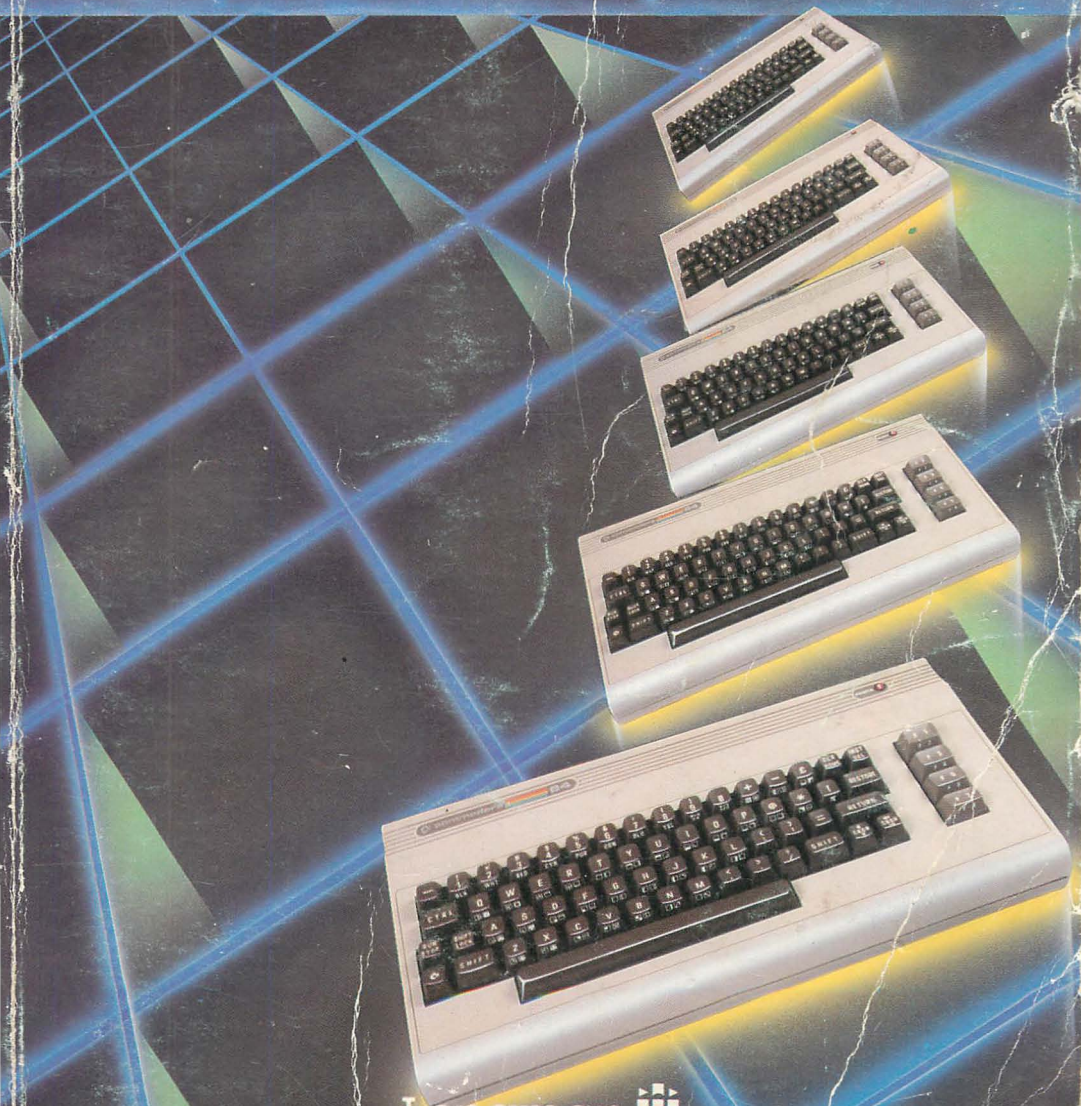
Introducing your Commodore 64

PK McBride



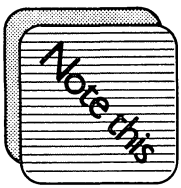
Longman

Introducing
your Commodore 64

Introducing your Commodore 64

PK McBride

Longman



This book is intended to help you understand how your Commodore 64 works, and how to get the best from it.

At various points you will be presented with programs illustrating features new to you, but these are all clearly explained in the text. Type in the programs very carefully, comparing each character with the character in the book. Remember to press the RETURN key at the end of each line, and type RUN and press RETURN to set the program running.

Longman Group Limited
Longman House, Burnt Mill, Harlow,
Essex CM20 2JE, England
and Associated Companies throughout the world.

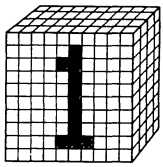
First published 1984

ISBN 0 582 91603 8

Printed in UK by Parkway Illustrated Press, Abingdon

Designed, illustrated and edited by Contract Books, London

Contents



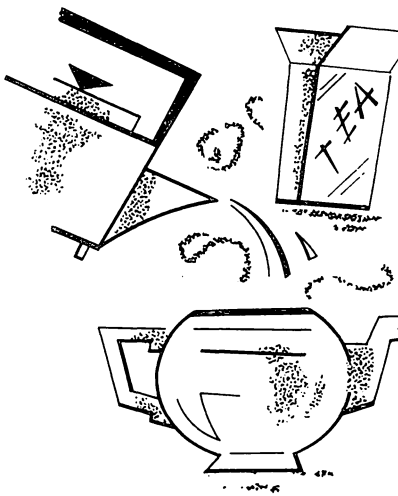
Programming

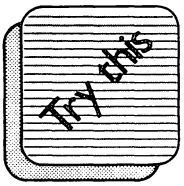
What is a

This book will show you how to write computer programs using your Commodore 64, but what is a program, and why do you need one? The simplest answer is that a program tells a computer to do something, and you need it because without it the machine can do only very simple things.

If you think of each of the computer's commands (the things it can do) as a brick, then writing a program is like building a house. That program-house can be almost any shape or size that you like. It's how you put the bricks together that counts.

Computers are very logical; some, like the Commodore 64, have large memories, but all of them are basically stupid. They will do exactly what they are told to do, without using any common sense or imagination at all. When you write a program to make your computer do something, you must set out your instructions simply and clearly, and in precisely the right order, because the machine will always do what you say, and not what you mean.



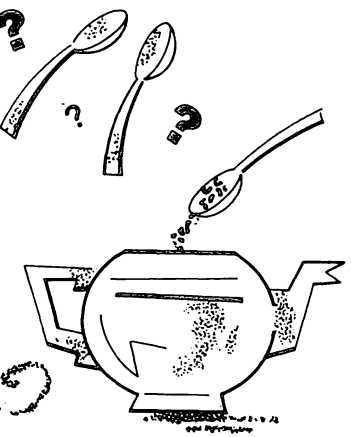


Here's an exercise to get you starting to think like a programmer. Take a simple job that you do quite regularly and do not really need to think about – making a cup of tea, for example – and break it down into the set of actions and decisions that make up the whole job.

program?

How do you make a cup of tea?

1 Find the kettle.
2 Fill it with water.
3 Plug it in and turn it on.
4 Find the tea pot.
5 Find the tea.
6 Work out how many spoonfuls of tea you need. One for each person and one for the pot.
7 Put the right amount of tea in the pot.
8 When the kettle has boiled, pour the boiling water into the teapot.
9 Shout 'Tea up!'

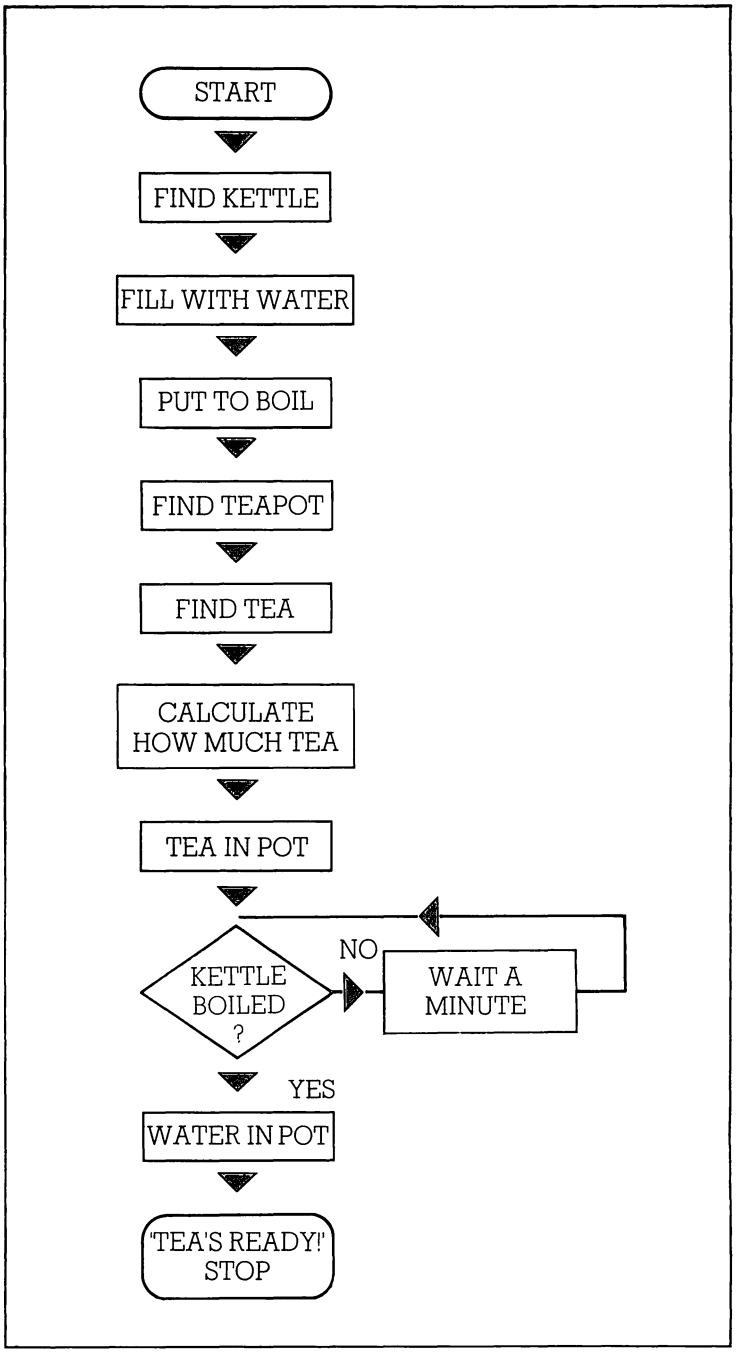


That's how I make tea – although not everybody likes it the way I make it.

How do you make tea? Work it out step by step, and write it all down. This is your tea-making program.

Your Commodore 64 won't make you any tea, no matter how carefully you program it, but it will do many other interesting things. What you have to do is work out exactly what it has to do, and type it all in, in carefully numbered stages.

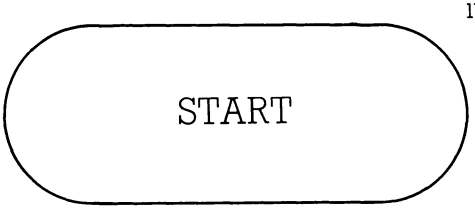
Follow the arrows



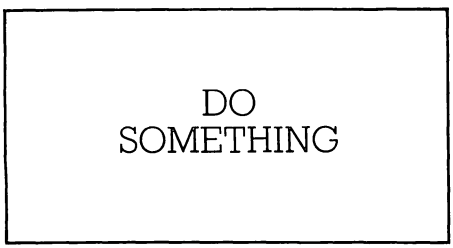
That construction of boxes and lines on the opposite page is known as a FLOWCHART. Follow the arrows from box to box. If you come to a diamond shape, then answer the question in it. Follow the arrow along the path that agrees with your answer, and carry on from there.

Flowcharts are a very useful way of planning a program. They make you work out what you mean, and that is the key to successful programming. You will meet a lot more of them as you work through this book, so let's have a look at the symbols now, to see what they all stand for.

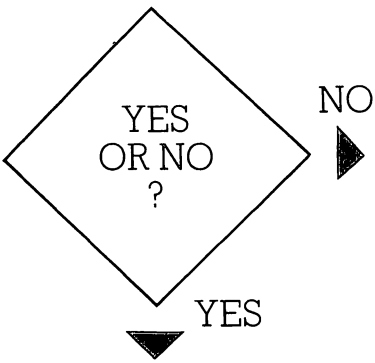
Oval shapes are used at the start and end of a program. You don't write the start instruction in the program. You start it from outside the program with the word RUN.



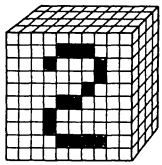
Rectangular boxes hold instructions.



Diamond shapes are for questions. When the computer comes to this part of a program it will branch off one way or the other, depending upon the answer it gets.



Arrows show the flow of the program. It is usual to write the flowchart from the top to the bottom of a page.



Printing

THE KEY TO BETTER PRIN

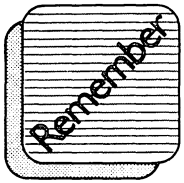
The easiest way to put anything on the screen is to use the PRINT command. Type in PRINT and then the quote marks (") – SHIFT and 2.

Now type in any characters you would like to see printed. As you work your way around the keyboard, you might like to explore what the keys can do. Those letter keys will produce more than just letters. Hold the Commodore key C= down and press a letter key. This gives you the graphics character on the front left of the key. Hold the SHIFT key down while you are typing and you get the graphic on the front right.

When you have typed enough – or when the machine cannot take any more (two full lines is your limit), type the quote marks again.

The PRINT instruction is ready, and there on the screen, but it won't be carried out until you press RETURN . So press RETURN and see what happens.

Do it again and explore the keyboard.



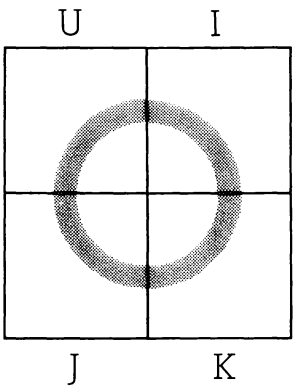
Press RETURN after each command and at the end of each program line.

TING

Press the Commodore and the SHIFT keys together and look at the screen. Everything changes! This is because your 64 has two character sets, not just one. Use the Lower Case mode when you want to print small and capital letters and the Commodore graphics. Use the Upper Case mode when you want to use all of the graphics characters.

If you have got lower case letters on your screen now, then press Commodore and SHIFT together, and push the keyboard back into Upper Case mode. We want those graphics.

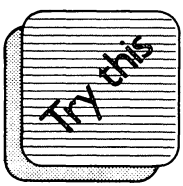
Here is your first program. It will print a circle on the screen. You can see the four graphics that make up the shape on the keys U,I,J and K. The circle will take two lines on the screen, so we use two PRINT instructions in a program. If it only needed one instruction, we could do it directly.



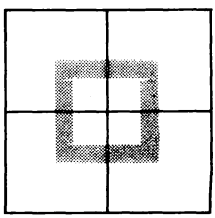
The instructions are numbered, so the 64 does them in the right order.

```
1 PRINT "◪◱"
2 PRINT "◹◪"
```

Type in each line, and press RETURN at the end of each line. Check that you have done it correctly, and you are ready to start. Type RUN (no line number) and RETURN .



Write a program to print the little square shown here.



GRAPHICS

You can produce some very good pictures using those graphics on the keys, if you take enough time and trouble. The examples here are rather crude, to keep the programs simple.

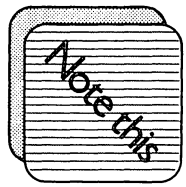
But your screen may be getting cluttered up. You have probably got program lines and bits of printing all over it. Wipe it clean by holding down **SHIFT** and pressing **CLR/HOME** . This clears the screen, and takes the cursor – the square that shows you where you are – back to its HOME position, the top left corner. If you press **CLR/HOME** without the **SHIFT** the cursor goes home, but the screen isn't cleared.

You have cleared the screen, but you haven't lost your program. Type in LIST, and **RETURN** . The program will reappear.

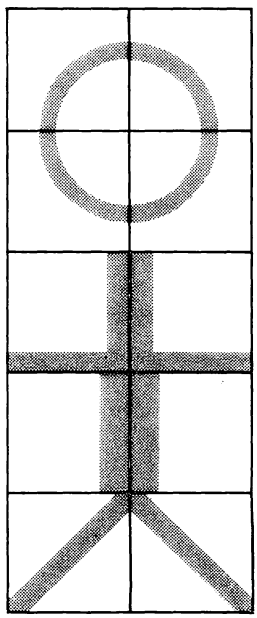
The CLR/HOME command can be included in your program, so that the printing is done on a nice clean screen. It must come before the PRINT lines, of course, and there you have a problem. How do you get a line before number 1? The answer is you can't. Always leave room for extra lines at the start of your programs, in case you want to add something afterwards. You should also leave spaces between the lines as well, just in case of any future change. To do this, start your line numbers from 10 (some people start from 100, so there is even more space), and number them in 10's.

```
10 PRINT....
20 PRINT....
```

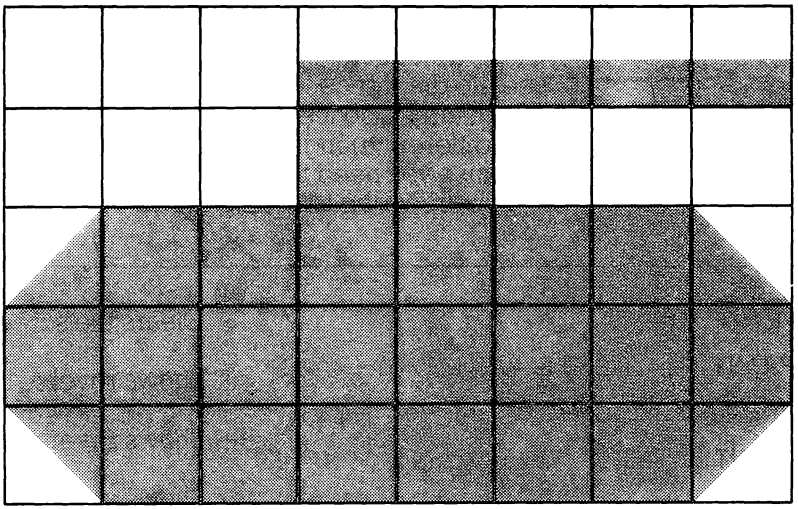
The 64 doesn't mind what numbers you use for the lines. It works from the lowest number to the highest, and if there are gaps between the numbers, it jumps them.



All graphics characters are shown in boxes like this ◲.



PRINT



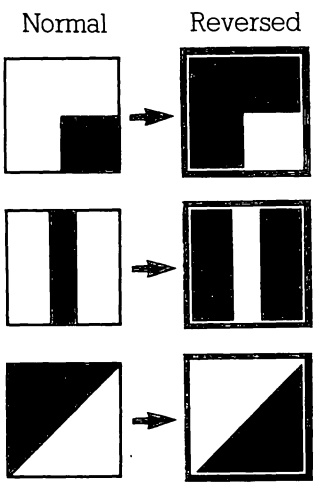
Program

Here's the program to PRINT the matchstick man. The graphics are all SHIFTed characters, except for those in line 50. The character in line 10 is got by pressing SHIFT and CLR/HOME. (It clears the screen and puts the cursor back to the HOME position, top left corner.) The characters in line 50 are Commodore key and L and J respectively.

```
10 PRINT "♥"
20 PRINT "◢◣"
30 PRINT "◥◤"
40 PRINT "▗▖"
50 PRINT "▝▘"
60 PRINT "▨▧"
```

Project

Type it in carefully (RETURN after each line) and RUN it. What do you think? Write a program of your own to PRINT a drawing. First, type NEW (and RETURN) to get rid of the original program.



So far we have only been using characters from half of the graphics set. Each of the graphics can be printed REVERSED, which produces quite different pictures.

At this point you must find a new key CTRL – control. This controls the commands on the front of the number keys. There are colour controls there as well, and we will get to them shortly. Let's sort out REVERSE first.

Type PRINT ". . then press CTRL and 9/REV ON . Now key in some of the graphics. Close with quote marks, and RETURN. You can see the difference that REVERSE makes. For a start it turns a space into a block, and gives us four different triangles, which is just as well, because you need them in the program below, which draws the tank on page 13.

```
10 PRINT "▼"
20 PRINT "_ _ _▄▄▄▄▄▄"
30 PRINT "_ _ _R_ _⊟"
40 PRINT "R◤_ _ _ _ _ _◥"
50 PRINT "R_ _ _ _ _ _ _ _"
60 PRINT "◥R_ _ _ _ _ _⊟◤"
```

▼ Clears the screen

R Shows reverse on

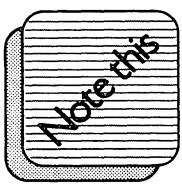
⊟ Shows reverse off

_ Shows a space

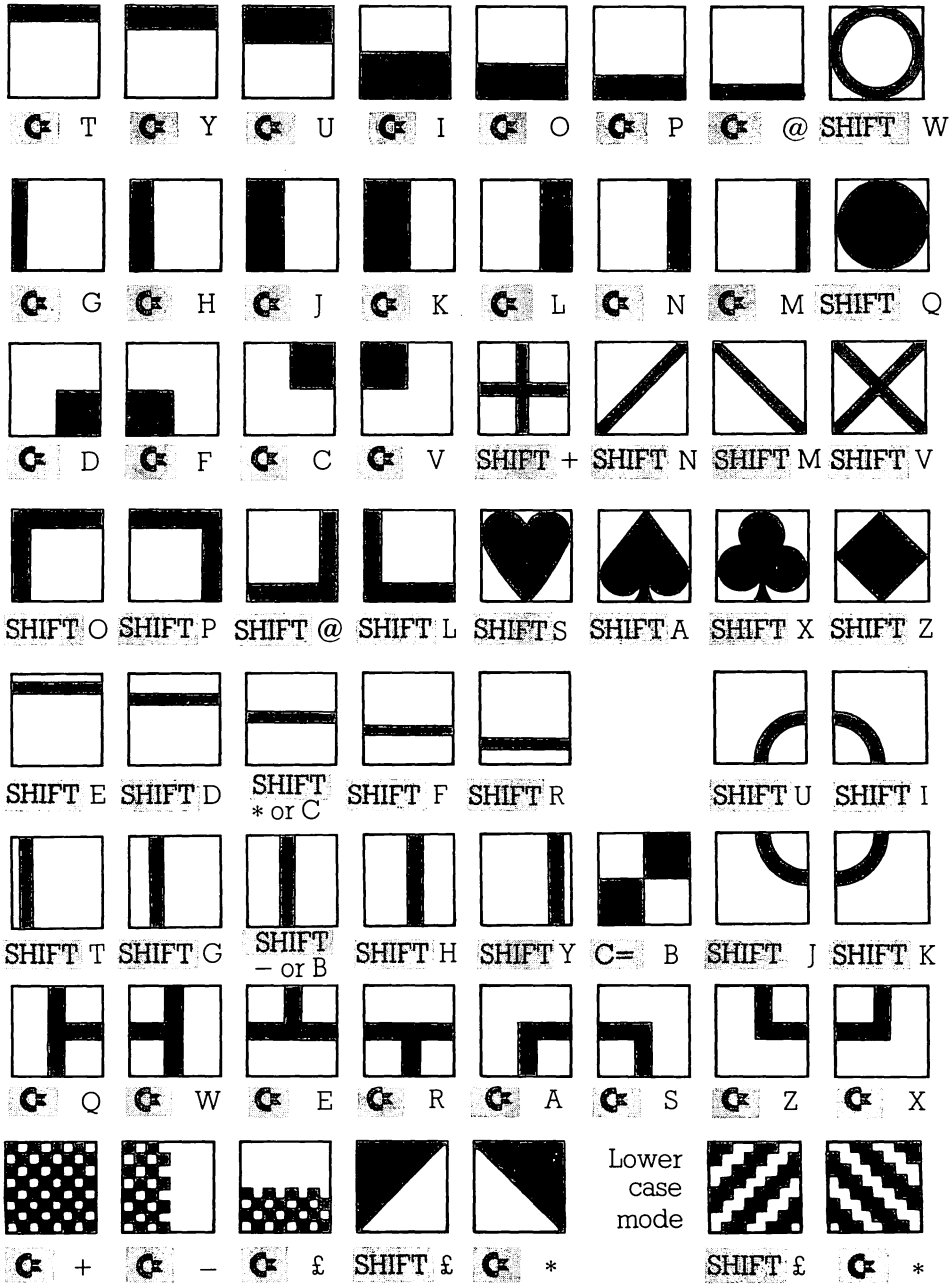
◤ Graphics character

The GRAPHICS FINDER is there to help you to find your way around the graphics set. The ones shown are normal (reverse off), and all but two of them are from the Upper Case Mode.

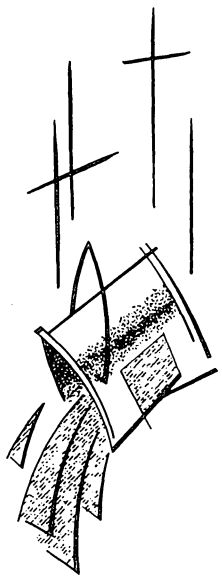
In Lower Case Mode the SHIFT key gives you capital letters, while the C= key still gives you left-side graphics.



The characters after a REVERSE ON command will *not* appear reversed in the program line. They are only reversed when you print.


C= T
C= Y
C= U
C= I
C= O
C= P
C= @
SHIFT W
C= G
C= H
C= J
C= K
C= L
C= N
C= M
SHIFT Q
C= D
C= F
C= C
C= V
SHIFT +
SHIFT N
SHIFT M
SHIFT V
SHIFT O
SHIFT P
SHIFT @
SHIFT L
SHIFT S
SHIFT A
SHIFT X
SHIFT Z
SHIFT E
SHIFT D
SHIFT * or C
SHIFT F
SHIFT R
SHIFT U
SHIFT I
SHIFT T
SHIFT G
SHIFT – or B
SHIFT H
SHIFT Y
C= B
SHIFT J
SHIFT K
C= Q
C= W
C= E
C= R
C= A
C= S
C= Z
C= X
C= +
C= –
C= £
SHIFT £
C= *
Lower case mode
SHIFT £
C= *

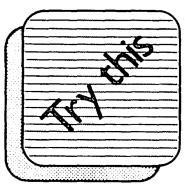
A splash of colour



If you are getting tired of writing in pale blue on a dark blue screen, change the colour of your ink by holding down CTRL and hitting one of the numbered keys (from 1–8). Type in a PRINT 'something' command now, and the 'something' will appear in the colour of your choice. Work your way through the colours, typing something in each. You will notice that some colours do not work well on a blue screen.

The CTRL key gets you the first eight colours. There are a further eight colours, and to get these, hold down the Commodore key and press a number. The full list of colours is shown opposite.

Those symbols are there for a purpose. You can change the colour of your printing, inside the PRINT line, just as you can change from normal to reversed characters.



Type in NEW (and RETURN) to clear any previous program. Now type:

CTRL and 1

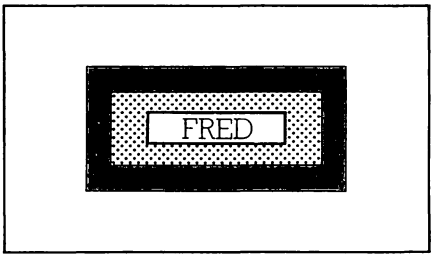
```
10 PRINT " ■ BLACK E WHITE"
```

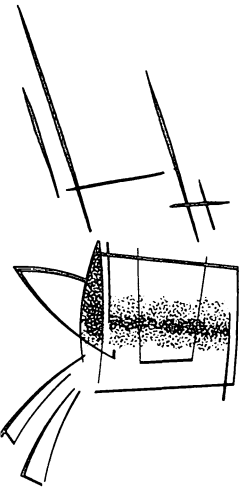
CTRL and 2

```
20 PRINT "R ■ BLACK E WHITE"
```

REVERSE ON

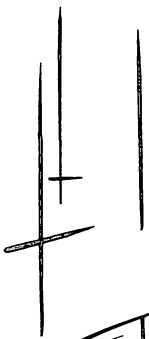
Add more lines to do the same for the other colours.

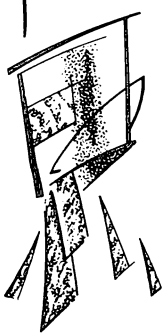




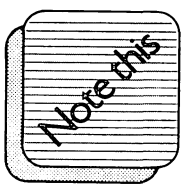
Project

Now that you have found where they all are, and what they look like, you can try to write a program to print your name in a coloured surround. Fred, here, has simply used the colour blocks you get from Reversed Space. You might like to use graphics characters to produce a fancier border.



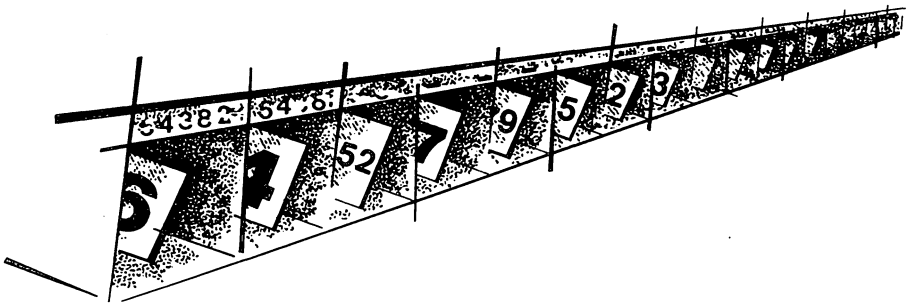


CTRL	KEY		CTRL	KEY	
	Black	1		Orange	
E	White	2		Brown	
£	Red	3		Pink	
	Cyan	4		Dark grey	
	Purple	5		Mid-grey	
	Green	6		Pale green	
	Blue	7		Pale blue	
π	Yellow	8		Pale grey	

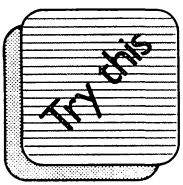


When you use a colour command inside a PRINT line, the colour of the ink does not change until the characters are printed. The program line itself remains in the current ink colour.

Take a po



Inside the Commodore 64 there are 65536 places where it can store information. That's what is meant by 64k. 64 kilobytes of memory gives you 64 × 1024 different stores. Most of these are left empty, for your use, but some are reserved for the computer. It has to store a lot of information to be able to manage your programs. For example, in store number 53280, the 64 keeps track of the colour number for the border, and in the store next door, 53281, it remembers the colour of the screen. You can change these numbers, if you like, by the use of a special command – POKE. This puts a number directly into a store.




```
POKE 53280,2          (RETURN)
```

Change the Border colour again by POKEing another number into store 53280. You can see the list of the colour codes at the top of page 19.

To change the colour of the screen, POKE the next store – 53281.

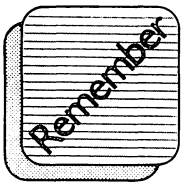
```
POKE 53281,10
```

will give you a pink screen.

ke at it!

Colour codes

0	Black	4	Purple	8	Orange	12	Mid-grey
1	White	5	Green	9	Brown	13	Pale green
2	Red	6	Blue	10	Pink	14	Pale blue
3	Cyan	7	Yellow	11	Dark grey	15	Pale grey



The screen colour you get when you switch on is Blue (6).

The next time you write a picture printing program, make these your first lines.

```
10  POKE 53280,...  (colour for Border)
20  POKE 53281,...  (colour for screen)
30  PRINT "■..."    (colour for ink)
```

CLR/HOME

Take a break

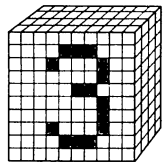
END and STOP will both stop the program. You might think they are both the same, but they are not. You can start again after a STOP, but the END is the end.

```
10  PRINT "HELLO"
20  STOP
30  PRINT "GOODBYE"
40  END
```

Run this, and the 64 prints 'HELLO', and then 'BREAK IN LINE 20'. Type in CONT, for CONTinue and RETURN. The program restarts, and you will see 'GOODBYE'. Now type in CONT. Nothing happens.

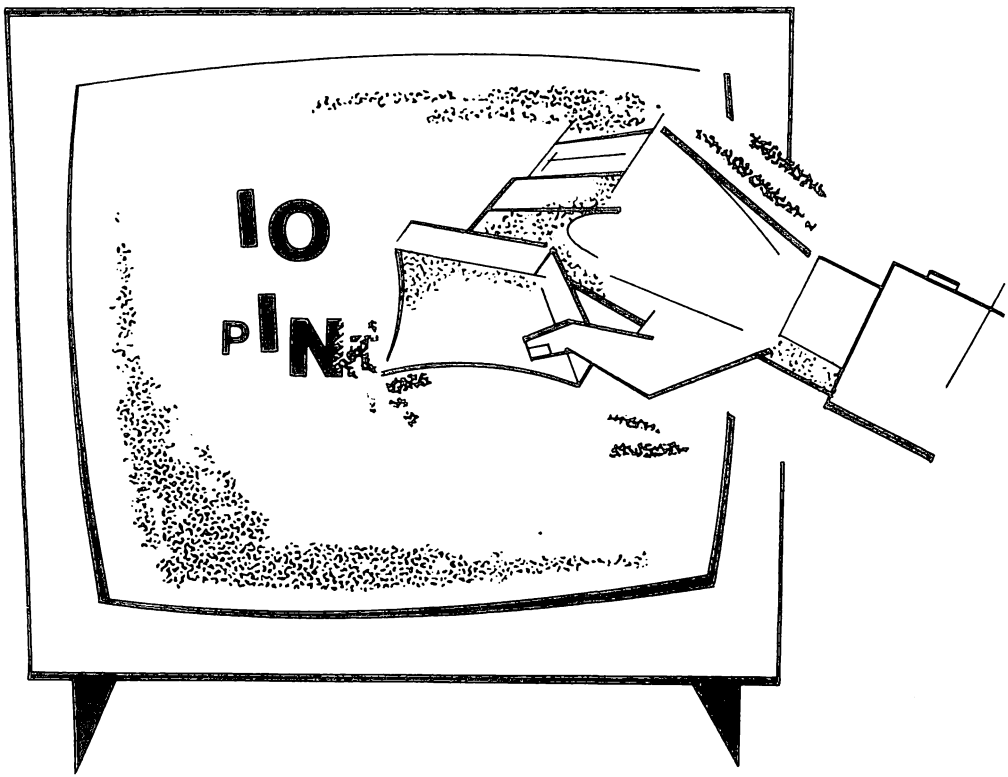
The STOP command is useful when you are working out your program. Write in a STOP after each routine that you want to look at more closely. When the program stops, you can examine the screen layout, or get it to print out any variables (so that you know what it's thinking!) or even LIST. As long as you don't make any changes, then CONT will restart from the point where it stopped. If you do correct an error, or add a line, then the CONTinue command won't work. When all is well with the routines, take out the STOP lines.

END is a much neater way of ending a program. It doesn't print a BREAK message.



Moving the cursor

EDIT IT!



