

# Turbocharge

your Commodore 64

Peter Worlock



Longman 



# **Turbocharge** **your Commodore 64**



# **Turbocharge** **your Commodore 64** Peter Worlock

Longman 

Commodore is a registered Trade Mark  
of Commodore Business Machines.

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

© Longman Group Limited 1984

All rights reserved. No part of this  
publication may be reproduced, stored in  
a retrieval system or transmitted in any  
form or by any means, electronic,  
mechanical, photocopying, recording or  
otherwise, without the prior permission of  
the Copyright owner.

First published 1984

ISBN 0 582 91605 4

Printed in UK by Parkway Illustrated Press,  
Abingdon

Designed, illustrated and edited by  
Contract Books, London

The programs listed in this book have been  
carefully tested, but the publishers cannot  
be held responsible for problems that  
might occur in running them.

---

# Contents

---

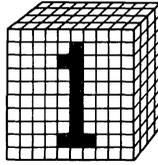
|                                     |     |
|-------------------------------------|-----|
| <b>CHAPTER 1</b>                    |     |
| Program structure and design        | 7   |
| <b>CHAPTER 2</b>                    |     |
| Built-in and user-defined functions | 19  |
| <b>CHAPTER 3</b>                    |     |
| Interactive programming             | 27  |
| <b>CHAPTER 4</b>                    |     |
| Information handling                | 39  |
| <b>CHAPTER 5</b>                    |     |
| Introduction to graphics            | 53  |
| <b>CHAPTER 6</b>                    |     |
| Advanced colour                     | 69  |
| <b>CHAPTER 7</b>                    |     |
| High resolution graphics            | 75  |
| <b>CHAPTER 8</b>                    |     |
| Introduction to sprites             | 83  |
| <b>CHAPTER 9</b>                    |     |
| Advanced sprites                    | 91  |
| <b>CHAPTER 10</b>                   |     |
| Animation                           | 105 |
| <b>CHAPTER 11</b>                   |     |
| Sound                               | 119 |
| <b>CHAPTER 12</b>                   |     |
| Interfacing                         | 137 |
| <b>APPENDIX</b>                     |     |
| Design aids                         | 149 |
| <b>INDEX</b>                        | 155 |

## Note

In some of the example programs, instructions appear in square brackets like this:

```
10 PRINT "[HOME]"
```

Do not type these words. Instead substitute the relevant control symbol from the keyboard.



---

# Program structure and design

---

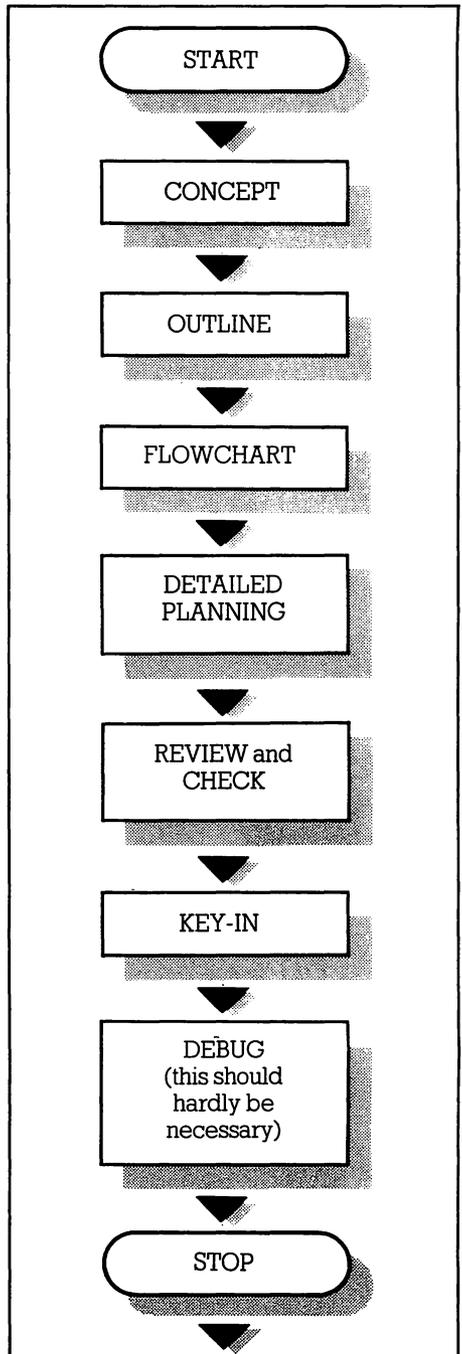


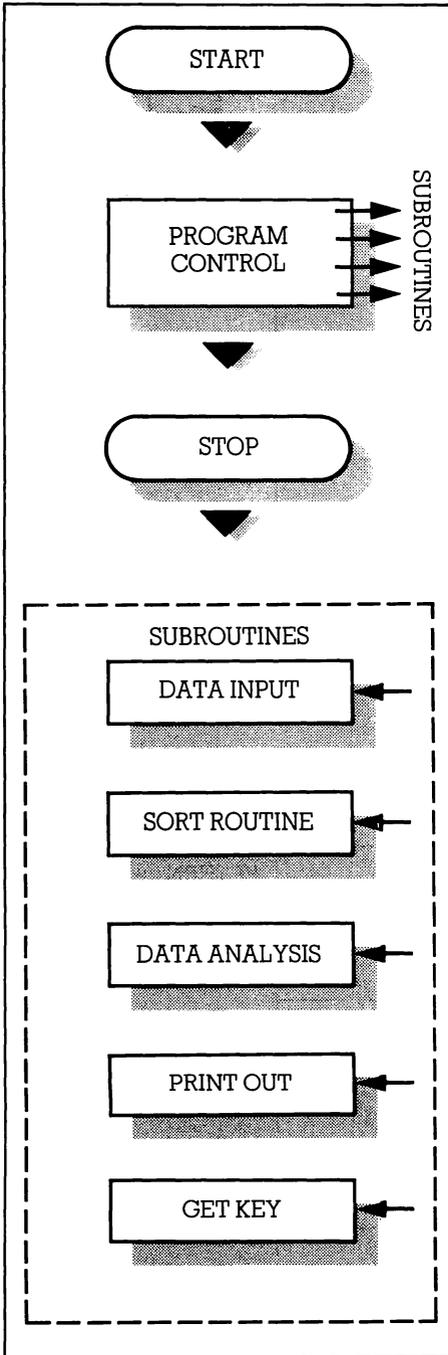
There are things in programming that are obviously good – a routine that speeds things up, a way of doing a task that saves programming time or a way of producing some special graphics effect. It's harder to see the benefit of structured programming although if you believe some people, it's impossible to write a good program in Basic because of its lack of structured commands. On the other hand there is a body of opinion that says if a program works it doesn't matter if it looks a mess or what language it's written in.

### The benefits

However, structure is good for you and in the long run it brings a number of benefits – once you get into the habit, structured programs are easier to write because you will be able to make extensive use of programs you have already written. Also, very few of your programs will be perfect first time – you'll want to make use of new techniques as you learn them, or update your programs as you add to your system with disk drives, printers etc. This updating is a lot easier if you can go back to your program and see immediately how it works. Unstructured programs which GOTO different sections apparently at random, which jump out of subroutines and loops and which introduce new variables in the middle of routines will take so long to untangle that you'll probably waste time writing them from scratch, or give up in frustration and never update the thing at all.

The easiest way to write a bad program is to get an idea, sit down at the keyboard and start pounding away.





The ideal way to write a good program is to have it all but finished *before* you turn on the computer. This, of course, is a counsel of perfection but it's something to aim at.

## Subroutines

The key to structured programming is the subroutine. Subroutines bring a number of advantages not least of which is that if you need a routine to sort a list of items you can write it, save it to tape or disk and then use it whenever you need a similar routine. Also when you write a new program you do not have to break off the new writing to reinvent the wheel. You can use a stub like this:

```

100 GOSUB 1000: REM SORT
    ROUTINE
1000 REM *** SORT GOES
    HERE ***
1010 RETURN
  
```

This means that the program will work as it stands (up to a point) and you can add the sort later.

Think of programs in terms of control sections calling all of the required subroutines and it becomes easier to update them later. The use of GOSUB also eliminates the greatest source of confusion in programs – a mass of GOTOs. Regardless of arguments about elegance the best reason for avoiding GOTO is that when debugging a program you can only get about 20 lines of code on screen and a GOTO will force you to list another section, then jump back again. GOSUBs indicate self-contained routines that you can often ignore at that point.

An acceptable use of GOTO is something like this:

```
100 GET A$: IF A$ = ""  
    THEN 100
```

Subroutines will also save you time and memory. If your program has dozens of lines like that one you only need to write it once as a subroutine and then replace the others with GOSUBs.

### AS A RULE

The main rule of subroutines is that there should be only one way in and one way out. Never do this:

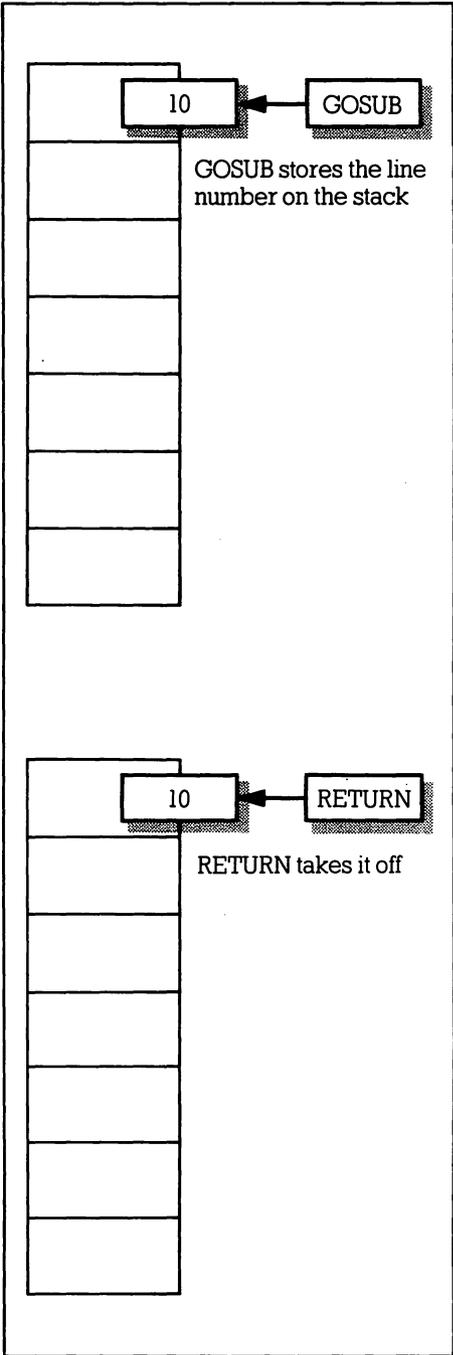
```
10 GOSUB 100  
20 :  
30 :  
100 START OF ROUTINE  
110 :  
120 IF A$ = "EXIT" THEN 20  
130 :  
140 RETURN
```

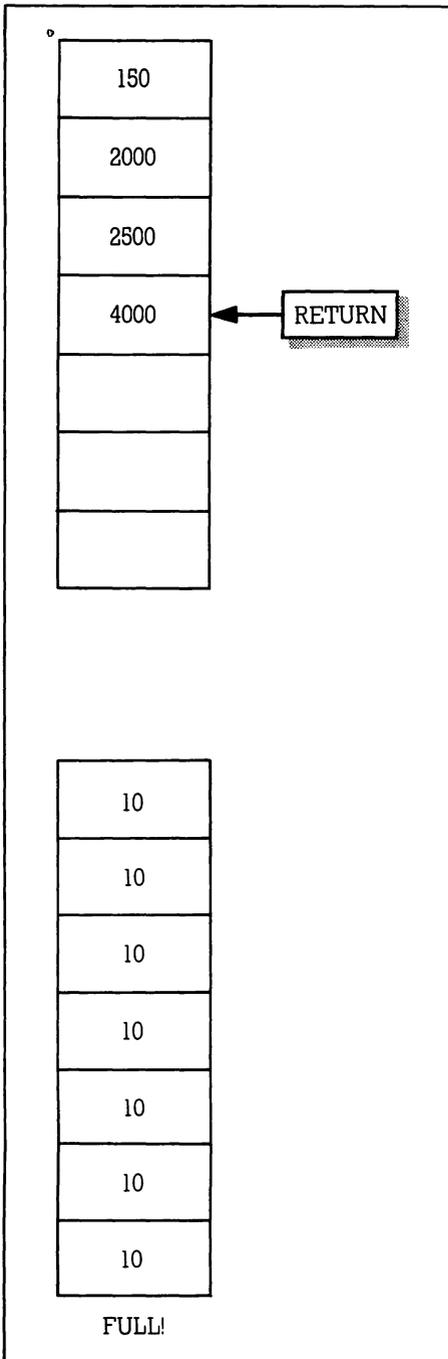
### TRY THIS

If it's necessary to make an early exit from the routine jump to the RETURN line. Not only does this make it easier to follow the logic of the program, there is a practical reason. Enter this and run it:

```
10 GOSUB 100  
20 END  
100 GOTO 10
```

Obviously you haven't run out of memory. What has been exhausted is a part of memory called the stack.





Every time the computer GOSUBs it saves the address of the branching point on the stack so it knows where to go to when it meets the RETURN statement. If it never gets to the RETURN the stack fills up.

### Branches

Subroutines of the kind we've discussed so far are a convenience. But they can be made a powerful tool if the computer can decide which of several routines to use. The most common method of determining a branch is the IF . THEN statement which can be thought of like this: IF condition THEN action.

