in the range 0-511 and refers to a position on the text screen:

```
screen address = column no. + 32*row no.
                 0 to 31      0 to 15
```

You would normally end a PRINT @ statement with a semicolon:

```
10 PRINT@43,CHR$(207);
```

Refer to Appendix Section 13 for text screen graphics.

The PRINT USING statement

line number PRINT USING image; value, value, value, . . .

The image is a character string — constant, variable or expression — which gives a specification for the fields that the values are printed in. Described at length in Chapter 22.

12 Functions and subroutines

(a) Functions (Chapter 20)

You can define a one line function

line number DEF FNxx(variable) = expression where xx is any ordinary variable name allowed by BASIC, e.g. FNA, FNRO, FNZ7.

When a program uses a function, the expression on the right hand side is evaluated using the given value of the function variable, but using the true values of any other variables. A given function name can only be defined once, and must be defined before it is used. Functions may use other functions if they were defined earlier.

(b) Subroutines (Chapter 21)

Subroutines are called by the GOSUB statement:

```
     line number a GOSUB line number b
orby
     line number a ON expression GOSUB linenumber b,...
```

The program jumps to the subroutine at line number b (or other in the ON... GOSUB version) and executes it until a RETURN is given. It then resumes execution at the line after line number a.

Subroutines can call other subroutines, but should not set up an endless loop of calls.

Subroutines are ended by the RETURN statement:

```
     line number RETURN
```

The running program returns to the line following the latest GOSUB.

13 TheTextScreen

The text screen of the DRAGON Computer contains 16 lines numbered 0-15 and 32 columns numbered 0-31.

The colours available on the text screen are

0	Black	5	'Buff' — looks white to me
1	Green	6	Cyan
2	Yellow	7	Magenta
3	Blue	8	Orange
4	Red		

The PRINT statement in all its forms uses the text screen. Graphics on the screen can be organized with the following statements:

(a) line number PRINT@ screen address, information

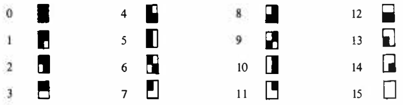
The screen address is in the range 0-511 and is

```
screen address = column no. + 32*row no.
                  0 to 31        0 to 15
```

The CHR$ function can be used to place graphics shapes on the text screen:

```
PRINT @ screen address, CHR$(code)
```

The codes are tabulated in Appendix Section 15, and include all the various symbols and characters available in the computer. Some of these are graphics symbols, as follows. The basic codes are:



All the shapes are some colour and black. To produce a coloured shape, add on:

128 + symbol number + (colour number−1)*16

Therefore codes
128 to 143 are green
144 to 159 are yellow
160 to 175 are blue
176 to 191 are red
192 to 207 are 'buff'
208 to 223 are cyan
224to239 are magenta
240to 255 are orange

(b) line number SET (column no., row no., colour no.)
0 to 63 0 to 31

This statement subdivides the 32x16 text screen so that each screen location is 4 subunits, hence the different row and column numbers. It sets a subcell to a colour. However only one colour and black are available in a screen position. For example Screen location 0, which is text column 0, row 0, contains four positions in the SET statement: (0,0), (0,1), (1,0), (1,1). These four must be the same colour or black. There are illustrations of this in Chapter 12.

(c) CLS Colour
Clears the textscreen to colour no. 'colour'

14 The GraphicsScreen

When the computer is turned on, or whenever you use a PRINT statement the text screen is displayed on the television. The graphics screen can be selected using the SCREEN statement

line number SCREEN screen, colour set

where 'screen'=0 for the text screen, 1 for the graphics screen. The 'colour set' selects a set of colours which depends on the graphics mode set in the PMODE statement.

The resolution and memory position of the graphics screen is selected by the PMODE statement:

line number PMODE resolution, page

where 'resolution' selects the resolution and 'page' selects an area of memory to be used for the screen.

The combinations available are:

Resolution code	Resolution on graphics screen columns x rows	Colours set 0	Colours set 1
0	128x96	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff
1	128x96	0,4,8=red 1,5=green 2,6=yellow 3,7=blue	0,4,8=orange 1,5=buff 2,6=cyan 3,7=magenta
2	128x192	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff
3	128x192	0,4,8=red 1,5=green 2,6=yellow 3,7=blue	0,4,8=orange 1,5=buff 2,6=cyan 3,7=magenta
4	256x192	0,2,4,6,8=black 1,3,5,7=green	0,2,4,6,8=black 1,3,5,7=buff

The PSET statement always refers to columns 0-255 and rows 0-191, regardless of the resolution in use:

```
line number  PSET (  column,     row,     colour  )
                     0 to 255   0 to 191   0 to 8
```