Correcting errors is no problem as the 64 has an ON-SCREEN EDITOR. This means that you can rub out your mistakes, and rewrite the words, just as if you were working with pencil and paper – and a rubber. It also means more than that. Type a short program with lots of mistakes in it. Make it something like this:

```
10  PINT "Whhop
20  POKER 5321,0
```

Spot the mistakes!

Delete

Rubbing out is easy, so do that first. Move the cursor from wherever it is now to the space just after the R in POKER. To move the cursor, use the keys with the arrows on. The Up and Down key will normally move the cursor down, but if you hold SHIFT at the same time, it goes up. Similarly, the Left/Right key normally goes to the right, but SHIFTed, it goes left.

In place? You should be on the square after the one you want to delete. Now press INST/DEL . The DEL is short for DELete.

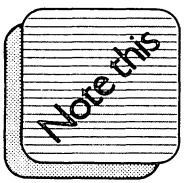
Insert

That same key can also be used to INSerT extra characters. There is an '8' missing from the number in line 20. Move the cursor along until it is on top of the square to the right of the missing character. Here it should be over the '1', as the number that should be there is '53281'. Now press SHIFT and INST/DEL again. All the characters to the right of the cursor, and the one under it, shuffle one place to the right. Type in the missing number.

When the line is as it should be, press RETURN, and the corrected line replaces the old one.

Line 10 has several mistakes. The INST/DEL key will allow you to clear up most of these. You also need to add an 's' and quotes at the end of the line. That's easy. Just put the cursor where you want to start typing, and type.

One last point. If a line, or part of a line is very badly mistyped, or you want to change a whole chunk of it, type over the part you don't like.



If you want to add a colour command to a line you have already written, then you must create a space for it with INST/DEL. You cannot simply type it over something else.

PRINT

You can include cursor moving commands in your PRINT lines, just as you can include colour or reverse commands. It is sometimes easier and neater to produce small blocks of print by using a single line with cursor movers in it, than using several different lines.

Try this

```
PRINT "[]]S[]]P[]]A[]]C[]]E[]]D[]]O[]]U[]]T  "
```

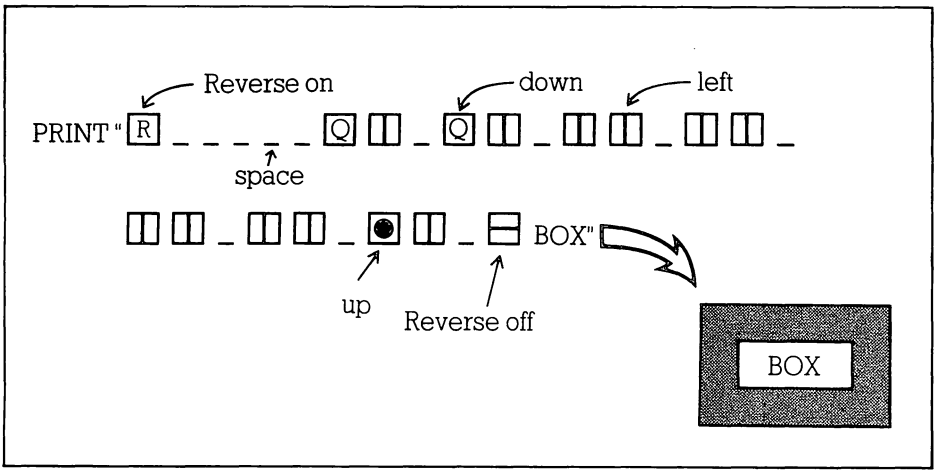
(cursor right)

That may look to you as if the 'cursor right' does the same job as a space. It doesn't. SPACE actually prints a space, rubbing out anything that was there before. Cursor right jumps over anything that may be there. You can prove this easily enough.

Go back up to the PRINT line and DELete the first cursor right. It will now print the same message, but starting a space to the left. Press RETURN, and the message will appear on the same line as the original, but to the left. You should see this:

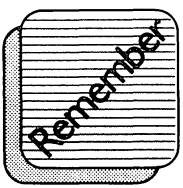
```
"SSPPAACCEEDDOOUUTT"
```

Here's a PRINT line that uses the other cursor movers to produce the BOX. Type it in carefully, and see if your BOX looks like the one here. You can colour the box by including a colour command at the start of the line.



Anywhere you like

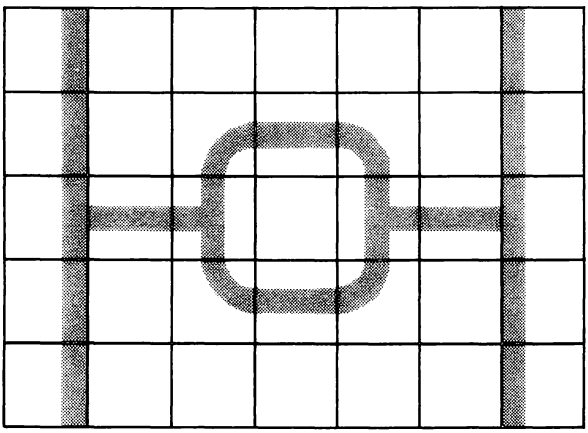
You can put the box anywhere you like on the screen by including cursor movers in the print line before the other characters. Try different combinations of down and right moves, and see what it takes to make the box appear in the centre of the screen.

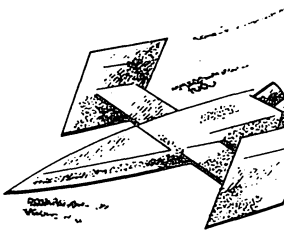


Cursor move commands are just like colour commands when it comes to editing lines. You cannot type them over other characters – try it and see what happens. You must INSerT them.

The box was quite small, and could be done comfortably in one PRINT line. If you want a larger picture, it is going to take several separate lines, as you cannot have a program line of more than 80 characters – 2 lines of type on the screen. Normally, after a PRINT command, the cursor moves on, to the start of the next screen line. This could be awkward if you were trying to produce a block picture. It would mean that your next line would need a set of cursor movers at the start to shuffle the cursor across to the right place. Fortunately it can be avoided. Finish your PRINT line with a semicolon (;) and the PRINT position stays wherever that line leaves it.

Tie fighters



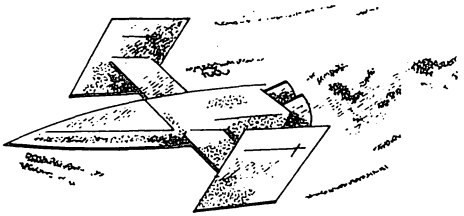


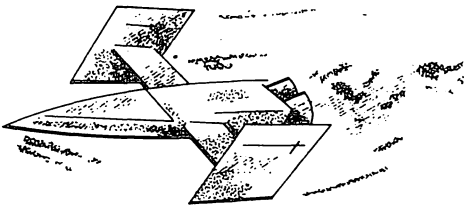
This is how to get the picture of the Tie fighter.

Program

```
10 PRINT "♥QQQQQQ]]]]]]]";
20 PRINT "  Q                Q                ";
30 PRINT "  Q                Q                ";
40 PRINT "  Q                Q                ";
50 PRINT "  Q                Q                ";
60 PRINT "                   "
```

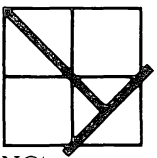
Type it in and try it. Count the cursor movers carefully. There should be one down move and seven left moves after the graphics in each of lines 20 to 50.



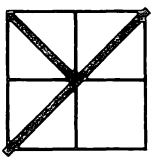


Project

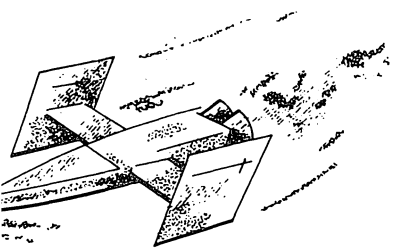
Now design your own spaceship or use the one below and work out a set of PRINT lines to put it on the screen. If you are designing your own, use squared paper, and start with a faint pencil sketch of the outline that you want. Check through the available graphics to see how close you can get to your sketch. Sometimes you will find that it helps to move the picture slightly on the grid. Here, for example, you want two diagonals to join. The junction on the left is impossible, but a small change in the position of the diagonals allows them to meet as you wanted.

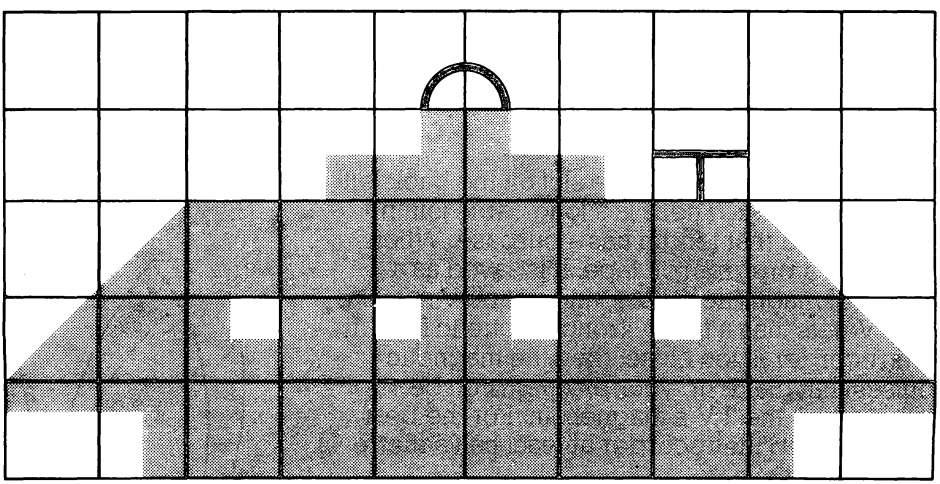


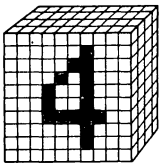
NO!



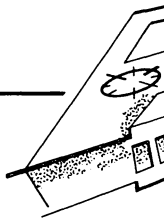
O K



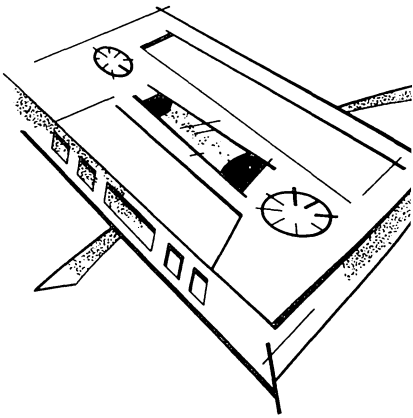




Save your effort



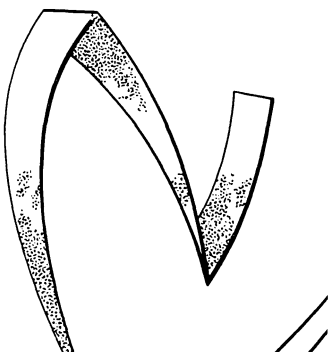
Using a tape recorder

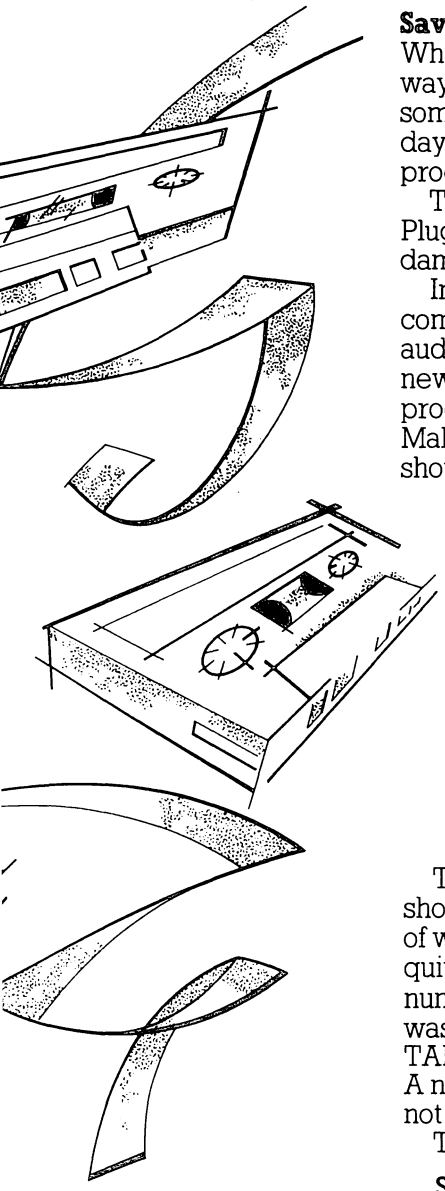


You should be getting to the point now where you are producing some pleasing pictures, and it's a shame to lose all the hard work and effort that has gone into them. So, it's about time to get to grips with the tape recorder. If you haven't got a cassette player – the Commodore C2N unit – start saving up for one now.

Set up

The C2N unit plugs into the slot on the back left of the computer. There is a long cable joining the two, and that length has a purpose. When you save a program on tape, it is saved as a series of magnetic pulses, and your TV set, like all TV sets, has a fairly strong magnetic field. If the recorder is too close, then the magnetic pulses that make up your program can be affected by the TV's magnetism. For the best results, keep the recorder at least half a metre from the set.





Save

When you have a program that is running the way you want it, and you want to go on to something new, or when you have to stop for the day, but would like to come back to the program later, then SAVE your program.

The C2N unit must be plugged in already. Plugging it in when the 64 is turned on can damage the computer.

Insert a cassette into the unit. The C15 computer tapes are ideal, but any good quality audio tape will do. Start at the beginning of a new tape, or wind on to a point past your other programs, if you have already saved some. Make a note of the tape counter number. This should be 000 on a new tape.

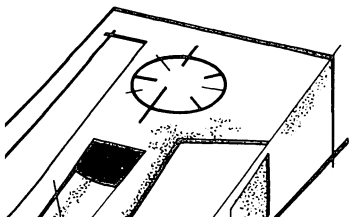
	Side A
000	TIE FIGHTER
020	SAUCER
035	JIM'S TANK
055	SUE'S HOUSE

Think up a name for your program. Keep it short and simple. It should instantly remind you of what the program is about. The 64 will be quite happy to save a program called 'Program number 1' – but you will have forgotten what it was in a week. 'TIE FIGHTER', 'SAUCER', 'JIM'S TANK', 'SUE'S HOUSE' are much better names. A name can be anything you like, as long as it is not more than 16 characters.

Type:

```
SAVE "TIE FIGHTER"
```

(or whatever) and RETURN



The 64 will print

```
PRESS RECORD & PLAY ON TAPE
```

Record
Do it. The red light on the C2N will glow to show that it is saving. The screen will blank, the same colour as the border, and the tape will turn.

When the program is saved, you will get the screen back, with a READY message on it.

Check
Normally, the C2N will save your program perfectly, but always check that it has. To do this, rewind the tape to the beginning of your program (find this using the tape counter numbers).

Type:

```
VERIFY "TIE FIGHTER"
```

(or whatever) and RETURN

The 64 now prints

```
PRESS PLAY ON TAPE
```

Do it. The screen blanks again, while the tape turns. The 64 is searching the tape for your program. Sometimes when it finds it, it will print:

```
FOUND TIE FIGHTER
```

When it prints a FOUND message, it will wait for you to tell it that this is the program you want. If it is, press the Commodore key, and it will go on to check the program. If it has found the wrong program – perhaps you wound the tape too far back – then press space, or one of the letter keys. It will look further to find another program.

Sometimes when it finds the program, the 64 will carry on and check it without waiting to ask you. This doesn't matter.

If all is well, and the program that it has found on the tape agrees with the one it has in its memory, you will get an OK report. Your program is safe on tape. Write its name and tape counter number on the cassette label. If you don't get the OK, then rewind and start again.

Possible problems

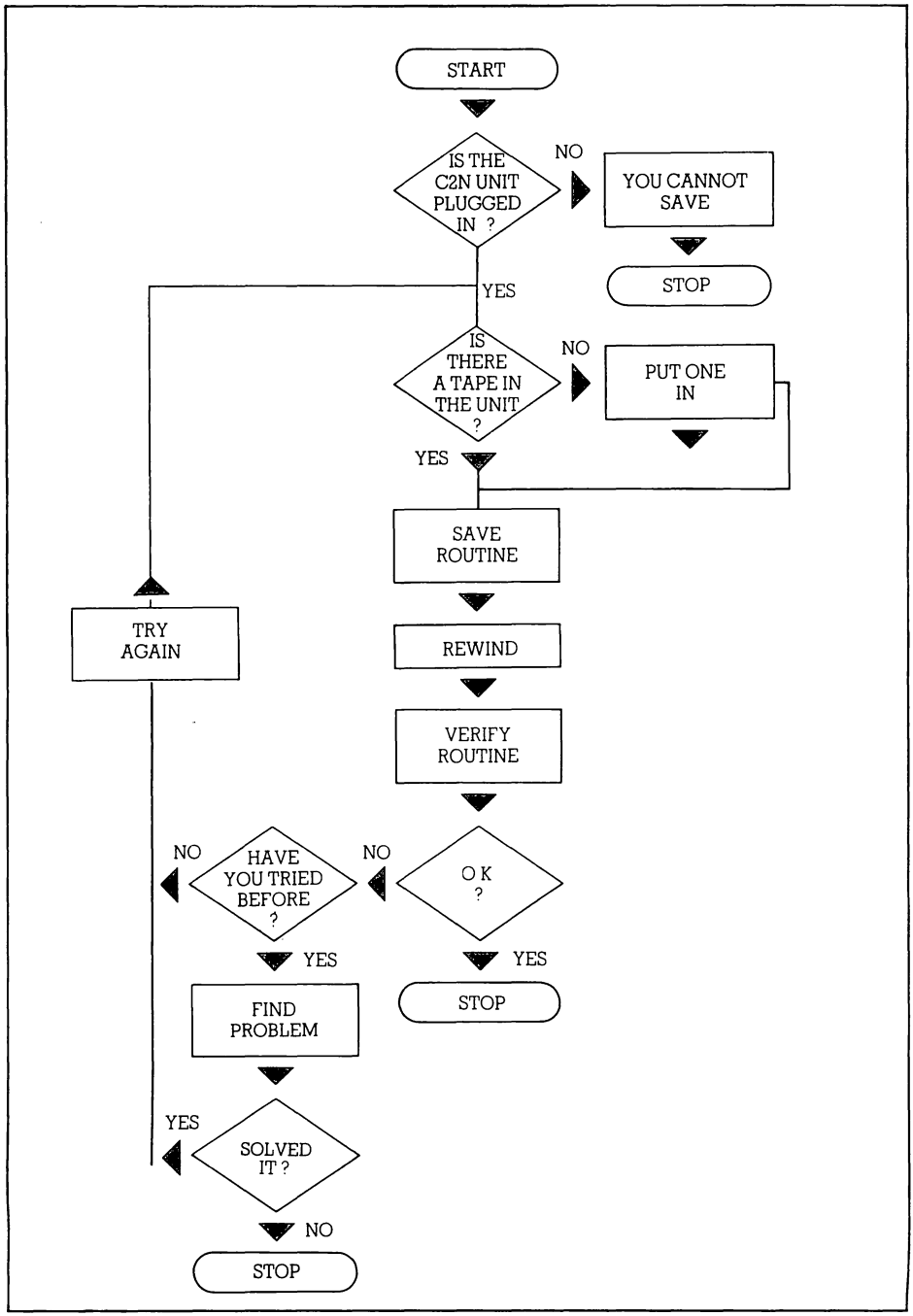
1 The connection isn't plugged in properly – make sure it is next time.
2 The tape is of poor quality, or you recorded again and again over the same piece of tape – try a new tape.
3 The C2N needs attention. The recording and reading head can sometimes move out of its proper alignment – have it checked by your Commodore dealer.

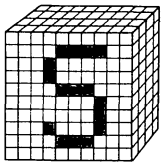
Load
To LOAD the program back into the 64 another day, follow the same routine as for VERIFY, but start with the command:

```
LOAD "TIE FIGHTER"
```

(or whatever)

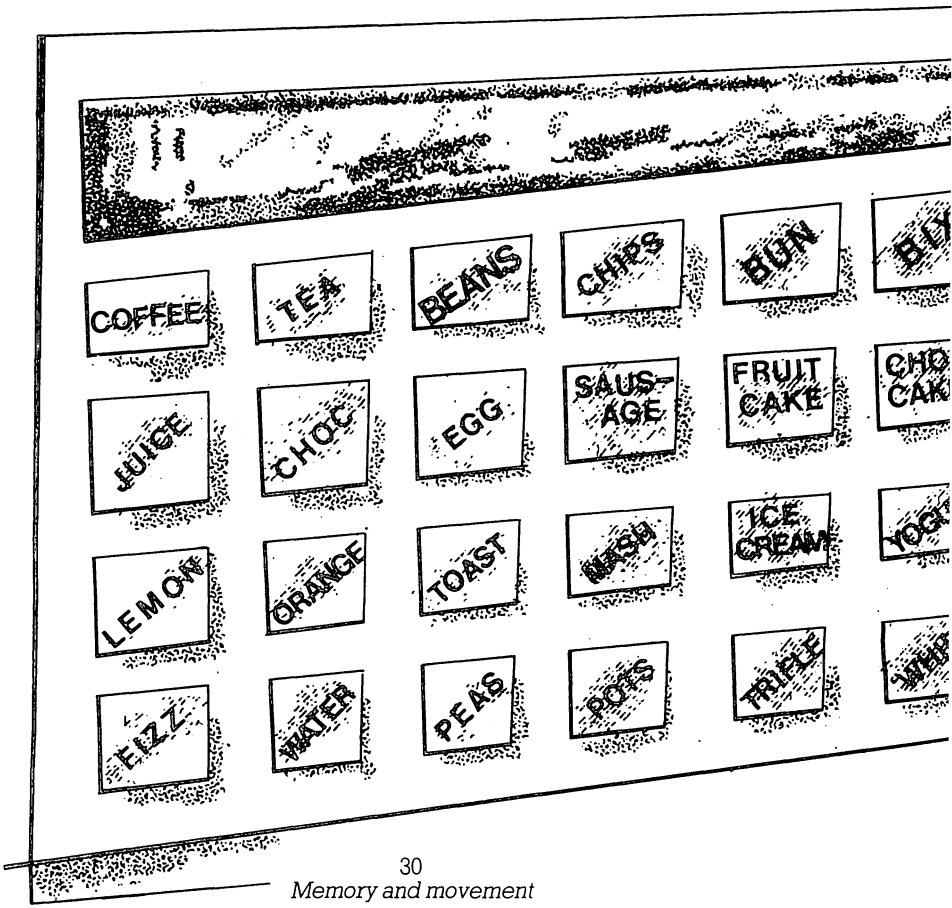
If you don't know where the program is on the tape, then start at the beginning. The 64 will tell you when it has found it.


START
IS THE C2N UNIT PLUGGED IN ?
NO
YOU CANNOT SAVE
STOP
YES
IS THERE A TAPE IN THE UNIT ?
NO
PUT ONE IN
YES
SAVE ROUTINE
REWIND
VERIFY ROUTINE
O K ?
NO
YES
STOP
HAVE YOU TRIED BEFORE ?
NO
YES
FIND PROBLEM
SOLVED IT ?
YES
NO
STOP
TRY AGAIN



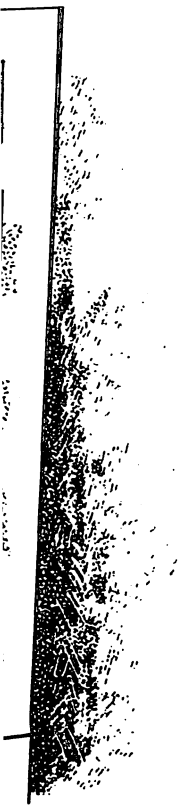
Memory and movement

USE YOUR



Some motorway service stations have recently introduced a new type of till, which you may have noticed. Instead of the usual keyboard with rows of numbers, these have a large grid with the squares marked 'Coffee', 'Tea', 'Fruit Juice', 'Egg & Chips', and so on. The assistant looks at your tray and presses the squares for the things that you have collected. As she presses the board, the prices flash up on the display, and the machine works out the total at the end.

MEMORY



How it works

How does it work? Obviously it is computerised and the machine has been told that coffee = 57p, tea = 45p, fruit juice = 50p, etc. The till is using its memory. Somewhere in its memory is a store labelled COFFEE, and containing the price (57), another labelled TEA, with 45 in it, and a great many more such stores.

These stores are known as VARIABLES, because what is in them can vary. The service station might decide to put up the price of its drinks, and it can do this by changing the variables. It will, no doubt, have a program that allows someone to type in new prices for all the things that it sells. A section of that program might look like this:

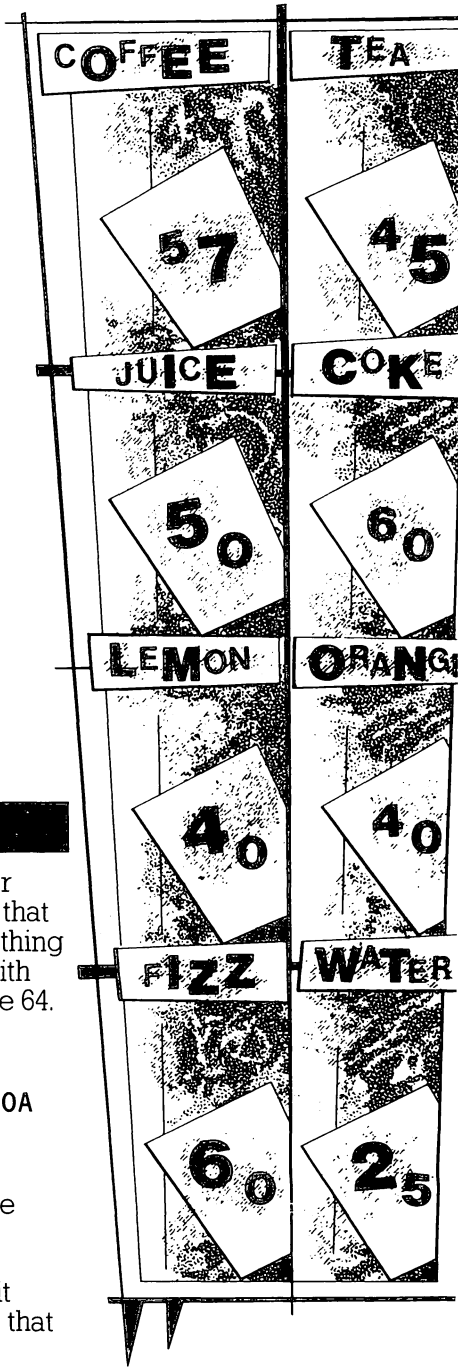
```
9000  INPUT COFFEE
9010  PRINT "COFFEE PRICE =";COFFEE
9020  INPUT TEA
9030  PRINT "TEA PRICE=";TEA
```

New word

There is a new command there that you have not yet met – INPUT. This tells the 64 to wait for someone to type something in. COFFEE and TEA are the labels given to the stores in which it will remember the numbers that are entered.

Key the program in and run it. Don't worry about the line numbers being so big. The 64 doesn't mind what line numbers you use, as long as they are not over 65535. If you prefer, change the line numbers to 10, 20, 30, and 40. It's the order that counts.

When the program runs, you will see a ? and a flashing cursor. The 64 is waiting for an INPUT. Type in a number, and press RETURN. The new coffee price will be printed. INPUT another number for the tea price.



Project

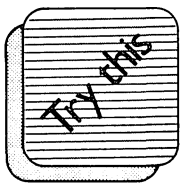
Add more lines to collect the prices of other foodstuffs at the service station. The names that you use for the variables can be almost anything you like, but don't start any two variables with the same two letters. This would confuse the 64. You can prove this by typing in:

```
9040  INPUT COCOA
9050  PRINT "COCOA PRICE =";COCOA
9060  PRINT "COFFEE PRICE
      =";COFFEE
```

Run it now, and enter different prices for the drinks. Notice what happens to your coffee price at the end.

Whan the 64 looks at the variable stores it only actually looks at the first two letters, so that here it sees CO(COA) and CO(FFEE).

The COFFEE and TEA variables are both examples of NUMBER VARIABLES – stores in which numbers are kept. The computer has a second type of store, STRING VARIABLES, in which characters are stored. While number stores are fixed in size – each store holds only one number – string stores can hold single characters, or whole sets of characters. If you can put quotes around it and print it on the screen, then you can put it into a string variable.




```
10  INPUT NAME$
20  PRINT NAME$
```

Run it, and when you see the flashing cursor, enter your name, or any name. Run it again and enter anything you like from the keyboard – letters, graphics, or numbers.

Storage space to let

INPUT is not the only command that puts data into memory stores. You can also give values to variables from within the program, by using the LET command.

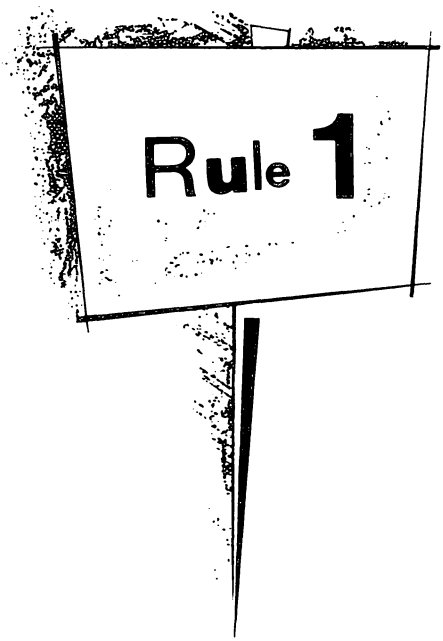
```
10  LET NAME$="FRED"
20  LET WEIGHT=55
30  PRINT NAME$;" WEIGHS";WEIGHT;
    "KILOS."
```

Take care with your typing when you key in this example. The different items in that PRINT line are separated by semicolons (;).

WHO'S MINDIN

The answer is, the computer does all the work, but you are the manager. There are certain rules you must follow, but only a few.

Rule 1
You have already met this one. No two variables of the same type can start with the same two letters. This is because the 64 only looks at the first two letters. Your variable names can be longer – but that is for your benefit only.



Rule 2
You cannot use BASIC words as variable names, or in variable names. So if you wanted to store a number that was to be printed on the screen, you couldn't call its store "PRINT".

```
LET PRINT=4
```

would give you a SYNTAX ERROR report.

This isn't always obvious. For example, the service station 'Bacon & Eggs' store could not be called BACON, because this contains "ON", and ON is a BASIC word. You will find a (more or less) complete list of the 64's BASIC words on the card at the back of your User Manual. If you ever do get a SYNTAX ERROR report from a line containing a variable, then try a different name.

G THE STORE?

Rule 3
All variable names MUST start with a letter. The other characters in the name can be letters or numbers, but not graphics characters or symbols, except for % and $.

The computing computer

In the early days of computers the machines were built to do one thing – compute, that is, handle numbers. You will now get about as much number-crunching power in a pocket calculator as there was in the huge machines of thirty years ago.

The 64 can do all the things a calculator can do – and more. For straightforward arithmetic, enter the sums in the usual way, using these signs:

+	Addition
–	Subtraction
*	Multiplication
/	Division

```
PRINT 2+2
```

makes the computer print "4"

For powers use the form X ↑ n to raise X to the power of n. So, 4 ↑ 2 means $4^2 = 16$.

To find square roots, use the function SQR(). Try this:
PRINT SQR(64). You will see 8.

Cube roots, and other roots are a bit trickier. You have to use the power sign with a fraction.
$8 \uparrow 1/3 = \sqrt[3]{8} = 2$.

$81 \uparrow 1/4 = \sqrt[4]{81} = 3$. (3*3*3*3 = 81)

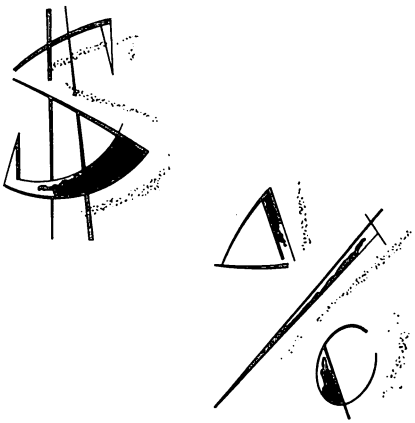
The 64 will happily handle any number up to 999999999 without fuss. When you get over this it starts to show the numbers using SCIENTIFIC NOTATION. Ask the 64 to PRINT 1000000000 and you will see this "1.0E+09". This means 1.0 × 1 billion (1000000000).

3.6E+13 would mean 3.6 × 10000000000000 (13 0's) = 36000000000000.

However the computer has its limits – it cannot cope with numbers of more than 39 digits.

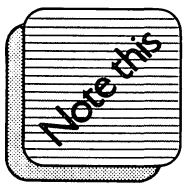
Types of variables

String Stores – which must end with $ – are essentially used for words, but any characters can be stored there, including graphics and numbers. LET N$="1" is a valid command. The N$ store now holds the character '1', though it has no number value.



Number stores are of two types. REAL variables can hold any type of number, whole or decimal, up to an astronomical size. REAL variables are labelled with the name only. INTEGER variables can store whole numbers only, up to a maximum size of 65535. These must have a percent sign (%) at the end of the name. LET NUM%= 999 is O.K. LET NUM%=99.9 would not work.

INTEGER variables take up only 2 bytes in memory, while REAL variables take 5. This can be useful with a very large program, or where you want to store great quantities of numbers. In the types of programs that we are working out in this book this advantage does not count very much, and using real variables saves us from having to remember the % sign all the time.



'LET' is not actually necessary. The line 'X=10' works just as well as 'LET X=10'. The 64 assumes you mean LET and writes it in for you. Including LET in the line helps to make the program list more readable, but miss it out if you want to save typing time.

If you ever forget to open a store, but then try to use it in the middle of a program, the 64 takes it in its stride. Suppose it met the line 'X=X+2', and you had never at any point told it what X was worth to start with. The 64 assumes that the store is to be opened with a value of 0. The result of that particular line would be that the X variable would now be worth 2. Likewise, if you typed 'PRINT Z$' without having said what Z$ was, it would create a store labelled Z$, leave it empty and print nothing.

The value of strings

If you have a program that requires the user to enter a number, and the user is the sort who might press the wrong sort of key by mistake, then you could have a problem.

```
INPUT "ANSWER PLEASE ";N
```

If he typed 'TEN', or '1 AND A BIT', the 64 would recognise that this was not a proper thing to be put into a number variable, and would print 'REDO FROM START'.

This won't matter if you don't mind your screen being messed up, but it would spoil a carefully designed screen.