The power of IF . THEN comes from the number of ways the computer can examine the condition. The symbols =, <, >, <> are called logical operators and mean equal, less than, greater than, and not equal respectively:

```

10 A = 10: B = 5
20 IF A = B THEN PRINT
   "A = B"
30 IF A < B THEN PRINT
   "A IS LESS THAN B"
40 IF A > B THEN PRINT
   "A IS GREATER THAN B"
50 IF A <> B THEN PRINT
   "A DOES NOT EQUAL B"

```

We can also combine conditions using the operators AND and OR:

```

60 IF A = 10 OR B = 10
   THEN PRINT "ONE
   CONDITION IS TRUE"
70 IF A = 10 AND A = B
   THEN PRINT "BOTH
   CONDITIONS ARE TRUE"

```

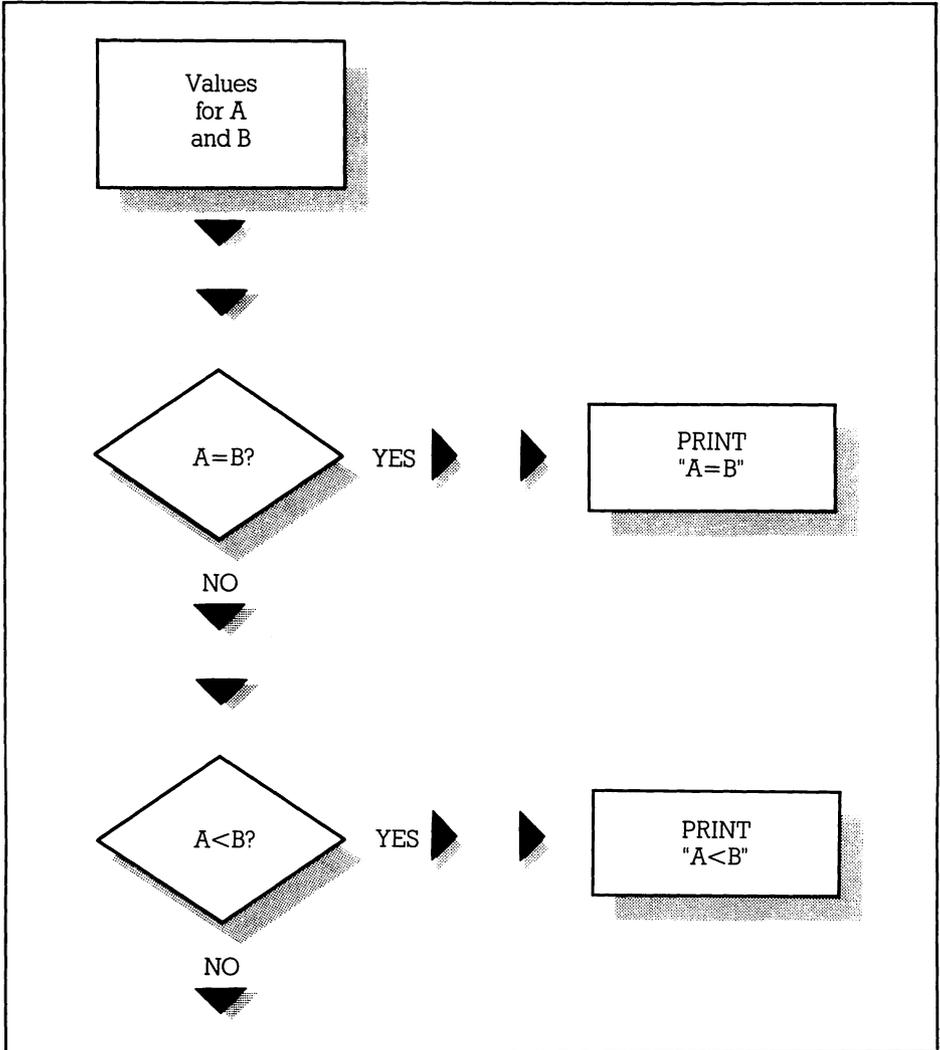
This range of checks gives computers most of their 'intelligence' in the sense that they can perform an action appropriate to the conditions.

However, if there is a wide range of possible conditions the program will need many IF . THEN statements to determine what to do. If you write the actions as a series of subroutines,

however, you can make things simpler using ON . GOSUB. This takes the form:

```
100 ON X GOSUB 200,300,  
400,500,600
```

If X = 1 then the subroutine at 200 will be performed, if X = 2 then routine 300 is called and so on.



| A? | GOSUB' |
|----|--------|
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |

```
40 ON A GOSUB 100, 200,
300
```

This has obvious uses in menus where the user is given a choice of actions and can select one by pressing a number. However, you can use quite complex expressions to determine the branch. As an example, think of a game where, depending on the player's score, a different hazard will appear. Let's say hazard 1 appears at 1000 points and higher, hazard 2 at 2000 points and so on. We can work things out like this:

```
10 A = INT(SCORE/1000)
20 ON A GOSUB 1000, 2000,
3000
```

The INT function in line 10 simply strips off the numbers after the decimal point so if the score is 1539 then  $1539/1000 = 1.539$  and  $INT(1.539) = 1$  so the subroutine at line 1000 is performed.

## Loops

### TRY THIS

Try this to see the principle in action:

```
10 PRINT "PRESS 1, 2 OR 3"
20 GET A$: IF A$ = ""
THEN 20
30 A = VAL (A$)
40 ON A GOSUB 100, 200,
300
50 GOTO 10
100 PRINT "YOU PRESSED 1"
110 RETURN
200 PRINT "YOU PRESSED 2"
210 RETURN
300 PRINT "YOU PRESSED 3"
310 RETURN
```

Note that if A has a value other than 1 to 3 the ON .GOSUB is ignored.

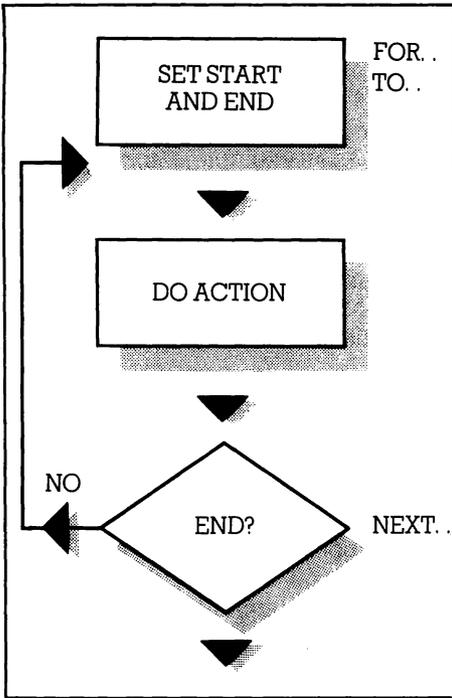
Another tool to help you to produce structured programs is the FOR .NEXT loop. This allows you to tell the computer to do something a certain number of times:

```
10 FOR I = 1 TO 10
20 PRINT "THIS IS "I
30 NEXT
```

It is useful for repetitive tasks. Imagine a program to POKE a space into every screen position:

```
10 POKE 1024, 32
20 POKE 1025, 32
30 POKE 1026, 32
```

and so on. You'd need 1,000 lines like that.



```

20 POKE SC, 32
30 NEXT

```

You can make the STEP as large as you want, or make it negative and the 64 counts backwards:

```

10 FOR I = 100 TO 0 STEP
   - 5
20 PRINT I;
30 NEXT

```

You can also have loops inside other loops, a process calling nesting:

```

10 FOR I = 1 TO 12
20 FOR J = 1 TO 12
30 PRINT I*J
40 NEXT J
50 NEXT I

```

This will print out the multiplication tables up to  $12 \times 12$ .

## AS A RULE

Or you could do it this way:

```

10 SC = 1024
20 POKE SC, 32
30 SC = SC + 1
40 IF SC = 1024 + 999
   THEN END
50 GOTO 20

```

This gets the 64 to do some of the work but it's still a bit long and complicated.

```

10 FOR SC = 1024 TO 2023
20 POKE SC, 32
30 NEXT

```

is much neater and faster.

You can also get the 64 to count in stages using the STEP extension. To put a space in every alternate screen position use this:

```

10 FOR SC = 1024 TO 2023
   STEP 2

```

The important rule with nested loops is that you must finish the last loop first. Try changing lines 40 and 50 like this:

```

40 NEXT I
50 NEXT J

```

You can remove one NEXT like this:

```

40 NEXT J, I

```

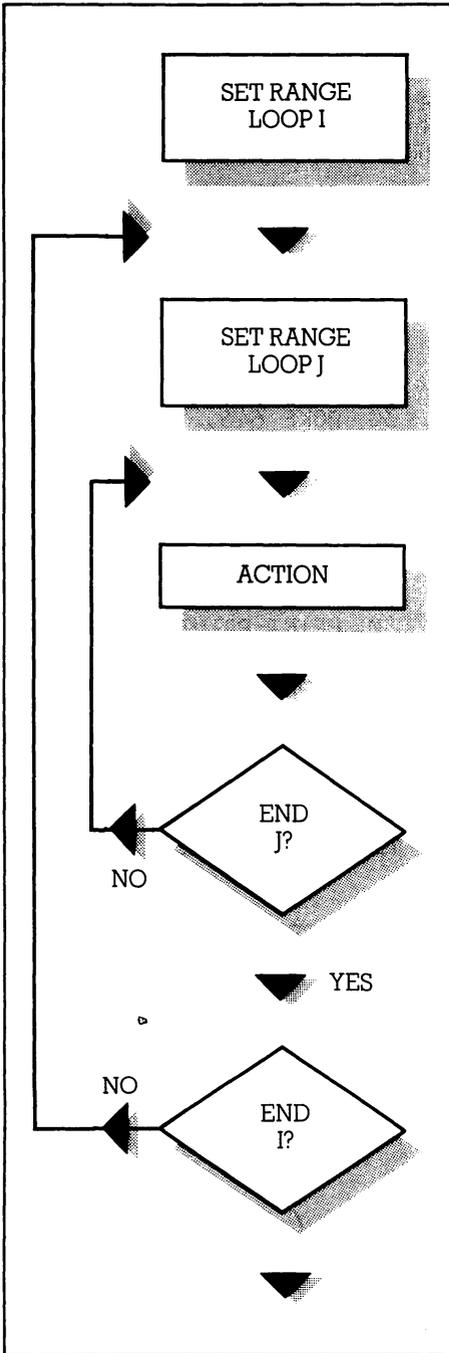
or remove the variable names like this:

```

40 NEXT
50 NEXT

```

If you leave out the variable names from the NEXT statement the 64 will automatically finish the loops in the correct order. The only reason for including them is to help during debugging – you can take them out after that.



## AS A RULE

The last rule concerning loops is the same as the one about GOSUBS: never jump out of a loop without finishing it. This is wrong:

```

10 FOR I = 1 TO 50
20 IF I = 25 THEN GOTO 40
30 NEXT
40 REM *** REST OF PROGRAM
***
  
```

The reason is the same as for GOSUBS: the 64 uses stack space to keep track of loops and the stack will fill up if you leave early. However, quite often you will want to skip the remainder of a loop for reasons of speed. Do it this way:

```

10 FOR I = 1 TO 50
20 IF I = 25 THEN I = 50
30 NEXT
40 REM *** REST OF PROGRAM
***
  
```

This way the last NEXT is always executed and the 64 can keep its books tidy.

## Keeping time

One of the most common uses of loops is as delays in a program – in music to let a note sound for the right length of time, or to let the user read something on screen. However, FOR .NEXT is a very fast command and trying to guess the amount of time a loop will take is a very haphazard affair. To help you out the 64 has a built-in clock which counts in hours, minutes, seconds and 60ths of seconds so you can create delays of very precise lengths.

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 4 | 3 | 1 | 5 | 7 |
|---|---|---|---|---|---|

The 64 has been on just over four and a half hours.

The clock is based on an interval timer which is set to zero when the 64 is turned on and is updated every 60th of a second. The timer is held in the variable TI so:

```
10 PRINT TI/60 "SECONDS
    SINCE POWER ON"
20 GOTO 10
```

will give a constant display of computing time in seconds.

TI is a 'read only' variable which means you can't alter it. However, there is another clock held in TI\$ which is updated by TI and you can alter this. Try it:

```
10 PRINT TI$
20 TI$ = "000000"
30 PRINT TI$
```

The string of zeroes can be thought of as hours, minutes and seconds like this: hh/mm/ss and the clock works on a 24-hour basis so 13 hours is one o'clock. Using the string operators (see chapter 4 for more details) we can split TI\$ into its parts:

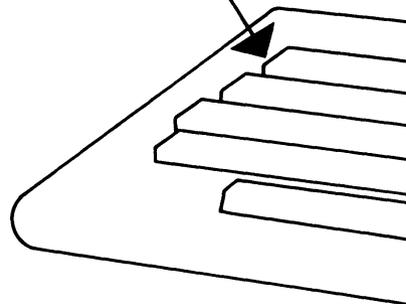
```
10 H$ = LEFT$(TI$,2)
20 M$ = MID$(TI$,3,2)
30 S$ = RIGHT$(TI$,2)
40 PRINT H$ " HOURS"
50 PRINT M$ " MINUTES"
60 PRINT S$ " SECONDS"
```

## TRY THIS

As a more practical example here's a simple reaction timer that should help you to get to know the keyboard a little better. The computer will print a word at random and measure how long it takes you to type it.

```
5 PRINT CHR$(147)
10 DIM W$(10)
20 POKE 53280,0: POKE
    53281,0
30 POKE 646, 7
40 GOSUB 500
50 PRINT "PRESS SPACE BAR
    WHEN READY"
60 GET A$: IF A$ <> " "
    THEN 60
70 X = INT(RND(1)*10+1)
80 PRINT:PRINT W$(X)
90 TI$ = "000000"
100 INPUT IN$
```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 4 | 2 |
|---|---|---|---|---|---|



```

105 IF IN$ <> W$(X) THEN
    PRINT "TRY AGAIN": GOTO
    100
110 T = VAL(TIS)
120 PRINT CHR$(147)
130 PRINT "YOU TOOK ";
140 IF T>100 THEN PRINT
    "MORE THAN A MINUTE":
    GOTO 160
150 PRINT T " SECONDS"
160 PRINT: PRINT "PRESS
    RETURN TO CONTINUE"
170 GET AS$: IF AS$ <> CHR$
    (13) THEN 170
180 GOTO 50
199 END
500 FOR I = 1 TO 10
510 READ W$(I): NEXT
520 RETURN
600 DATA TYPE,HELP,
    QUIT,NEXT,ZEAL,
610 DATA BURN,JEST,
    WALK,ROOM,COST

```

## Checklist

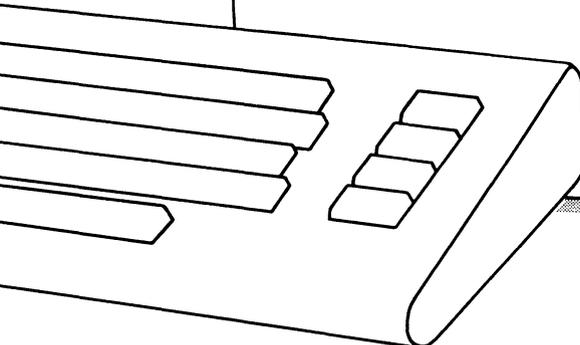
In this chapter you've learned:

- Why you should try to write structured programs.
- How and why to use subroutines (and how not to use them).
- The power of branches and how to program them.
- How to use the IF. .THEN statement and some of its limitations.
- How the ON. .GOSUB statement gives you extra power and flexibility.
- How and why to use loops for repetitive tasks.
- How to use the internal clock for delays and measuring time.