The following statements are used with graphics on the graphics screen:

CIRCLE	draw a circle (Chapter 14)
COLOR	set foreground and background colours (Chapter 13)
DRAW	sophisticated string driven line drawer (Chapter 23)
GET	save picture in array (Chapter 24)
LINE	draw a line (Chapter 14)
PAINT	colour in an area (Chapter 15)
PCLEAR	reserve memory for graphics (Chapter 15)
PCLS	clear screen to a colour (Chapter 13)
PCOPY	copy memory pages (Chapter 15)
PMODE	select resolution and memory page (Chapter 13)
PRESET	turn off graphics blob (Chapter 13)
PSET	turn on graphics blob (Chapter 13)
PUT	put picture from array to screen (Chapter 24)
SCREEN	select screen and colour set (Chapter 13)

15 Character Codes

Symbols and characters are stored as codes in the memory of the DRAGON Computer, each occupying one byte. Here is a table of the codes that can be used on the text screen — except that on the screen lower case letters are shown as upper case in reverse video (green or black). If you want to use upper and lower case, press SHIFT and 0 and then all the letter keys will work in upper and lower case.

The symbol corresponding to a code can be obtained as CHR$(code)

The code corresponding to a symbol is ASCII(symbol)

Codes 0-31 are not much used:

3 is 'Break'	10 is 'New line'	13 is 'enter'	
8 is ←	9 is →	10 is ↑	21 is →

None of these will print anything.

All the codes from 96 to 127 are lower case versions of the codes 64-95.

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
32	SPACE	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	−	69	E	93	]	117	u
46	.	70	F	94	↑	118	v
47	/	71	G	95	←	119	w
48	0	72	H	96	@	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	[
52	4	76	L	100	d	124	\
53	5	77	M	101	e	125	]
54	6	78	N	102	f	126	↑
55	7	79	O	103	g	127	←

Codes 128 to 255 represent coloured graphics symbols for the text screen. Refer to Appendix Section 13.

16 Sounds and Music

The DRAGON computer contains a tone generator, accessed by the SOUND statement.

```
line number  SOUND (  tone,      duration  )  (Chapter 10)
                      1 to255   1 to255
```

The musical notes obtained by these tones are not exactly in tune, and are:

Note	Bottom Octave	Middle Octave	High Octave	Higher Octave	Top Notes
C		89	176	218	239
C#		99	180	221	241
D		108	185	223	242
D#		117	189	225	243
E		125	193	227	244
F	5	133	197	229	
F#	19	140	200	231	
G	32	147	204	232	
G#	45	153	207	234	
A	58	159	210	236	
A#	69	165	213	237	
B	78	170	216	238	

Music is generated by the PLAY statement which can control volume and tempo more gradually — refer to Chapter 23.

17 Error Messages

The following error messages may appear when you run a program — correct the error (not always easy) and run the program again. A message will be similar to

```
?SN ERROR IN 50
```

which tells you that the error whose code is SN occured in line 50.

Error Code	Meaning
/0	You tried to divide a value by zero.
AO	You tried to OPEN a file that was already OPEN.
BS	You tried to use a subscript too large for the array.
CN	Sorry, program can't CONT.
DD	You already DIMed an array in this DIM statement.
DN	No such device number in an Input/Output statement.
DS	Computer is loading a program and found a direct statement — ie no line no.
FC	Illegal function call — the argument is out of range, eg RND(0) isn't allowed.
FD	You tried to read a number from a file and the computer found a string (or vice versa).
FM	You tried to INPUT from a file opened for output (or vice versa).
ID	This statement (INPUT or DEF FN) can't be used in direct mode.
IE	You've tried to read data past the end of the file.
IO	Input/Output Error. Computer can't understand the data — you may have started in the middle of a record or the tape or disc may be bad. If you're doing CLOAD or SKIPF, try again.
LS	A string is too long. Strings can only be 255 characters. Too much concatenation can cause this.
NF	There wasn't a FOR statement for this NEXT, or your NEXT is out of order.
NO	Sorry, can't use this file until you OPEN it.
OD	Out of data — you have tried to READ more items than your DATA list contains.
OM	Out of memory — you've used it all up. Try to use less for graphics (ie PCLEAR 1 or 2).
OS	Out of string space. Use CLEAR to reserve more.
OV	Overflow. A value is too big for the computer.
RG	You said RETURN, but there wasn't a GOSUB, sir!
SN	Syntax error. Your statement is wrong — spelling, punctuation, wrong number of brackets or whatever.
ST	This string operation is too complicated.
TM	Type mismatch. You have tried to use a number where a string is wanted or vice versa.
UL	You asked the computer to GOTO or GOSUB or IF. . . THEN to a line number that isn't there.

Index