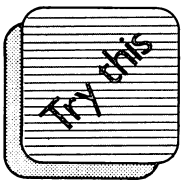
You can prevent this by taking the INPUT into a string variable.

```
INPUT "ANSWER PLEASE ";N$
```

Then use the VAL function to find out what, if anything, the string is worth. If the user has entered a numeral or a numeral followed by letters, then VAL(N$) will give the value of the numeral only. If the answer is anything else, then VAL(N$) will give a value of 0. Here's the complete crash-proofing routine.

```
1000  INPUT "ANSWER PLEASE ";N$
1010  IF VAL(N$)=0 THEN 1000
1020  LET N=VAL(N$)
```

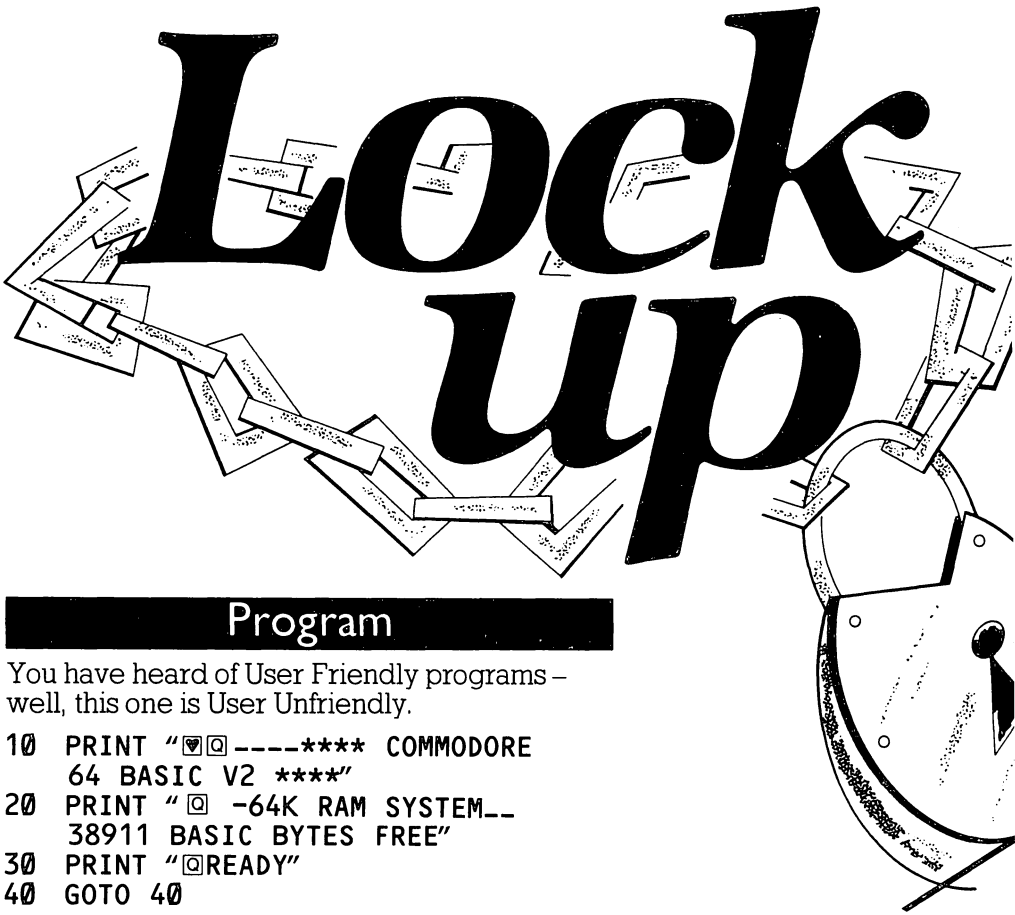
VAL turns strings into numbers. There's another function to turn numbers into strings —STR$(N).




```
LET A$ = STR$(9): PRINT A$
```

One small point here: Positive numbers are always printed with a space to their left. It is in this space that the minus sign is printed for negative numbers. This space is taken into the string when you use the STR$() function. Prove this by typing in: LET A$ = STR$(9): PRINT "[R]";A$

Lock up



Program

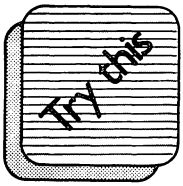
You have heard of User Friendly programs – well, this one is User Unfriendly.

```
10  PRINT "■■----**** COMMODORE
    64 BASIC V2 ****"
20  PRINT "■ -64K RAM SYSTEM__
    38911 BASIC BYTES FREE"
30  PRINT "■READY"
40  GOTO 40
```

Type it in, making sure that you have four spaces at the start of the first line, and one at the start of the second. Now run it. It will, if you have got your spacing right, give you a screen that looks just like the start screen. What happens if you press a key?

The answer is nothing. The program is still running – running round in a tiny circle. The 64 will normally work through a program one line at a time. As it finishes one line it goes onto the next one down. It doesn't have to be this way. You can make the program go to any line you want, by using the GOTO command. If you make it go back to the start of the same line, it will never get anywhere, but neither will it stop trying!

You can stop this by breaking into the program. Press RUN/STOP. GOTO can also do useful things, and we will come back to it shortly. Meanwhile, let's really lock the machine up.



Change line 40.

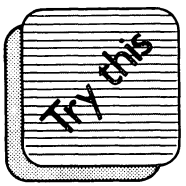
```
40 POKE 1,4
```

Now run it. Can you get any of the keys to do anything? There's only one thing to do. Turn off the machine and start again. This line shows that you must take care when POKEing about in the 64. The wrong number in the wrong place can cause chaos. Next time one of your friends, or family, says that they fancy doing a bit of programming, offer to set the machine up for them, and type in a lock-up program. Only use the second version when you are feeling really unkind.

How long is a string?

The classic question is 'How long is a piece of string?' The smart answer to this one is 'Twice as long as half of it.'

Things are different with computer strings. The 64 can tell you instantly the length of any string, or string variable, by using its LEN function.




```
PRINT LEN("THIS IS A
LONG STRING.")
```

(Remember that strings always have to be enclosed by quotes, and when you are using functions like LEN, you have to wrap the whole lot in brackets.)

Did you get 22 for the LENgth of that string?

Here's another for you to try:

```
A$="SHORT": PRINT LEN(A$)
```

This should give you 5.

The LEN function has many uses. For example, you are laying out a screen, and you want to make sure that a particular string appears centrally. This is no problem if you know beforehand just how long that string will be, but you might not. It could be the user's name, or one of several possible messages. Suppose the string, whatever it might be, is held in the variable W$. Here's the line that would print it centrally.

```
PRINT TAB(20-(LEN(W$)/
2));W$
```

20 is the mid point of the screen. LEN(W$)/2 makes sure that the print position goes left far enough for half of the word to be printed before the middle.

PASSWOR

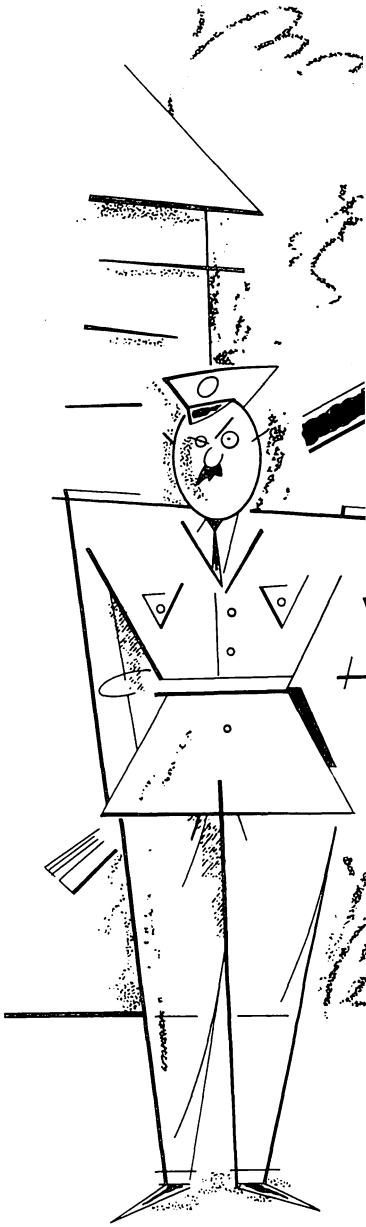
Passwords
All the best computer systems in big offices have security systems to stop the wrong people from looking at certain parts of the program. If they can have security, why not you? Here's how to start protecting your programs. The routine below can be tacked onto the start of any secret programs.

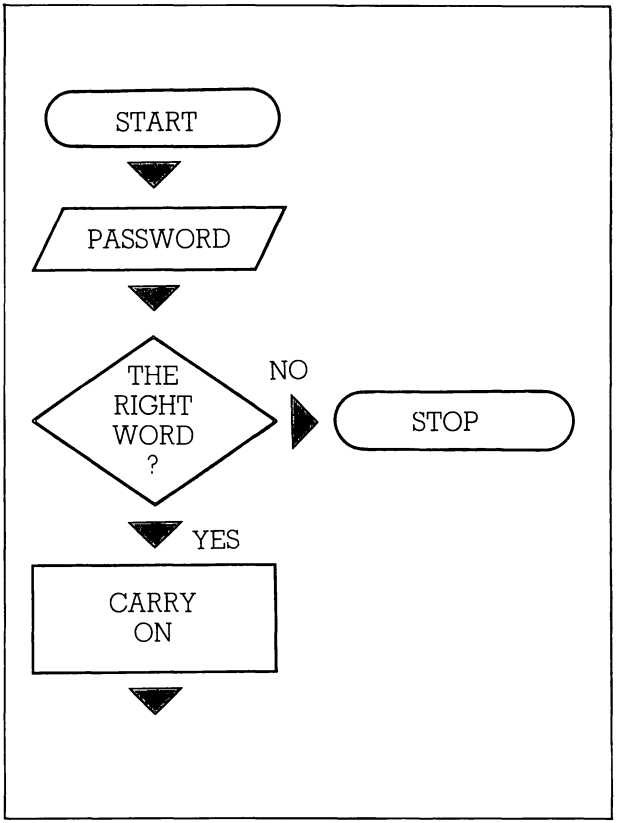
```
 1  INPUT "PASSWORD ";W$
 2  IF W$="OPEN SESAME" THEN
    GOTO 10
 3  POKE 1,4
10  ..... <-----------+
                      |
start of main program -+
```

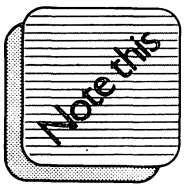
New points
In line 1, you are using an INPUT PROMPT. This is a word or phrase that will be printed on the screen at the INPUT line. The ? and flashing cursor appear directly after it. INPUT prompts must be enclosed by quotes, and be followed by a semicolon and the variable.

Line 2 works the diamond shape in the flowchart. It does exactly what you would expect it to. IF the person typed in the right password, so that W$ was 'OPEN SESAME', THEN the program jumps to line 10. If he gets the password wrong, then the program goes straight on to the next line instead. In this case the next line locks up the keyboard and stops the program.

Time saving tip
You can miss out the GOTO in line 2.
IF. . . . .THEN 10, works just as well.







You should now know how to:

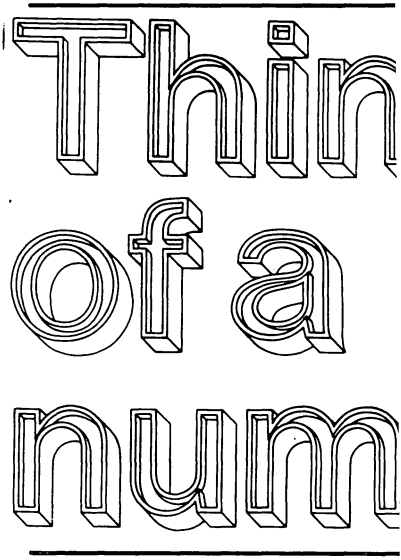
PRINT – using colour and cursor commands.
POKE – screen and border colours.
INPUT – with and without prompts.
GOTO.
Use IF. . . . .THEN. . . lines.
Use variables.

Project

Write a program that asks for the user's name, and then either prints a friendly message if it is your name, or an unfriendly one if it is someone else's.

Add passwords to some of your programs. Think up some hard-to-guess words to include in those IF W$ =". . . . . " THEN. . . lines.

Program



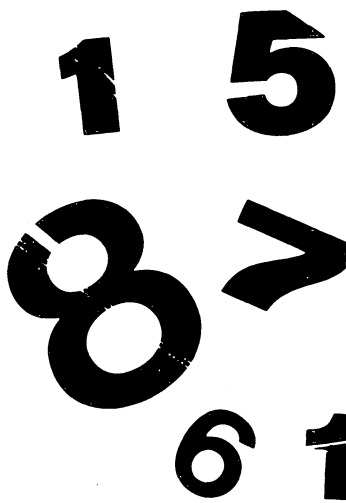
Here's a program for a simple number game. The 64 will 'think' of a number between 1 and 20, and the player has to guess what it is. The program's numbers start at 100 for a very good reason – you are going to add a title screen to it later.

```
100  LET X= INT(RND(0)*20)+1
110  INPUT "GUESS THE NUMBER ";G
120  IF G>X THEN PRINT "TOO BIG!":
     GOTO 110
130  IF G<X THEN PRINT "TOO
     SMALL!": GOTO 110
140  PRINT "THAT'S RIGHT"
150  STOP
```

Type it in and try it. Take care over the first line. This finds a RaNDom number. RND(0) gives a random number between 0 and 1. It's a decimal, 9 digits long, and not the sort of number that anyone would ever guess. If you want to see what these random numbers look like, type in:

```
PRINT RND(0)
```

*20 multiplies the number by 20 to give a random number between 0 and 20.
INT chops off all the decimal bits, leaving a whole number – an INTEGER.
+1 is needed to push it into the 1 to 20 range. Without it, the numbers would be between 0 and 19 (after you had chopped off the decimal bits).

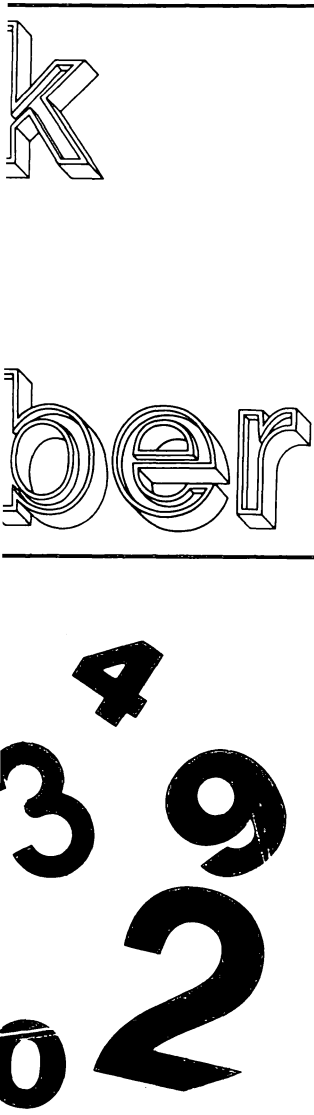


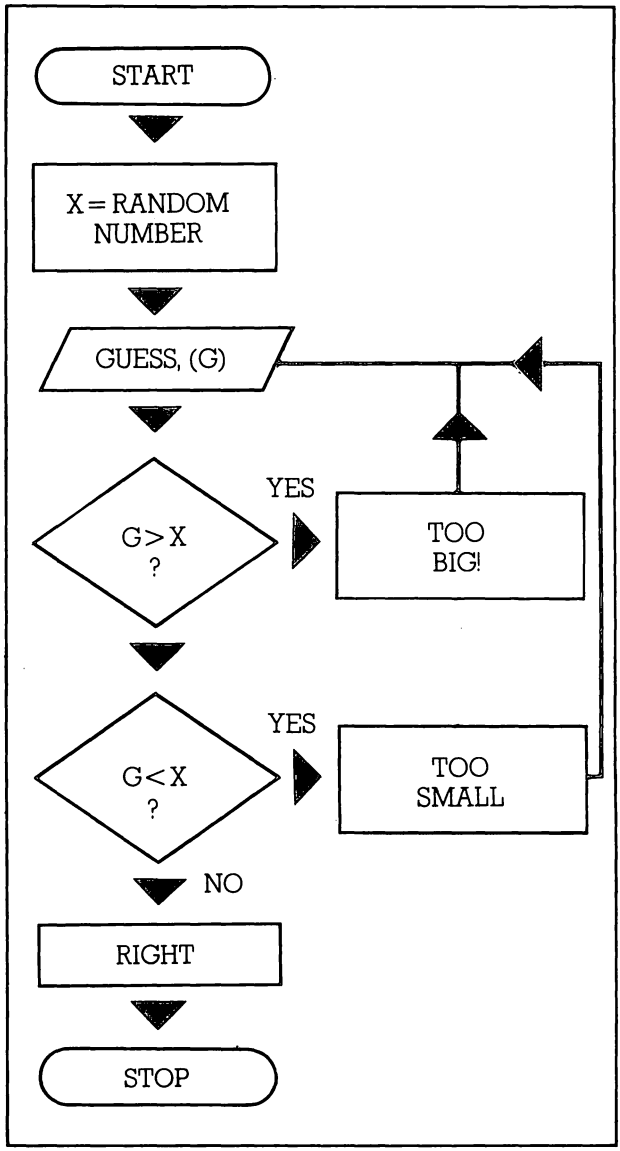
Try this

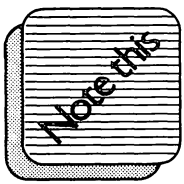
This program, as it stands, has to be RUN every time the number has been guessed correctly. Add some lines at the end to ask the player if he wants another go. IF the answer is 'YES' THEN GO back TO 100. If not, STOP.

You could tidy up the screen by clearing it before you start the new go.

```
105  PRINT "■"
```





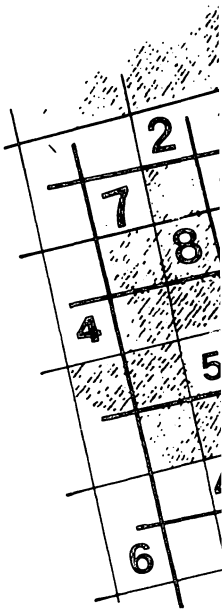


Multi-statement lines. You can put more than one command on a line, as long as there is enough space, and you separate them with colons(:). This is very handy after IF. . . . .THEN. . . statements.

Program

And now, a start screen for your number game. Every good game deserves a title page! This scatters random numbers all over the screen, then prints the title in the middle.

```
10  PRINT "■"
20  C=0
30  T= INT(RND(0)*40)
40  N=INT(RND(0)*20)+1
50  PRINT TAB(T);N
60  C=C+1
70  IF C<22 THEN GOTO 30
80  PRINT "[S][Q][Q][Q][Q][Q][Q][Q][Q][Q][Q][J][J]
    [J][J][J][J][J][J][J] THINK OF A NUMBER"
90  GET A$: IF A$="" THEN GOTO 90
```



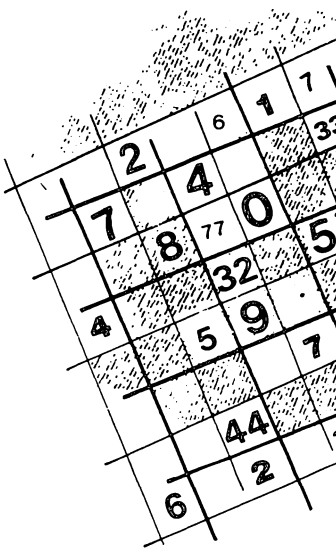
How it works

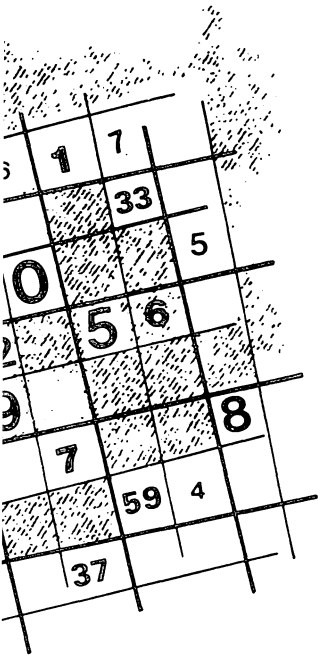
There are two new things in this program. TAB is the TABulator command, used for drawing up TABles. PRINT TAB(12) tells the computer to start printing at column number 12. The 64 has 40 columns across the width of the screen, and these are numbered from 0 to 39. Line 30 gives you a random number for the TAB position.

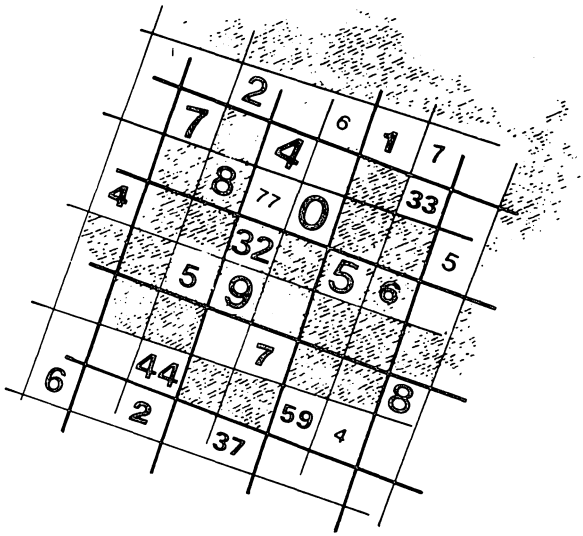
Line 90 introduces a whole new idea. There are two ways of getting information from the keyboard, not just one (INPUT).

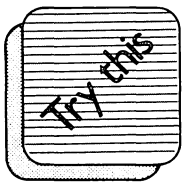
GET A$ tells the 64 to look at the keyboard and see if any key has been pressed. The key's character (if one has been pressed) is to be stored in the A$ variable. The second half of the line checks to see if A$ has a value. If it hasn't then the 64 goes back to the start of the line to check the keyboard again.

This sort of line is a very neat way of giving the user control of the program. The screen appears, and when he is ready, the user can press any key to move on to the game.









It's a variation on the User Unfriendly READY screen. Instead of locking the machine up, hold everything with a
GET A$:IF A$ ="" THEN GOTO . . . line.

Follow that with a set of lines to print a surprise message. 'Go away. I am thinking.' or 'Hang on. Not quite ready.' Another good line would be this:

```
PRINT " OUCH. THE ";A$;" KEY IS
SORE."
```

Notice the semicolons (;) in that line.

Your routine can have a series of GET A$ traps. One of these could act as a password. Make the key letter 'P' for 'Pass' and include a line. . . . .IF A$="P" THEN. . . carry on.

TABLES

Program

Now here is something that you have always wanted – a program that prints out the Two Times Table. Well, it could be useful for a younger member of the family, and it doesn't have to be the Two Times Table. You could adapt the program to print out the 279 Times Table, always assuming that you wanted to learn the 279 Times Table.

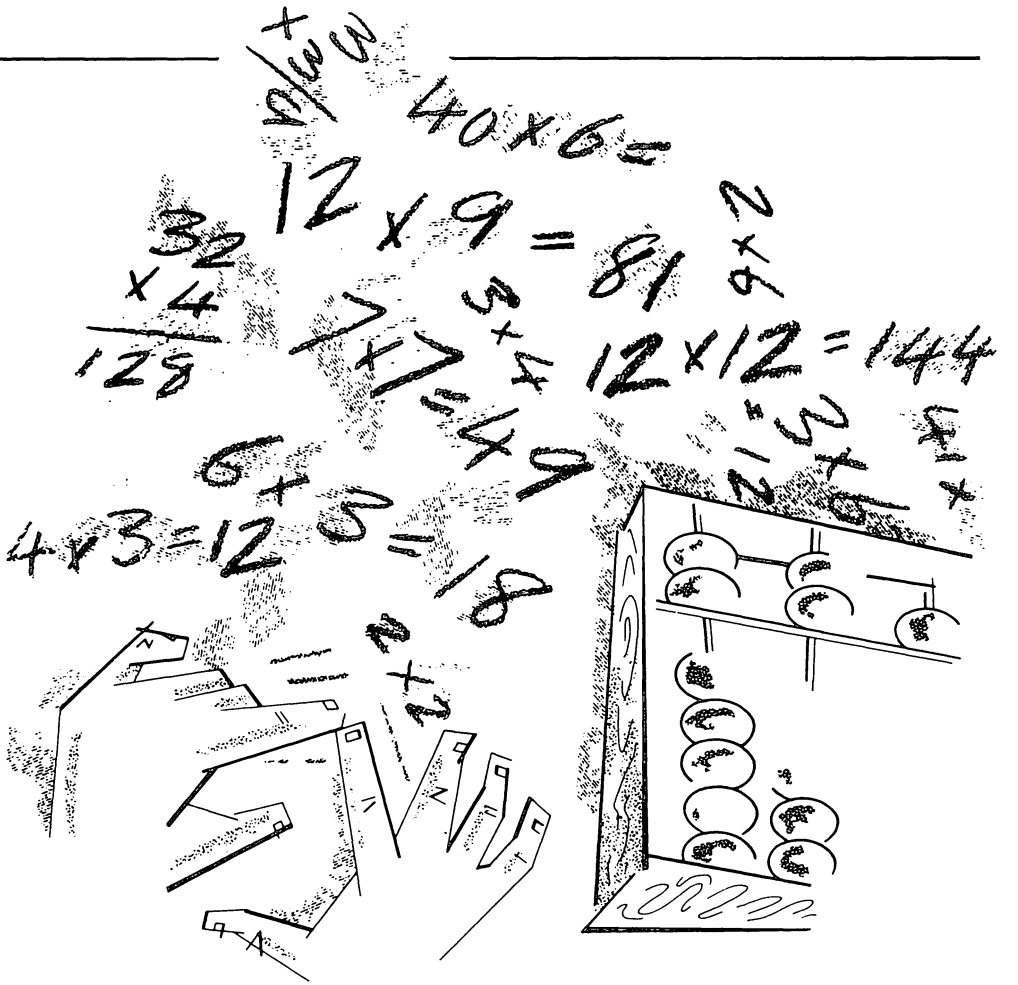
The program has been adapted, so that it prints Two, Three and Five Times.

```
10  PRINT TAB(1);"NO.";TAB(10);"2X";
    TAB(20);"3X";TAB(30);"5X"
20  FOR N = 1 TO 12
30  PRINT TAB(1);N;TAB(10);N*2;
    TAB(20);N*3;TAB(30);N*5
40  NEXT N
```

FOR N=1 TO 12 means, for every number from 1 to 12 . . . . It is the first line of a FOR. . .NEXT loop. The other end of the loop is line 40, and NEXT N means go back for another number, until we reach the last one.

What exactly does line 30 do? Type the program in and run it, just so that you can see how those TAB's give neat columns of figures. There may come a time when you want to produce this sort of display.

FOR. . .NEXT loops are very useful for ANIMATION, and we will get to that shortly, but first another look at TAB, and its close relation SPC.



Project

There are only 39 columns on the screen. What happens if you try to use a higher number with TAB?

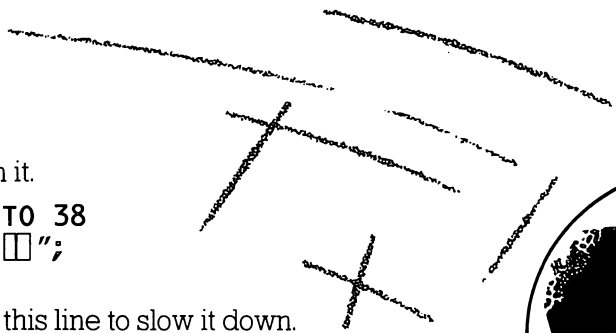
```
10  FOR N=0 TO 255
20  PRINT TAB(N);"*"
30  NEXT N
```

What happened? Did you notice how the asterisks became more and more spread out? The 64 counts its way to the TAB position. Whenever it reaches the end of a line, it drops down to the next. TAB can simplify your PRINT lines.

PRINT TAB(55);. . is easier to type than PRINT "Q]]]]]]]]]]]]]]]]]]]]"

SPC stands for SPaCes. PRINT. . SPC(10). . means leave 10 spaces. While TAB always starts to count from the beginning of the line, SPC counts from the current print position. Retype that Times Table program so that you have SPC in place of TAB. What difference does it make?

Speed



Type this in, and run it.

```
10  FOR C= 0 TO 38
20  PRINT " ◉◫";
30  NEXT C
```

Did you see it? Add this line to slow it down.

```
25  FOR D=1 TO 100: NEXT D
```

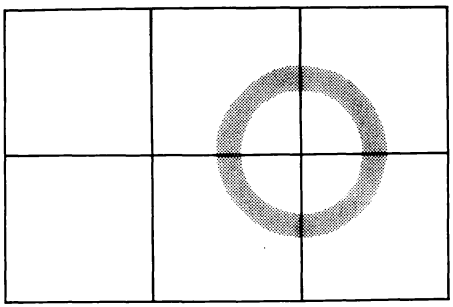
This is a Delay Loop. You make the computer do nothing – which takes next to no time, but you make it do it for 100 times. That slows it down a bit.

Try it again. You should now be able to see the little ball as it moves across the screen. Change the number in your delay line to alter its speed.

Try the same thing with a slightly more complicated graphic – the circle made from the characters on U, I, J and K. Make sure you get the right number of cursor movers in the line, and don't forget the semicolon at the end.

```
20  PRINT"_◪◹Q◫◫◫_◹◪◉◫◫";
```

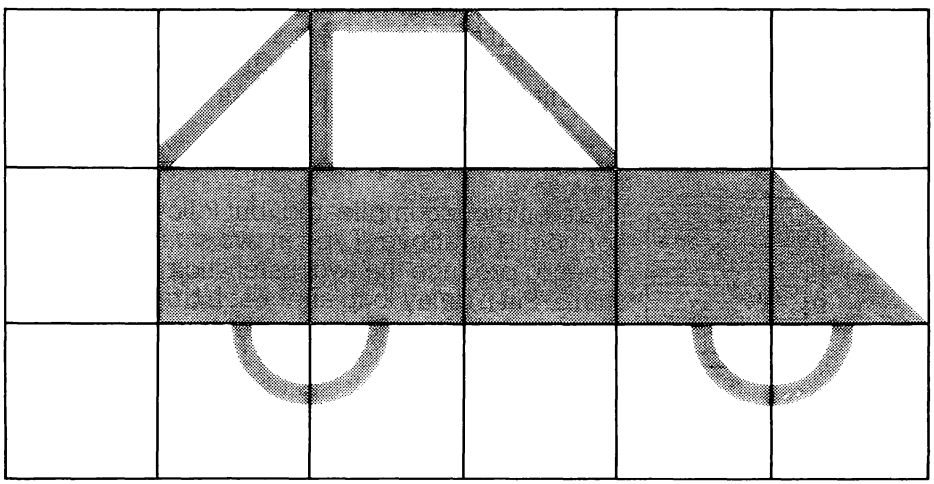
ball



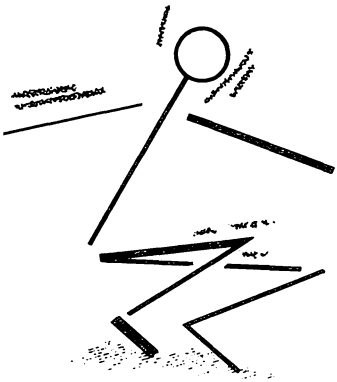
D.I.Y. animation

Time to Do It Yourself. Design a small image – nothing too elaborate, and about the same size as the car shown here. Work out the PRINT line that you would need to produce the figure, and make that the new line 20 for the program. You must make sure that the cursor movers push the print position back to the start of the picture, but one space to the right. You must also make sure that you include a line of blanks down the left hand side, to rub out the old image.

Tidy it up. If you have typed in the program exactly as given, then your screen will be getting cluttered. Add a line to the start of the program to clear the screen. You could push the print position down a few lines, and add some colour to your picture to improve the effect.

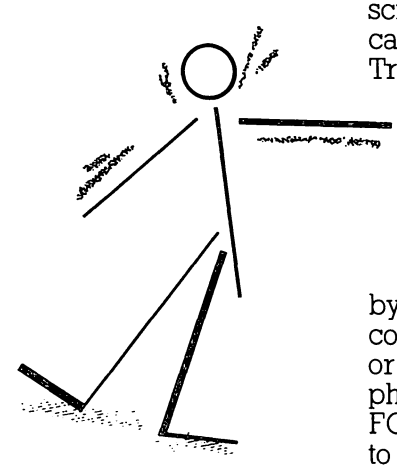


knees bend

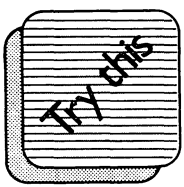


Animation doesn't just mean whizzing across the screen. It means putting life into something. You can put life into text by making words flash. Try this:

```
10  PRINT "■THIS WILL FLASH."
20  FOR D=1 TO 100:NEXT D
30  PRINT "■";TAB(10);"■FLASH"
40  FOR D=1 TO 100:NEXT D
50  GOTO 10
```



You can make the flash even more effective by printing both words reversed, but in contrasting colours. With careful use of the TAB or cursor movers, you can pick out any words or phrases in a screen of text, in this way. Use a FOR. . .NEXT loop, rather than a simple GOTO to limit the number of times your words flash.



It's a bit more complicated, but follows the same principle as above. Type in two sets of PRINT lines to produce the two matchstick men shown here. Put a delay loop after each drawing, and loop the whole program so that the matchstick man bobs up and down. The PRINT lines should print the whole block, spaces and all, to make sure that each image totally erases the previous one. Start each set of lines with a 'cursor home' command.

Exhaustion

When you have got the man bobbing nicely, alter your program to include this idea. The Delay loops that you have at the moment are fixed – FOR D=1 TO 200 makes him move at a reasonable speed. Make the delay variable instead. Add this line to the start of the program:

```
LET WHEN=500
```

Now change the Delay loops to:

```
FOR D=1 TO WHEN: NEXT D
```

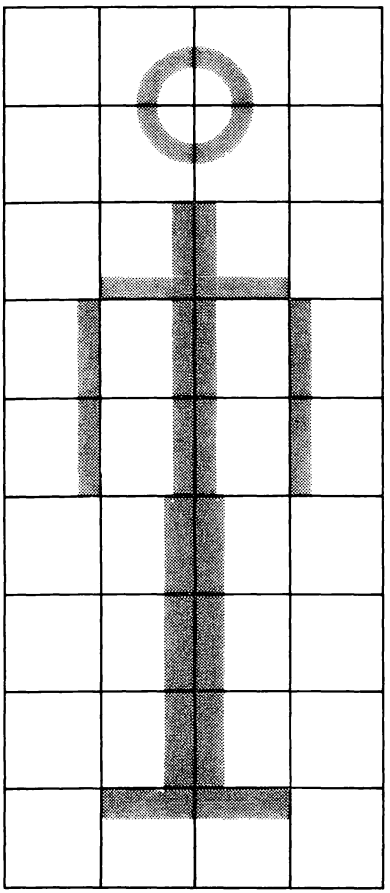
and add this just before you loop back to print the first figure again:

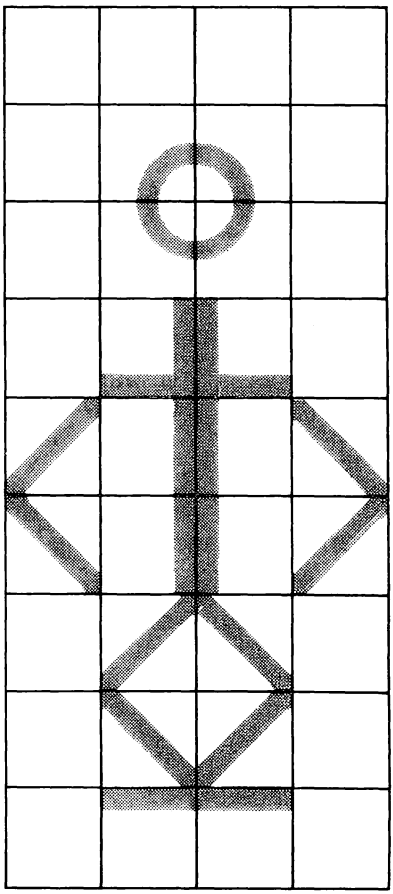
```
LET WHEN =WHEN -10
```

Each time round the loop, the delay gets shorter. It's quite an interesting effect. Design some matchstick gymnastics of your own.