HELP

HELP

YOU TOOK MORE THAN A MINUTE  
PRESS RETURN TO CONTINUE



---

## Projects

---

- Examine some of your earlier programs for unstructured techniques and try to amend them to incorporate branches, subroutines and loops.

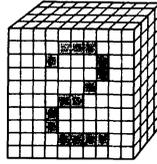
---

- Start a subroutine library on one dedicated tape or disk. Look out for new, slicker routines in books and magazines and add them to your collection. (This is not to suggest that you write programs as nothing more than amalgamations of other people's ideas. But there are too many new programs to be written to waste time looking for ways to solve old problems.)

---

- Write a program to display a clock on screen (either digital or using some kind of graphics screen) which allows the user to set the time in hours, minutes and seconds.

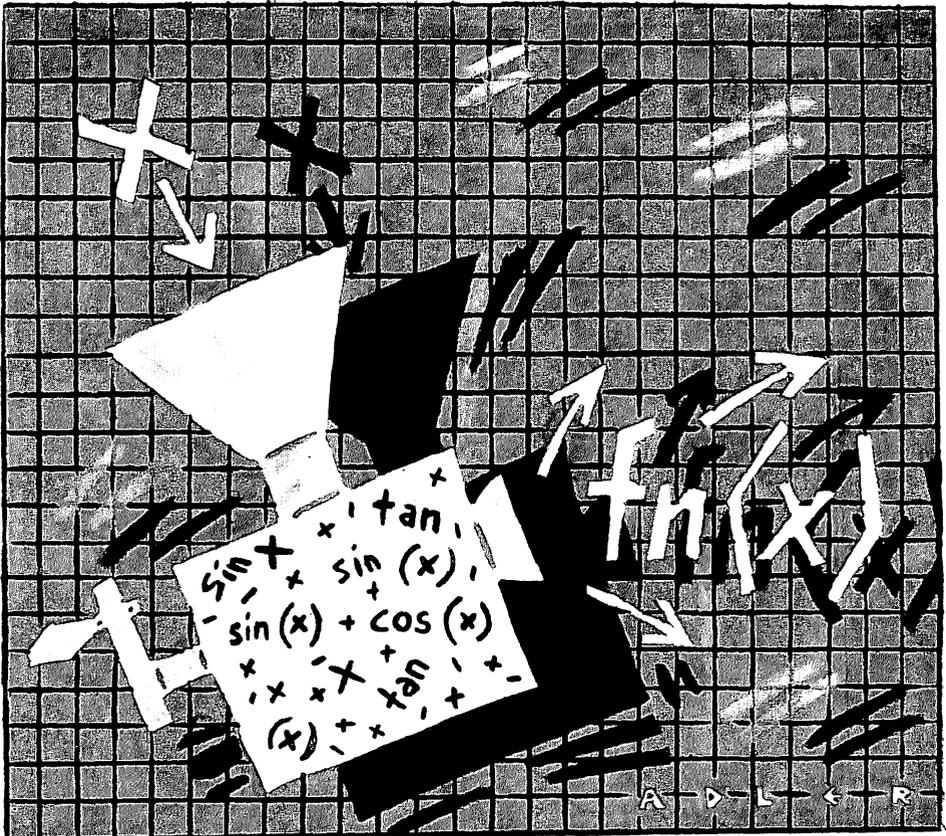
---



---

# Built-in and user-defined functions

---



## Thinking of a number...

One of the mistaken beliefs that many people have about computers is that you need to be good at mathematics to use them. This is completely untrue, although it's easy to see how it came about. After all, at their lowest level computers work with nothing but numbers. But one of the reasons for having a micro is that it can do maths quicker and more reliably than us humans.

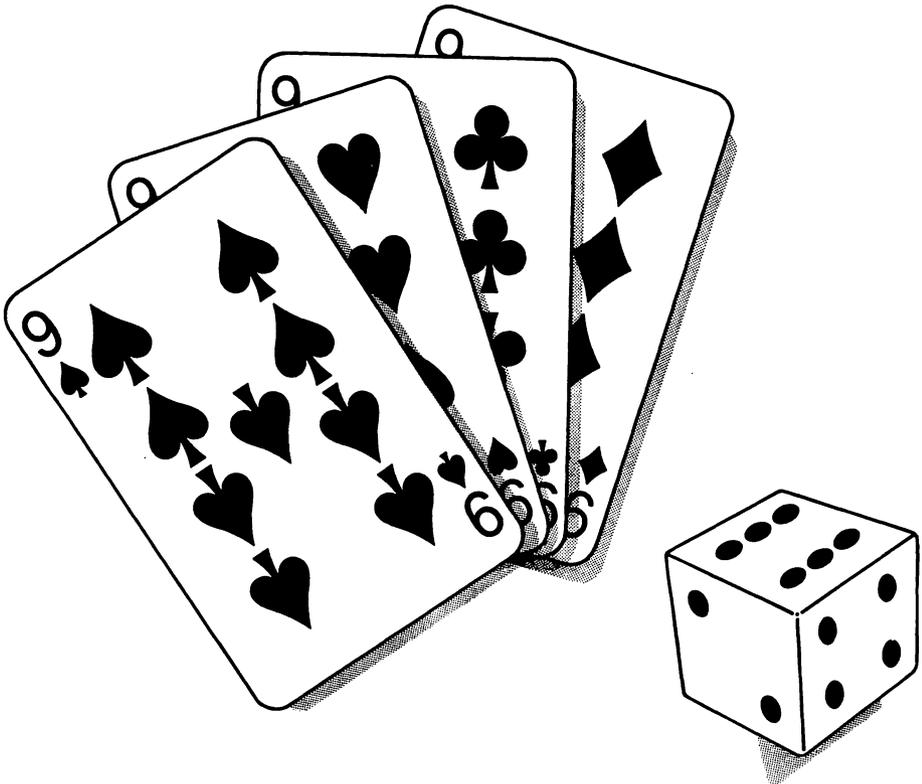
That said, the more you know about numbers the better you can program your 64. Numbers are necessary for all sorts of applications, from simple FOR..NEXT loops, to working out

display positions for simple graphics, all the way to complex trigonometry and solid geometry for advanced high resolution graphics.

In this chapter we'll look at some of the built-in commands – called functions – that you can use to let the 64 handle the hard maths in your programs.

## RND

Random numbers are just what they sound like – numbers that are determined by chance and should be impossible to predict. There are all sorts of uses for them – in card games



where you want a random shuffle; in simulations – games or serious – where you want to introduce the element of luck or unpredictability; in adventures where things like the weather can influence the play. In real life we put these things down to luck; in computers it's down to random numbers.

The function for random numbers is the RND statement and looks like this:

```
A = RND(1)
```

This produces a number between 0 and .99999999. You may have noticed the first sentence of this section says random numbers 'should be impossible to predict'. But try this program: first switch your computer off and then on again.

```
10 FOR I = 1 TO 10  
20 PRINT RND(1);  
30 NEXT
```

This will print 10 numbers across the screen in an apparently random order. Save the program, switch off, then load and run it again. If you compare the numbers generated by these two runs, you'll see that they are the same. But edit line 20 as follows and try the experiment again:

```
20 PRINT RND(0);
```

The reason for the difference is that computers do not generate true random numbers but what are called pseudo random numbers. Using the form RND(1) starts the generation from the same place in the sequence, but using RND(0) picks numbers from elsewhere in the sequence.

**There are two points to note from this:**

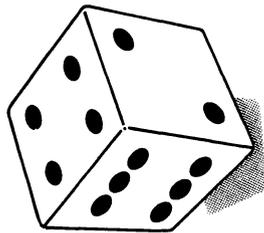
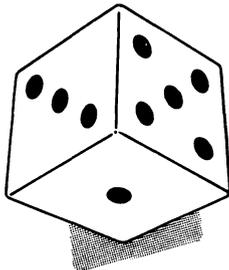
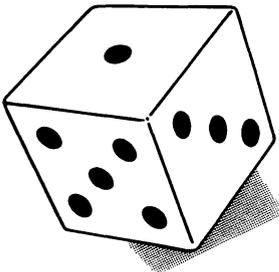
- ☐ For truly random numbers make your first RND statement in a program RND(0), and RND(1) thereafter.
- ☐ Using RND(1) throughout can be useful in debugging a program since you always get the same sequence of numbers.

To generate a range of numbers use the following form of statement:

```
A = INT(RND(1) * 10 + 1)
```

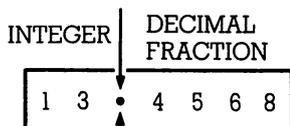
This would give numbers in the range 1 to 10. To get a middle range, for example between 10 and 15, use this:

```
A = INT(RND(1) * 5 + 10)
```



## INT

The last few programming examples introduced another mathematical function, the INT statement. This takes a number and chops off everything after the decimal point:  
INT(1.53965) becomes 1  
INT(5.32333) becomes 5



Note that it does not round numbers to the nearest whole number. To get true rounding add .5 and then take the INT value:

$\text{INT}(1.33 + .5) = 1$

$\text{INT}(1.61 + .5) = 2$

Also take care when dealing with negative numbers. INT continues to give the lowest value but this works 'upside down' with negatives:

$\text{INT}(12.6) = 12$

$\text{INT}(-12.6) = -13$

You will most often see INT used in conjunction with RND but there are other uses. For example, you cannot POKE fractions into memory locations so if you want, say, the nearest screen location to a number use  $\text{INT}(X + .5)$ . This has the same effect as using integer variables (eg,  $A\%$ ,  $D\%(1,5)$  etc). Note that in the following statement, the INT is redundant:

**10 A% = INT(X)**

A% will automatically perform the INT function.

## ABS and SGN

These two functions are connected to plus and minus values of numbers. SGN will reveal whether a number is positive or negative. For example:

$A = \text{SGN}(3)$  results in  $A = 1$

$A = \text{SGN}(-3)$  results in  $A = -1$

$A = \text{SGN}(0)$  results in  $A = 0$

It is not the most used Basic function but possible uses include games to see if the player has run out of fuel or

| ABS |    | SGN |
|-----|----|-----|
| 5   | 5  |     |
| 4   | 4  |     |
| 3   | 3  | 1   |
| 2   | 2  |     |
| 1   | 1  |     |
| 0   | 0  | 0   |
| 1   | -1 |     |
| 2   | -2 |     |
| 3   | -3 | -1  |
| 4   | -4 |     |
| 5   | -5 |     |

ammunition, or accounts programs to see if a bank account has gone into the red.

ABS returns the absolute value of a number, regardless of whether it is positive or negative:

A = ABS(25) results in A = 25  
 A = ABS(-25) results in A = 25

Again, it is not a common function but it has one extremely helpful use – in toggles. A toggle is a handy form of switch and works like this: a toggled switch becomes on if it was off, and off if it was on. It eliminates the need for two switches or keys to control one condition. In the following example, ABS allows you to switch between reverse letters by pressing the F1 key. Try typing something and then toggle the F1 key a couple of times.

```

10 PRINT CHR$(147)
20 GETAS$: IF AS$ = "" THEN
   20
30 IF AS$ = CHR$(133) THEN
   GOSUB 100: GOTO 20
40 IF REV = 1 THEN PRINT
   CHR$(18);
50 PRINT AS$;: PRINT
   CHR$(146);
60 GOTO 20
99 REM *** TOGGLE ROUTINE
   ***
100 REV = REV - 1
110 REV = ABS(REV)
120 RETURN
  
```

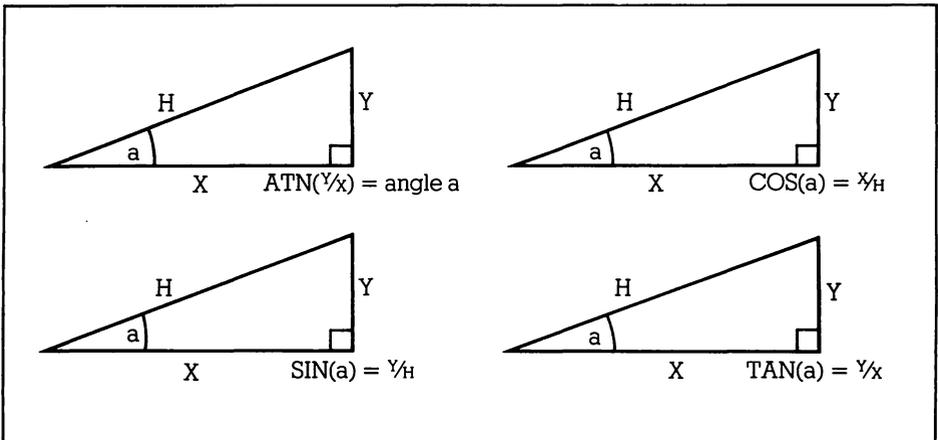
Can you work out the logic of the subroutine, bearing in mind that REV is automatically set to 0 when the program is run? Write a program to get the same effect without using ABS.