Time saving tip

When you have two lines the same in a program, don't type out the second. Use the cursor to go back and renumber the first line. Try it, and see the result by LISTing.





Going for a walk

Program

Time now to draw together the aspects of PRINTed animation that have been covered in the last few sections. This shows the way to move a changing picture across the screen – in this case the matchstick man is going for a walk. Altering the delay time makes him speed up as he goes.

```
10  LET WHEN = 400
20  FOR T = 0 TO 37
30  PRINT "[♥]";TAB(T);"[◪◱]-[Q][□□□□][◔◰]-[Q][□□□□][□]▙-
    [Q][□□□□][□]--[Q][□□□□][□]◹-[Q][□□□□][◿][□]▂[Q][□□□□]▔
    __"
40  FOR D= 1 TO WHEN : NEXT D
50  PRINT "[♥]";TAB(T);"-[◪◱][Q][□□□]-[◔◰][Q][□□□]-[◿][□]
    [Q][□□□]-[□][□][Q][□□□]--[□][Q][□□□]-[▔][▂][Q][□□□] ---"
60  FOR D= 1 TO WHEN : NEXT D
70  LET WHEN =WHEN - 10
80  NEXT T
90  END
```

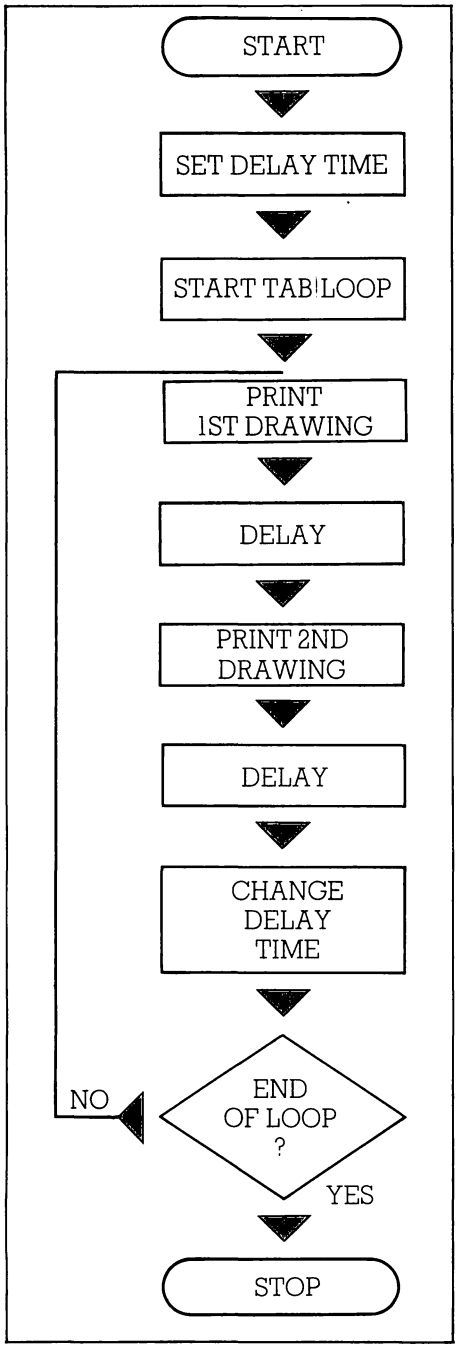
Type this in and run it. If it doesn't work properly then check through it in this order.

Debugging

1 Are you getting any SYNTAX ERROR reports? If you are, then look at the line number indicated, and check the spelling of the BASIC words. Check also that you have semicolons (;) and colons (:) in the right places.

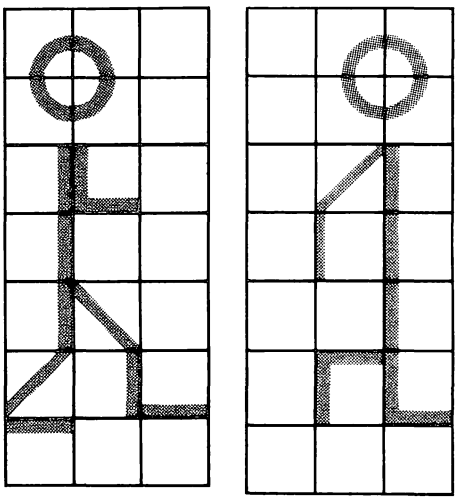
2 Are you getting a NEXT WITHOUT FOR report? This would show that either you had missed out the FOR. . .TO. . . line, or, more likely, that you had mistyped the variable name.

3 In this type of program the most likely cause of error will be in the cursor move commands. One extra, or one missing, will destroy the alignment of the pictures.



Project

Design your own pictures to move with this routine. They really need to be of animals or other figures where movement involves a change of shape. Normal round-wheeled vehicles can be animated with the simpler routine shown earlier. You could of course, try a square wheeled car. . . Routines like this can be included in title sequences of bigger programs, or as a 'reward' in some form of testing program.



Checklist

You now know how to:

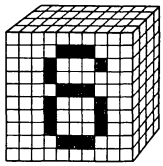
Use random numbers.

Compare numbers.

GET characters from the keyboard.

Use TAB and SPC in PRINT lines.

Animate pictures using loops.

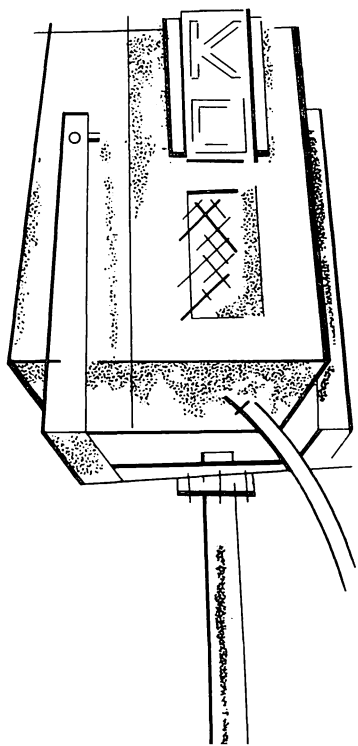


Sound on the 64

INTRODUC

The Commodore 64 can produce an amazing variety of sounds, but, unfortunately, even the simplest sound requires a certain amount of hard work. The SID chip (Sound Interface Device) takes up a set of addresses starting at 54272, in the 64's memory. To program the chip you have to POKE numbers directly into SID.

If you intend to do much with the 64's sounds, you will soon find that coping with 5-figure numbers can be a real headache. One solution to this is to put the base address into a variable: LET SID = 54272. The addresses of stores higher up in SID can be referenced to the base address. POKE SID + 4,. . . is the same as POKE 54276,. . . and much easier to understand.



Program

Type in this program and listen to the sound.

```
 10  LET SID = 54272
 20  FOR N= SID TO SID+24:
     POKE N,0: NEXT N
 30  POKE SID + 24,15
 40  POKE SID + 0,0
 50  POKE SID + 1,34
 60  POKE SID + 5,9
 70  POKE SID + 6,75
 80  POKE SID + 4,17
 90  FOR D= 1 TO 1000: NEXT D
100  POKE SID + 4,16
110  FOR D=1 TO 1000: NEXT D
120  GOTO 80
```

You should hear a slow electronic chime. Break out of the loop with RUN/STOP. If you break out in mid-chime, the sound will continue. Stop it by holding down RUN/STOP and pressing RESTORE. This restores all the system variables to their normal starting values, turning off the sound, putting the screen back to dark blue, and so on.

ING SID

THE SID CHIP

ADDRESS	SID+?	EFFECT
54272	SID+0	} Pitch of note. Voice 1.
54273	SID+1	
54274	SID+2	} Pulse rate
54275	SID+3	
54276	SID+4	Waveform. Voice 1
54277	SID+5	Attack/decay. Voice 1
54278	SID+6	Sustain/release. Voice 1
54279	SID+7	
54280	SID+8	
54281	.....	
54282	.....	} Voice 2
54283	.....	
54284		
54285		
54286	SID+14	
54287	SID+15	
54288	SID+16	
54289	.....	} Voice 3
54290	.....	
54291	.....	
54292		
54293		
54294		} Special effects
54295	SID+23	
54296	SID+24	Volume control (0–15)

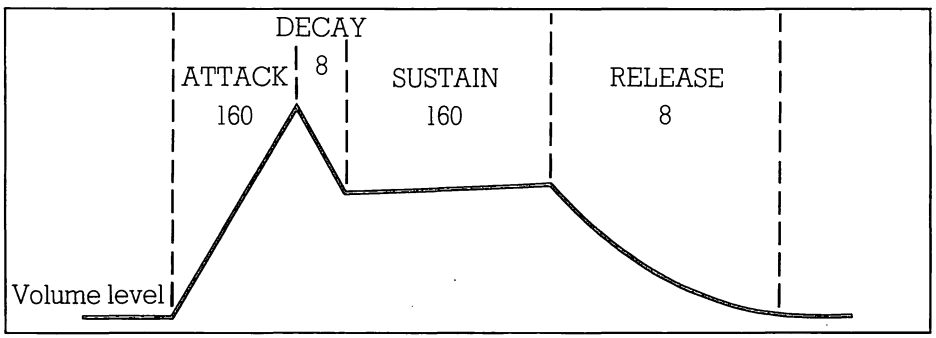
How it works

Line 20 goes through all of the addresses in the SID chip, putting 0 in all of the stores. You must clear the chip before you start to use it.

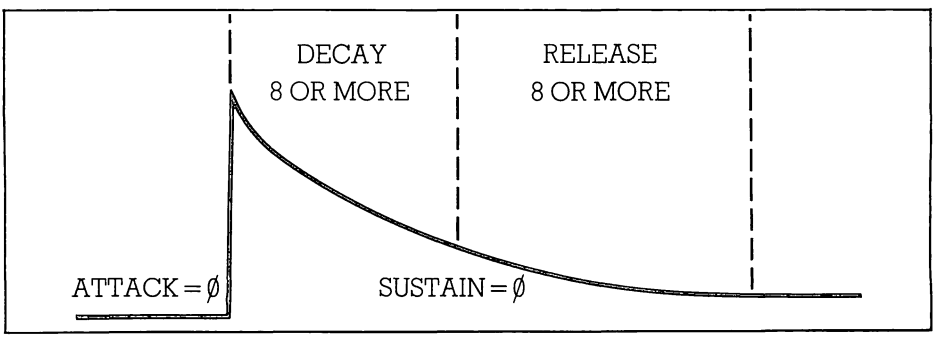
Line 30 sets the volume to maximum, (15). 0 is the minimum. The actual volume you get depends also on the TV volume control. In fact, if the TV volume is turned right down, you won't hear anything!

Line 40 and 50 together set the pitch of the note that is played. The numbers used here can be anything between 0 and 255. Of the two, the most important is the second – SID + 1. POKE a low number here for a low note, a high number for a high note. SID+0 gives the fine tuning to the notes. Try changing the numbers in these lines to see what difference it makes.

Line 60 and 70 determine the ENVELOPE – the SHAPE of the sound. This is in four parts, even though it is managed by only two POKEs.



The shape of the volume level for a wind or string instrument.



The shape of the sound of a percussion instrument.

The ATTACK rate is how long it takes the sound to reach maximum volume. Percussion instruments – drum, bell, piano – reach maximum volume instantly. Wind instruments like the flute take a fraction of a second to reach their peak.

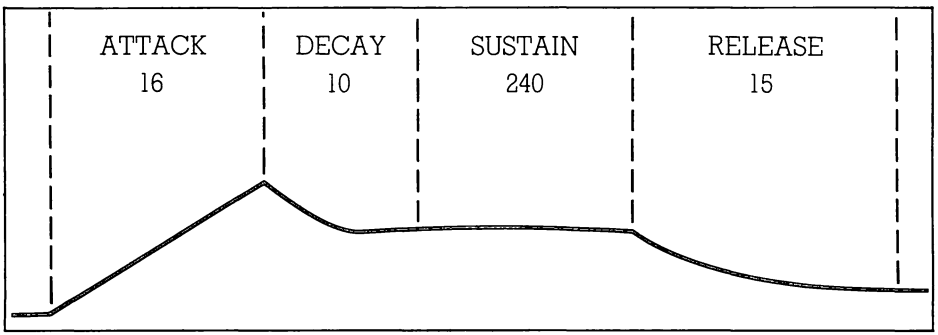
The DECAY rate is the time taken for the volume to go down from its peak to a lower level at which it will continue for a while.

The SUSTAIN rate is how long that lower volume lasts.

The RELEASE rate is how slowly the note dies away.

Those four parts of the shape are controlled by SID +5 and SID +6 in this way.

SID +5 controls ATTACK and DECAY. The numbers that determine the length of the ATTACK phase are 128,64,32 and 16, or any combination of these. The higher the number, the longer the phase. 16 is a very short ATTACK time, 240 (128+64+32+16) is the longest. The DECAY numbers are 8,4,2 and 1, or combinations of them. 15 gives the slowest possible DECAY. The ATTACK and DECAY numbers are added together and POKEd as a single number. POKE SID +5,9 gives an ATTACK rate of 0, and a DECAY of 9. POKE SID+5,95 gives a medium length ATTACK (80=64+16) and long DECAY (15).



A shape suitable for a rocket engine sound effect.

POKE SID+5

Any sum of →

ATTACK				DECAY			
128	64	32	16	8	4	2	1

The same procedure applies to the SUSTAIN and RELEASE rates that are POKEd into SID +6. In the example, the number is 75, which gives a SUSTAIN value of 64 and a RELEASE value of 11.

If you understand about binary numbers, you will recognise the control numbers as being the decimal values of the bits in a byte. If you have yet to meet binary numbers, then come back to this point after you have tackled Sprites, where these are covered.

Lines 80 and 100 are both concerned with the same address – SID +4. This is the one that controlled the waveform, the nature of the note. POKEing this address makes the sound appear, and turns it off. The waveform used here is the TRIANGLE. It has a mellow sound. POKE SID+4,17 turns it on, and POKE SID +4,16 turns it off.

The other waveforms and their reference numbers are shown in the table. The SAWTOOTH (33,32) has a 'beepier' sound, and WHITE NOISE (129,128) will produce a range of hisses and crackles for sound effects. Note that the PULSE waveform is rather unusual, and needs extra programming to make it work.

POKE SID+6

Any sum of →

SUSTAIN				RELEASE			
128	64	32	16	8	4	2	1

WAVEFORM	ON	OFF
Triangle	17	16
Sawtooth	33	32
Variable pulse	65	64
White noise	129	128

Project

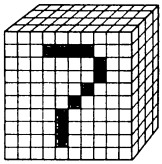
Try changing the values in these lines, one at a time, to see what the effects are. You could also try changing the lengths of the delays. The rate at which sounds are repeated affects the overall impression given by the sound.

Make a note of any interesting sounds that you find, and record the values for each of the SID addresses in a table like this.

Postscript

If this much work is involved in producing just one note, you might think that playing a tune would be an immense task. Fortunately there are two programming techniques which make it much easier. It is time to go 'Back to BASICs'.

SOUND	SID+0	SID+1	SID+5	SID+6	SID+4
Chime – electronic	0	34	9	75	17
Bell	0	34	10	12	33
Rocket	0	7	25	249	129
Steam Hiss	0	99	15	249	129
Steam train	0	25	24	16	129
Piano	0	4–200	10	9	33



Back to BASICs

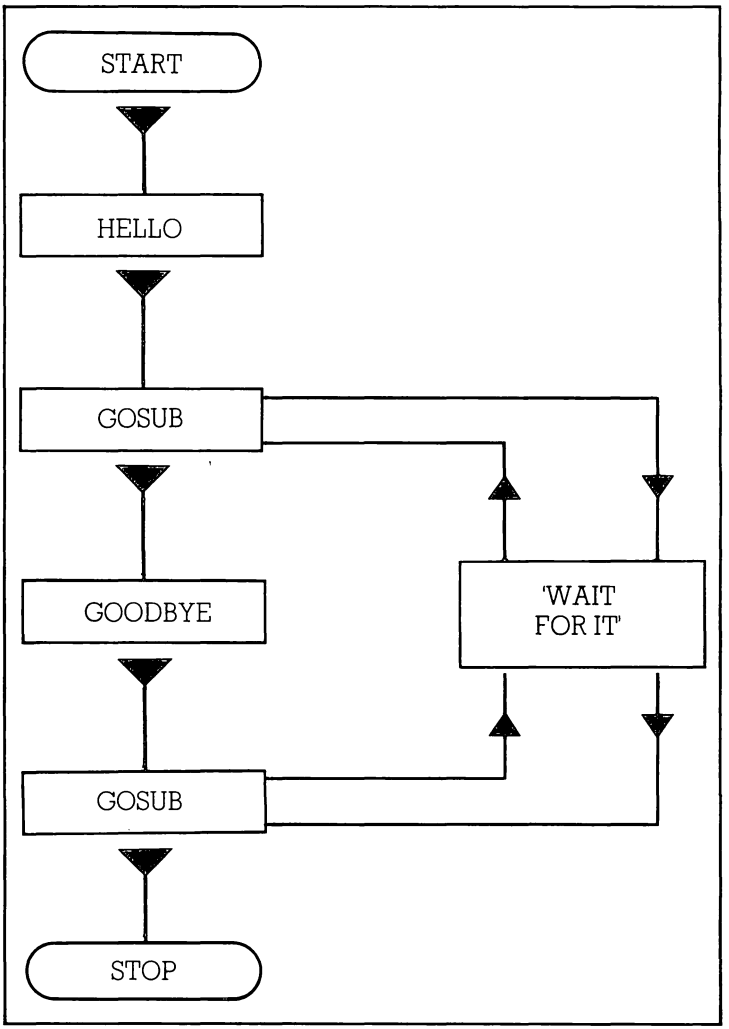
Subroutines

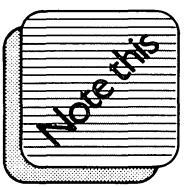
It very often happens in long programs that there are particular routines that you need to use several times – 'Press any key to go on', play a note, or whatever. It would be tedious to have to type them every time they were needed. Subroutines are a way to solve that problem. You send the program to the subroutine with a GOSUB command, and make it come back with a RETURN command.

Type this in, and you will see how it works.

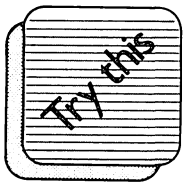
```
 10  PRINT "HELLO"
 20  GOSUB 100
 30  PRINT "GOODBYE"
 40  GOSUB 100
 50  STOP
100  PRINT "PRESS ANY KEY TO GO
     ON"
110  GET A$: IF A$<>""THEN 110
120  GET A$: IF A$ = "" THEN 120
130  RETURN
```

As the 64 heads for the subroutine, it makes a note of the line number it was at. When it meets the RETURN command, it checks the GOSUB stack, to pick up that line number.





Line 110 is a safety precaution. If you miss it out, you will find that a heavy key press will send the program shooting through line 30 and the subroutine and back without stopping. That line tells the 64 to wait there if someone is already pressing a key, and in this way it makes sure that two separate key strokes are needed to work the two trips through the subroutine.

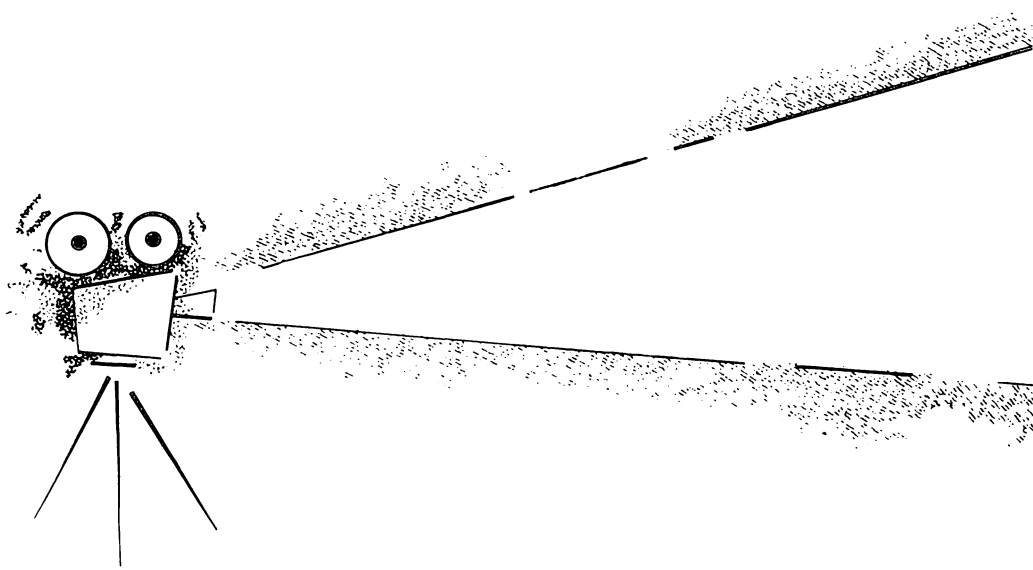


If you have a program where you have added a title screen, renumber the lines that produce that screen, so that the routine fits at the end of the program. Add a RETURN line at the very end. Now you can use the same screen at the beginning and at the end of the program, by sending the computer to it with GOSUB commands.

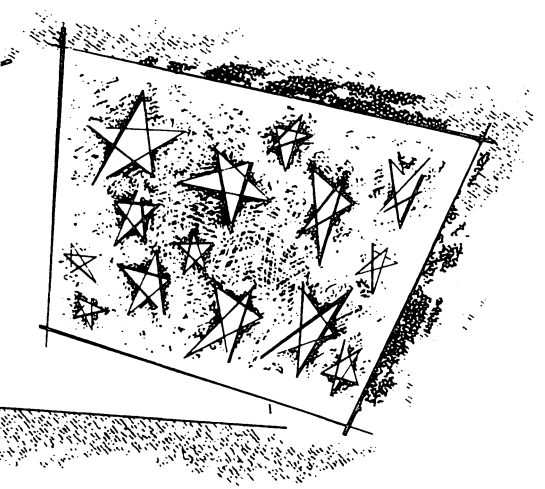
Changing the variables

A subroutine doesn't have to do exactly the same thing every time. If it contains variables, then these can be changed in the main program. Here's an example of this idea at work. The two variables used are T which holds the TAB position, and W$ which stores a word. The subroutine flashes the given word at the given TAB position.

Flashy titles



```
10  PRINT " ♥ THIS IS A
    FLASHY TITLE"
20  LET T= 0: LET
    W$="THIS": GOSUB 100
30  LET T= 5: LET
    W$="IS": GOSUB 100
40  LET T= 8: LET W$=
    "A": GOSUB 100
50  LET T = 10: LET W$=
    "FLASHY" : GOSUB 100
60  LET T= 17 : LET W$=
    "TITLE": GOSUB 100
70  STOP
100 FOR N= 1 TO 10
110 PRINT                    Colour –
    " [S] ";TAB(T);" [E] ";W$ white.
120 FOR D= 1 TO 50 :
    NEXT D
130 PRINT                    Colour –
    " [S] ";TAB(T);" [ ] ";W$ black.
140 FOR D = 1 TO 50 :
    NEXT D
150 NEXT N
160 RETURN
```



Type it in and try it. Remember that when you have lines which are the same or very similar, it is often quicker to renumber the line and alter it with the on-screen editor, than to type out a whole new line. Line 20 can be easily altered to fit any of the next four lines.

Note the cursor home and colour commands in those PRINT lines in the subroutine. These are essential if it is to work properly.

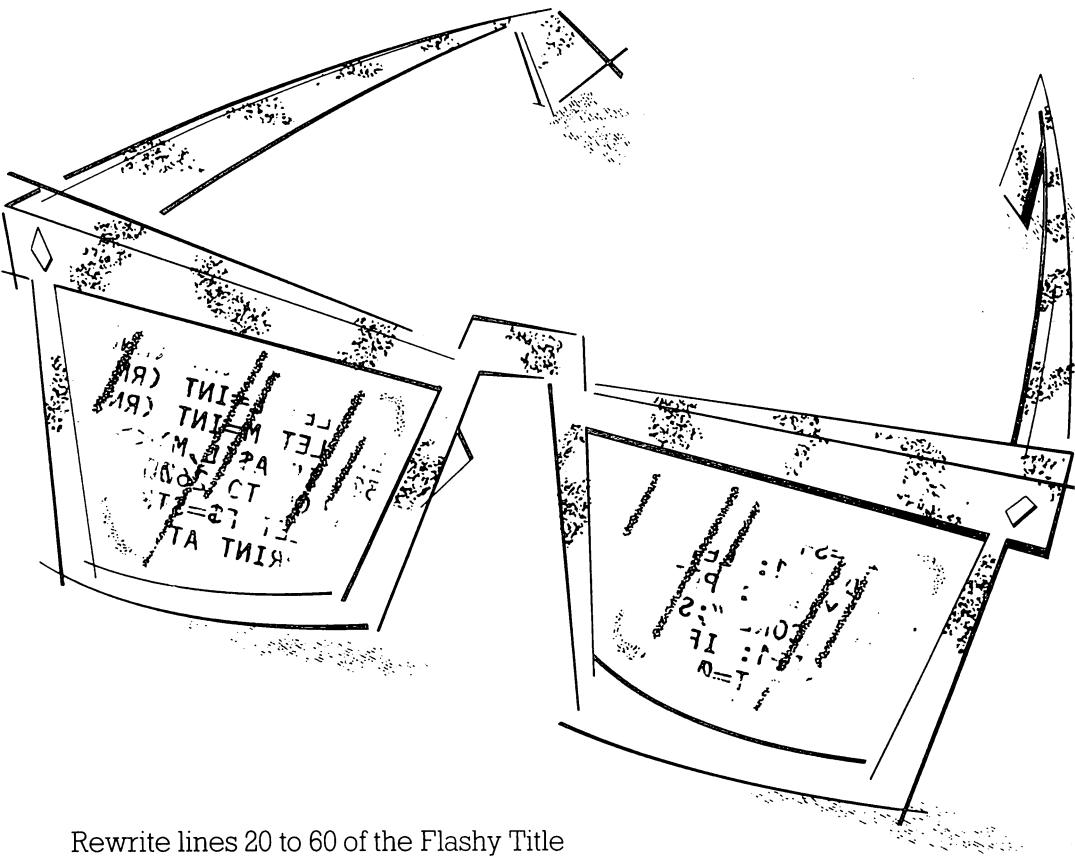
When the program is running as it should, SAVE it, as we will be returning to it later to demonstrate the next BASIC idea.

Subroutine bugs

If you ever see a RETURN WITHOUT GOSUB message, then one of two things has happened. Either you have used GOTO rather than GOSUB when you redirected the program, or there is nothing to stop the computer from carrying on from the end of the main program into the subroutine. That's why line 70 is there in the Flashy Title program.

If a NEXT WITHOUT FOR error comes up, and you are using subroutines, then check that you haven't used the same variable name in the FOR. .NEXT. . loops in both the main program and the subroutine.

Teach the



Rewrite lines 20 to 60 of the Flashy Title program so that they look like this:

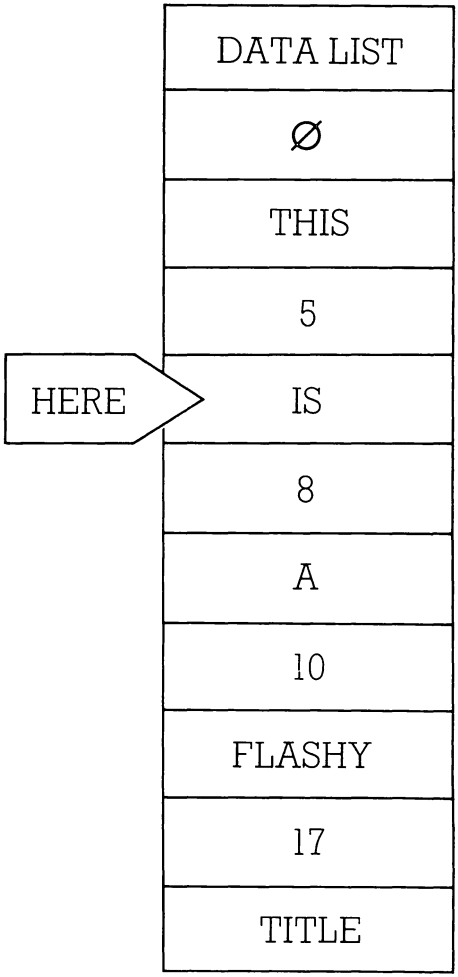
```
20  FOR G= 1 TO 5
30  READ T,W$
40  GOSUB 100
50  NEXT G
60  DATA 0,THIS,5,IS,8,A,10,FLASHY,17,TITLE
```

64 to read

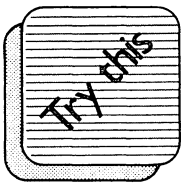
When the 64 meets the READ command, it goes to the line marked DATA and READs the first thing it finds there, storing the information into a variable. In this case, it is READing two things at once. The first piece of DATA is stored in T, the next in W$. On the first time through the G loop, it picks up 0 and 'THIS', and hands them over to the flash subroutine.

When it comes back to the READ command for the second time, it READs the next two pieces of DATA, 5 and 'IS'. Each time it READs, it also pushes a DATA marker along, so that it knows where to look next time. It carries on through the list, until either it reaches the end of the loop, or it runs out of data, in which case you would get an error report.

Reading loops offer a very efficient way of transferring large quantities of information into variables, and can dramatically reduce the workload in a program where the same sort of thing is to happen many times. You can see this in 'Write your own Quizzes'.



Write your own QUIZZES



Watch those DATA lines. Each chunk of data has quotes round it, and these serve two purposes. They make clear to you where each question and answer ends, and they are sometimes essential. If your data has punctuation in it, you must enclose it in quotes, or the computer will be confused. The quotes are not needed when the data consists of simple words and phrases.

Program

```
 10  READ Q$,A$
 20  PRINT Q$
 30  INPUT ANSWER$
 40  IF ANSWER$ = A$ THEN 70
 50  PRINT "WRONG. THE ANSWER
     IS ";A$
 60  GOTO 80
 70  PRINT "ABSOLUTELY CORRECT."
 80  INPUT "ANOTHER GO (Y/N)"; G$
 90  IF G$="Y" THEN 10
100  STOP
110  DATA "WHO IS THE PRIME
     MINISTER OF GREAT
     BRITAIN?","MRS.THATCHER"
120  DATA "HOW MANY BASIC BYTES
     ARE FREE ON A
     COMMODORE 64?","38911"
130  DATA "WHO WRITES GREAT QUIZ
     PROGRAMS?",
     "     (insert your name here)          "
```

You can, and I hope will, write your own questions and answers in the DATA lines. You can have as many as you like. The program will run on until it runs out of DATA.

Quiz programs can get very boring if the questions come up in the same order every time. They really need to be chosen at random. To do that, start your program like this. (I am assuming you have 20 questions. Change that number to fit your program.)

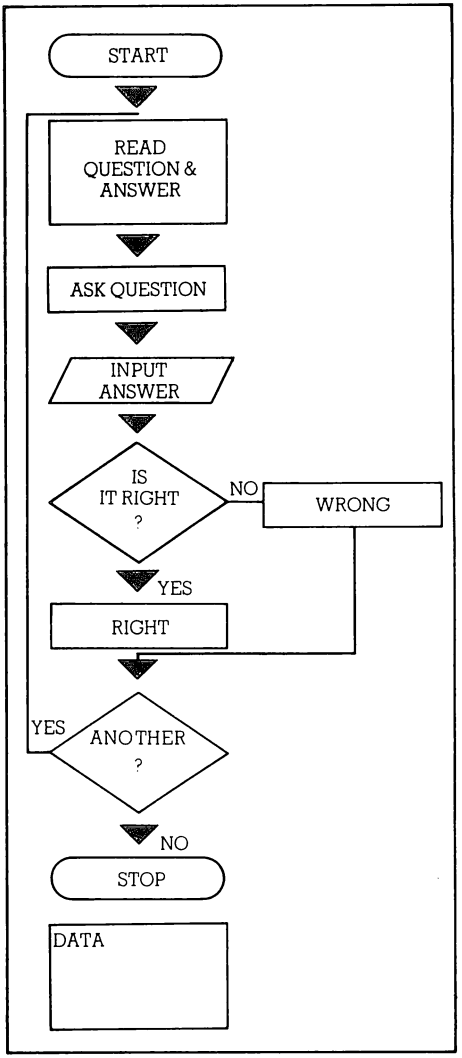
```
 5  RESTORE
10  LET X =
    INT(RND(0)*20)+1
15  FOR N=1 TO X: READ
    Q$,A$:NEXT
```

Change line 90 to send the program back to line 5.

RESTORE moves the DATA marker back to the beginning of the DATA list.

The next two lines get the computer to READ through the list a random number of times, stopping after the Xth time. If the RESTORE line was not there, then the 64 would very rapidly run out of things to read. The question and answer for the go are the last ones that were READ.

Using this routine, you will find that you start to get repetitions after about ten goes. The more data you have, the less likely you are to have questions repeated.



Checklist

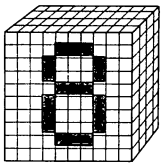
You know now how to:

1 Produce simple sounds.

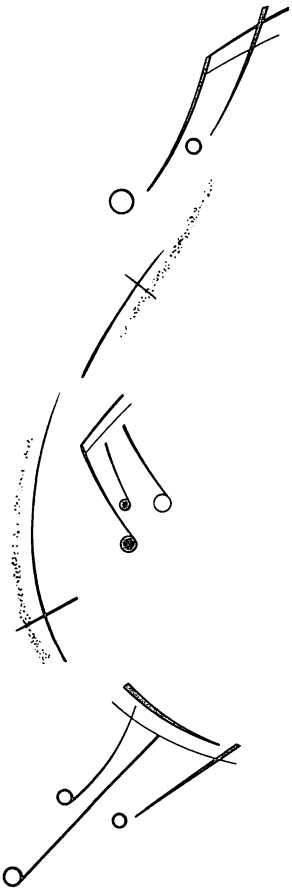
2 Use subroutines to repeat actions. The BASIC commands are GOSUB and RETURN.

3 Put DATA into variables using the READ command, with RESTORE as a means of resetting the data marker to the start of the list.

You are ready to move onto more complex sound programs and sprite graphics.