## ATN, COS, SIN, TAN

These are the trigonometry functions and are used in geometry. In computing you will come across them most often in high resolution graphics

and they all take the form:

A = ATN(B): C = COS(D):  
 E = SIN(F): G = TAN(H)



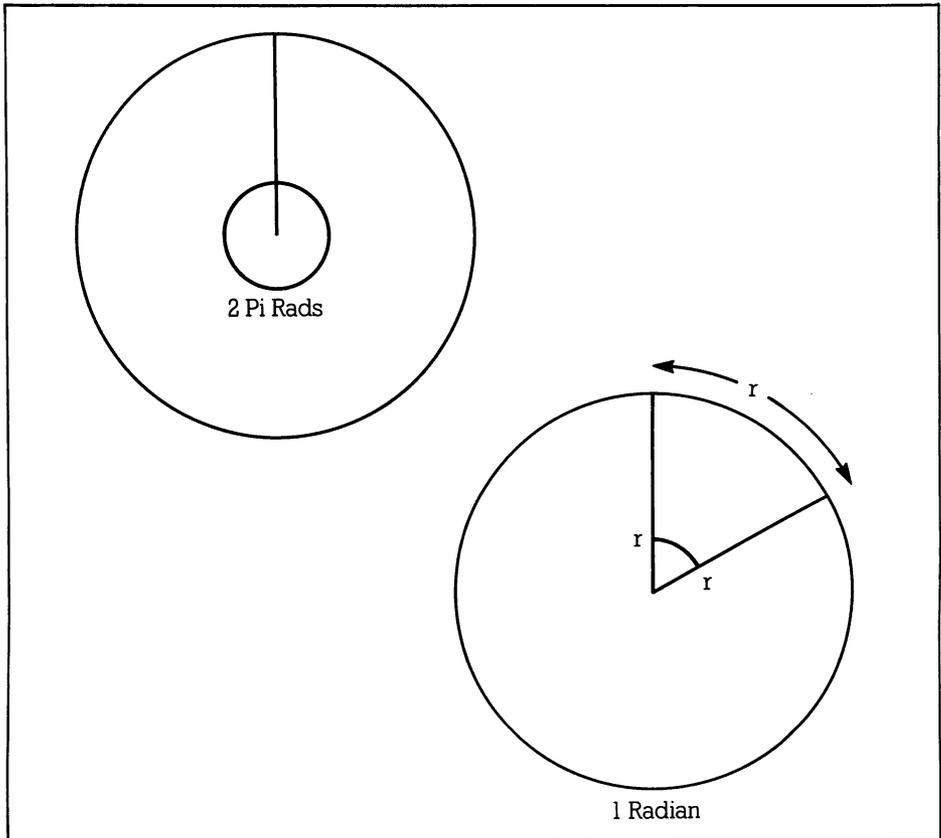
A point to beware of here concerns the way your 64 handles angles. Most of us think of angles in degrees with 360 degrees in a circle and 90 degrees in a right angle. The 64 measures angles in the mathematician's form of radians, so when using any of the trig functions, you must convert the results into degrees if that's what you want.

- To convert degrees to radians multiply by  $\text{Pi}/180$ , eg 45 degrees =  $45 * (\text{Pi}/180) = 0.79$  radians

---

- To convert radians to degrees multiply by  $180/\text{Pi}$ , eg 2 radians =  $2 * (180/\text{Pi}) = 114.59$  degrees

Trigonometry functions, along with some of the other math functions like  $\uparrow$  (exponentiation) are among the slowest Basic functions on the 64. A program that heavily relies on these is going to run very slowly. One way of speeding things up is to work out as many values as possible in advance and use these as tables in arrays. This is especially crucial in games where speed is of the essence and cutting down on processing time can make all the difference between an enjoyable game and a boring, jerky time-waster.



## User-defined functions

The ability to define your own mathematical functions is among the most useful and least used tools on the 64. If your programs are full of repeated lines like  $A = \text{INT}(\text{RND}(1) * 100 + 1)$  then user-defined functions can save memory and your own programming time.

One possible reason for the under-use of this feature is the clumsy syntax required. We need the statement, a name for the function, an argument then the expression to be evaluated. Let's see if we can sort this lot out.

There are two stages to using user-defined functions, hereafter referred to as FN. The first step is to define the function using the DEF FN statement which goes something like this:

```
DEF FN A (X) = 2 + 4
```

This is about as simple as we can make things. This function is called FN A and is used like this:

```
A = FN A (X)
```

where A will equal 6 in this case.

Obviously there's more than this and we can make the function a great deal more complicated (the 64 will let you know when you've gone too far with a FORMULA TOO COMPLEX error). In the RND example given on page 16, we could reduce it to this:

```
DEF FN A (X) = INT(RND(1) *  
10 + 1)
```

and then replace the statement throughout the program with:

```
A = FN A (X)
```

So far X has been a dummy argument, having no effect on things. But if we bring X into play we can increase the power and convenience even further. Suppose we want a general function to calculate a range of random numbers. We can change FN A like this:

```
DEF FN A (X) = INT(RND(1) *  
X + 1)
```

Now to get a number in the range 1 to 10 we can use:

```
A = FN A (10)
```

The name of the function can be any valid numerical variable (eg, X, YY, Z1), and the argument in brackets works like any other variable. The 64 will evaluate it automatically, eg:

```
Z = FN G2 (X/45+6)
```

If the program encounters an FN statement before the function has been defined it will produce an UNDEF'D FUNCTION error, so like the DIM statement, all your DEF FN commands should be at the start of the program. However, unlike DIM there will be no error if the DEF FN is executed twice (although obviously it's a waste of time).

Using user-defined functions does not offer a great speed improvement, but if you get into the habit of using them your programs will be neater, tighter, and less wasteful of memory. Try converting some of your programs and see the difference.

---

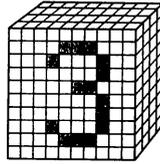
## Checklist

In this chapter you've learned:

- How RND works and how to generate a range of random numbers.
- How the INT function works and when to use it.
- How to use ABS and SGN and when they can be useful.
- How to define your own mathematical functions using DEF FN.
- How to use these functions in your programs with FN(X).

## Projects

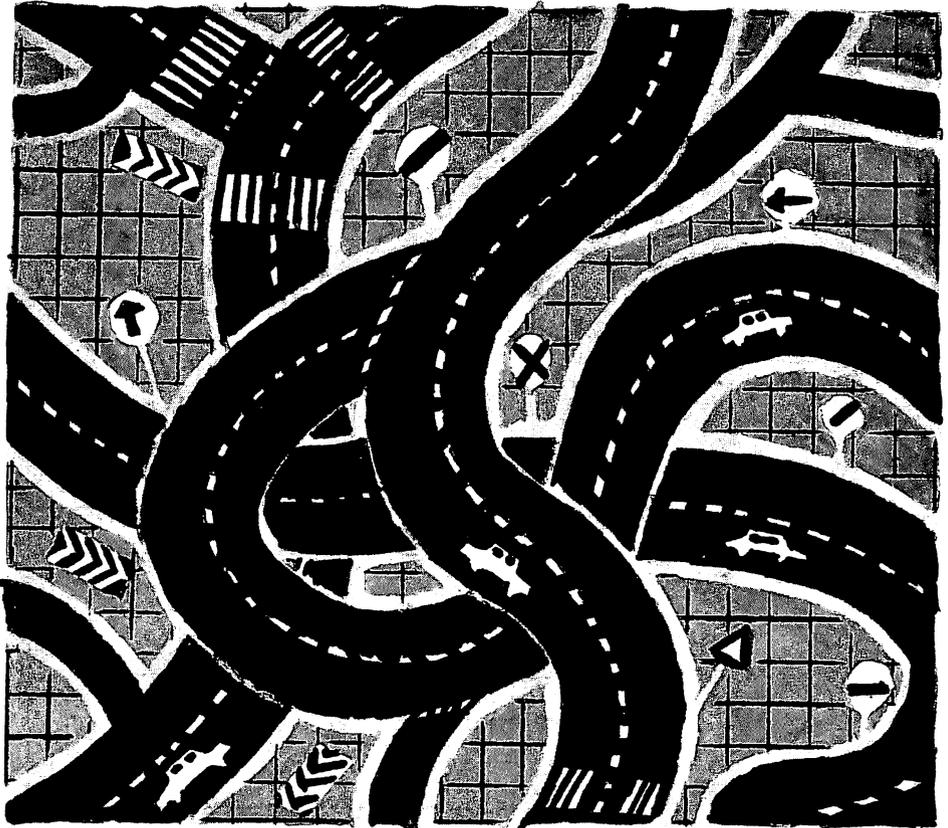
- Write a program to POKE random characters at random positions on screen and in random colours.
- Use your own functions with DEF FN to perform the calculations.
- Convert either a program you've written or one from a book or magazine to use DEF FN and FN(X).



---

# Interactive programming

---



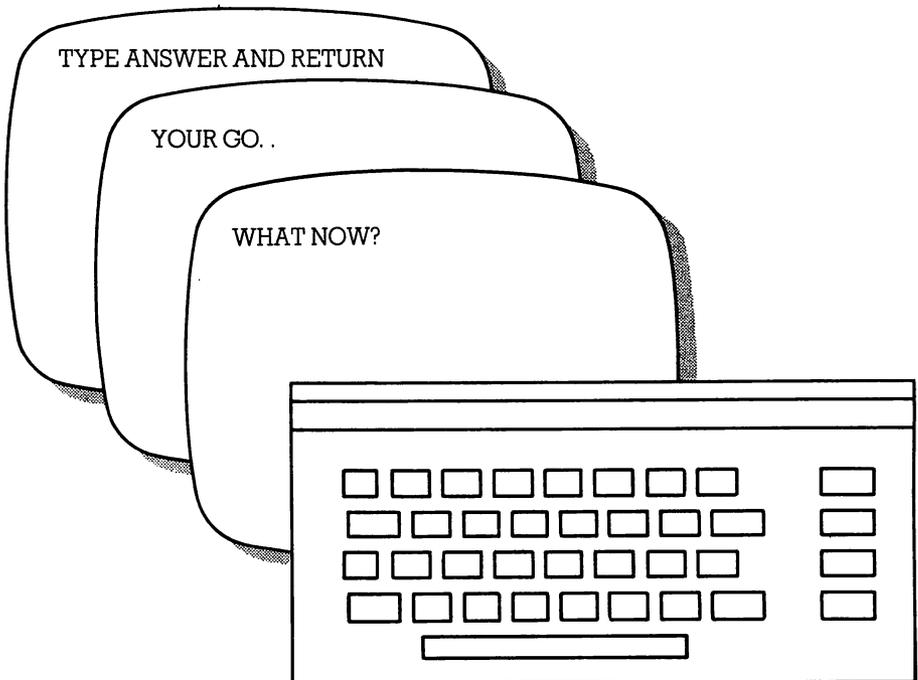
---

With the exception of the shortest of utilities, all programs are interactive to some degree. A program that searches a mass of data is not very interactive, games are highly interactive and word processing programs perhaps most interactive of all. Interaction in this sense means the way in which the program gets information from the user, and how it presents information back via the screen, printer or other peripherals.

It can be summed up in that most abused piece of computer jargon, 'user friendliness'. From the programmer's view, it means making your programs easy to use and easy to understand, debugged and failsafe.

In this chapter we'll look at ways of making your programs helpful,

interesting and consistent, and at ways of debugging and error trapping. One point to note immediately is that it is difficult if not impossible to make a program completely foolproof. As the old saying goes, you just haven't imagined a creative enough fool, but that doesn't mean you shouldn't try to anticipate potential errors and eliminate them, a process also known as mug-trapping. More important, perhaps, is that your programs should be failsafe – if an inventive idiot does use your program the consequences should not cause the destruction of data by overwriting tapes and disks, and the program should not crash leaving the user with the tedious process of reloading the program and re-entering all of the lost data.



## Two rules

'User friendly' programs have two main rules:

- ❑ **Be consistent.** Error messages, instructions etc should not appear first at the top of the screen, then at the bottom. Do not mix GET and INPUT too much. Do not require the user to enter Y or N at one point then Yes or No at another.
- ❑ **The user comes first.** You might argue that nine times out of ten you will be the user.

Then think of yourself using the program a year from now. You won't remember the instructions, your brilliant colour display might be unreadable when you've bought that monochrome monitor. Your exciting minute-long title sequence will bore even you after the hundredth time of viewing. And aren't your programs so good that all your friends want a copy?

## Getting information

The two main ways of acquiring information or commands from the user are the INPUT and GET commands. However, there are many more ways to skin this particular cat. What about joysticks and light pens? Even at the keyboard there is a third way – a direct PEEK to the keyboard. There are pros and cons to GET and INPUT. GET is very quick, requiring only a single keypress, but what if you want more than a single character? The usual choice is INPUT, but this has complications. Enter this two-line program and run it:

```
10 INPUT "ENTER A NUMBER";A
20 GOTO 10
```

Now type in a number and press RETURN. So far so good. Next type in 2,000 including the comma. You might know that the micro wants that as 2000

– no comma – but our inventive idiot doesn't. Finally type in ONE HUNDRED in letters. Another good display ruined.

## Another rule

**Never use INPUT A. Use INPUT A\$ instead.**

By using a string variable we avoid the chance of a REDO FROM START error. It is only marginally longer to use this form:

```
10 INPUT "ENTER A
NUMBER";A$: A =
VAL(A$)
20 GOTO 10
```

The second disadvantage to INPUT is the necessity to press RETURN as well. If you have used INPUT throughout the program, it is momentarily confusing to encounter a GET where the program gallops away before your hand reaches RETURN.

A minor point concerns the 'Press any key' kind of routine. Bear in mind that our inventive idiot will hit keys like CTRL, SHIFT, RESTORE and your program will wait forever for almost any key to be pressed.

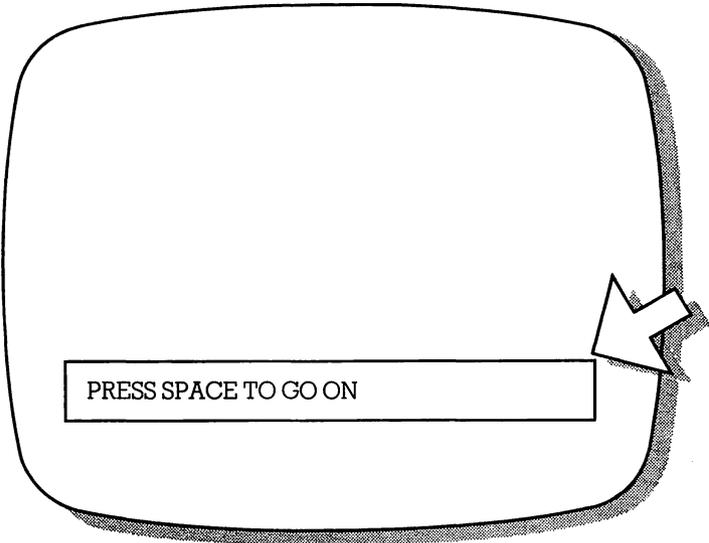
**Instead ask the user to press a specific key – why not the space bar, the biggest and most obvious key on the keyboard, or the return key, often used to enter commands.**

There are similar drawbacks in the yes/no responses. Try this:

```
10 PRINT CHR$(14): REM  
   LOWER CASE  
20 PRINT "CONTINUE? (Y/N)"  
30 GET A$:IF A$ = "" THEN  
   30  
40 IF A$ = "Y" THEN PRINT  
   "YES"  
50 IF A$ = "N" THEN PRINT  
   "NO"  
60 GOTO 30
```

Run the program and when asked the question hold down the shift key and press "y" or "n". Nothing happens because the 64 sees a distinct difference between "Y" and "y". Line 40 would have to look something like this:

```
40 IF A$ = "Y" OR A$ = "y"  
   THEN etc
```



One final point in this area: what happens if the user accidentally presses SHIFT and Commodore keys together? Your carefully designed screen display becomes a mess. However briefly, it destroys the image of professionalism. PRINT CHR\$(8) prevents this from happening and PRINT CHR\$(9) puts things back to normal.

## TRY THIS

Finally on this subject, here's an input routine that you can make as safe as you wish by making a variety of characters illegal.

```

1000 IN$ = "": IN = 0
1010 GET A$: IF A$ = ""
      THEN 1010
1020 IF A$ = CHR$(13) AND
      IN = 0 THEN 1010
1030 IF A$ = CHR$(13) AND
      IN > 0 THEN 1090

```

```

1040 IF ASC(A$) < 33 OR
      ASC(A$) > 95 THEN
      1010
1050 PRINT A$;: IN$ = IN$
      + A$: IN = IN + 1
1060 GOTO 1010
1090 RETURN

```

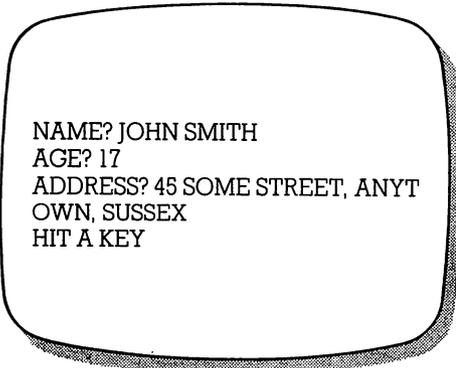
As it stands this routine will accept only the alphanumeric characters in normal text mode (ie, no colour codes or graphics characters). You can amend this by altering the parameters in line 1040. It will not accept a null string.

Further changes you could make include the addition of a flashing cursor, the ability to delete some of the input and the option of using upper and lower case letters.

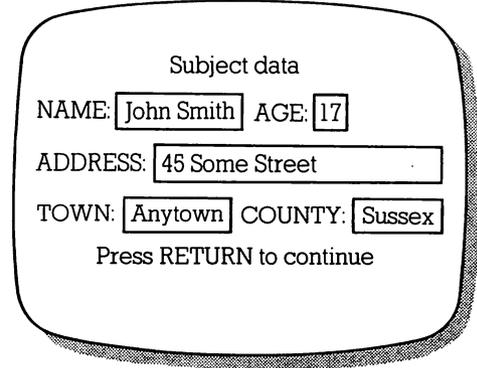
The input string is returned to the main program in IN\$.

Using this routine it's possible to add an extra professional touch by setting up defined boxes on screen to accept the input.

Wrong:



Right:

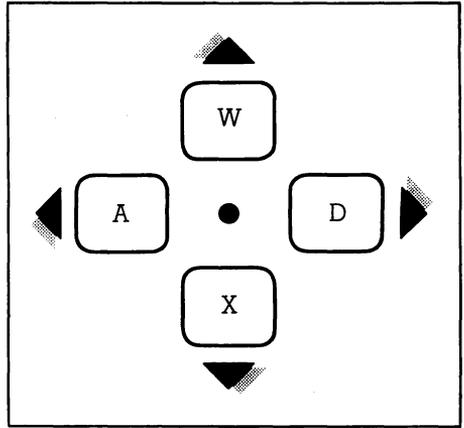


## Quick commands

So far we've looked at getting information but what about commands, particularly in the realm of game playing? Generally, the difference is one of speed – the game should respond to the command as quickly as possible (adventures are an obvious exception).

INPUT is ruled out by the need to press the RETURN key so GET is the more common method. Here's a simple example which shows the main limitation. Use W to go up, X to go down, A to go left and D to go right.

```
10 POKE 53280, 0: POKE
   53281, 0
20 SC = 1024: L = 40
30 CO = 54272
40 P = SC + 20 + L * 12
50 POKE P, 81: POKE P +
   CO, 1
60 P1 = P
70 GET AS$: IF AS$ = ""
   THEN 60
80 IF AS$ = "W" THEN P1 =
   P - L
90 IF AS$ = "X" THEN P1 =
   P + L
100 IF AS$ = "A" THEN P1 =
   P - 1
110 IF AS$ = "D" THEN P1 =
   P + 1
```



```
120 IF P1 < SC OR P1 >
   2023 THEN 60
130 P = P1
140 GOTO 50
```

This moves a ball around the screen, but for fast movement try holding a key down. Nothing happens because once the 64 has GOT the keypress it won't recognise another until you let go. You have to press the key repeatedly. We can get round this problem by making all the keys repeat in the manner of the space bar and the cursor keys. Add the following:

```
45 POKE 650, 255
```

Now try again.

## A better way

There is another way to achieve the same effect and that is by PEEKing the keyboard direct. Location 197 holds the value of the current key pressed and returns a value of 64 if no key is

held down. The snag is that these values do not correspond to either the screen display codes, or the CHR\$ codes. The table contains the values for all of the keys.

|      |       |       |       |       |       |       |
|------|-------|-------|-------|-------|-------|-------|
| ←=57 | 1=56  | 2=59  | 3=8   | 4=11  | 5=16  | 6=19  |
| 7=24 | 8=27  | 9=32  | 0=35  | + =40 | - =43 | £=48  |
| Q=62 | W=9   | E=14  | R=17  | T=22  | Y=25  | U=30  |
| I=33 | O=38  | P=41  | @=46  | *=49  | ↑=54  | A=10  |
| S=13 | D=18  | F=21  | G=26  | H=29  | J=34  | K=37  |
| L=42 | : =45 | ; =50 | = =53 | Z=12  | X=23  | C=20  |
| V=31 | B=28  | N=39  | M=36  | , =47 | . =44 | / =55 |

cursor up=7 cursor right=2 return=1 space=60 F1=4 F3=5 F5=6 F7=3

Among the other advantages of this method are that shifted, CTRLed and unshifted keys have the same values so the need for additional checks is reduced, and we get auto-repeat using this routine. Amend the program above as follows:

```

70 Q = PEEK(197):IF Q =
   64 THEN 70
80 IF Q = 9 THEN P1 =
   P - L
90 IF Q = 23 THEN P1 =
   P + L

```

```

100 IF Q = 10 THEN P1 =
    P - 1

```

```

110 IF Q = 18 THEN P1 =
    P + 1

```

This method also makes using the function keys as controls easier to evaluate since they have consecutive values (although beware F7). They become a natural choice for ON. .GOSUB routines.

## Displaying information

This is perhaps even more important than the way in which the program gets information from the user because

there are even more danger areas, and more ways in which you can make things easy for the user.

### DO

- Make your displays readable and attractive.
- Use colour to signal different processes.
- Use sound to warn or reassure.
- Be consistent in the positioning and wording of instructions and messages.

### DON'T

- Use excruciating colour combinations or graphics displays which vanish in black and white.
- Leave the user wondering what to do next.
- Ignore the details because you can't be bothered.

## Attractive displays

By using reverse video, colour and the careful use of graphics, and the positioning of items on screen, you can give your displays a very professional feel. Think how boring those programs are that have everything – input and output – in the same colour, all in capital letters, all down the side of the screen.

Even worse are the programs that try to liven things up and get it wrong with clashing colours, misaligned headings and cluttered masses of data.

### TRY THIS

This short program displays all the foreground/background colour combinations on the 64. You can see which work best together but generally the plain backgrounds – black, white, and the greys – are preferable to oranges and bright greens.

```
10 A$ = "THE QUICK BROWN  
FOX"  
20 FOR B = 0 TO 15  
30 FOR F = 0 TO 15  
40 PRINT CHR$(147)  
50 POKE 53280, B: POKE  
53281, B  
60 POKE 646, F: PRINT A$  
70 FOR D = 1 TO 500:  
NEXT D  
80 NEXT F, B  
90 POKE 646,1
```

For programs which have lots of text, keep the border colour the same as the screen. This is also good with some types of games since it gives the

impression of a larger screen.

However, the border can be used to good effect in some ways. For example, use different colours to help the user to keep track of where he or she is in the program. Use one colour for data entry, another for data retrieval and use red for danger when deleting information. You can also step through the border colours to signify that the computer is still working although nothing appears on screen. This is much more effective than a simple 'please wait . . .' message.

If you have only a black and white TV or monitor, try your programs on a friend's colour TV if you want to include lots of colour. The rest of us have less excuse since we can turn down the colour and see how our programs look in black and white.

BEEP BUZZ  
BEEP BUZZ  
BUZZ  
BEEP

## Sound

We can use sound in a similar fashion. Chapter 11 will discuss sound in a great deal more detail – for now we'll confine ourselves to beeps and buzzes in much the same way as TV quiz shows greet right and wrong answers.

### TRY THIS

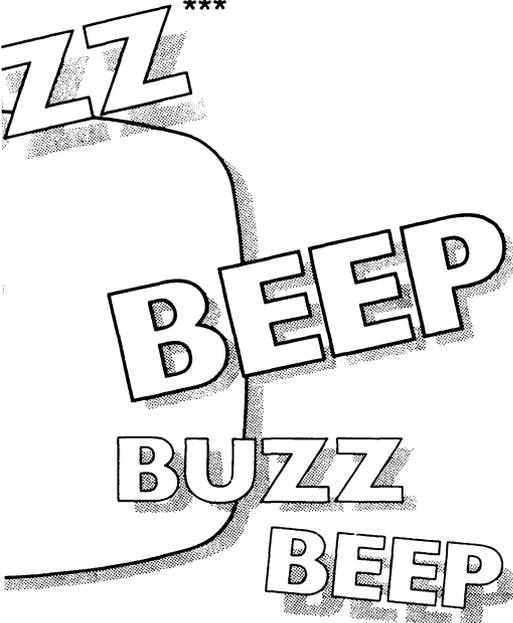
```
10 PRINT CHR$(147): GOSUB
  500
20 PRINT "DO YOU LOVE YOUR
  MICRO?"
30 GET A$: IF A$ = ""
  THEN 30
40 IF A$ = "Y" THEN GOSUB
  200
50 IF A$ = "N" THEN GOSUB
  300
60 GOTO 10
199 REM *** PLEASANT CHIME
  ***
```

```
200 POKE S + 5, 12: POKE
  S + 6, 10
210 POKE S + 1, 65: POKE
  S, 50
220 POKE S + 4, 17
230 FOR I = 1 TO 250: NEXT
240 POKE S + 4, 16
250 RETURN
299 REM *** UNPLEASANT
  BUZZ ***
300 POKE S + 3, 118: POKE
  S + 2, 35
310 POKE S + 5, 62: POKE
  S + 6, 0
320 POKE S + 1, 5: POKE S,
  50
330 POKE S + 4, 65
340 FOR I = 1 TO 250: NEXT
350 POKE S + 4, 64
360 RETURN
499 REM *** INITIALISE SID
  CHIP ***
500 S = 54272
510 FOR I = 0 TO 24
520 POKE S + I, 0: NEXT
530 POKE S+24,15
540 RETURN
```

This simple use of sound is one of the easiest yet most effective ways of making your programs more professional and pleasant to use.

## Error trapping

Some micros have a Basic command ON ERROR which directs the program to jump to a routine written by the programmer to handle mistakes. Unfortunately the 64 does not have this command so it is up to you to make sure no errors can arise, otherwise the program will crash with an error message.



## Formatting screens

There are a number of Basic tools at your disposal to help in setting up your displays to be readable.

**Cursor controls:** These are useful for the relative positioning of text and graphics but can be clumsy in working out absolute positions. For example, if you want to put a word in the middle of the last screen line you need 25 cursor downs, and 15–20 cursor rights.

**TAB()** and **SPC()**: can do away with the strings of cursor rights. **TAB** can have an argument from 0 to 255 where 0 is at the extreme left of the screen. Note that if the cursor is already past the position designated it will skip to that position on the next line.