Time and music



Program

This program shows how you can use a subroutine and a READ loop to produce a series of notes. In this example, you will hear a clock striking the half-hour.

```
10  SID = 54272
20  FOR N= SID TO SID + 24
30  POKE N,0
40  NEXT N
50  POKE SID +24,15
60  FOR N= 1 TO 8
70  READ HF,LF
80  GOSUB 200
90  NEXT N
100 DATA 43,52,34,75,38,126,25,
    177,25,177,38,126,43,52,34,75
199 STOP
200 POKE SID + 0,LF
210 POKE SID + 1,HF
220 POKE SID + 5,9
230 POKE SID + 6,15
240 POKE SID + 4,33
250 FOR D=1 TO 750: NEXT D
260 POKE SID + 4,32
270 FOR D= 1 TO 250: NEXT D
280 RETURN
```

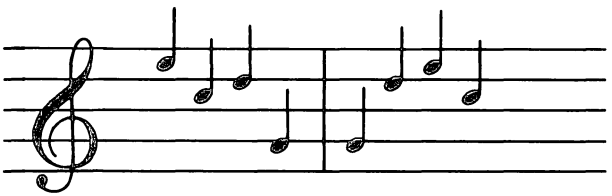
The musical

How it works

Lines 10 to 50. You only have to clear the SID chip and set the volume once for each musical session. In fact, you will find that doing this every time you played a note would create an irritating crackle.

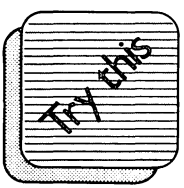
Line 70. The two values for each note are labelled High Frequency and Low Frequency. The High Frequency is the most important number. The Low Frequency is the fine tuning part of the note. The values for the eight notes are given in the DATA line.

Line 270. This creates a short silence between notes to separate them.



E C D G G D E C

	HF	LF
E	43	52
D	38	126
C	34	75
G	25	177



Convert the program so that it strikes the hour, rather than the half-hour. To do this, you must enclose the set of lines from 60 to 90 in another loop, so that the sequence of chimes is played twice. Don't forget to RESTORE at the start of each trip through the loop.

Next add an INPUT line to ask what hour it is, and a loop to create the main chimes. Middle C (HF= 17,LF=37) sounds about right.

```
110  INPUT "WHAT TIME IS IT ";T
120  FOR N= 1 TO T
130  LET HF = 17: LET LF = 37
140  GOSUB 200
150  NEXT N
```

Write your

Find a simple tune, or compose one of your own, and work out the values for the notes. The central part of the scale of C is shown here, and a full list of values is given in Appendix M of the User Manual.

Count how many notes there are in the tune, and put this number into the READ loop. Type in the values as DATA lines, taking care to get the High Frequency and Low Frequency in the right order.

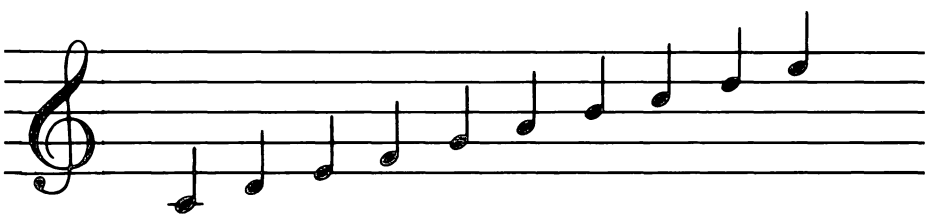
Time changes

As it stands, the note playing program always plays notes of the same length. The length of the note is fixed by those two delay loops. If the number here is made into a variable, then the time value can be READ in the same way as the note values.

```
250  FOR D= 1 TO WHEN : NEXT D
270  FOR D= 1 TO WHEN/4: NEXT D
```

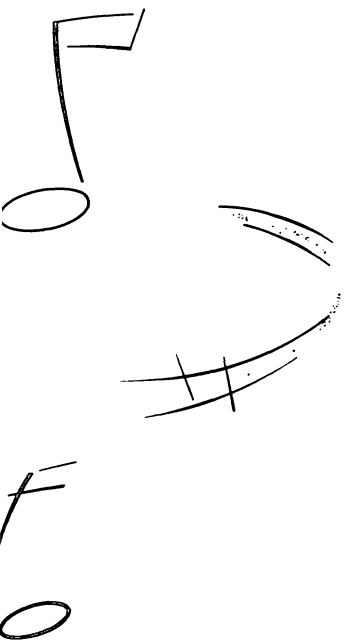
Line 70 should now be:

```
READ HF,LF,WHEN
```



C D E F G A B C′ D′ E′

own music



Fine tuning

You will often find with music programs, that your first stab at a tune doesn't sound quite right. The time values may be altogether too slow or fast, or individual notes may have the wrong time or pitch values. It may help to write a line into the playing loop to PRINT the notes' number, pitch and time values. This will make it easier to identify the bugs.

NOTE	HF	LF
C	17	37
D	19	63
E	21	154
F	22	227
G	25	177
A	28	214
B	32	94
C'	34	75
D'	38	126
E'	43	52

		WHEN?
MINIM		1500
CROTCHET		750
QUAVER		375

WHAT'S THE TIME

You don't need a clock to answer that question – there's one built into the Commodore. It's not normally visible, but it's easy enough to find. Try this.

```
PRINT TI$
```

The 64 will print a six figure number, and this tells you the hours, minutes and seconds since the machine was turned on. If it said '023607' this would mean 2 hours, 36 minutes and 7 seconds.

```
02 36 07
```

This clock ticks over all the time the machine is running, except when the cassette is in use, quite independently of anything else that it might be doing. It is not changed by any programs – unless you decide you want to change it. TI$ is a variable, and like all variables it can be reset.

Program

This will turn your computer into an alarm clock. Write the time in line 10 and line 20 as six figure numbers. In the example here they are set for 4.55 and 5 o'clock.

```
 5  PRINT "■"
10  TI$="045500"
20  IF TI$="050000" THEN 40
30  GOTO 20
40  SID =54272
50  FOR N= SID TO SID+24: POKE
    N,0: NEXT N
60  POKE SID+24,15: REM VOLUME
70  POKE SID+1,120: REM HI FREQ
    VOICE 1
80  POKE SID+15,40: REM HI FREQ
    VOICE 3
90  POKE SID+5,9: POKE SID +6,0:
    REM SHAPE OF NOTE
100 POKE SID+4,21: REM WAVEFORM
110 FOR D=1 TO 50 : NEXT D
120 POKE SID+4,20
130 FOR D=1 TO 20: NEXT D
140 GOTO 100
```

You will need to wait five minutes before the alarm goes off when you RUN the program.

The bell like sounds are produced by a technique known as 'ring modulation', which combines the output of two oscillators (the parts of the circuit which produce the waveforms). The normal waveforms only use one oscillator. Try altering the values in lines 70 and 80 to see the range of notes that you can produce.

When it's all typed in and checked, set the program running and go off and make a cup of tea. When the 64's clock reaches the time you set in line 20, bells will ring. You have succeeded in converting your 64 into an expensive alarm clock, because you can't do anything else with it while this program is running.

Time for a game

The 64 doesn't actually keep track of time in hours, minutes and seconds. TI$ just turns it into this form for your benefit. The clock works by a straight-forward counter ticking over 60 times a second. You can read this counter directly, by typing PRINT TI. Convert this to seconds by typing PRINT TI/60. You can't reset TI directly. 'TI=0' won't work. You can reset it by changing TI$. Try this and see.

```
TI$="000000": PRINT
TI$,TI
```

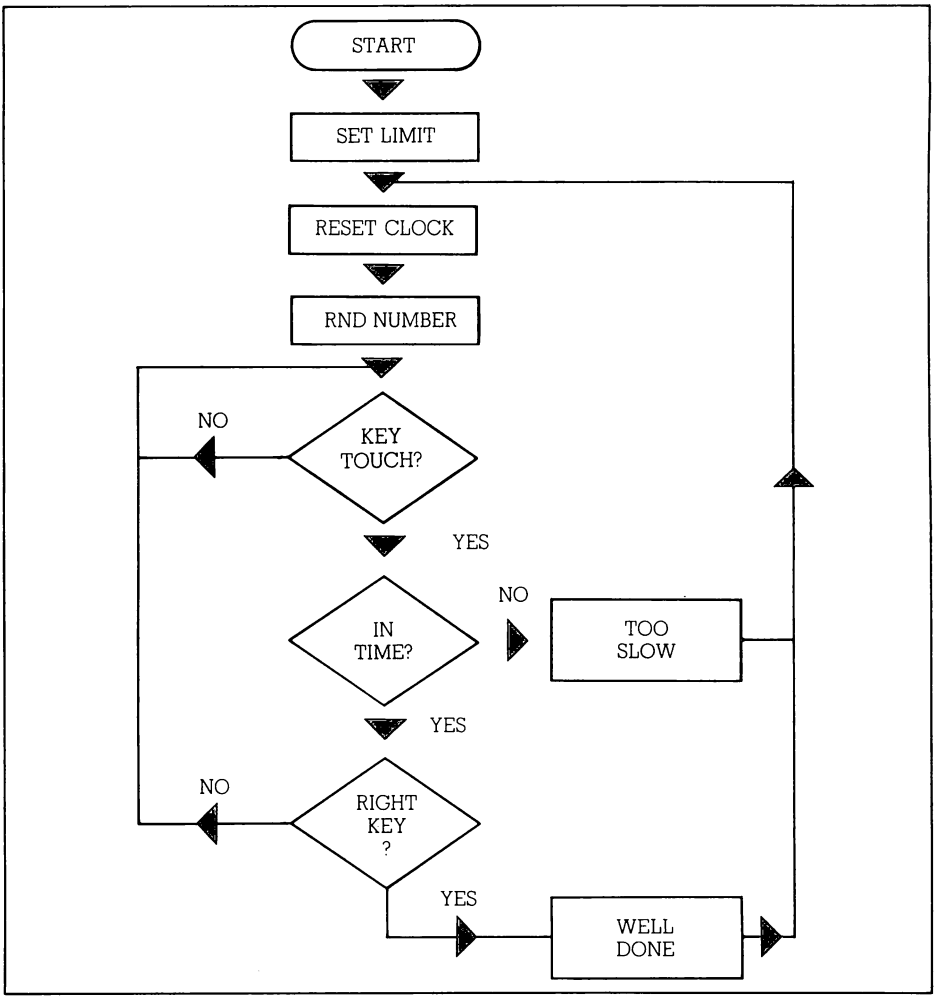
Because TI works in such small units of time, it is quite a flexible way of setting time limits in a game. Try this one.

Speed test

The 64 prints a random number on the screen. The player has to press that number key within a time limit. This limit is raised or lowered depending upon how successful he is. The object of the game is to see how low a limit you can reach.

Program

```
  5  PRINT "■"
 10  LIMIT =120
 20  X= INT(RND(0)*10)
 30  PRINT X, LIMIT
 40  TI$="000000"
 50  GET A$: IF A$<>"" THEN 50
 60  GET A$: IF A$ ="" THEN 60
 70  IF TI>LIMIT THEN 100
 80  IF VAL(A$) =X THEN 120
 90  GOTO 50
100  PRINT "TOO SLOW" : LIMIT =LIMIT + 10
110  GOTO 20
120  PRINT "WELL DONE ": LIMIT =LIMIT -10
130  GOTO 20
```



Projects

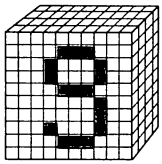
This is only the skeleton of the program; you can make a number of improvements to it.

1 After each go, send the program off to a routine to print the player's new limit, and to ask if he wants another go. At the moment, there is hardly time to read the limit.

2 Make this into a letter game by changing the RND line to

```
20  X=INT(RND(0)*26)+65:
    X$=CHR$(X)
```

Then print X$ rather than X. The check line is now a simple "IF A$ =X$. . ." This could be used for touch typing practice.

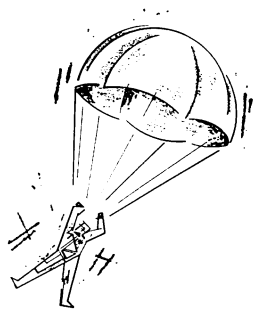


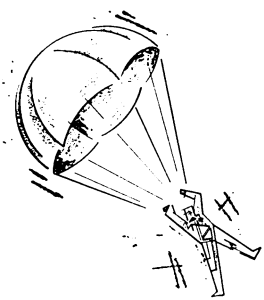
Sprites

PARACHU

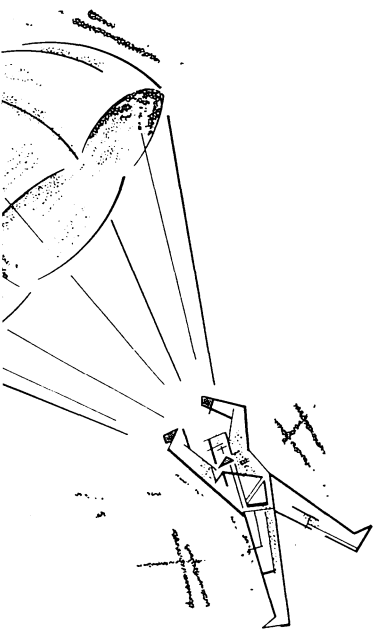
A FIRST LOOK AT

This program creates a sprite in the shape of a parachutist, and makes him drop slowly down the screen. It's rather long, but then it takes a lot of DATA to define a sprite.





TIST SPRITES



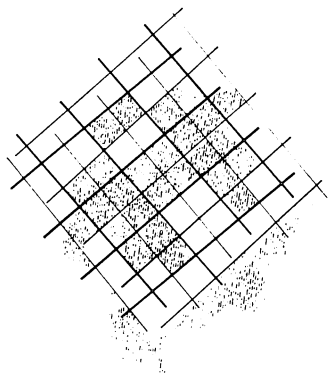
Program

```
  5 PRINT "▽"
 10 FOR N=0 TO 62
 20 READ B
 30 POKE 12800+N,B
 40 NEXT N
 50 DATA 1,255,0
 60 DATA 15,255,224
 70 DATA 127,255,252
 80 DATA 255,255,254
 90 DATA 192,124,6
100 DATA 128,16,2
110 DATA 64,16,4
120 DATA 32,16,8
130 DATA 16,16,16
140 DATA 8,16,32
150 DATA 4,16,64
160 DATA 2,16,128
170 DATA 1,57,0
180 DATA 1,187,0
190 DATA 1,85,0
200 DATA 1,255,0
210 DATA 0,56,0
220 DATA 0,56,0
230 DATA 0,40,0
240 DATA 0,68,0
250 DATA 0,130,0
260 POKE 2040,200
270 VIC = 53248
280 POKE VIC +0,100
290 POKE VIC +1,0
300 POKE VIC + 39,1
310 POKE VIC + 21,1
320 FOR Y = 0 TO 255
330 POKE VIC +1,Y
340 NEXT Y
350 END
```

The program is really in two parts. The lines down to 260 are concerned with defining the shape of the sprite. The rest of the lines put the sprite on screen and move it. Let's take each part in turn.

SPRITE DEFINIT

The sprite that you saw on the screen was composed of a pattern of dots of light in a grid 24 dots by 21. The information that determines this dot pattern is held in memory as a set of numbers. What the program cannot show is the relation between the dots and the numbers, and to understand this we have to look at binary numbers.



Binary numbers

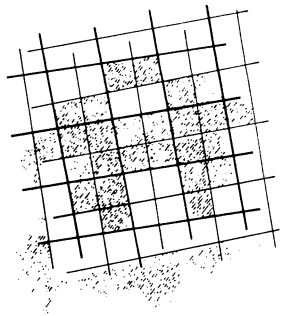
Our normal (decimal) method of counting was developed to fit the fact that humans have ten fingers. The shepherd counting his sheep, would count the first ten on his fingers, then put a pebble in his pocket to remind him of that group of ten, and start from one again on his fingers. When all the flock had passed through the gate, he could count the number of pebbles in his pocket, counting ten for each pebble, and add on the extra ones that were recorded on his fingers.

In the decimal system, each digit in a large number is worth 10 times the amount of the same digit to its right. So, 999 is worth $900 + 90 + 9$, or $9 \times 10 \times 10 + 9 \times 10 + 9$.

A computer has no fingers, and no pockets in which to put pebbles. What it does have are lots of circuits where an electric current can be turned on or off. These are called BITs, and they are grouped in eights, into BYTEs.

BITs and BYTEs

In a BYTE, the rightmost BIT has a value of 1 if it is turned on. The BIT to its left is worth 2, and as you work across to the left, each bit is worth twice as much as the next, when it is turned on.



128	64	32	16	8	4	2	1
0	1	0	0	1	1	1	0

ION

This byte is worth the same as 78 in decimal. The bits that are turned on are 64, 8, 4 and 2.

The largest value any byte can have is 255 = 128+64+32+16+8+4+2+1. Of course, the computer can handle much larger numbers than this, but not in a single byte.

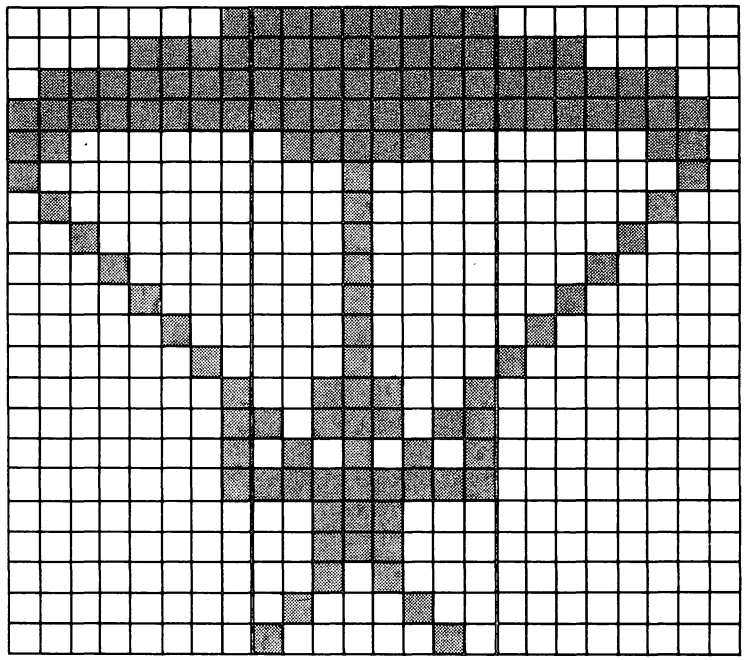
The sprite pattern has 24 dots in each row, and these are managed by 3 numbers, one for each set of 8 dots. When defining sprites, therefore, we have to convert each row of dots into three binary numbers, and change these to decimal numbers.

With practice, you can get quite fast at this. You may find the Dot pattern to number conversion table (pages 82–83) useful at first.

Stage 1

The first stage of defining a sprite is to sketch out your design on squared paper, and shade in the dots that will give you the shape you want. Divide the sprite up into three columns of eight dots each, and convert the patterns to numbers. Keep a record of these numbers in a table like the one shown on page 81.

Dot pattern



Stage 2

The second stage is to transfer those numbers into memory. Exactly where you put them is up to you. The free memory area starts at address 2048, and goes through to 40959. However, your BASIC program has to fit in the first part of that space, and if you use the addresses over 16383 (the 16k mark) you will have problems. The VIC chip – the part of the 64 that handles graphics – can only look at 16k of memory at a time, and it is normally set to look at the first 16k.

Sprite definition numbers have been put from address 12800. It's far enough up memory to leave lots of space for the BASIC program, and there's still space for another 54 sprite pictures as well.

The loop at the beginning of the Parachutist program READs the sprite DATA into the block of memory starting at 12800, and going on to 12862. If you compare the numbers in those DATA lines with the Number Set worked out

Binary pattern																							Number set		
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		1	255	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0		15	255	224
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0		127	255	252
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0		255	255	254
1	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	0		192	124	6
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0		128	16	2
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0		64	16	4
0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0		32	16	8
0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0		16	16	16
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0		8	16	32
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0		4	16	64
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0		2	16	128
0	0	0	0	0	0	1	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0		1	57	0
0	0	0	0	0	0	1	1	0	1	1	1	0	1	1	0	0	0	0	0	0	0		1	187	0
0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0		1	85	0
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0		1	255	0
0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0		0	56	0
0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0		0	56	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0		0	40	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0		0	68	0
0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0		0	130	0

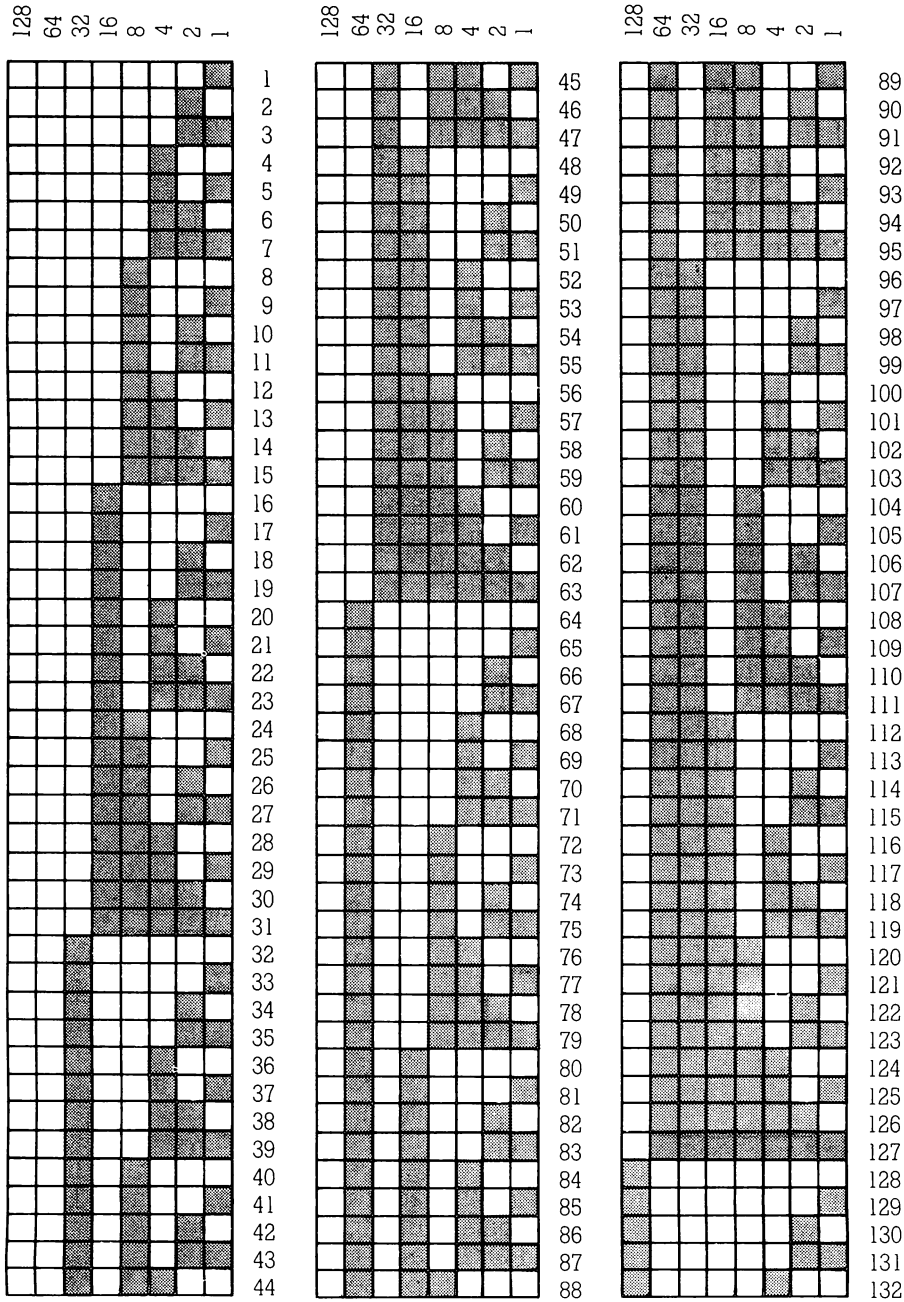
next to the dot pattern, you will see that the numbers are stored one row at a time, working from left to right and from top to bottom.

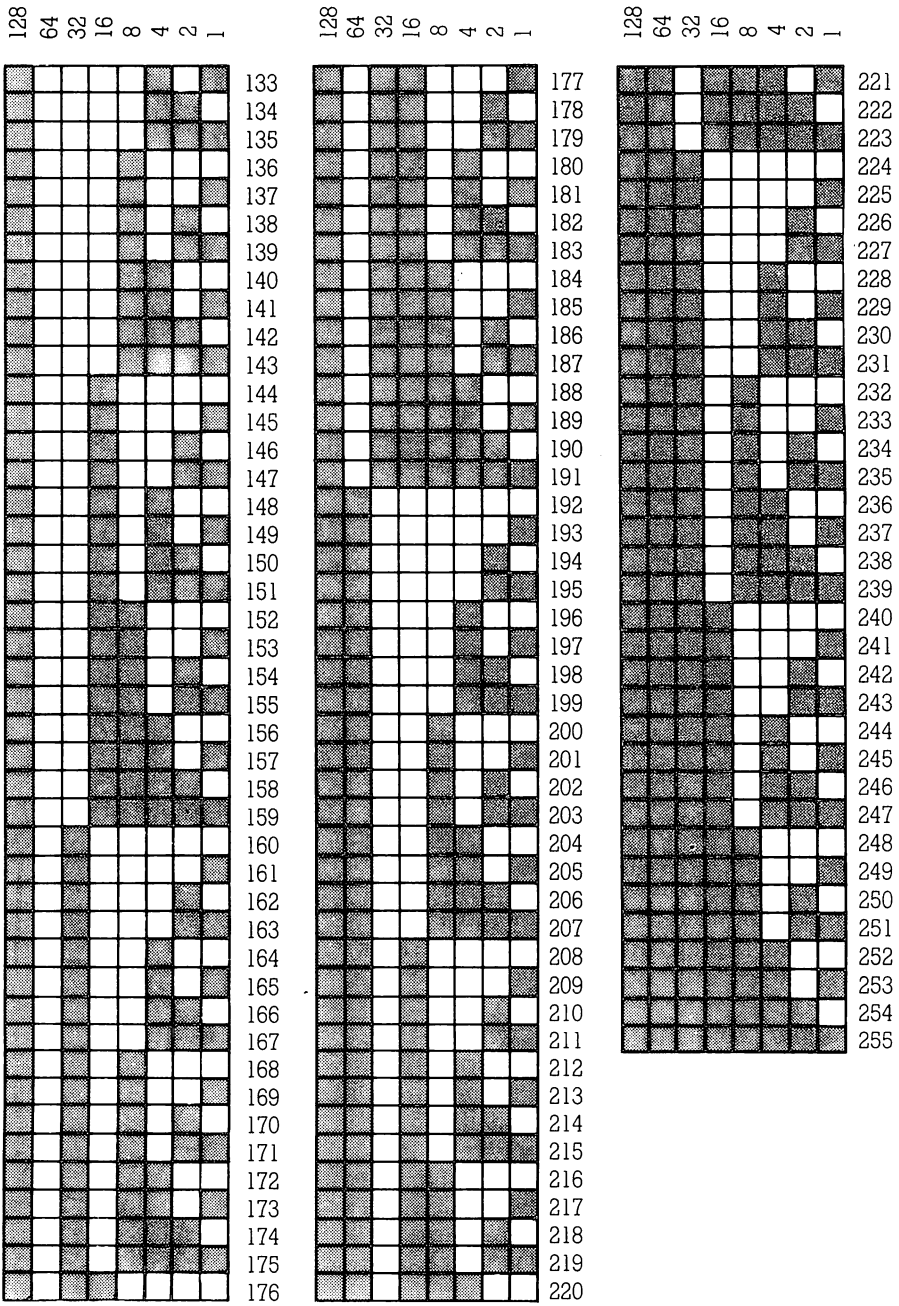
Stage 3

The last, and an important, part of the sprite definition routine is to tell the 64 where you have put the data for the sprite. This is line 260. POKE 2040,200.

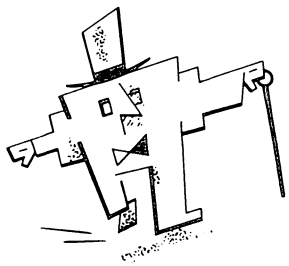
This sets the SPRITE POINTER for the first sprite to the address 12800. 2040 is the address of the sprite pointer. 200 × 64 gives the number 12800. If you wanted to define a second sprite, then you would put its data in the block starting at 12864, (201 × 64 = 12864). Bear this in mind when deciding where to put sprite data. Whatever address you choose it must be a multiple of 64, or you will not be able to set the sprite pointers properly.

Dot pattern to number conversion table



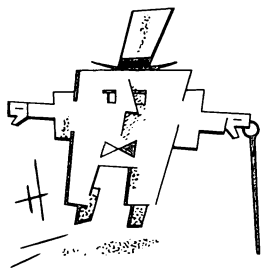

128 64 32 16 8 4 2 1
133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
128 64 32 16 8 4 2 1
177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220
128 64 32 16 8 4 2 1
221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255

Meet VIC



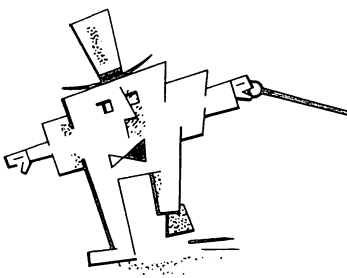
VIC – the Video Interface Chip – is the part of the 64 which controls the screen; its colours, graphics and other characters and, most importantly for you, the sprites.

It is located in the 64's memory at the addresses starting from 53248. You will have noticed in the Parachutist program how this address was stored in the variable VIC, just to make life easier. Four numbers were POKEd into the VIC chip to make the sprite appear. This is the absolute minimum. You must give the X position (horizontal), and Y position (vertical) and the sprite colour code, and you must turn the sprite on.



Look at the table and work out which POKE did which. (We were working with sprite 0.)

As you can see from the table, there are eight sprites, numbered 0 to 7. You could, if you wanted, alter the Parachutist program so that there are eight parachutists. You do not have to write in READ and DATA lines for those extra seven sprites. They can all share the same sprite pattern. The lines needed to add another three sprites are shown here.



	X	Y	Colour	Turn on	Pointer address
Sprite 0	VIC+0	VIC+1	VIC+39	VIC+21,1	2040
Sprite 1	VIC+2	VIC+3	VIC+40	VIC+21,2	2041
Sprite 2	VIC+4	VIC+5	VIC+41	VIC+21,4	2042
Sprite 3	VIC+6	VIC+7	VIC+42	VIC+21,8	2043
Sprite 4	VIC+8	VIC+9	VIC+43	VIC+21,16	2044
Sprite 5	VIC+10	VIC+11	VIC+44	VIC+21,32	2045
Sprite 6	VIC+12	VIC+13	VIC+45	VIC+21,64	2046
Sprite 7	VIC+14	VIC+15	VIC+46	VIC+21,128	2047

Program

```
260  FOR P=2040 TO
     2043:POKE
     P,200:NEXT P
280  POKE VIC +0, 50
281  POKE VIC +2,75
282  POKE VIC +4,100
283  POKE VIC +6,125
...  ...
291  POKE VIC +3,0
292  POKE VIC +5,0
293  POKE VIC +7,0
...  ...
301  POKE VIC +40,2
302  POKE VIC +41,7
303  POKE VIC +42,9
...  ...
310  POKE VIC +21,15
...
331  POKE VIC +3,Y
332  POKE VIC +5,Y
333  POKE VIC +7,Y
```

Set sprite pointers.

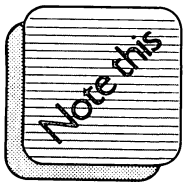
X positions – Sprite 0 has been moved to the left.

Y positions.

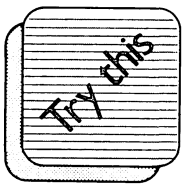
Colours – choose your own.

Turn on sprites – see below.

Y positions in loop.



Line 310 needs some explanation. For all of the other POKEs here, there is a different address for each sprite. Only ONE address is used for turning on sprites. Each BIT of the byte at VIC+21 acts as an ON/OFF switch for a sprite. Look down the fifth column in the table and you will see that the numbers form a binary pattern. "POKE VIC +21,15" turns on sprites 0,1,2 and 3. (1+2+4+8 = 15)



1 Use the table to work out the POKEs needed to produce the other four parachutists.

Set

— sprite pointers to 200 to get the pattern

— X and Y values. Either number can be anything between 0 and 255. I have spaced them out in gaps of 25 dots, and started all of them at 0, which is off the top of the screen. Try your own placings.

— colours. Use the same colour codes here that you use when setting screen and border colours.

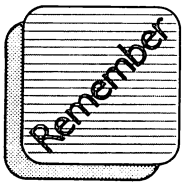
Change line 310 so that all the sprites are turned on. POKE VIC + 21,255 for all eight.

2 Line 320 (FOR Y = 0 TO 255) starts the parachutists above the top of the screen, and makes them fall below the bottom. At what Y value is a sprite first fully on screen, and what value would leave it standing on the bottom edge?

Add

```
321  PRINT "[S]";Y
322  FOR D = 1 TO 250:NEXT D
```

and make a note of the Y values.

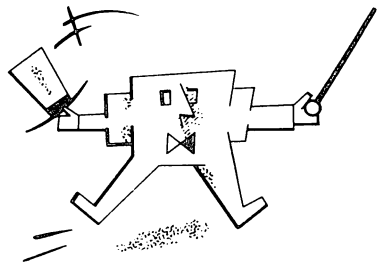


When you are working with the VIC or SID chips, it makes a lot of sense to put the base address into a variable, as we have done, but there is a catch. If, for any reason, the base address is lost, then you finish up POKEing a number into an address at the very start of the system. This could be disastrous. You saw the effect of POKE 1, 4 in the Lock-Up program! There are several ways in which this might happen, so be on your guard.

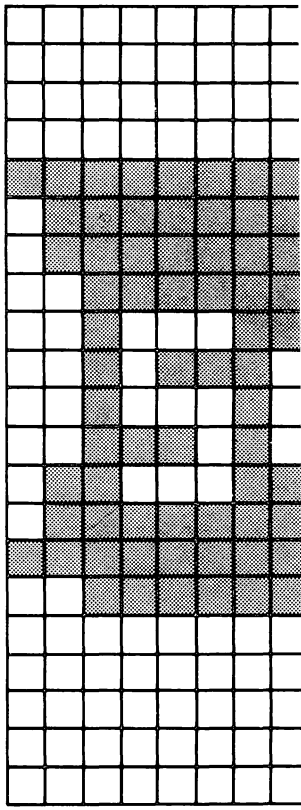
1 Mistyping the variable name. VUC + 1, . . BIC + 1, . . VJC + 1, . . all have the same effect. The 64 assumes you want a new variable called VUC, BIC or VJC and sets it up for you – with a value of 0. Add 1, and there you are with a Lock-Up line. You might find it safer to label the variable V, rather than VIC. There's less room for error.

2 Erasing the line in which the base address is set. It can happen.

3 Starting the program with GOTO, not RUN, and missing the line in which the base address is set. It is perfectly safe to start the program by instructing the 64 to GOTO the first line of a routine that you want to test, as long as the variables are set within the lines, and before they are used. Whenever you make a change to a program all the variables are wiped clear.