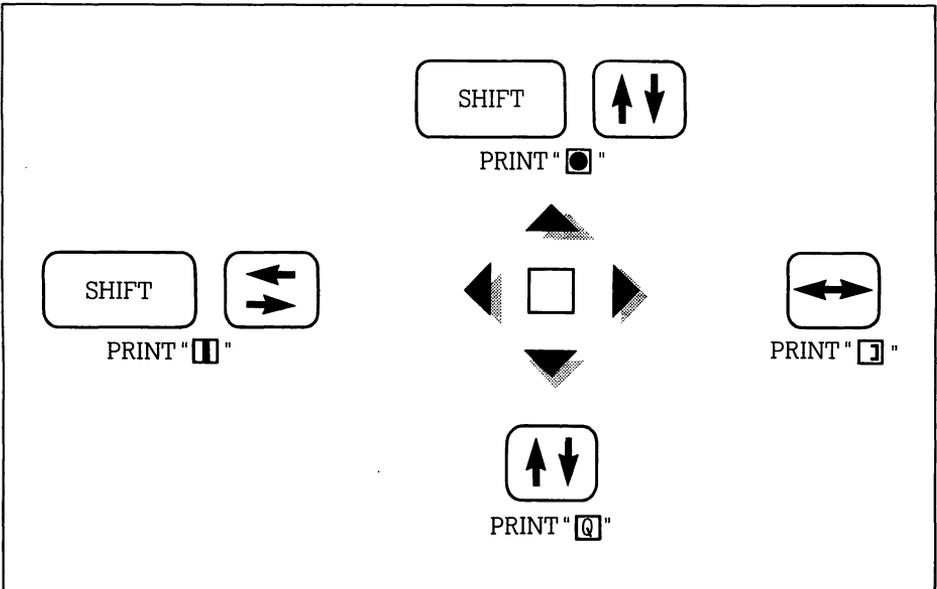
**SPC** works relative to the current position, so **SPC(5)** will move the cursor five places to the right. It does not print five spaces so anything

already printed in those five spaces will not be overwritten.

**Punctuation:** easily overlooked for screen formatting are the , and ; operators. The comma works like a pre-set tab stop on a typewriter. Think of the screen as divided into four columns of ten places. The comma will set the cursor position to the next available column.

The semicolon (;) keeps the cursor position immediately next to the last thing printed. This program demonstrates the different effects:

```
10 PRINT CHR$(147)
20 FOR I = 1 TO 12
30 PRINT INT(RND(1)*10);
40 NEXT
50 FOR I = 1 TO 12
60 PRINT INT(RND(1)*10),
70 NEXT
```



## A utility

Undoubtedly the greatest help to screen formatting is a PRINT AT command but again the 64 does not have one. This combines the speed of the PRINT statement with the ease of POKE in locating a screen position. Fortunately PRINT AT is fairly easy to implement using a short machine code routine. Enter the following program:

```

10 PRINT CHR$(147): GOSUB
   1000
20 AT = 49152
30 SYS AT, 10,10,"TEST
   STRING"
900 END
999 REM *** LOAD MACHINE
   CODE ***
1000 FOR I = 0 TO 38:
   READ A
1010 POKE 49152 + I, A:
   NEXT
1020 RETURN
1029 REM *** MACHINE CODE
   DATA ***

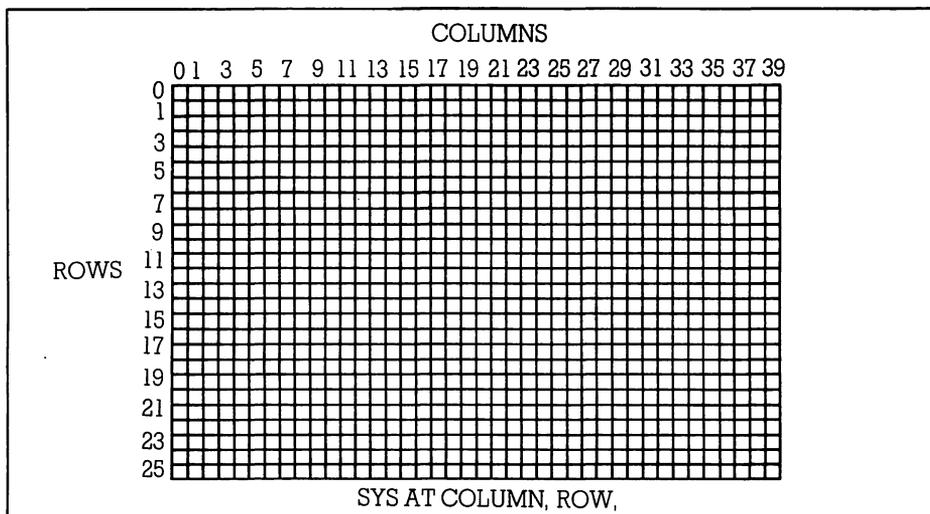
```

```

1030 DATA 32, 241, 183,
   134, 87, 32, 241
1040 DATA 183, 134, 88,
   165, 87, 201, 40
1050 DATA 176, 6, 165, 88,
   201, 25, 144
1060 DATA 3, 76, 72, 178,
   166, 88, 164
1070 DATA 87, 24, 32, 240,
   255, 32, 253
1080 DATA 174, 76, 160,
   170

```

Double check the numbers in the DATA statements and save this before you run it. One mistake in machine code is generally enough to lock up the machine and the only way to recover is to switch off. It won't hurt the 64 but it does your mental health no good at all.



## TRY THIS

Now let's try a simple demonstration using our version of PRINT AT, SYS AT. Our version uses the syntax

```
SYS AT, X, Y, "STRING"
```

where X is between 0 and 39 and Y is between 0 and 24.

Add the following routine:

```
40 A$ = " DEMO "  
50 B$ = CHR$(18) + A$ +  
   CHR$(146)  
60 Y = 12: FOR X = 0  
   TO 17  
70 SYS AT, X, Y, A$  
80 FOR D = 1 TO 50:  
   NEXT D, X  
90 SYS AT, X, Y, B$  
100 FOR D = 1 TO 100: NEXT  
110 SYS AT, X, Y, A$  
120 FOR D = 1 TO 100: NEXT  
130 GOTO 90
```

**REMEMBER:** better boring but useful than clever and useless. But best of all is a colourful, attractive and readable display.

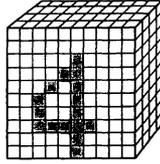
## Checklist

In this chapter you've learned:

- Why you should error-trap your programs and some techniques for doing so.
- An alternative to INPUT and GET for getting information from the user.
- A faster way of getting single-key commands using PEEK(197).
- Some techniques for making your programs more professional using colour and sound.
- A fast and easy way to format displays using SYS AT.

## Projects

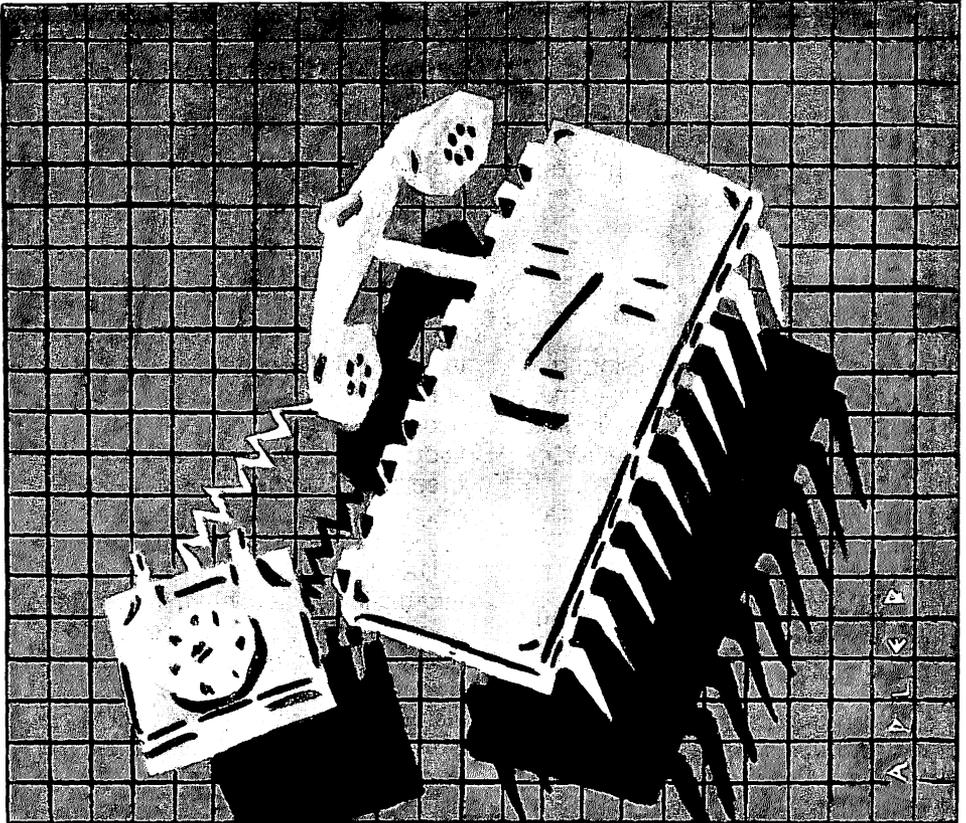
- Write a program to accept a name and address from the user using the foolproof INPUT routine.
- Develop the INPUT routine so that cursor and colour controls are illegal (check for CHR\$ codes).
- Use SYS AT to display the information back again with the screen neatly divided into information, command and message areas.



---

# Information handling

---



Reduced to the simplest level computers are nothing more than information processors, whether that information is in pure number form or numbers representing a company's accounts, a memo to the managing director or the relative positions of a space invader and your missile base. Some of that information is looked after automatically by the operating system – the Basic language ROM in the case of the 64. Variables and strings are stored at the top of memory above your Basic program while information to be displayed on your TV or monitor is kept in a special part of RAM called screen memory.

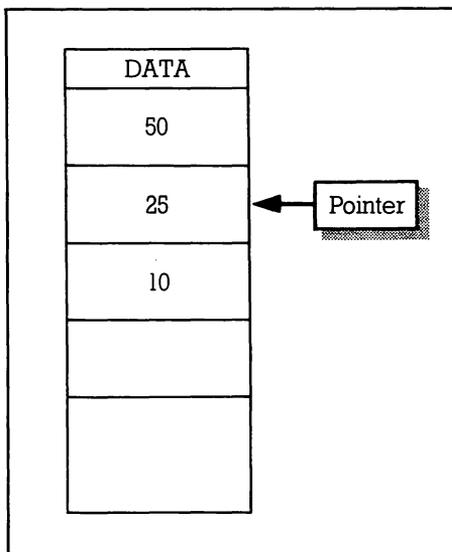
However, to get the most from the 64 it is up to you to organise the information to suit the different tasks. In the last chapter we looked at ways of getting information from the user of a program. Here we'll see how to organise the data so that the program can make the best use of it.

### One way

The most obvious means of storing data is by way of the DATA statement. This Basic keyword tells the program that what follows is not to be acted on but is merely a list of numbers or strings or both. The program gets at this information using the READ statement:

```
100 FOR I = 1 TO 3: READ
    A: NEXT
110 DATA 50, 25, 10
```

This program successively sets A to 50, 25 and 10. If for some reason you want to set A to 50 again you need to RESTORE the data pointer to the



beginning and then READ A once more. This obviously has limitations. On some micros you can RESTORE the data pointer to a given line number but the 64 doesn't have this facility. So to get at information in the middle of a group of DATA statements you have to go back to the start and then use a dummy loop to READ past the items you don't want before reaching the target number or string.

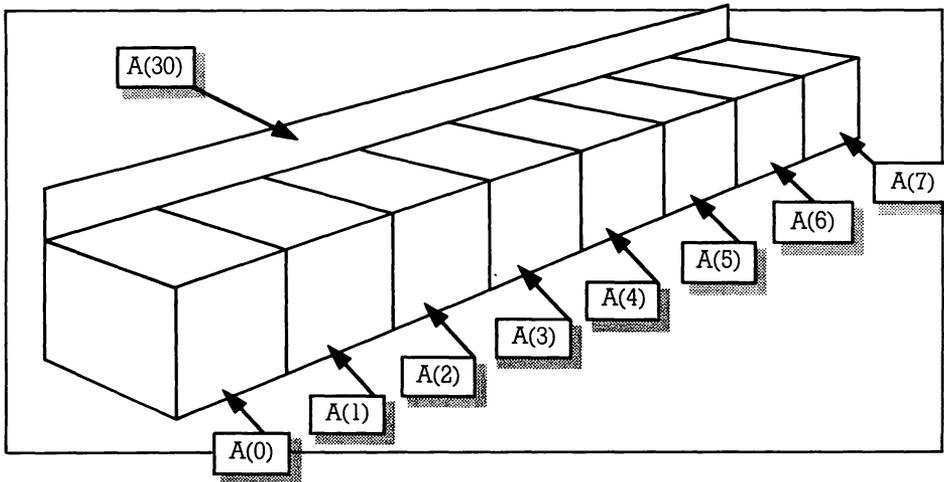
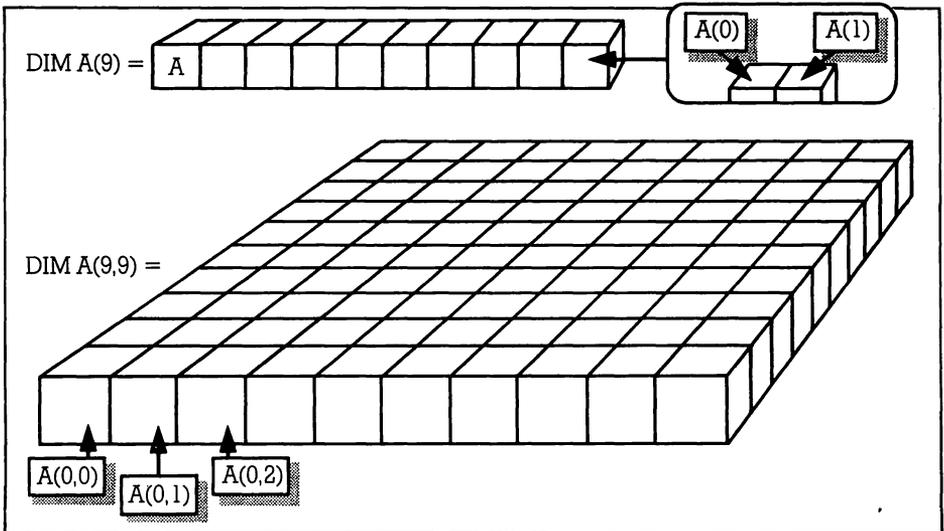
READ and DATA can be used to good effect when you have the information to start with: for example, in setting up sprites or user-defined graphics, or for holding the data for a piece of music. The obvious limitation of DATA statements is that the information is fixed when the program is written. Although it is possible to manipulate the way the 64 stores a program to incorporate new DATA statements, it is little more than a programmer's trick and it remains a very inflexible way of storing and using information.