Zoom



If you tried the second exercise in the last section, you should have found that the sprites are first fully visible (going downscreen) with a Y value of 50. When Y is 229 the parachutist appears to stand on the bottom edge of the screen area, and he has completely disappeared when Y reaches 250. Let's explore the X values.

The parachutist sprite isn't really appropriate for this – unless you like to imagine that a strong wind is blowing him across the screen. Create a new sprite; one more suited to horizontal motion. The one here is a Class S4 Scout Vehicle of the Andromedan Imperial Space Force. You may prefer to design one of your own. If you do, then record the number set beside the pattern.

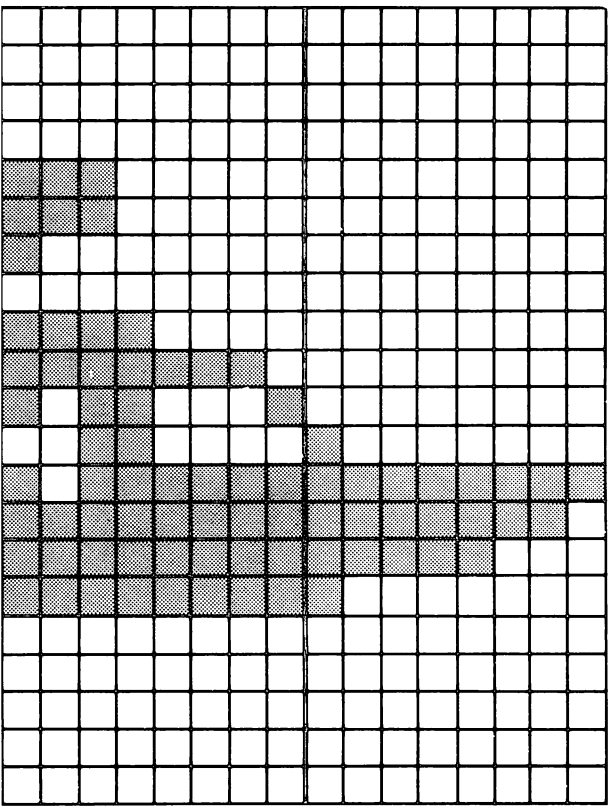
Use a routine like the one in the Parachutist program on page 77 to store your sprite data in the block of memory from 12800, then add these lines.

```
300  POKE 2040,200
310  VIC = 53248
320  POKE VIC +0,0
330  POKE VIC +1,100
340  POKE VIC +39,7    (choose your own colour)
350  POKE VIC +21,1
360  FOR X = 0 TO 255
370  POKE VIC +0,X
380  NEXT X
```

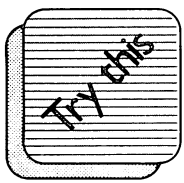
Now run the program and see what happens.

The sprite stops about three-quarters of the way across the screen. The screen is 40 character columns across, which is 320 dots, and the largest number you can POKE is 255. When you want to take something onto the right of the screen, you have to bring another address into play – VIC + 16. This works like VIC + 21. One address manages all the sprites. Add these lines to your program:

```
390  POKE VIC +16,1
400  FOR X = 0 TO 100
410  POKE VIC +0,X
420  NEXT X
430  POKE VIC +16,0
```

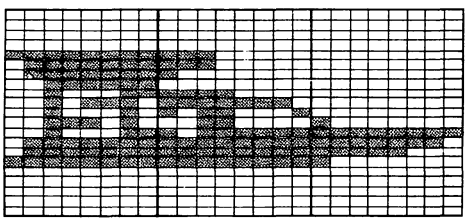


0	0	0
0	0	0
0	0	0
0	0	0
255	224	0
127	224	0
127	128	0
63	0	0
35	240	0
46	254	0
34	177	0
58	48	128
99	191	255
127	255	254
255	255	248
63	255	128
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

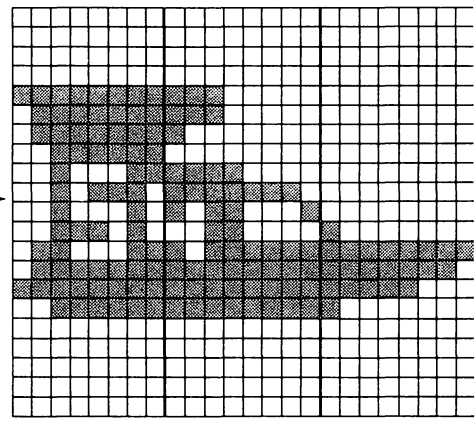


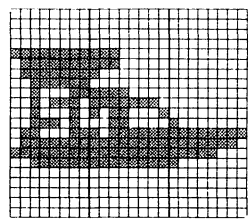
Add lines to slow the action down and print out the X values, so that you can find the points at which the sprite enters and leaves the screen.

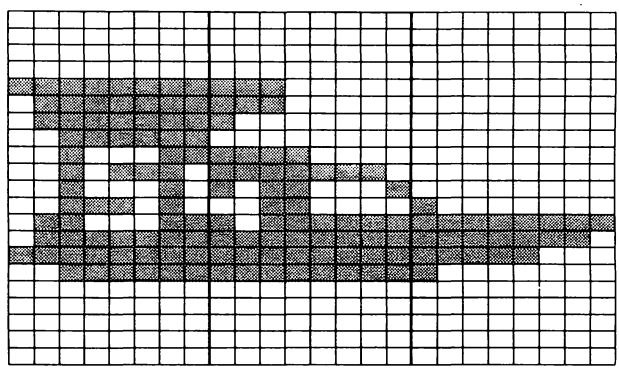
SIZE and SP

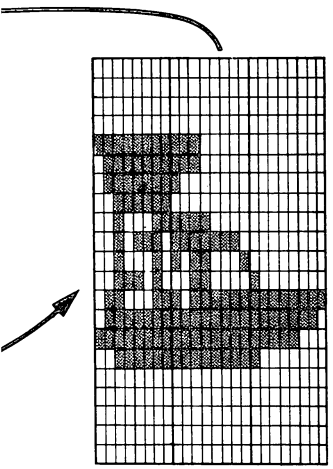


POKE VIC+29,1









POKE VIC+23,1

1 Size

Sprites can be made to appear double length, double height or double size. To enlarge your sprite in the X direction, use this line:

```
POKE VIC + 29,1
```

To enlarge in the Y direction:

```
POKE VIC +23,1
```

Use both lines to create a double size sprite. Both these addresses work the same way as VIC + 21 and VIC + 16. POKE VIC + 23,2 would make sprite 1 double height. If you decide to use a sprite that is to be enlarged in one direction only, then your best approach is to design it on a stretched grid. This will help to keep the proportions right.

2 Speed

You already know how to slow down action by including a delay line in the routine, but how about speeding up?

Normally a FOR. . .NEXT. . . loop works through the range of numbers one at a time. You can alter this by the use of STEP.

```
FOR N=1 TO 15 STEP 2
```

will only go through 1,3,5,7,9,11,13 and 15.

It covers the same range, but misses half the numbers out, thus halving the time it takes to reach the end. The STEP can be any size you like. Change lines 360 and 400 in your program so that they read:

```
FOR X= Ø TO 255 STEP 4
```

The S4 Scout vehicle has found a new burst of speed. Increase the STEP still further and watch the effect.

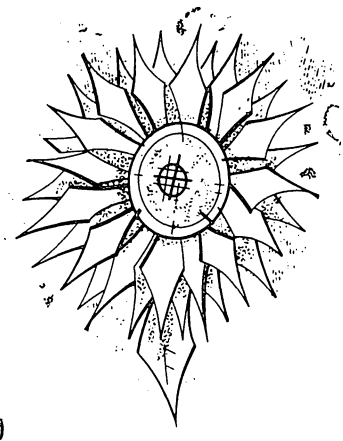
You can also STEP backwards. Your movement does not have to be left to right.

```
FOR X = 255 TO Ø STEP −1
```

This will move the sprite (at normal speed) from right to left.

Multi-SPR

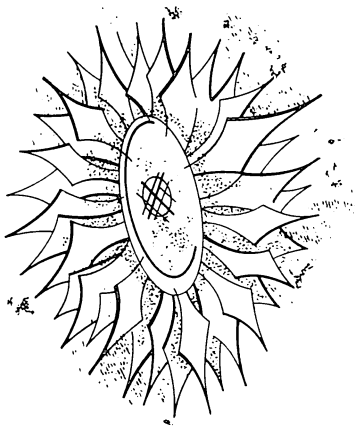
It is possible to use three different colours on the same sprite. This program shows how. It creates a sprite in the shape of a flower with a red and yellow head, and green leaves. The sprite has also been made double sized so that you can see it better.



Program

```
  5  PRINT "■"
 10  FOR N= 12800 TO 12862
 20  READ B: POKE N,B
 30  NEXT N
 40  DATA 2,138,0,10,170,128,10,170,
     128,10,170,128,10,186,128,10,254,
     128,2,254,0
 50  DATA 10,254,128,10,186,128,10,
     170,128,74,154,128,82,154,1,16,
     16,5,20,16,20
 60  DATA 20,16,84,21,16,84,21,17,
     84,21,81,84,5,85,80,1,85,64,0,84,0
100  POKE 2040,200
110  VIC = 53248
120  POKE VIC+0,100:POKE VIC+1,100
130  POKE VIC+39,7
140  POKE VIC+37,5
150  POKE VIC+38,2
160  POKE VIC+28,1
170  POKE VIC+23,1: POKE VIC+29,1
180  POKE VIC+21,1
```

coloured ITES



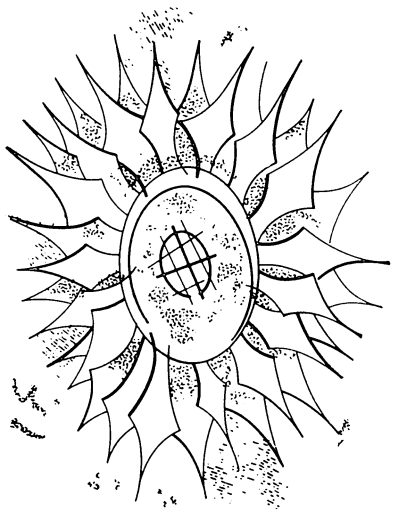
How it works

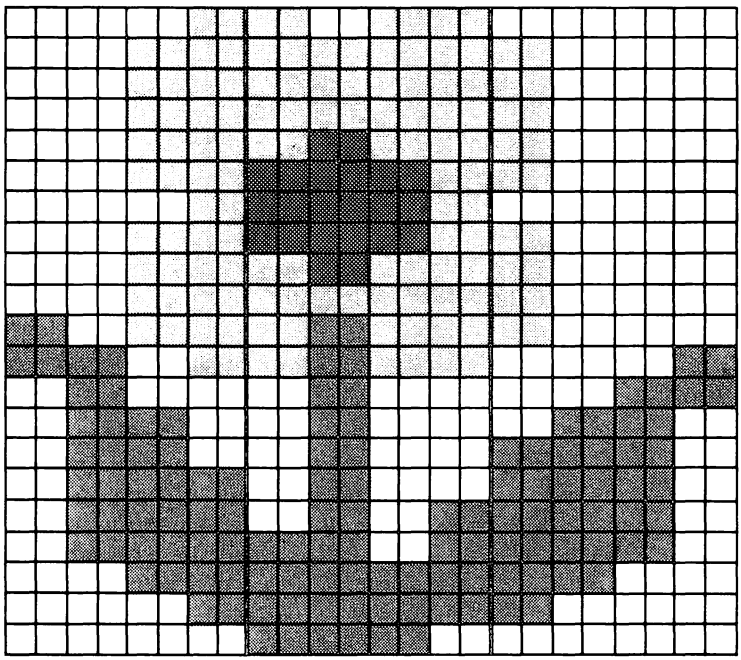
Lines 40 to 60 carry all the data for the sprite. Squeezing the pattern into a few lines like this saves typing and saves memory space, but it is not quite so easy to spot your mistakes and correct them. Using a separate DATA line for each row of pattern numbers is a safer but slower way to enter your data.

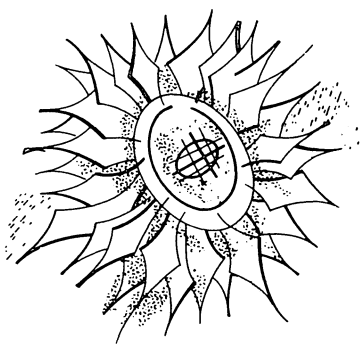
Lines 130 to 160 set the colours. VIC + 39 is the normal sprite colour. VIC + 37 is multi-colour 1, and VIC + 38 is multi-colour 2. POKE VIC + 28,1, turns on the multi-colour mode for sprite 0.

Type in the program and run it. Look at the sprite carefully and you will notice that the outline is not as smooth as it has been on other sprites. This is still true if you reduce it back to its normal size by POKE VIC +23,0: POKE VIC +29,0.

When a sprite is in multi-colour mode, the smallest possible unit that you can define is two dots wide. A single dot (or rather a single BIT) is enough to tell the VIC chip whether to colour a point or leave it blank. In multi-colour mode there are four possibilities for each part of the sprite – leave blank, use multi-colour 1, use multi-colour 2 or use sprite colour. It takes a pair of BITs to carry this information.



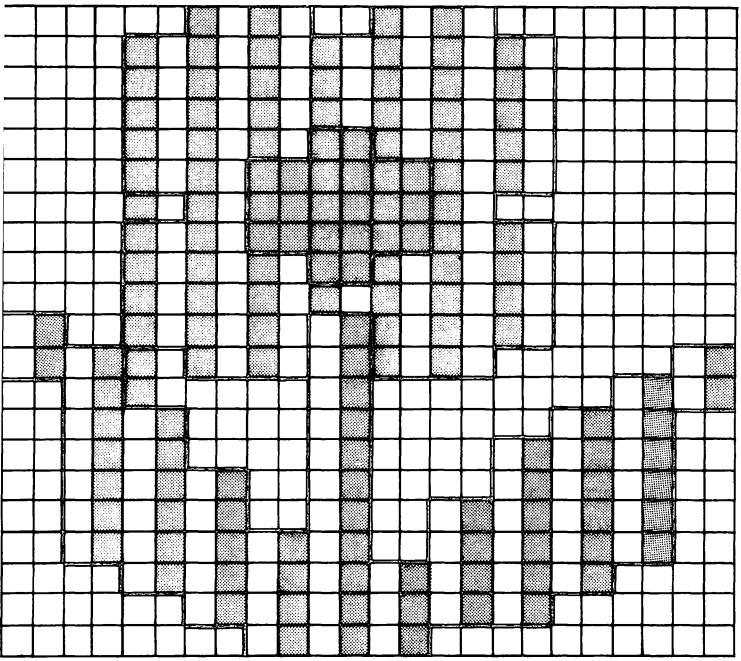




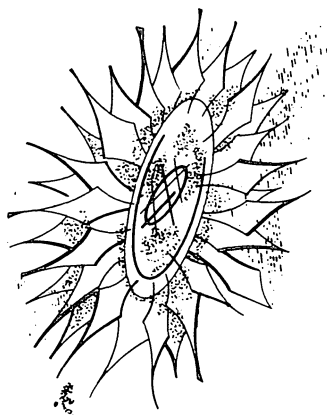
MULTI-COLOUR 1	POKE VIC+37,5
SPRITE COLOUR	POKE VIC+39,7
MULTI-COLOUR 2	POKE VIC+38,2
BACKGROUND	

If both BITs in the pair are 0, then the corresponding two dots will be left blank. If the BIT on the right of the pair is 1, but the other 0, then both dots will be given the colour set by multi-colour 1. When the left BIT only is 1, the two dots are sprite colour. If both bits are 1, then both dots become multi-colour 2.

You need a slightly different grid from normal when designing multi-coloured sprites. Draw your vertical lines so that you have twelve columns of two dots each. When you are first



2	138	0
10	170	128
10	170	128
10	170	128
10	186	128
10	254	128
2	254	0
10	254	128
10	186	128
10	170	128
74	154	128
82	154	1
16	16	5
20	16	20
20	16	84
21	16	84
21	17	84
21	81	84
5	85	80
1	85	64
0	84	0

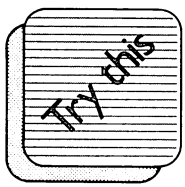


working with multi-colour mode, you will find it helpful to draw up two grids for each sprite. On one you will design the coloured image, and then shade it in. The second grid is used to translate the colours into dot-pair patterns. Copy the outlines of your design onto this, and then work through each colour area, shading in the appropriate dot or dots to give the colour you want there. This dot pattern can then be changed into numbers in the usual way.

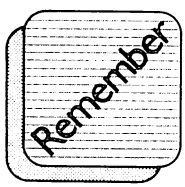
Multi-coloured multiple sprites

When you set the multi-colours, you set them for all the sprites though the sprite colours can be changed as usual. Add these lines to your program. They will put two more flowers on the screen, one of them blue, the other pale green. Both the new flowers have red centres and green leaves, because these are multi-colours.

```
100  POKE 2040,200:POKE
     2041,200:POKE 2042,200
...
121  POKE VIC+2,150:POKE VIC+3,100
122  POKE VIC+4,200: POKE VIC+5,100
...
130  POKE VIC+39,7:POKE VIC+40,14:
     POKE VIC+41,13
...
160  POKE VIC+28,7
170  POKE VIC+23,7: POKE VIC+29,7
180  POKE VIC+21,7
```



Add more lines to fill the screen with brightly coloured flowers. Try different colours for the centres. (Multi-colour 2). Red does not combine very well with some colours.



You will usually have to turn your C64 off to return to normal – before you do, remember to SAVE the program.

Sprite look up tables

SPRITE	SPRITE POINTER	X POSITION 0–255	Y POSITION 0–255	COLOUR
0	2040	VIC+0	VIC+1	VIC+39
1	2041	VIC+2	VIC+3	VIC+40
2	2042	VIC+4	VIC+5	VIC+41
3	2043	VIC+6	VIC+7	VIC+42
4	2044	VIC+8	VIC+9	VIC+43
5	2045	VIC+10	VIC+11	VIC+44
6	2046	VIC+12	VIC+13	VIC+45
7	2047	VIC+14	VIC+15	VIC+46

Multi colour sprites

POKE VIC+37, colour code, multicolour 1
POKE VIC+38, colour code, multi-colour 2
POKE sprite colour as normal, then turn on multi-colour mode (VIC+28)

Single address controls

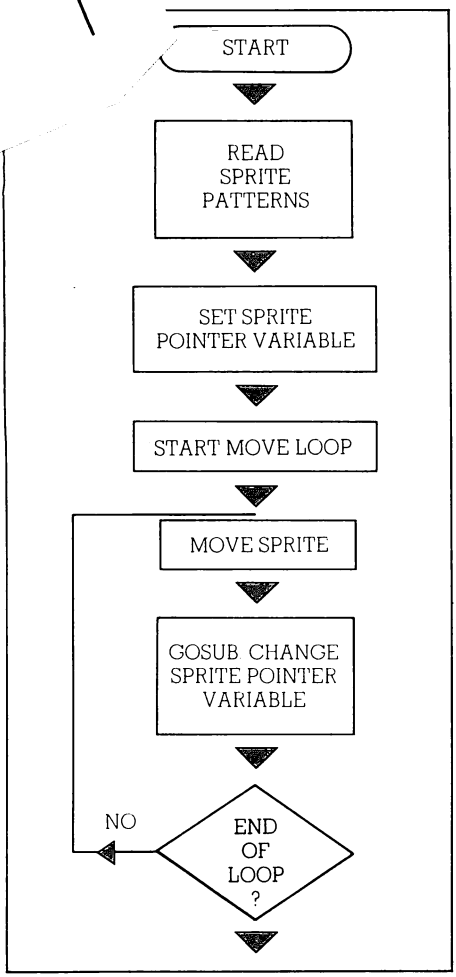
ENLARGE X ↔	VIC+29
ENLARGE Y ↕	VIC+23
SPRITE ON FAR RIGHT	VIC+16
MULTI-COLOUR MODE	VIC+28
TURN ON SPRITES	VIC+21

The number POKE'd here should be the sum of the codes of the sprites that you want to affect.

Thus, POKE VIC+23,12 would enlarge upwards sprites 2 and 3.

Sprite codes

NUMBER	0	1	2	3	4	5	6	7
CODE	1	2	4	8	16	32	64	128



COMPUT

You saw in the section on graphics how to animate your pictures by printing blocks of graphics. Sprites allow for even better animation effects. You can get smoother movement, and more detailed changes in your figures. The next program is fairly complex, so we will take it in stages.

The first stage is to get these two designs into the memory. You could do this by writing two READ loops:

```
FOR N=12800 TO 12862
READ B: POKE N,B
NEXT N
FOR N= 12864 TO 12926
READ...etc.
```

You can also do it by creating a second loop around the READ loop.

New idea

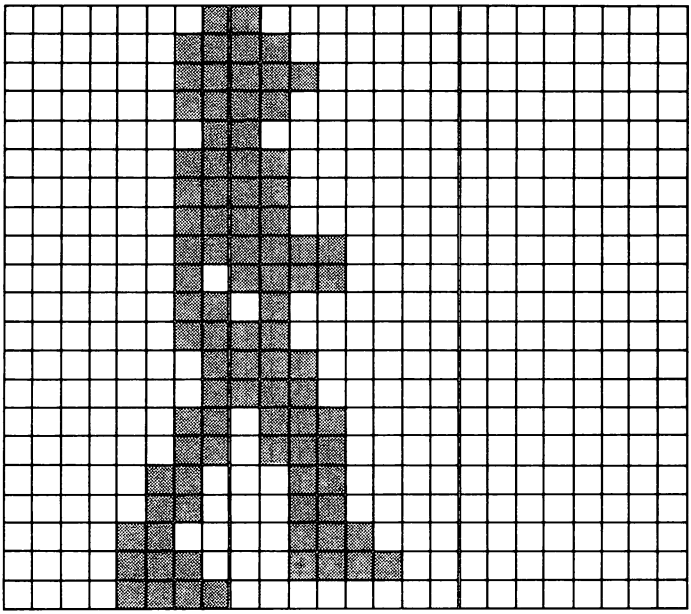
```
FOR T=0 TO 1
FOR N=0 TO 62
READ B: POKE 12800 +
64*T +N,B
NEXT N,T
```

How it works

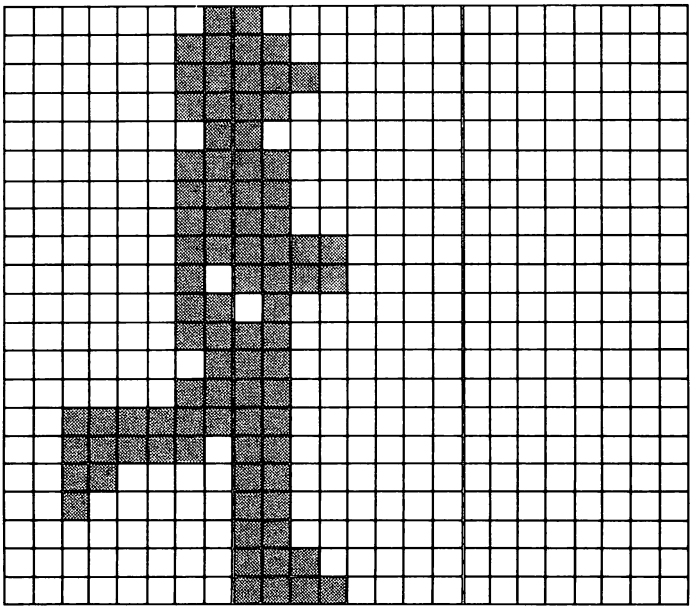
The T loop makes the program go through the N loop two times. That complex expression in the POKE statement works out the address. The very first number, when T =0 and N=0, will go to address 12800 (=12800 + 64*0 +0). The next goes to 12801 (=12800 + 64*0+1). When it comes to the data for the second pattern, the formula gives 12864 as the first address. (12800 + 64 *1 +0)

It may seem a lot of trouble to take for the sake of saving the odd line, but in fact, this is a very compact piece of programming. You can use this for any number of sprites. The only change you have to make is to the number in the first line.

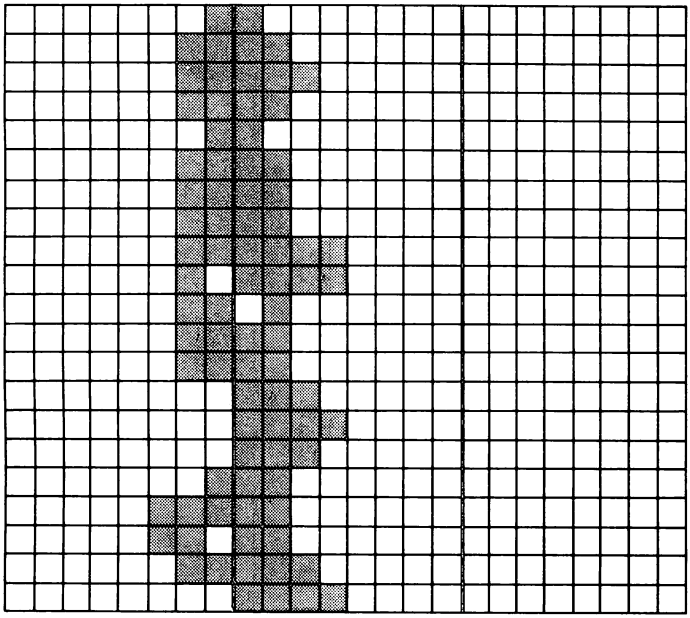
ER ANIMATION



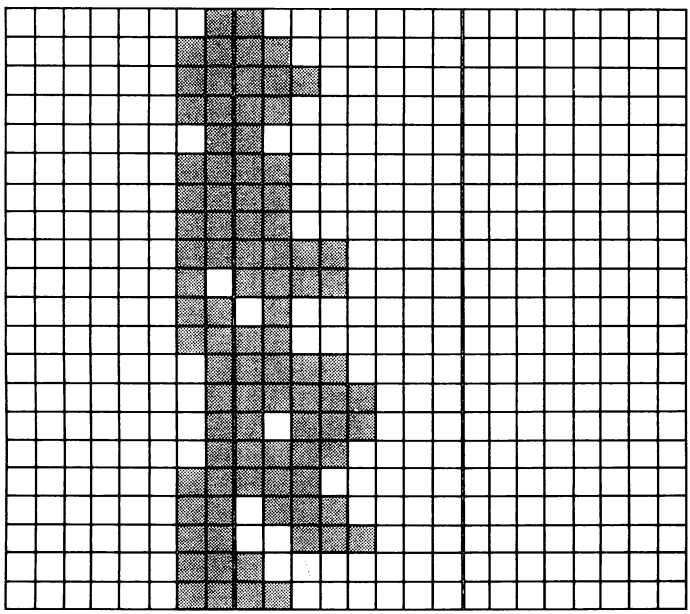
1	128	0
3	192	0
3	224	0
3	192	0
1	128	0
3	192	0
3	192	0
3	192	0
3	240	0
2	240	0
3	64	0
3	192	0
1	224	0
1	224	0
3	112	0
3	112	0
6	48	0
6	48	0
12	56	0
14	60	0
15	0	0



1	128	0
3	192	0
3	224	0
3	192	0
1	128	0
3	192	0
3	192	0
3	192	0
3	240	0
2	240	0
3	64	0
3	192	0
1	192	0
3	192	0
63	192	0
62	192	0
48	192	0
32	192	0
0	192	0
0	224	0
0	240	0



1	128	0
3	192	0
3	224	0
3	192	0
1	128	0
3	192	0
3	192	0
3	192	0
3	240	0
2	240	0
3	64	0
3	192	0
3	192	0
0	224	0
0	240	0
0	224	0
1	192	0
7	192	0
6	192	0
3	224	0
1	240	0



1	128	0
3	192	0
3	224	0
3	192	0
1	128	0
3	192	0
3	192	0
3	192	0
3	240	0
2	240	0
3	64	0
3	192	0
1	240	0
1	248	0
1	184	0
1	240	0
3	224	0
3	112	0
3	56	0
3	128	0
3	192	0

Here's the rest of the program. Notice that the address for the sprite pointer is held in the variable SP, and that the routine which changes SP is written as a subroutine.

```
300  SP=200: POKE 2040,SP
310  VIC = 53248: POKE VIC +0,0:
     POKE VIC+1,100
320  POKE VIC +39,0:POKE VIC +23,1:
     POKE VIC+29,1:POKE VIC+21,1
330  FOR X=0 TO 255 STEP 4:
     POKE VIC+0,X
340  GOSUB 500 : POKE 2040,SP
350  FOR D=1 TO 150: NEXT D
360  NEXT X
370  END

500  IF SP=200 THEN SP=201: RETURN
510  IF SP=201 THEN SP=200: RETURN
```

Each time the program goes through the loop at lines 330 to 360, the sprite is moved four dots forward and the sprite pointer variable is changed so that it appears that the legs are actually moving. The delay time can be increased so that you can see the individual moves more clearly.

Projects

1 The movement is rather jerky at the moment. This is because there are only two pictures per step. Walt Disney cartoons normally have sixteen frames (separate pictures) for every second. Smooth the movement by adding the two sprite patterns shown here. If you used the 'compact' READ loop, you will need to change the first line to:

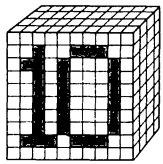
```
FOR T=0 TO 3
```

Add in your extra DATA lines. This can be done quite quickly by renumbering and editing the existing lines. The first twelve rows of all these patterns are the same.

Alter the SP subroutine to this:

```
510  IF SP=201 THEN
     SP=202: RETURN
520  IF SP=202 THEN
     SP=203:RETURN
530  IF SP=203 THEN
     SP=200:RETURN
```

2 Add a further routine so that the man walks right across the screen and off the far side. You must POKE VIC + 16,1 to do this. Loop the whole of that routine so that he starts off again from the left. POKE VIC + 16,0 before he starts again.



Advanced BASIC

HANDLING DATA

For some years, big businesses have been using computers to store and process their accounts and files. Now that computers are so much cheaper and more efficient, an increasing number of smaller businesses, and even clubs and societies are acquiring their own machines. What do they use them for?

In business, the first thing to computerise is the wage packets. Calculating wages is a matter of strict routine and simple arithmetic – at which computers excel. The routine to calculate the gross weekly wage needs only to take the hourly rate and multiply it by the number of hours worked. Calculating tax, overtime and bonuses is only a little more complicated.

```
1000  BLOGGWAGE=BLOGGRATE*BLOGGHOURS
1010  BROWNWAGE=BROWNRATE*BROWNHOURS
1020  GREENWAGE=GREENRATE*GREENHOURS
.....
```

This program could get very long if the firm was of any size. There has to be a more efficient way of doing it and there is. The answer is to use ARRAYS.

An array is a set of memory stores, all with the same name, but with different reference numbers. Here, the EMPLOYEE$ array holds the names of all the workers. The RATE, HOURS and GROSS arrays hold the necessary figures. EMPLOYEE$(1) is Mr Bloggs, RATE(1) is 4.00, HOURS(1) is 40 and WAGE(1) is 160.00. The computer can handle all of the staff wages with a simple looped program.

```
1000  FOR N=1 TO 5
1010  LET WAGE(N)=RATE(N)*HOURS(N)
1020  PRINT "WAGE FOR ";
      EMPLOYEE$(N);"=";WAGE(N)
1030  NEXT N
```

Remarks to remind you

Readability is very important in programs. You must be able to read through your program listing and understand what is going on. This isn't too much of a problem when you are writing the program because everything is still fresh in your mind. You know what all the variable names mean, and how they link together. You know what each part of the program does. But will you still know when you come back to it in a few months' time?

The REM statement allows you to write remarks in the program list. You can write anything you like after a REM. The 64 will ignore it all.

```
100  LET A= 6
101  REM NUMBER OF
     ALIENS
...  ...
300  IF B$ = "H" THEN
     GOTO 4000
310  REM HELP PAGE AT
     4000
```

Arrays give you a very flexible and powerful way of handling data. Suppose, for example, that you wanted to know which employee had earned most in that week. When there are only five names on the list, it is very simple to scan the table and pick up the highest figure in the WAGE column, but what if the firm had a hundred, or a thousand employees? The computer could soon find the answer for you.

```
3000  BASE=0: MOST=0
3010  FOR N=1 TO 5
3020  IF WAGE(N)>BASE THEN MOST=N:
      BASE=WAGE(N)
3030  NEXT N
3040  PRINT MOST
```

This starts by opening two stores, BASE and MOST. The MOST store will be used to collect the reference number of the highest WAGE figure. The BASE is used for comparison. On the first run through the loop, it finds that Blogg's WAGE is higher than the BASE figure (0), so it changes the BASE to 160, and stores 1 in MOST. Next time round, it discovers that Brown's WAGE is higher than 160, and it changes the BASE, and MOST stores again. Green's figure is lower than the BASE, so the computer moves on to Lewis' WAGE. This is higher than the 201 now stored in BASE, and the two checking stores are reset. The final WAGE is lower, so nothing happens that time.

ARRAYS

EMPLOYEE	RATE	HOURS	WAGE	1 . . .
Bloggs, F. D.	4.00	40	160.00	
Brown, T. S.	6.30	32	201.60	
Green, I.	4.00	35	140.00	
Lewis, B. S.	8.00	30	240.00	
Prior, J.	3.60	40	144.00	

	EMPLOYEE$	RATE	HOURS	WAGE
1	Bloggs, F. D.	4.00	40	160.00
2	Brown, T. S.	6.30	32	201.60
3	Green, I.	4.00	35	140.00
4	Lewis, B. S.	8.00	30	240.00
5	Prior, J.	3.60	40	144.00

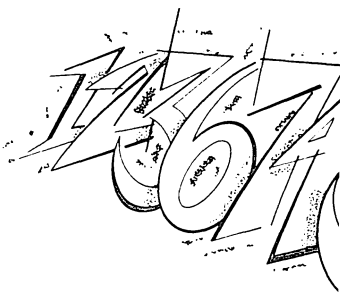
At the end of the loop, the computer has 4 in the MOST store. It can pick out the name of the highest paid worker by using this as a reference number in the EMPLOYEE array. EMPLOYEE$(4) is 'Lewis'.

Arrayed variables can be handled just like ordinary variables as well. They can be changed by LET and INPUT commands. At the end of the year, the boss of our little business may decide to do a salary review, rewarding his best workers with improved hourly rates.

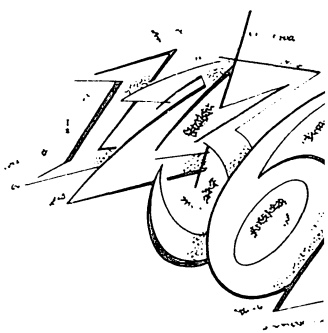
```
7000  REM SALARY REVIEW: RATE
      RESET
7010  INPUT "EMPLOYEE'S REFERENCE
      NUMBER ?";ERN
7020  PRINT " NAME IS ";
      EMPLOYEE$(ERN)
7030  INPUT "NEW RATE ?";RATE(ERN)
```

ARRAY

Arrays can be for strings, integers (whole numbers) or real numbers. A single array can have as many elements as the memory will allow. This means you could store about 8000 real numbers, or 20000 integers or 40000 characters.