## A better way

Arrays provide us with a much better way and are perhaps the most powerful programming tool available to the Basic programmer. They provide a neat and logical way of storing information and can be changed according to need.

The command to set up an array is

the DIM statement, which tells the micro to DIMENSION an area of memory and to give it a name. Think of it like this: DIM A(30) instructs the 64 to set up a large box called A containing 31 smaller boxes called A(0), A(1), A(2) and so on. Technically, the smaller boxes are called subscripts of the



array. In this case the computer expects the boxes to store numbers. You can also define string arrays using DIM A\$(X). This will hold up to X+1 strings (never forget the A(0) subscript).

Using arrays has some obvious benefits. For example, if you have three related variables you don't have to give them three separate names such as A\$ = JOHN: B\$ = ALAN: C\$ = GEORGE. You can use A\$(1), A\$(2), A\$(3). Note that A\$(1) and A1\$ are completely different variables and you can use them both in the same program – you may get confused but the 64 will cope perfectly.

Suppose you want to keep a record of the best selling records each week. Without using an array the program is

going to get complicated and might go something like this:

```

10 INPUT T1$: REM RECORD 1
20 INPUT T2$: REM RECORD 2
.
.
.
100 INPUT T0$: REM
    RECORD 10

```

You'd also need a separate group of strings to hold the names of the artists. If, at a later date, you need to find out whether world-famous band The Bloggs appeared in the chart and at what position, you'll need a morass of IF...THEN constructions to get the answer.

Using an array, things are much easier:

### The program

| T\$(9,1) |          |          |
|----------|----------|----------|
|          | T\$(.,0) | T\$(.,1) |
| 0        |          |          |
| 1        |          |          |
| 2        |          |          |
| 3        |          |          |
| 4        |          |          |
| 5        |          |          |
| 6        |          |          |
| 7        |          |          |
| 8        |          |          |
| 9        |          |          |

```

10 DIM$(9): REM 10 TITLES
20 FOR I = 0 TO 9: INPUT
   T$(I): NEXT: REM ENTER
   10 TITLES

```

We have reduced a dozen lines of program to two. To keep a matching record of the artists simply requires the addition of another dimension. We can amend our two-line program as follows:

```

10 DIM T$(9,1)
20 FOR I = 0 TO 9
30 INPUT T$(I,0): REM GET
   TITLE
40 INPUT T$(I,1): REM GET
   ARTIST
50 NEXT

```

Now checking on The Bloggs is easy:

```

100 A = 11: FOR I = 0 TO
   9: REM SET FLAG AND
   BEGIN LOOP
120 IF T$(I,1) = "THE
   BLOGGS" THEN A = I:
   I = 9: REM SET FLAG
   AND EXIT LOOP
130 NEXT
140 IF A = 11 THEN PRINT
   "NOT FOUND": END
150 PRINT "THE BLOGGS ARE
   AT NO.": A

```

If you want more than one week – a whole year for example – just add another dimension: DIM T\$(9,1,51). This aspect of arrays – being able to have several dimensions – gives them even more power. The only danger with using multi-dimensional arrays is that you may become confused about which subscripts represent what information. Liberal use of REM statements and a little care should keep everything straight.

## Caution

Another consideration in using arrays is the amount of available memory. Arrays can take up a frightening amount of RAM which may not be used so be careful to dimension arrays to a reasonable size. If at the start of the program you have to guess at the size of array you need, be sure to go back later and set it correctly. Enter the following to see the principle in action:

```
10 DIM A$(600,20)
```

Enter PRINT FRE(0)-(FRE(0)<0)\*65536 and note the answer. Now run this one-line memory eater and check memory again. This is why you cannot get a real idea of a program's size unless it has been run.

## AS A RULE

Two other points about arrays. Wherever possible DIM all arrays in the first line of a program. This should ensure that you don't dimension them twice resulting in a REDIM'D ARRAY ERROR. If you need to reset an array you must perform a CLR first which will wipe out all your other variables as well. Secondly, although it is not necessary to DIM an array of fewer than 11 elements, you should do so. It keeps things tidy.

## String operators

So far we've looked at storing and manipulating data in large chunks. What about smaller pieces of information like individual words?

Basic provides a number of commands that let us examine information and change things around at this level. The three most important are the string operators RIGHT\$, MID\$ and LEFT\$.

You may read elsewhere that it is possible to add strings: A\$ = B\$ + C\$. This is misleading – what is actually happening is that the strings are joined and while you cannot subtract strings you can divide them up again.

RIGHT\$ operates on the rightmost portion of a string and takes the form A\$ = RIGHT\$(B\$,5). In this case A\$ will be set to the five characters on the right of B\$.

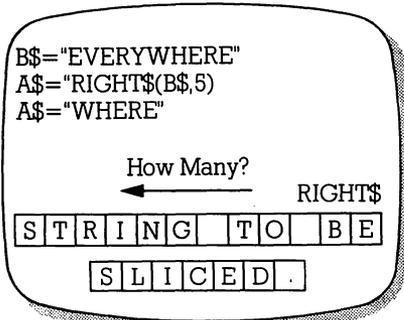
LEFT\$ works in a similar manner at the other end of the string.

The potentially confusing one of the three is MID\$. This takes the middle portion of a string but here 'the middle' can be any part of a string or even all of it.

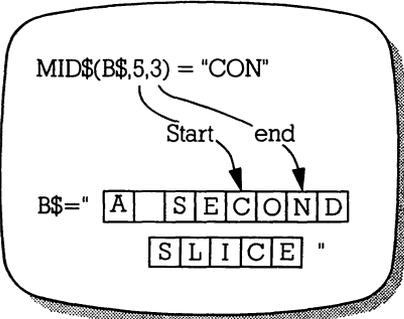
If B\$ = "123456789", MID\$(B\$,4,3) will produce "456", the three characters starting at position 4. This is the obvious use but MID\$ almost makes RIGHT\$ and LEFT\$ redundant: MID\$(B\$,1,4) will produce "1234", MID\$(B\$,5,5) gives "56789" and MID\$(B\$,1,9) returns the whole of B\$. So why use RIGHT\$ and LEFT\$ at all? Because you can see at a glance that RIGHT\$(B\$,6) is doing something to the end of a string, while unless you know exactly how long B\$ is, MID\$(B\$,3,25) tells you very little.

At this point there is one other string operator that we need and that is LEN. This gives the length of a string. If B\$ is once more "123456789" then LEN(B\$) will return the number 9.

```
B$="ANY"
C$="THING"
A$=B$+C$
A$="ANYTHING"
```



```
B$="EVERYWHERE"
A$="LEFT$(B$,3)
A$="EVE"
```

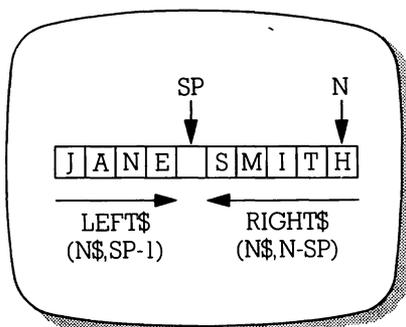


## TRY THIS

Using string operators we can get the computer to work some of the information needed by a program. In the following example the computer will determine Christian names and surnames from a single input string:

```
10 SP = 0: INPUT "WHAT IS  
   YOUR NAME";N$  
20 N = LEN(N$)
```

```
30 FOR I = 1 TO N: IF MID$(  
   N$,I,1) = " " THEN SP  
   = I: I = N  
40 NEXT  
50 IF SP = 0 THEN 10  
60 C$ = LEFT$(N$,SP-1)  
70 S$ = RIGHT$(N$,N-SP)  
80 PRINT "CHRISTIAN NAME  
   = "C$  
90 PRINT "SURNAME = "S$
```



## How it works

In line 10 the user enters his or her name, for example "Jane Smith". Line 20 gives the length of the string, in this case 10. Line 30 does the work for us, stepping through the name string a character at a time until it finds a space. At this point  $I = 5$  so it sets the space pointer to five and  $I$  equal to  $N$  and exits the loop in line 40. If no space had been included the entry would be invalid so the program goes to line 10 for another try.

Line 60 takes the letters to the left of  $SP$  and calls them the Christian name. Line 70 gets the other end and calls it the surname.

The VAL function is also a string operator. It gives the numeric value of a string:  $A = \text{VAL}("45.2")$  sets  $A = 45.2$ . However, it also takes only the numbers from a string:

```
10 INPUT "AGE AND NAME";N$  
20 PRINT VAL(N$)
```

is a simple way of extracting the numbers from the input string. But it will only work if the number is at the start of the string. PRINT VAL("XYZ123") will return a zero.

## The shortest adventure in the world

In order to see how these ideas work in practice let's take a look at one of the most common types of programs to make use of arrays and string manipulation: an adventure game.

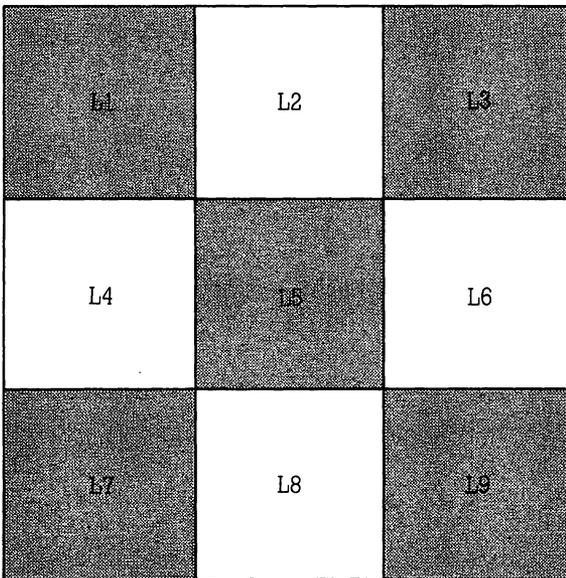
Since most adventures are quite long, and it's not much fun trying to solve an adventure you've typed in yourself, this one will be quite short, hence its title: the Shortest Adventure in the World. However, it works in much the same way as larger games of this type and if you follow it through you should be able to tackle a larger project. It's only a question of scale.

First let's look at what we need.

Obviously, a map is necessary to be able to locate what's where. We also require a number of descriptions of locations and objects. The other major requirement is the vocabulary of the game – what words the computer will understand.

Next, we need to consider the structure of the program. The map can be most simply imagined as a square building, three rooms wide by three deep. All of the information will be held initially in DATA statements – room descriptions, objects and the vocabulary of nouns and verbs.

However, it would be unworkable to



operate on these using READ and RESTORE so we'll transfer them to arrays where they'll be more accessible.

Keeping track of the player's location is a matter of having a pointer while, as is customary in these games, commands will be in the form of verb, noun (go west, get knife) and can be handled quite simply by our string slicing routine to sort out LEFT\$ and RIGHT\$. Apart from a few other variables to keep track of odds and ends, that's all there is to it but as you'll see, adventures involve large amounts of programming and even a miniature like this takes a lot of code.

```

10 PRINTCHR$(147)
   CHR$(30)
20 POKE53280,0:
   POKE53281,0
30 L=1:GOSUB1000

```

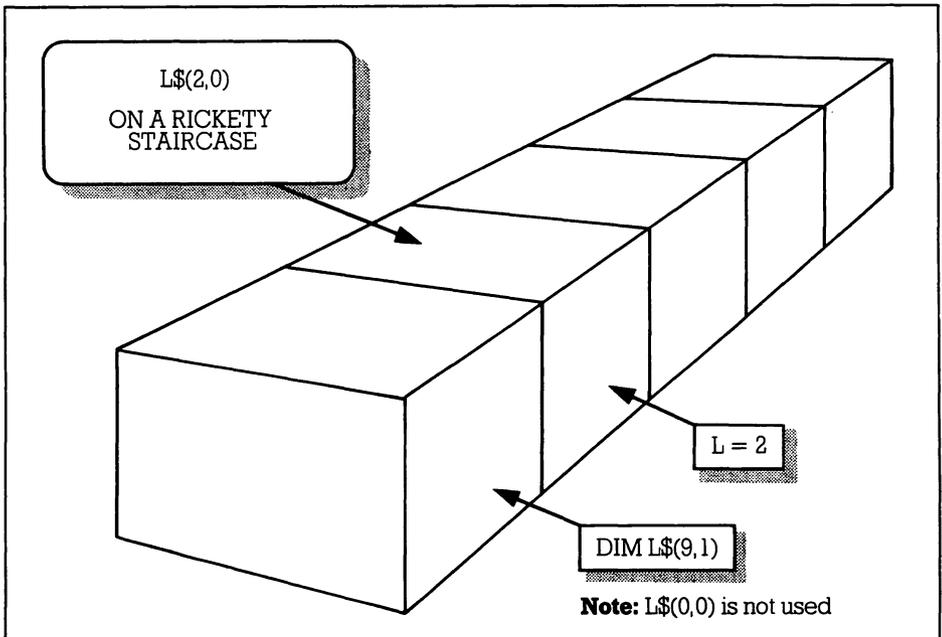
A straightforward start. These lines simply set up the screen colours (green text on a black background), set the player's location L to the start and go to the initialisation routine at 1000.

```

40 PRINT"YOU ARE ";
   L$(L,0)
50 PRINT"POSSIBLE EXITS
   ARE ";
60 FORI=1TOLEN(L$(L,1))
70 ES=MID$(L$(L,1),I,1):
   E=VAL(ES)
80 PRINTES(E)" ";:
   NEXT: PRINT
90 PRINT"WHAT SHALL I DO
   NOW?"