1 You must tell the 64 what size of array you want using the DIM (DIMension) command. DIM GAME$(5) sets up a string array with 5 stores. DIM N(2000) sets up a number array to hold a maximum of 2000 numbers. The DIM command must only be given once – it's no good deciding you want to make it bigger in the middle of a program.

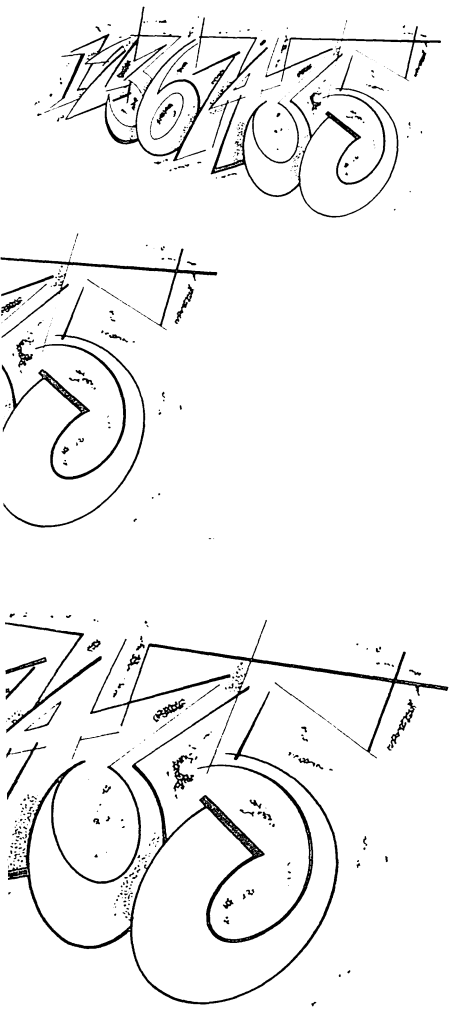


2 Exception. If you only want small arrays – eleven stores or less – then you can miss out the DIM command, and the 64 will set up the array for you, just as it will set up variables if you miss out the LET. . . line. Missing out your DIM lines is not a good habit to get into.

3 The same rules apply to array names that apply to variable names. Start with a letter; don't start two with the same two letters; end with a $ if it's a string array, and don't include any BASIC words in the name.

4 The array reference number must be a whole number. All arrays start from 0 and the highest number is fixed by the DIM line. DIM A(6) creates seven stores A(0),A(1). .A(6).

RULES



W$(5) ARRAY	
W$(0)	"HELLO"
W$(1)	"GOODBYE"
W$(2)	"HOW ARE YOU?"
W$(3)	"IT's 5.45 pm"
W$(4)	" "
W$(5)	" "

N(12) ARRAY	
N(0)	3.33
N(1)	297
N(2)	43.2
N(3)	1
N(4)	0
N(5)	52673
N(6)	66
N(7)	12.532
N(8)	778965321
N(9)	0
N(10)	0
N(11)	0
N(12)	0

Highest and

This program uses the same kind of routine as the WAGES sorter. When you run it, you will be asked to type in eleven numbers, one at a time. Use any numbers you like, and in any order that you fancy. The program will sort through and pick out the highest and the lowest.

Program

```
  5  PRINT "♥"
 10  DIM N(10)
 20  HBASE =0: HI =0
 30  LBASE =999: LO=0
 40  FOR T=0 TO 10
 50  INPUT "NUMBER ";N(T)
 60  NEXT T
 70  FOR T=0 TO 10
 80  IF N(T)>HBASE THEN HBASE
     =N(T):HI =T
 90  IF N(T)<LBASE THEN LBASE =
     N(T): LO=T
100  NEXT T
110  PRINT "HIGHEST ";N(HI)
120  PRINT "LOWEST ";N(LO)
```

It doesn't much matter what value you give to LBASE at the start, as long as it is bigger than at least some of the numbers that it will find in the array.

Just to show how flexible this array handling is, add these few lines to the program, and it will calculate the total and average as it runs through the loop:

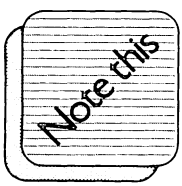
```
 35  SUM=0
 95  SUM =SUM +N(T)
130  PRINT "TOTAL =";SUM
140  PRINT "AVERAGE =";SUM/11
```

(You couldn't call the variable TOTAL, as it includes TO)

owest

"HIGH" & "LOW" STRINGS
"A" < "B" "AARDVARK" < "ANTELOPE" "MARGARINE" > "BUTTER" "999" > "1000000"

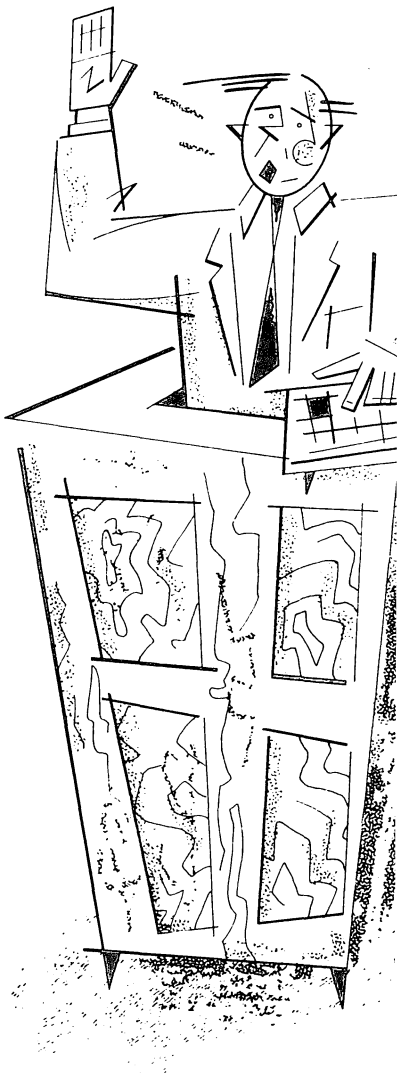
You can compare strings as well as numbers, but with these 'highest' and 'lowest' refers to their ASCII codes, so the comparison is essentially alphabetical. Adapt the program to take in a list of names and find the first and last. Change the array name to N$(10), HBASE=0 should become HBASE$="AAA"; LBASE=999 becomes LBASE$="ZZZ". Change all the variables names in the same way, and alter the printed comments to something more suitable. Try the program with a variety of different words and names. You will find that the comparison works on the whole word, not just the first letter. "ANIMAL" is lower than "ANT" because "I" (the third letter, and the last in ANT) has a higher ASCII code than "I". If you compare numbers inside string variables, then you can get odd results, as it is the ASCII codes that count here, not the values of the numbers.



For a full list of ASCII codes, see Appendix F in your User Manual.

TRUTH AND LOGIC

Truth



Computers value truth – don't we all – but being computers they value it in terms of numbers. A false statement is worth nothing – a true statement is worth −1. You can see this by typing:

```
X=99: PRINT X=99
```

Now type:

```
PRINT X=66
```

What does it print this time?
The evaluation works on strings as well.

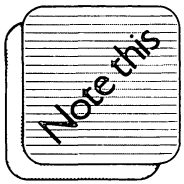
```
A$="TEST": PRINT A$="TEST"
```

Did it print '−1'?
You can use the value of truth idea as a way of compressing IF. . .THEN. . . lines. Suppose you wanted to control the movement of a sprite, so that its X position was increased when the player pressed '2', and decreased when he pressed '1'. You could use a pair of IF. . .THEN. . . lines.

```
100  GET A$ : IF A$="" THEN 100
110  IF A$ ="1" THEN X=X-1
120  IF A$="2" THEN X=X+1
...
```

The two lines 110 and 120 could be replaced by this:

```
110  X=X+(A$="1")-(A$="2")
```



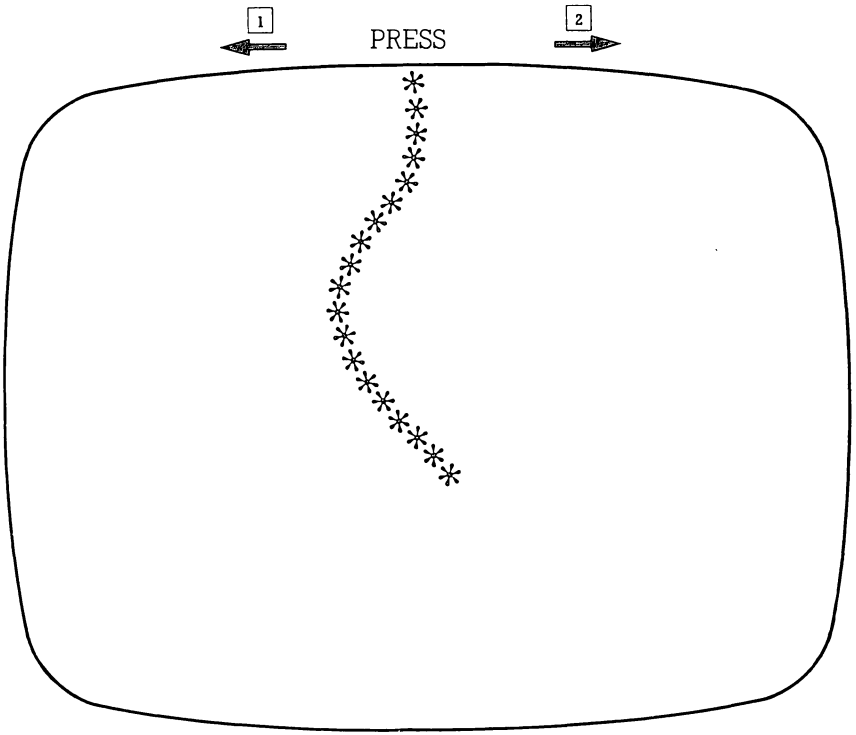
As the truth function gives you −1 for a true statement, you have to reverse this with the sign outside the bracket. That line takes 1 from X when A$ = '1' and adds when it is '2'.

If you want to change X in steps of more than 1, then include a multiplier in the line. Here the keystroke will add or take 4.

```
110  X=X+(A$="2")*4  -(A$="1")*4
```

This little routine shows truth at work.

```
10  PRINT " ♥ "
20  X=20
30  PRINT TAB(X);"*"
40  GET A$
50  X=X+(A$="1")*2-(A$="2")*2
60  GOTO30
```



Logic

The Commodore 64, like most computers, has LOGICAL OPERATORS. These are another means of testing truth.

AND checks to see if two statements are true.

OR compares two statements to see if either, or both are true.

Add this line to the last program:

```
55  IF X<=8 OR X>=30 THEN GOTO 40
```

If the value of X reaches either of the limits then the program will loop back to miss the PRINT line.

Write in a line of your own so that the program will stop if either '3' or '4' is pressed.

Add another line. This shows the use of AND. It tests for the two conditions – X must be less than or equal to 10 AND A$ must be '5' to make the program stop.

```
53  IF X<=10 AND A$="5" THEN STOP
```

You can combine AND and OR, but think the line through carefully first, and test it afterwards to make sure it does what you want. This line ought to let you stop the program by pressing '5' when the X value is either 10 or less, or 30 or more.

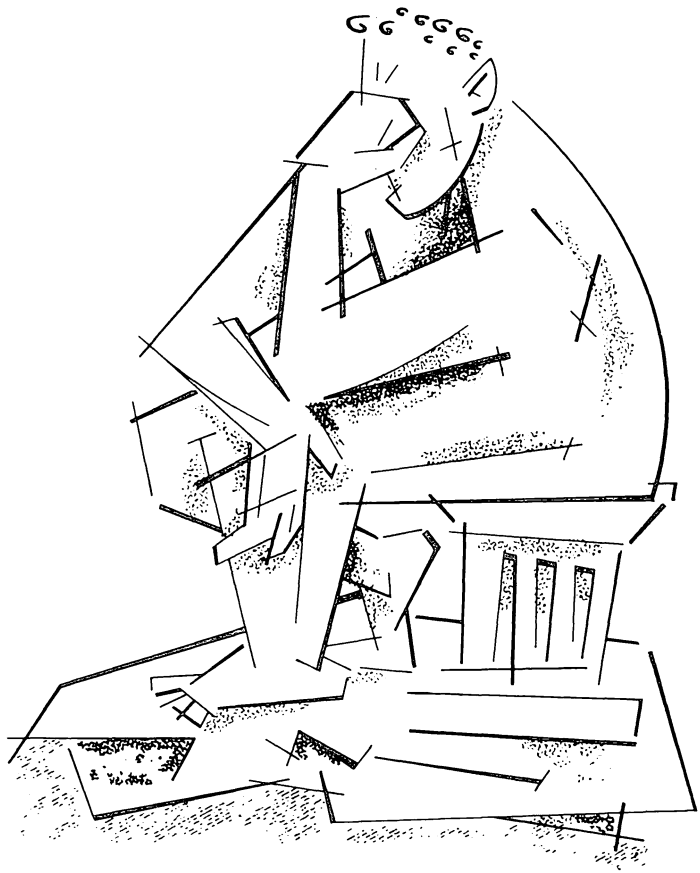
```
53  IF X<=10 OR X>=30 AND A$="5"
    THEN STOP
```

Test it. What happens when you push the asterisk to the left limit? Run it again and push it to the right. This line actually tests for two conditions – either X<=10 (and nothing else) OR X>=30 AND A$="5". This wasn't quite what you wanted.

Enclose the first part of the line in brackets. That way it will test the two values of X before it looks at the A$.

```
53  IF (X<=10 OR X>=30) AND
    A$="5" THEN STOP
```

COMPARISONS	LOGIC
=equals <>is not equal to >=is more than or equal to <=is less than or equal to If true −1 If false Ø	AND — are both statements true? OR — is either true? are both true?



Bubble sorts

Have you ever played those games where there is a 'Roll of Honour'? If your score is high enough, then you are included in the list, and the list is sorted into order each time there is a new entry.

There are several ways in which you can sort a list of numbers, or strings, and of these, the Bubble Sort technique is probably the easiest to explain. It is similar to the Highest – Lowest routine. Type it in and enter a set of eleven assorted numbers.

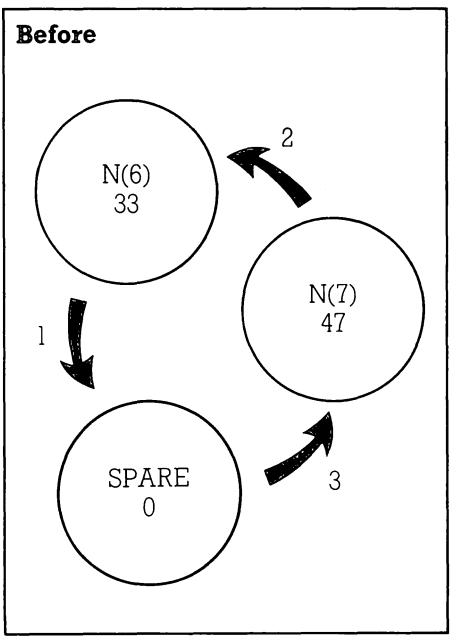
```
 10  DIM N(10)
 20  FOR T=0 TO 10
 30  INPUT "NUMBER ";N(T)
 40  NEXT T
 50  SPARE =0
 60  FOR T=0 TO 9
 70  IF N(T)<N(T+1) THEN
     SPARE =N(T):
     N(T)=N(T+1):
     N(T+1)= SPARE
 80  NEXT T
 90  IF SPARE <> 0 THEN
     50
100  FOR T=0 TO 10
110  PRINT N(T)
120  NEXT T
```

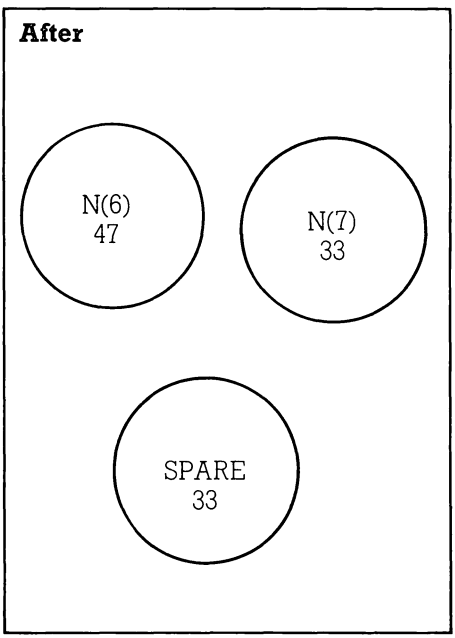
ROLL OF HONOUR

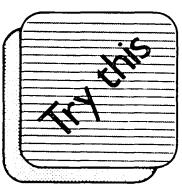
GRANDAD	50000
EUGENIUS	40000
BILL	35240
BAD TED	30000
FRED	26540

How it works

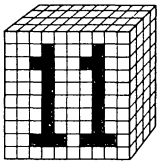
This sorts the numbers into order, highest first. Look at one trip through the loop, when T=6. Let's suppose that N(6)=33 and N(7)=47. N(6) is less than N(6+1) so it reorders the pair. The 33 from N(6) is put into the SPARE store for safe-keeping. N(6) then takes the 47 from N(7), and N(7) gets its new value – 33 – from the SPARE. On any one trip through the loop several pairs of numbers may be shuffled in this way. At the end of the loop, the computer checks to see if the SPARE store has been used. If it has, then it goes back and runs through them again. There will come a time when all the numbers have been sorted into order, and on the next trip through the loop, the SPARE will not be changed from the 0 given to it at line 50. This routine will not sort a list that includes a 0. Why not?





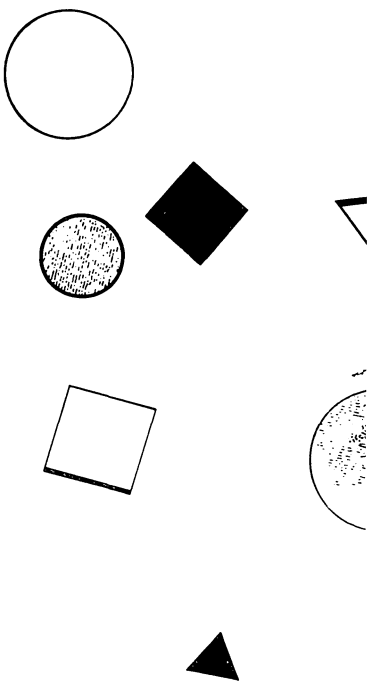


Alter the bubble sort program so that it will sort a set of names into order. You will need to make the same sort of changes here that you did when making the Highest – Lowest program into an alphabetical one.



Games

Poke a picture



The PRINT statement is not the only way to get characters on the screen. You can also POKE characters directly into the screen memory. The 64 keeps a track of what's on the screen in a 1000-byte block of memory starting at address 1024. Try it.

```
POKE 1024,81: POKE 1024+54272,0
```

This will make a solid circle appear in the top left corner.

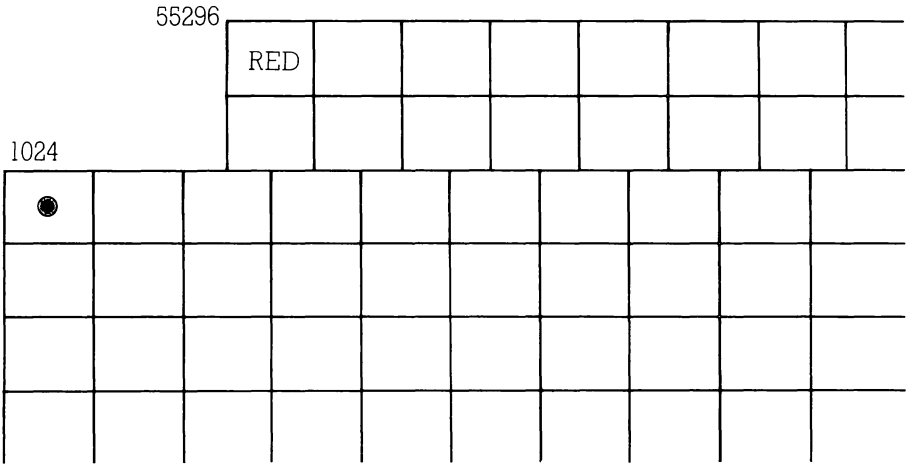
The memory block holds the screen information in logical order. The numbers run from left to right, working down from the top.

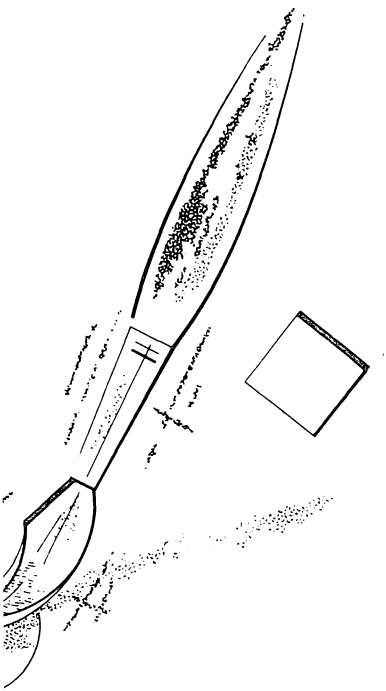
```
POKE 1025,81: POKE 1025+54272,0
```

This puts a second blob to the right of the first. Add 40 to move down a line.

```
POKE 1065,81: POKE 1065+54272,0
```

A third blob appears below the second.





Try this program if you want to see the way the screen fills, and the characters that are available. It works through the screen and through the character set at the same time. The screen is run through the N loop. The character codes are held in X.

```
 5  POKE 53281,1
10  X=0
20  FOR N=1024 TO 2023
30  POKE N,X
40  X=X+1
50  IF X=256 THEN LET X=0
60  NEXT N
```

Type it in and run. Now press the Commodore and SHIFT keys together to see the second character set.

The codes used in screen POKEs are not the same as the ASCII codes. Look them up in Appendix E of the User Manual when you want to use them.

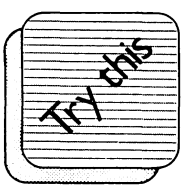
POKE A

The screen memory only holds the character codes. It knows nothing about colour, as this is stored in a separate block of memory starting at 55296. To put a red blob in the top left, using POKEs, you would need this.

```
POKE 1024,81: POKE 55296,2
```

The colour codes used here are the same as the colour codes for the screen and border POKEs.

Having to calculate both screen and memory positions could be hard work, and it certainly leaves lots of room for error, but fortunately there is a simple way to make sure that you colour the right square. The colour memory follows the same pattern as the screen memory, and the difference between the screen address and the colour memory address for any square is always 54272.

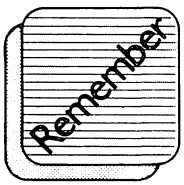



```
POKE 1500,42: POKE 1500+54272,1
```

This should give you a white asterisk about halfway down the screen. Using a variable to hold the screen address saves a little typing in the long run, and reduces the chances of error even more.

```
P=1600:POKE P,42:POKE P+54272,0
```

COLOUR



Do take care with your screen POKEs. If you mistype and POKE above the screen, or below the screen, then you will have problems.

```
POKE 1000,42
```

This will put a number in the tape buffer! Other low numbers can cause chaos. Those first 1000 bytes of memory hold vital information about the operating system.

```
POKE 2088,42
```

Now you have gone off the bottom of the screen, and the POKE has finished up in your BASIC program.

Look for the lines in the MINEFIELD program that prevent any accidents like these.

Characters and codes

Computers can only understand numbers. Letters have no meaning to them until they have been converted to numbers. Fortunately for you and me, they do this themselves, but sometimes you will want to use those code numbers directly.

In Appendix F of the User Manual there is a full list of the ASCII codes – the code numbers of the characters. If you want to know the code of a character, look it up here – or try this. It finds the code of the letter 'A'.

```
PRINT ASC("A")
```

The computer prints 65. Try another letter, or a number of a graphic. The character must be inside quotes, inside brackets.

You can reverse the process with the CHR$ function. Try it.

```
PRINT CHR$(65)
```

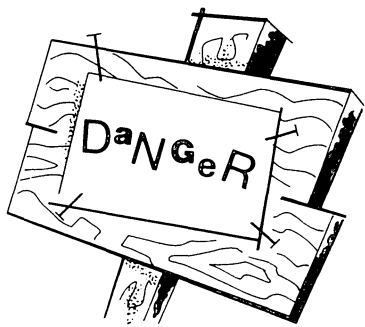
What do you get?

The ASCII codes apply to more than just characters. Anything which can be written into a PRINT line has an ASCII code.

```
PRINT
CHR$(18);CHR$(30);CHR$(65)
```

This turns Reverse on, (18), changes the colour of the ink to green (30) and prints the letter 'A'.

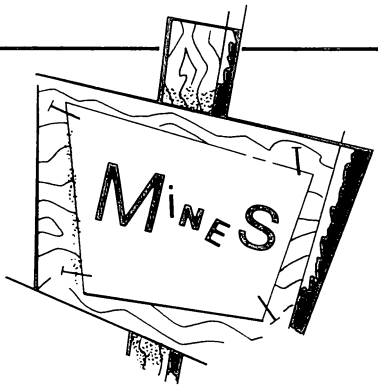
MINEFIELD

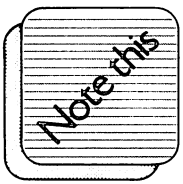


Here's a game that uses the screen and colour memories. The object of the game is to reach the bottom right corner, without treading on a mine. It's a bit bare at the moment, so that you can see the main routines more clearly. When you've got it running, you might like to improve the presentation and add some sound effects. I have used asterisks (code 42) to indicate mines, and a blob (code 81) to show the player's position.

Program

```
 10  POKE 53281,7:PRINT " ♥ "
 20  FOR N=1 TO 50: P=
     INT(RND(0)*1000)+1024
 30  POKE P,42: POKE P+54272,7:
     NEXT
 40  PL = 1024: LIVES = 5
 50  POKE PL,81:POKE PL+ 54272,1
 60  GET A$ : IF A$<>"" THEN 60
 70  GET A$ : IF A$=""THEN 70
 80  IF A$="U" AND PL>1064 THEN
     PL=PL-40
 90  IF A$="D" AND PL<1984
     THEN PL=PL+40
100  IF A$="L" AND PL>1025
     THEN PL=PL-1
110  IF A$ ="R" THEN PL=PL+1
120  IF PEEK(PL)=42 THEN 200
130  POKE PL,81: POKE PL+ 54272,1
140  IF PL = 2023 THEN 300
150  GOTO 60
```





Lines 20 and 30 put 50 asterisks on the screen at random places, but colour them yellow, the same colour as the screen.

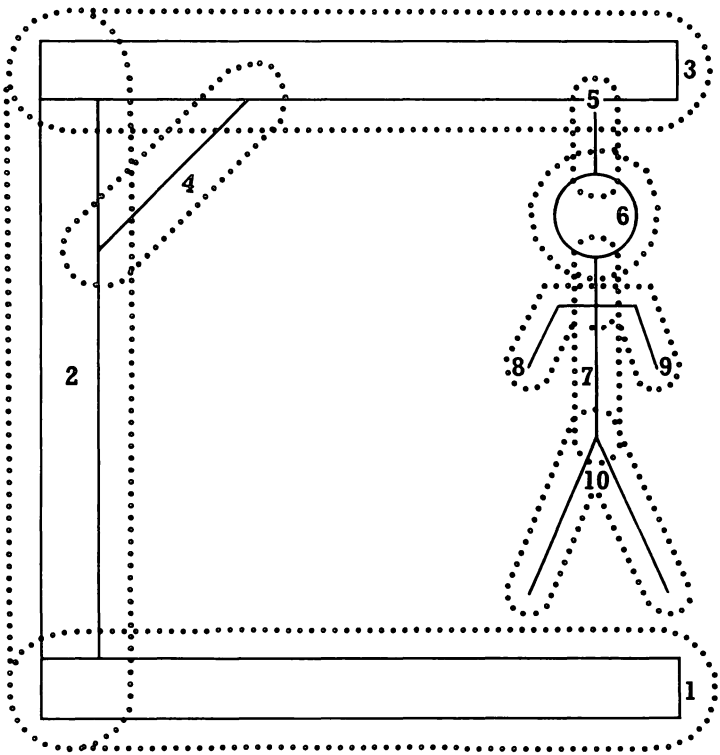
The routine from 60 to 110 works the key controls. The keys 'U,D,L,R' move the player. Notice the checks built into the lines to stop the player from wandering off the top, and bottom edge of the screen.

That triple loop at line 200 produces a flashing asterisk when you tread on a mine.

```
190  REM SHOW MINES AND LIVES LEFT
200  FOR T= 1 TO 20: FOR COL =0 TO 1:
     POKE PL+54272,COL: FOR D=1 TO 25
205  NEXT D: NEXT COL: NEXT T
210  LIVES = LIVES -1: PRINT
     "[S] LIVES";LIVES
220  FOR N=1024 TO 2023: IF PEEK(N)
     =42 THEN POKE N+54272,0
230  NEXT N
240  IF LIVES =0 THEN 400
250  PRINT "[S] ________"
260  FOR N=1024 TO 2023:
     POKE N+ 54272,7: NEXT N
270  GOTO 50
300  PRINT "[S] MADE IT WITH";LIVES;
     "   LIVES LEFT."
310  END
400  PRINT "[S] YOU HAVE RUN OUT OF
     LIVES."
410  END
```

HANGMAN

The gallows



Design your own gallows if you prefer, but make the drawing appear in 10 sections.

```
1000  PRINT "(First section)"
1010  RETURN
1100  PRINT "(Second section)"
1110  RETURN
      etc. to
1900  PRINT "(Last section)"
1910  RETURN
```

```
  5 PRINT "[♥]"
 10 DIM N(20)
 20 RESTORE
 30 X=INT (RND(0)*20)+1
 40 IF N(X)=1 THEN 30
 50 N(X) =1
 60 FOR T=1 TO X: READ W$ :
    NEXT T
 70 HIT =0: MISS=0
 80 L=LEN(W$) :Z$ =W$
 90 PRINT "[♥][Q][Q][Q][J][J]";: FOR T=1 TO L :
    PRINT "- ";: NEXT
100 GET L$ : IF L$<>"" THEN 100
110 GET L$ : IF L$="" THEN 110
120 FIND =0
130 FOR T=1 TO L
140 IF L$ = MID$(W$,T,1) THEN
    GOSUB 400: FIND=1
150 NEXT T
160 IF FIND =1 THEN 200
170 MISS =MISS+1
180 ON MISS GOSUB 1000,1100,1200,1300,
    1400,1500,1600,1700,1800,1900
190 IF MISS =10 THEN 250
200 IF HIT =L THEN 220
210 GOTO 100
220 PRINT "[S] WELL DONE.": GOTO 300
250 PRINT "[S] YOU ARE HANGED!"
260 PRINT "THE WORD WAS ";Z$
300 INPUT "ANOTHER GAME (Y/N) ";A$
310 IF A$="Y" THEN 20
320 IF A$ ="N" THEN 350
330 GOTO 300
350 END
400 HIT=HIT+1
410 PRINT "[S][Q][Q] ":PRINT TAB(T*2);L$
420 W$=LEFT$(W$,T-1)+" "+RIGHT$(W$,L-T)
430 RETURN
1000 REM DRAWING ROUTINE
2000 DATA COMPUTER,BASIC,LOOP,ARRAY,
     VARIABLE,STRING,INTEGER, RANDOM
2020 DATA RETURN,FLOWCHART,PROGRAM,
     ROUTINE,MEMORY,POKE,CURSOR
2030 DATA READ,NEXT,PRINT,GOSUB,
     CHARACTER
```

Hangman is a good game, and it's also a good way of introducing string slicing and program design.

This is the most complex program that we have tackled so far, so let's trace the process from idea to flowchart to BASIC. The first stage is to write down how the game works when it is played by two humans.

How it works

Game plan

1 The first player thinks of a word, and tells the other person how many letters it has – usually by marking a line for each letter on a piece of paper.

2 The second player tries to guess the letters that make up the word.

3 If he guesses a letter which is in the word, then the first player writes it into the correct place on the line.

4 If the guess was a bad one, then the first player draws a little more of the Hangman drawing.

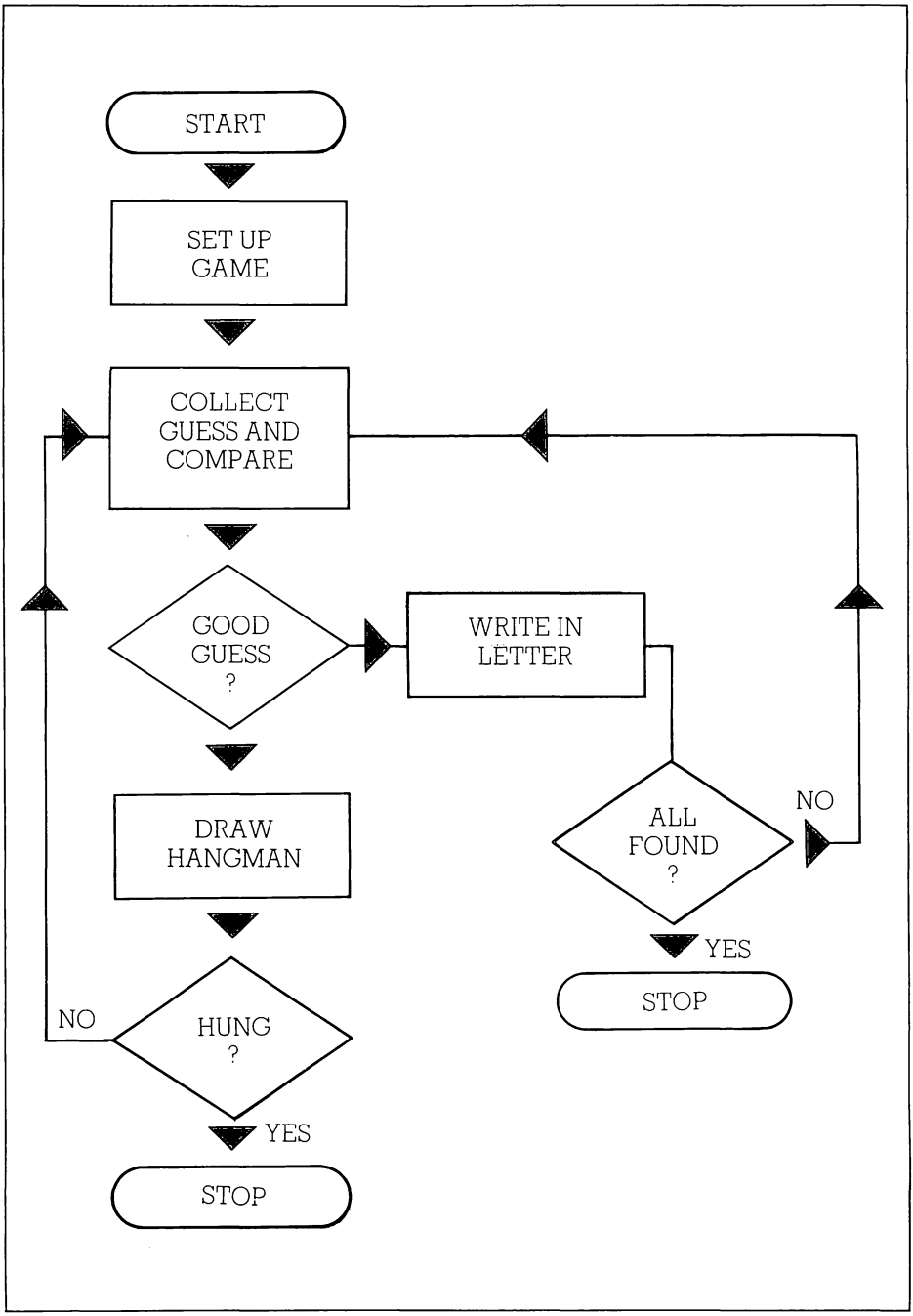
5 The game ends either when the second player has guessed all the letters, or when the hangman picture is complete.

This game outline can then be converted into a block flowchart. Some parts of that flowchart will convert to single lines of BASIC; others will become whole routines.

1 Set up game (lines 10 to 90)

The words for the game are all written into the DATA lines at 2000 onward. You want the computer to get one of these at random – but you also want to make sure that it does not pick the same word twice. This means that you have to keep a record of the words that have been used. The array N(20) is used to store the reference numbers of the words. Each time the computer finds a random number, it checks the N() array to see if it has been used. If it hasn't, then the number is marked off, and the DATA list is read to find the word.

Lines 70 to 80. You need HIT and MISS counters, the length of the word, and a copy of the word. Line 90 prints a spaced out row of dashes for the letters.


START
SET UP GAME
COLLECT GUESS AND COMPARE
GOOD GUESS ?
WRITE IN LETTER
ALL FOUND ?
NO
YES
STOP
DRAW HANGMAN
HUNG ?
NO
YES
STOP

2 Collect guess and compare

GET L$ is used, rather than INPUT L$. You could use an INPUT line, but you would have to make sure that you fixed its position with a PRINT line using cursor commands beforehand. If you didn't, you could ruin the display. GET L$ avoids the bother.

Line 140 introduces a new function – MID$. This looks at a section of the word.

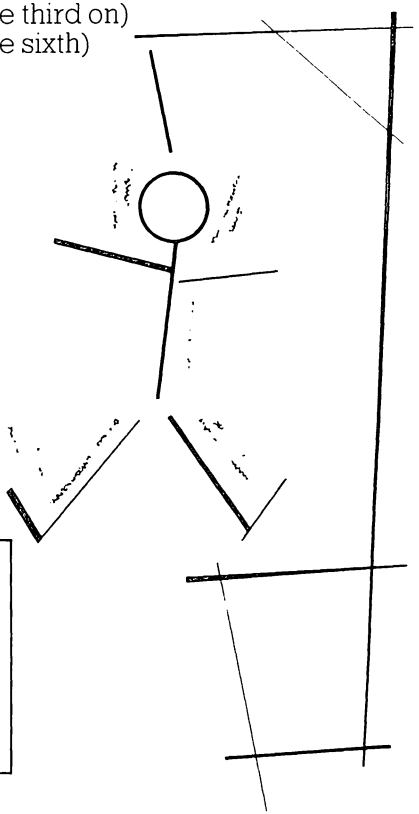
MID$ must be followed by the word (or string variable), the number of the first letter you want to look at, and the number of letter you want.

If W$ = "COMPUTER" then:

```
MID$(W$,3,3) = "PUT"   (3 letters from the third on)
MID$(W$,6,2) ="ER"     (2 letters from the sixth)
```

There are two other string slicing functions, and both are used in the subroutine at 400. LEFT$(W$,3) will slice off the first three letters of the word. RIGHT$(W$,3) slices off the last three letters.

In line 420 "LEFT$(W$,T−1)" slices off that part of the word up to the found letter. "RIGHT$(W$,L−T)" takes the remainder of the word to the right of the letter. That line doesn't just chop up the word, it puts it back together again. The plus signs join strings together. The effect of this line is to replace the found letter with a space. This is to stop cheats.



```
W$="PROGRAM"
L$="A"
L$=MID$(W$,6,1) so T=6
LEFT$(W$,T−1)="PROGR"
RIGHT$(W$,L−T)="M"
W$="PROGR"+" "+"M"="PROGR M"
```

3 All found?

A single line covers this. All you need to check is that there have been as many HITs (good guesses) as there are letters.

4 Draw hangman

This part of the program has been left for you to do, apart from the first two lines. Line 170 simply counts MISSes. Look closely at the next line. This uses a new BASIC command.
ON. . .GOSUB. . ..

When MISS=1, this line sends the program to the subroutine at 1000; when it is 2, the program goes to 1100; when MISS=3, it goes to 1200, and so on. ON. . .GOSUB. . . replaces a set of IF. . .THEN. . . lines.

```
IF MISS =1 THEN GOSUB 1000
IF MISS =2 THEN GOSUB 1100
......
```

The other form of this command is ON. . .GOTO. . ., where it works in exactly the same way. ON. . .GO. . . can be used wherever the values that could come after ON form a simple series, 1,2,3,4,5,6,. . . If the values for which you are checking are −3,65,999,2345 then you must use IF. . .THEN. . . lines.

To work out the drawing subroutines, use the sketch given with the program list, or design your own ten part picture. Each of the ten subroutines will then consist of a PRINT line, containing lots of cursor commands to get the print position to the right place, and the graphics for that part of the picture. A second line contains RETURN.

```
1000  PRINT "[S][ ][Q][Q][Q][Q][Q][Q][Q][Q][J]
      [J][J][J][J][J][J][J][J][J][J][J][J][R]■___"
1010  RETURN
```

5 Hung?

Another simple check line to see if the MISS counter has reached 10.

Stop

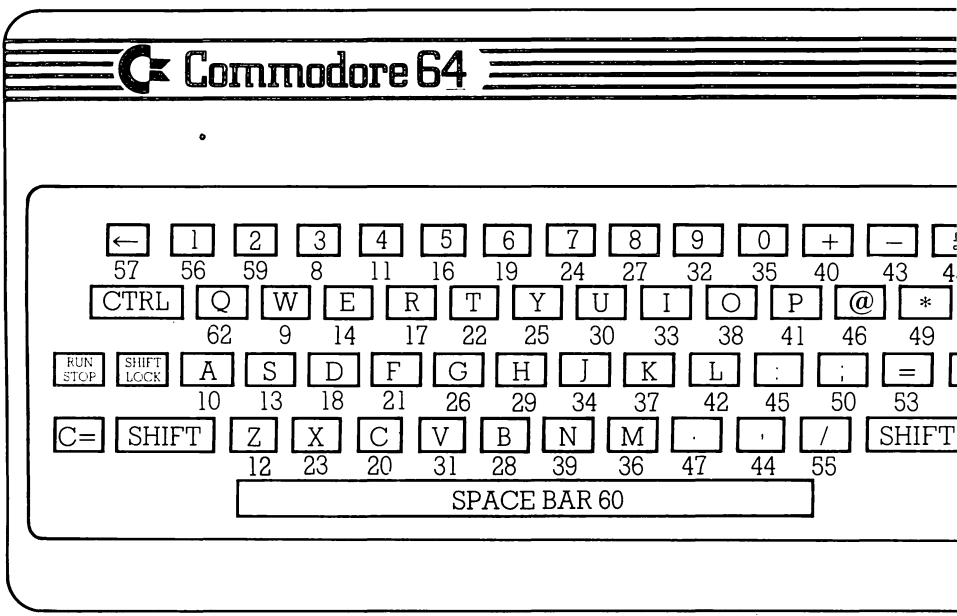
Rather than end the program after one run through, an 'Another Go?' routine has been included. You should note that the program restarts from the RESTORE line for the next go.

TAKE A

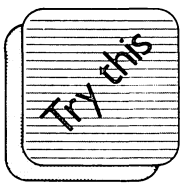
PEEK allows you to find out what is going on inside the 64. It will tell you the contents of any address, just as POKE will let you change the contents of an address. Most of the time, when we are working in BASIC, it shouldn't matter too much just exactly what is going on inside, as long as the 64 gets on with the job. Some of the more sophisticated sprite routines need PEEKs, and there is one particular address which is of great interest.

Address 197 picks up keystrokes. PEEK(197) is like the GET statement, but far more flexible.

The number that you get from PEEK(197) will always be between 0 and 64. '64' shows that no key is being pressed. The numbers produced by the other keys are shown here.



PEEK

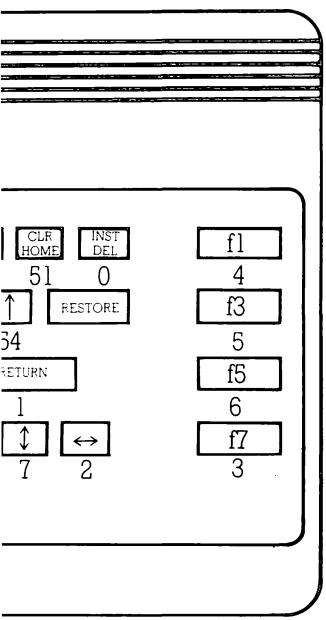



```
10  IF PEEK(197)= 64 THEN 10
20  PRINT PEEK(197): GOTO 10
```

Here's another routine to show how you could use the PEEK to control the flow of a program. Type it in and see.

```
10  X =0
20  PRINT X
30  X=X+1
40  IF PEEK(197)<>64 THEN 40
50  GOTO 20
```

As long as no key is touched, the program keeps printing. Hold down a key to stop it, and watch it start up immediately you take your finger off.



Projects

1 Try writing a GET line that would do the same.

2 Change line 40 to this to create the reverse effect. Now the program will only run when you hold down a key:

```
40  IF PEEK(197)=64 THEN 40
```

Look out for the use of PEEK(197) in the SKI-SPRITE game.

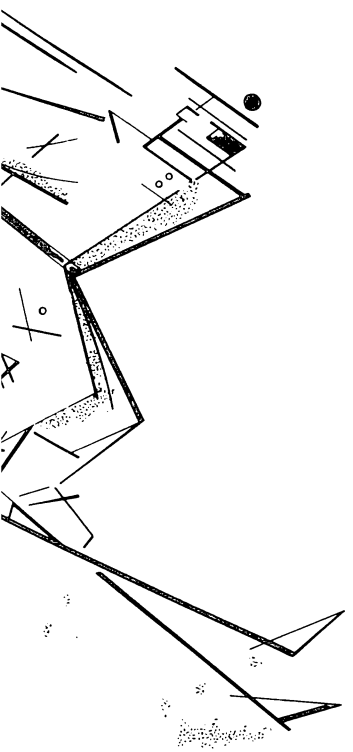
Ski sprite

In 'SKI SPRITE', the aim is to keep the skier clear of the asterisks as he rushes down the screen. Steer left and right with the [<][>] keys.

The game ties together several of the sprite handling techniques and keyboard control. It also introduces a new idea – COLLISION DETECTION. There are two PEEKs that will tell you if a sprite has 'hit' a character, or another sprite. The one we are using here checks for sprite-character collisions.

PEEK(VIC +31) will have a value of 1 if sprite 0 hits a character; a value of 2 if sprite 1 collides; 4 for sprite 2, etc.

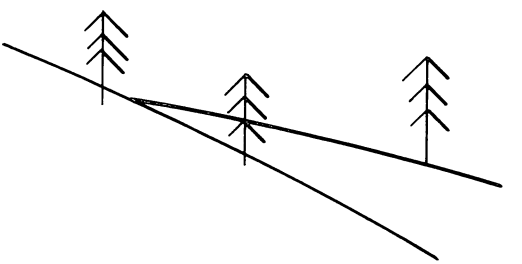
PEEK(VIC+30) checks for sprite–sprite collisions, using the same codes. So, a value of 3 here would show that sprites 0 and 1 had crossed paths.

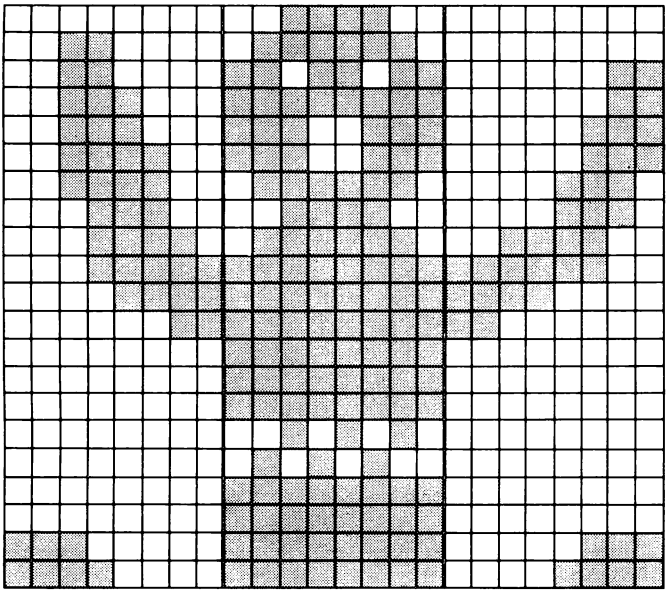


Collision codes

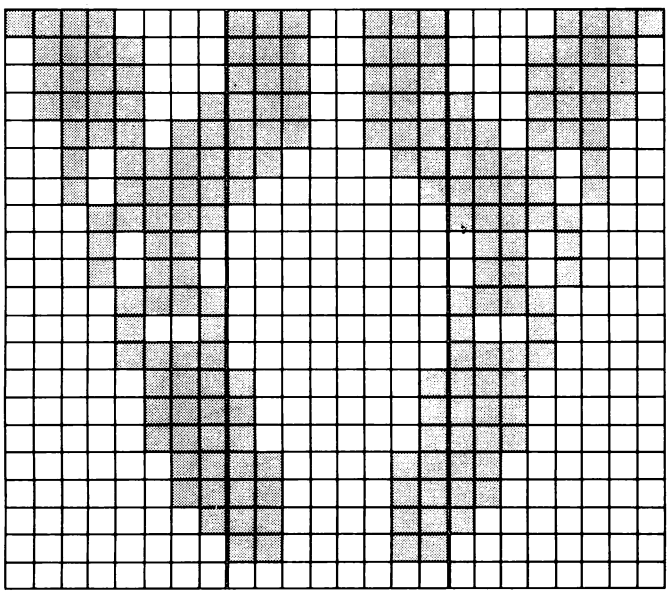
Sprite number	0	1	2	3	4	5	6	7
Collision Code	1	2	4	8	16	32	64	128

The sprite DATA lines have not been included in the program list. You will find it easier to type them in correctly if you work directly from the designs. You can, of course, always alter the designs to suit yourself.





0	60	0
48	126	0
48	219	3
56	255	3
56	231	7
60	231	7
60	126	14
28	60	14
30	126	60
31	255	252
15	255	240
3	255	192
0	255	0
0	255	0
0	255	0
0	42	0
0	84	0
0	255	0
0	255	0
224	255	7
240	255	15



240	231	15
120	231	30
120	231	30
121	231	158
59	231	220
47	195	244
47	129	244
31	0	248
22	0	104
22	0	104
15	0	240
9	0	144
15	0	240
7	129	224
7	129	224
7	129	224
3	195	192
3	195	192
1	195	128
0	195	0
0	0	0

Program

```
  5  PRINT "■"
 10  REM SKI SPRITE
 20  FOR N=0 TO 1:FOR
     T=0 TO 62
 30  READ B: POKE 12800
     + 64*N +T,B
 40  NEXT T: NEXT N
 50  DATA....
...
...
200  VIC =53248
210  POKE VIC,100:POKE
     VIC +1,100:REM
     SPRITE FOR TOP
     HALF
220  POKE
     VIC+2,100:POKE VIC
     +3,121:REM SPRITE
     FOR LEGS
230  POKE VIC+39,2:POKE
     VIC +40,2:POKE
     2040,200: POKE
     2041,201
240  HIT=0:POKE VIC
     +31,0: REM NO
     COLLISIONS YET
250  PRINT " ■ (24 down
     cursors)":POKE
     53281,1:POKE
     53280,1
260  POKE VIC +21,3
270  X=100
280  T=INT(RND(0)*40):
     PRINT TAB(T);"*"
290  IF PEEK(197)<> 64
     THEN GOSUB 1000
300  IF PEEK(VIC+31)>1
     THEN GOSUB 2000:
     IF HIT =5 THEN 400
310  IF RND(0)>.2 THEN
     PRINT:GOTO 290
320  GOTO 280
400  PRINT "YOU HAVE
     CRASHED ONCE TOO
     OFTEN !"
410  INPUT "ANOTHER GAME
     (Y/N) ";A$
420  IF A$ ="Y" THEN 200
430  IF A$<>"N" THEN 410
440  END

1000  REM MOVE SPRITE
      ACCORDING TO KEYS
1010  P=PEEK(197)
1020  X=X+(P=47)*2
      -(P=44)*2
1030  X=X+(X>255)*2-(X<0)*2
1040  POKE VIC,X : POKE
      VIC +2,X
1050  RETURN
2000  REM COLLISION
2010  HIT =HIT+1
2020  FOR N=1 TO 20 :FOR
      COL=6 TO 7
2030  POKE VIC +39,COL:
      POKE VIC+40,COL
2040  FOR D=1 TO 50:
      NEXT D
2050  NEXT COL: NEXT N
2060  POKE VIC +39,2:
      POKE VIC +40,2
2070  PRINT " ■ (24 down
      cursors)":
2080  POKE VIC+31,0
2090  RETURN
```

How it works

Two sprites are used here to give a larger and clearer image. Sprite 0 sits on top of sprite 1, and the horizontal position for both sprites is held in the variable X.

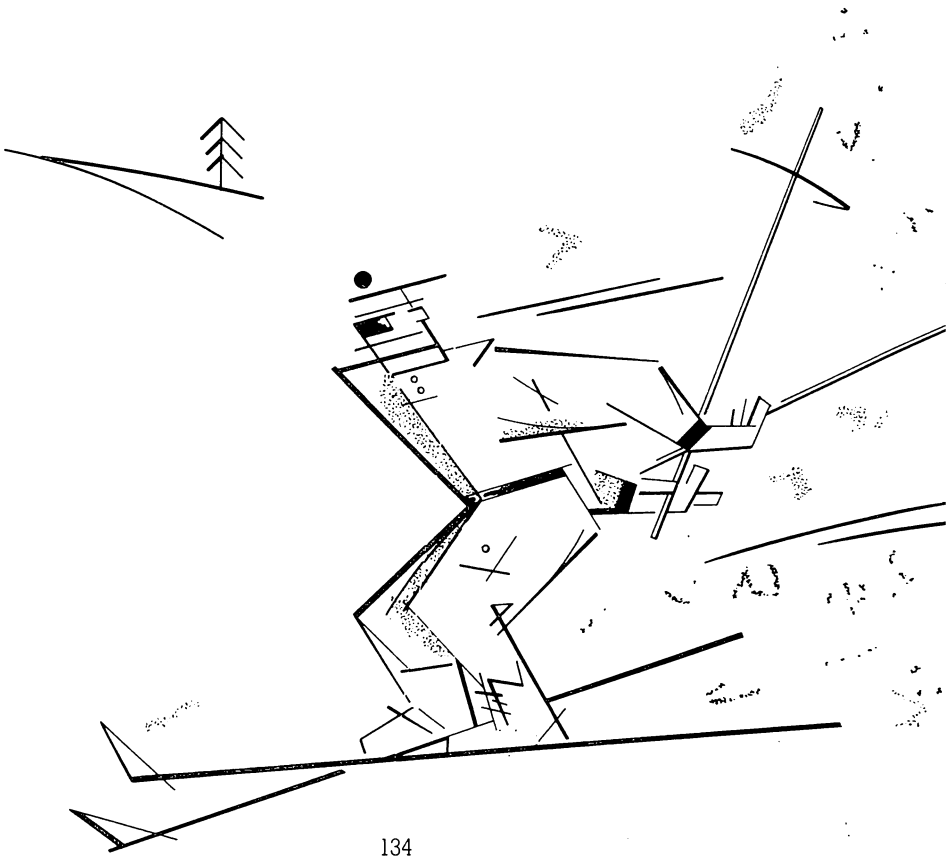
Line 240: You must clear the collision address before you try to use it. The HIT counter keeps a tally of collisions. You are allowed 5 on any one run. The limit can be set to anything you like.

Line 250: This pushes the print position to the bottom of the screen, so that the asterisks begin to print from there. The screen and border are set to snow-white.

Line 280: This prints an asterisk at a random TAB position.

Line 300: Here you are asking the 64 to check if the lower sprite (number 1) has collided – i.e., is PEEK(VIC+31)=2, but it might just happen that the top sprite collides at the same time, in which case the PEEK would give you 3. This caters for either possibility.

Line 310: This controls how many asterisks appear on the screen. Eight times out of ten, the program will go back to the keystroke line



(290) and miss the asterisk print line. Change that random limit to increase, or reduce the frequency of the asterisks.

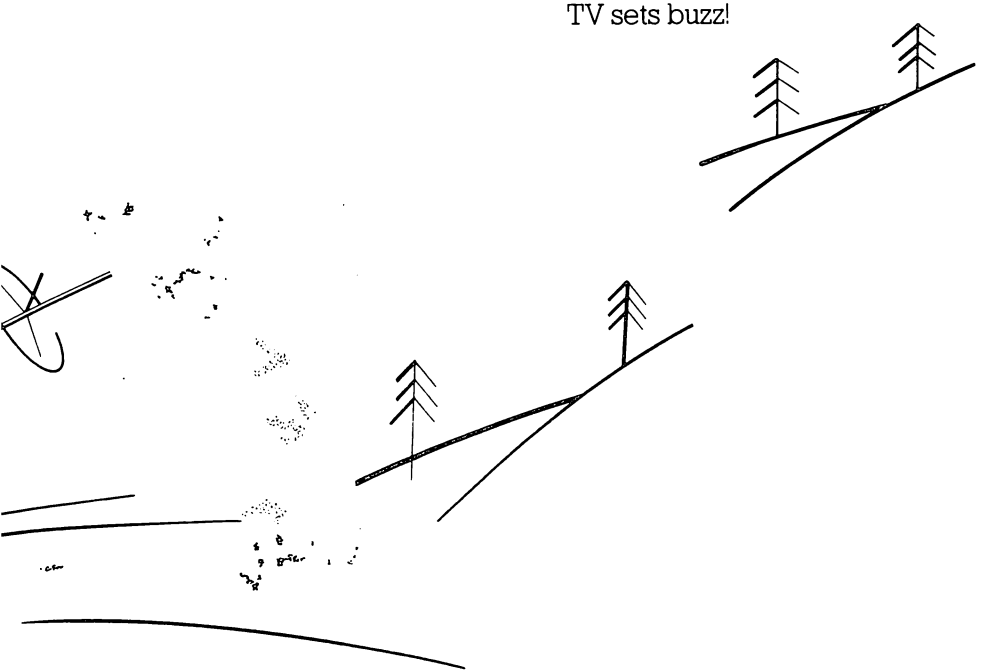
400–440 When the INPUT line appears, you will notice a lot of commas and full stops. These have been collected into the INPUT BUFFER during the course of the game. It looks a bit messy, so add this line to empty the buffer before the INPUT.

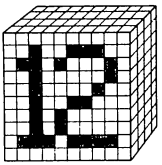
```
405 GET A$: IF A$<>"" THEN 405
```

1000–1050 The sprite is controlled by the [<] and [>] keys. If you would prefer to use other keys then look up their code numbers on the Take a Peek reference chart.

1030 stops the X value from wandering out of range. If it does go over 255 or under 0, then this corrects it before the POKE line.

2000—2090 You could add some sound effects in here. If you do, then you would be advised to recolour your screen. White screens and borders make some TV sets buzz!





The final chord

Make more music

It's time to put your improved BASIC knowledge to work and convert your computer keyboard into an organ keyboard. The following program makes the number keys (from 1 to 8) play notes in the key of C. Once it is working properly, you can extend this to make more music with the other keys.

Program

```
 5  PRINT "▣"
10  REM ORGAN
20  DIM P(8,3)
30  FOR T=1TO8: FOR
    N=1TO3:READ P(T,N):
    NEXTN,T
40  DATA 56,17,37,59,19,63,8,
    21,154,11,22,227,16,
    25,177,19,28,214
50  DATA 24,32,94,27,34,75
60  S=54272
70  FOR N=S TO S+24:
    POKE N,0: NEXT N
80  POKE S+24,15
90  POKE S+5,0:POKE S+6,32
100 PRINT " PRESS KEYS 1
    TO 8 TO PLAY"
110 IF PEEK(197)=64
    THEN 110
120 X=PEEK(197): FOR
    T=1 TO 8: IF
    P(T,1)=X THEN N=T:
    GOTO 140
130 NEXT T: GOTO 110
140 HI = P(N,2): LO=
    P(N,3): POKE S+1,HI:
    POKE S,LO
150 POKE S+4,17
160 IF PEEK(197)=64
    THEN POKE S+4,16
170 GOTO 110
```

To make this program work, we are using an array. The arrays you saw earlier were all one-dimensional – like a list. The array used here is two-dimensional. You could think of it as a table like the one on this page. The array we use is labelled P(8,3). It has 8 rows and 3 columns. In the first column is stored the PEEK code for the key, in the second column is the HI frequency for the pitch; and the third column stores the LOw frequency.

The table here shows the data for the array. The dotted lines enclose that part of the table which is written into the program. The rest can be added, by you, later.

NOTE	KEY/CODE	HI	LO
C	1/56	17	37
D	2/59	19	63
E	3/8	21	154
F	4/11	22	227
G	5/16	25	177
A	6/19	28	214
B	7/24	32	94
C'	8/27	34	75
D'	9/32	38	126
E'	0/35	43	52
F'	+/40	45	198
G'	−/43	51	97
A'	£/48	57	172
B'	CLR HOME /51	64	188

Projects

1 Extend the 'organ' keys to cover the top row from 1 to CLR/HOME. The table gives you all the data you need for this. Your range of notes will then be from middle C to top B.

2 Change the sounds. This has been written with the Triangle waveform (POKE S+4,17). Try using the Sawtooth waveform instead for a more 'electronic' sound

The Pulse waveform offers even more variety of textures. To use this, POKE S+4,65 to turn it on, and POKE S+4,64 to turn it off. You will also need to add a line. With the Pulse waveform, you have to set the Pulse rate before anything happens. This is controlled by two POKEs – S+2 and S+3. Of these S+3 is the most important. Its value can be anything between 0 and 15. The higher the number, the 'thinner' the sound. S+2 gives a fine tuning edge to the waveform. Add this line for a start, and adjust the numbers later to suit yourself

```
95 POKE S+3,7: POKE
   S+2,50
```

3 Change the envelope. The envelope is the shape of the sound – the ATTACK/DECAY, SUSTAIN/RELEASE rates. At the moment the envelope is shaped to give a flat, organ-like sound, as this allows for an easy slide between notes. Try changing line 90 to this: POKE S+5,9: POKE S+6,32.

Try different values in that line and see what effects you can produce.

All together

SID has three voices, though so far we have used only one. The second and third voices work in exactly the same way as the first, and the SID numbers follow an easy pattern. Look at the table. All the numbers for voice 2 are seven more than the equivalent numbers for Voice 1. Voice 3 numbers are seven more again.

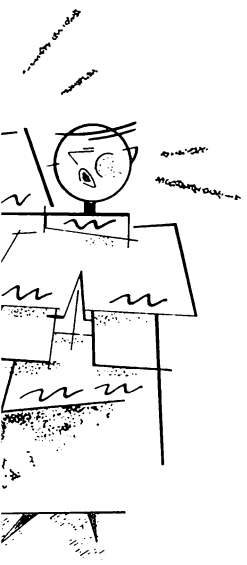
This plays a C chord:

```
 10  SID=54272: FOR N=SID TO SID
     +24: POKE N,0: NEXT
 20  POKE SID+24,15
 30  POKE SID+5,9:POKE SID+6,40
 40  POKE SID+12,9:POKE SID+13,40
 50  POKE SID+19,9:POKE SID+20,40
 60  POKE SID+1,17:POKE SID+ 0,37
 70  POKE SID+8,21:POKE SID+7,154
 80  POKE SID+15,25:POKE SID+14,177
 90  POKE SID+4,33:POKE SID+11,33:
     POKE SID+18,33
100  FOR D=1 TO 1000: NEXT D
110  POKE SID+4,32:POKE SID+11,32:
     POKE SID+18,32
```

If you don't like the sound of this chord, then try changing to a Triangle waveform. Alter the 33's in line 90 to 17's. (You will need to change line 110 to turn them off.) You can experiment with different values for the shapes of the notes – lines 30 to 50. They don't all have to be the same.

Whatever the final sound you come up with, it has all been rather a lot of work for one chord. It needn't be just for one chord, of course. This can be made into a subroutine, and called up whenever you need it in a program. You can change the chord that is played by putting the High and Low frequencies into variables, and send different sets of notes to your subroutine.

You can even write a routine that will play a chord on the strength of just one note value. All music follows very strict mathematical rules, and the relation between the notes of a chord of any one type is always the same, no matter what the chord. To understand this you have to look at those High and Low frequencies a little more closely.



	VOICE 1	VOICE 2	VOICE 3
Pitch – low	SID+0	SID+7	SID+14
– high	+1	+8	+15
Pulse rate – fine tune	+2	+9	+16
– main	+3	+10	+17
Waveform	+4	+11	+18
Attack/decay rate	+5	+12	+19
Sustain/release rate	+6	+13	+20
Volume	SID+24		

Frequencies

The FREQUENCY of a note is the number of times per second that it makes the air vibrate. The faster the vibrations, the higher the note sounds.

The Commodore 64 can produce notes with frequencies as low as 268 beats per second, and as high as 64814 beats per second. This covers a range of eight octaves – much the same as a piano. What it can't do is cope with any of these frequency values in a single byte. The highest number you can POKE into an address is always 255.

The 64, like most home computers, handles big numbers in two bytes – a HIGH BYTE and a LOW BYTE. If you look back at the last program you will see that the High and Low frequencies for C are 17 and 37. The frequency of C is 4389. Use your 64 to multiply 17 by 256, and then add 37 to the total. You should get a result of 4389.

To turn any two-byte number into a decimal, multiply the second byte (the High byte) by 256, and add the first (Low byte). If you don't fancy the maths, type this program in. It will work out frequencies for you.

```
10  INPUT " HIGH FREQUENCY ";HI
20  INPUT "LOW FREQUENCY ";LO
30  F = HI*256 +LO
40  PRINT "FREQUENCY =";F
50  GOTO 10
```

You can turn this on its head and convert big numbers to two-byte form with a program like this:

```
10  INPUT " BIG NUMBER ";BN
20  HI = INT(BN/256)
30  LO = BN-HI*256
40  PRINT "HIGH BYTE";HI,"LOW BYTE
    ";LO
50  GOTO 10
```

(This may seem like a long way to go to a shortcut, but we are getting there!)

CHORD SHAPES follow set patterns. A Major chord consists of the first, third and fifth notes of a scale. The chord C has the notes C, E and G, from the scale *C*, D, *E*, F, *G*, A, B, C. These intervals are mathematical. Multiply the frequency of the first note by 1.26 to get the frequency of the second. Multiply this by 1.18 to get the third.

Program

At last, here it is. The give-me-a-note chord-playing subroutine.

```
1000  S=54272:FOR N=S TO
      S+24:POKE N,0:
      NEXT N
1010  POKE S+24,15
1020  POKE S+5,9: POKE
      S+6,40
1030  POKE S+12,9: POKE
      S+13,40
1040  POKE S+19,9: POKE
      S+20,40
1050  GOSUB 1200:
      BN=BN*1.26
1060  POKE S+1,HI:
      POKES+0,LO
1070  GOSUB 1200:
      BN=BN*1.18
1080  POKE S+8,HI: POKE
      S+7,LO
1090  GOSUB 1200
1100  POKE S+15,HI: POKE
      S+14,LO
1110  POKE S+4,33: POKE
      S+11,33: POKE
      S+18,33
1120  FOR D=1 TO WHEN:
      NEXT D
1130  POKE S+4,32: POKE
      S+11,32: POKE
      S+18,32
1140  END
1200  HI = INT(BN/256)
1210  LO =BN-HI*256
1220  RETURN
```

To make this work, set a value for BN corresponding to the frequency of the bottom note of the chord, and a value for WHEN to fix the length of the sounds.

Projects

You can improve this by taking the first line out of the subroutine, and putting it at the very start of your program. This avoids the crackle as the computer clears the sound chip.

For a minor chord, change the second half of line 1050 to BN = BN*1.19, and the second half of line 1070 to BN=BN*1.25.

For a major seventh, change line 1070 to BN=BN*1.141.

SAMPLE FREQUENCY TABLE

	HI	LO	FREQ
C	17	37	4389
D	19	63	4927
E	21	154	5530
F	22	227	5859
G	25	177	6577
A	28	214	7382
B	32	94	8286
C	34	75	8779

A final word

I hope this book has helped to introduce you to your Commodore 64. You should by now have a reasonable grasp of most of the Commodore's BASIC, and of handling its sounds and sprites. There are still areas that have not been touched – machine code, file-handling and some of the more advanced mathematics. However, you know enough now to be able to write a whole range of programs of your own, and I hope you have gained enough confidence with the machine to feel ready to tackle the more complex areas of programming.

When you sit down to write your own programs, take them from the 'Top down'. Start with a very clear idea of what the program is supposed to do. Then break the program down into its main parts, and draw up an outline flowchart. The next stage is to look at the separate routines that will make up the blocks of the program. Some of these may have to do

things that you have never tried before. Other routines will be based on what you already know well.

Solve your problems before you start, if you can. Tackle the hardest parts first, and try and work out the routines by themselves – not as part of your overall program. It is much easier to test out a new technique in a 10 line program, rather than as a 10 line routine in a 200 line program!

Ideally, you should be able to assemble a program out of tried and trusted routines. That way, your de-bugging sessions will be limited to making sure that the routines link together properly. It may look quicker to scrap all this preparation, and just get in there and hack, but in the long run, it takes longer.

And finally, a thought for when things get tough. Nothing's impossible – it's just that you haven't found the way to do it – yet.

Index to BASIC keywords

The numbers refer to the first page on which the keyword is covered.

Launch yourself into programming

Introducing your Commodore 64

All the practical advice you need to start programming.

ACTION PACKED PROGRAMS

Introducing your Commodore 64 gives you instant on-screen results. Right from the start you will be working on powerful program listings - each one fully explained and illustrated. These listings will become the basis for your skills and give you results fast.

NEW PROGRAMMING SKILLS

In Introducing your Commodore 64 you will discover the starting points for powerful programming. You will discover how to explore the potential of your micro. Best of all you'll discover how to write dynamic and exciting programs.

NYBBLE BY NYBBLE, BYTE BY BYTE

Introducing your Commodore 64 works at making it easier for you. There's no unnecessary jargon. If you need to know it's clearly explained - and then practised. It's as simple as that.

GET RESULTS - FAST.

INTRODUCING YOUR COMMODORE 64

Longman
Computer
Books

ISBN 0-582-91603-8

9 780582 916036