```

This block is the main display section. L\$(L,0) holds the description of the current location. L\$(L,1) is a related string which holds the possible exits.



YOU ARE IN A DARK DAMP CELLAR  
THICK WITH COBWEBS.  
POSSIBLE EXITS ARE EAST  
WHAT SHALL I DO NOW?

Lines 70 and 80 slice the string and print the valid directions which are held in a master direction array, E\$( ).

```
100 GOSUB500
110 IFV$<>"GO"THENPRINT"I
    DONT KNOW HOW TO DO
    THAT":GOTO90
```

We'll come to the routine at 500 shortly. Line 110 checks to see whether the user has entered a valid verb command. In our very limited example the only allowable verb is GO. To extend the program you could call another subroutine here to check for all possible legal commands and act on them accordingly.

```
120 GOSUB600
130 IFD=-9THENPRINT"I
    CANT GO THAT WAY":
    GOT050
```

D is a direction flag with a default value of -9 which is changed by the entry of a valid direction, ie north, south, east or west.

```
140 EX=-1
150 FORI=1TOLEN(L$(L,1))
160 IFD=VAL(MID$(L$(L,1),
    I,1))THENEX=D
```

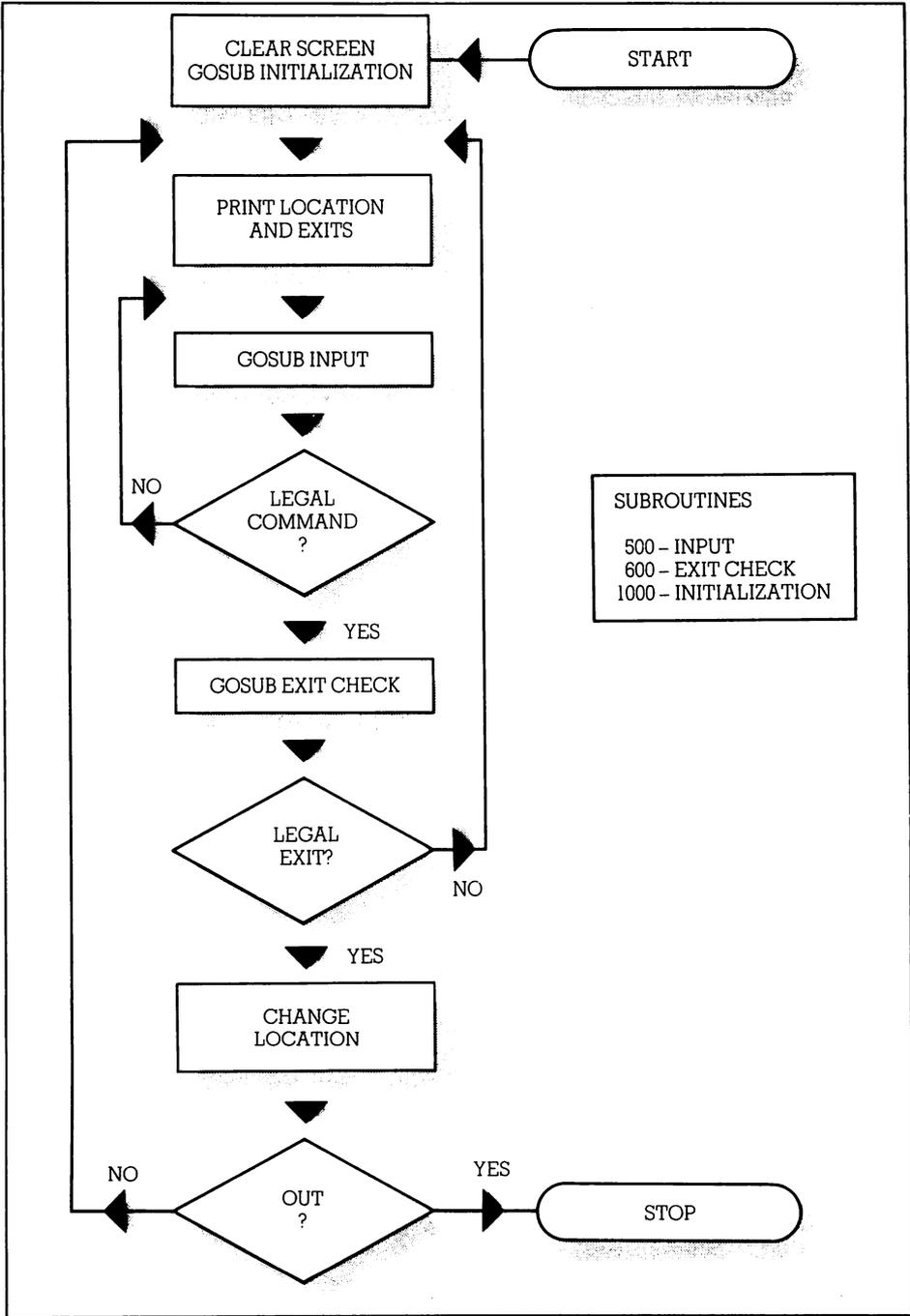
```
170 NEXT
180 IFEX=-1THENPRINT"I
    CANT GO THAT WAY":
    GOT050
190 L=L+D(D)
200 IFL=11THEN800
210 GOT040
```

These lines control the movement of the player. Having already checked that the direction is legal, the program now checks to see if the direction is valid at the current location. EX is another flag which is changed if the direction chosen matches one of the possible directions held in L\$(L,1). If it doesn't change then a suitable message is printed, followed by the legal directions again.

Line 190 updates the position in much the same way as screen locations work. Possible directions are up (-3), down (+3), right (+1) and left (-1). These values are held in array D( ).

Finally line 200 checks to see if you have left the house before returning to the core routine at line 40.

```
500 C$=" ":INPUTC$ :V=0
510 C=LEN(C$):FORI=1TOC
520 IFMID$(C$,I,1)≠" "
    THENV=I:I=C
```



```

530 NEXT:IF V=0 THEN 500
540 V$=LEFT$(C$,V-1)
550 N$=RIGHT$(C$,C-V)
560 RETURN

```

You should recognise this routine from earlier in this chapter. It is the same one that we used to get Christian names and surnames from an input string. It does exactly the same job for verb and noun, returned in V\$ and N\$.

```

600 D=-9:FORI=1TO4
610 IFN$=E$(I)THEND=I:I=4
620 NEXT
630 RETURN

```

A short routine to check whether the user has entered a valid direction (ie they haven't typed in GO UP, GO OUT or GO MAD). In a full-scale adventure you would need a longer section to check for all valid nouns, check to see whether the player had them or could see them and then take appropriate action depending on the verb command.

```

800 PRINT"YOU ARE OUTSIDE
      THE HOUSE. WELL DONE!"
999 END

```

The conclusion, reached when you step out of the front door.

```

1000 DIML$(9,1), E$(4),D(4)
1010 E$(1)="NORTH":E$(2)=
      "EAST":E$(3)="SOUTH":
      E$(4)="WEST"
1020 D(1)=-3:D(2)=1:D(3)=
      +3:D(4)=-1
1030 FORI=1TO9:READL$(I,0):
      READL$(I,1):NEXT
1040 RETURN

```

The set-up lines which dimension the arrays for locations, exits and direction values, and move the description strings from the DATA

statements into the array.

```

1050 DATA"IN A DARK DAMP
      CELLAR THICK
      WITHCOBWEBS",2
1060 DATA"ON A RICKETY
      STAIRCASE. IT DOES
      NOT FEEL VERY SAFE",24
1070 DATA"AT THE TOP OF THE
      STAIRS BEHIND AN OPEN
      WOODEN DOOR",43
1080 DATA"IN THE DINING
      ROOM, LONG SINCE
      ABANDONED",23
1090 DATA"IN THE KITCHEN.
      THERE IS A
      CUPBOARD HERE",24
1100 DATA"AT THE CORNER OF
      A CORRIDOR. THERE
      IS A DOOR LEADING
      NORTH",134
1110 DATA"IN THE LIVING
      ROOM THOUGH NO-ONE
      HAS LIVED HERE FOR A
      WHILE",12
1120 DATA"IN THE FRONT
      HALL. THE DOOR IS TO
      THE SOUTH",243
1130 DATA"IN THE STUDY
      WHICH APPEARS TO
      HAVE BEEN
      RANSACKED",14

```

Finally, the DATA statements themselves. Here we have only minimal descriptions of nine rooms. A full-scale adventure might have 50 or 100 rooms and each requires a description. You would also need DATA and arrays to hold all of the objects appearing in your adventure and all the valid commands. It is this mass of data which makes adventures so demanding on memory.

Although this adventure will take you longer to type in than to solve, it demonstrates the tools you need to

---

write this kind of program. In the Projects you'll find some ideas on how to extend it into something more entertaining.

---

### Checklist

---

In this chapter you've learned:

- How to use DATA statements for information handling and their limitations.
  - How to set up arrays using the DIM statement.
  - How to manipulate the information held in arrays.
  - The extra power of multi-dimensional arrays and how to use them.
  - How to use the string operators LEFT\$, RIGHT\$ and MID\$.
  - How to put these ideas into practical use in a simple demonstration adventure game.
- 

---

### Projects

---

- Write a simple filing system which could be used by enthusiasts in their hobby (record and stamp collections, or to catalogue books and magazine articles). These are the steps you might take:
    - 1** Set up multi-dimensional arrays to hold the information.
    - 2** Use the foolproof INPUT routine from chapter 3 to get the information and then put it into the arrays, perhaps using some kind of loop to update pointers.
    - 3** After reading chapter 12 you'll be able to incorporate routines to save the information to tape or disk so write the program in structured, modular fashion. This will make it easy to add new routines.
    - 4** Add a search facility using the string operators to find a given item.
- 

STAMP FILE  
PROGRAM OPTIONS

- 1 UPDATE FILE
- 2 SEARCH FILE
- 3 LOAD FILE
- 4 SAVE FILE

YOUR CHOICE?



---

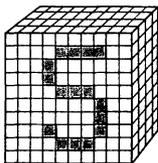
□ Extend The Shortest Adventure in the World into a more enjoyable game. Here are some ideas:

---

- 1** Add a list of verbs like LOOK, TAKE, DROP etc. These can be held in an array V\$( ) and add a routine to search the array for a match.
- 2** Add a few objects (nouns) that the player can collect or use along the way. You might have treasures, food or a tool to overcome some obstacle. These will be held in array N\$( ).
- 3** Perhaps you could put some food in the kitchen and a vicious dog at the front door. The player would have to keep the food and feed the dog to escape. (Allow for the player ruining his chances by letting him eat the food.) This could be handled by a dedicated subroutine.
- 4** Last but not least, open things up by adding extra rooms. You could create a 4 × 4 map then add the new room descriptions and alter the direction values to +1, -1, +4 and -4.

If you have a friend with a 64, swap your different versions as puzzles to be solved. You can share the problems and it encourages you to be inventive.

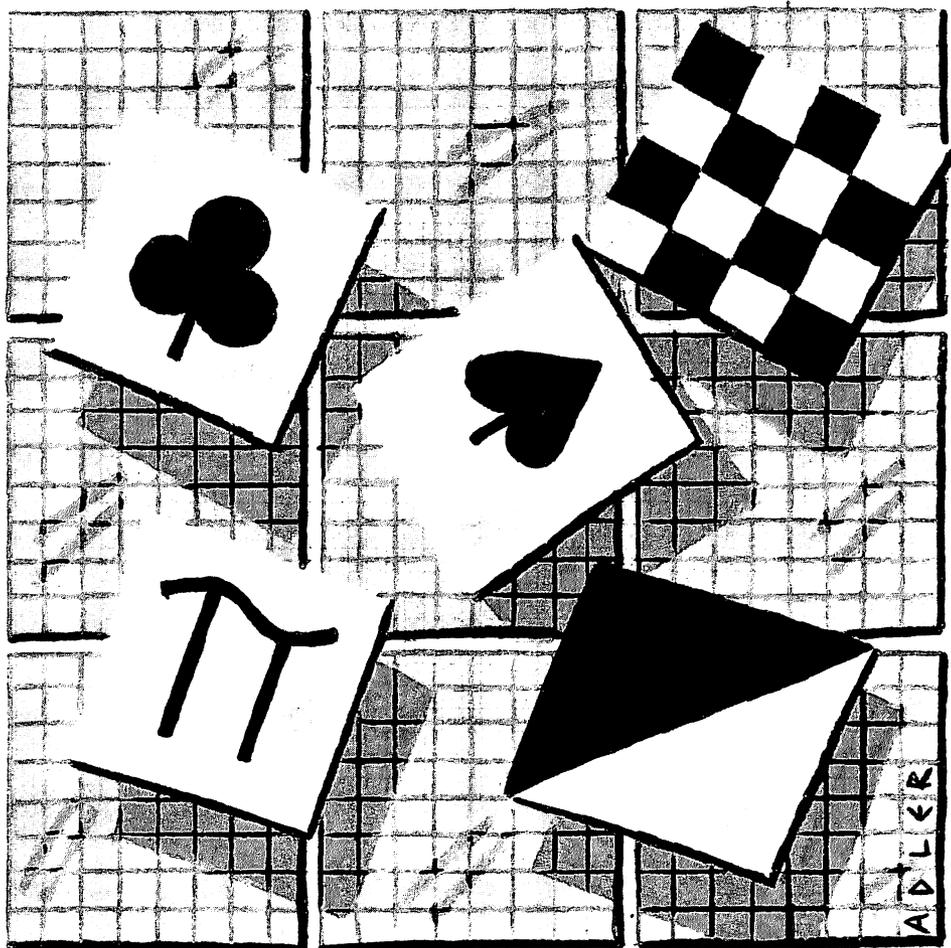
---



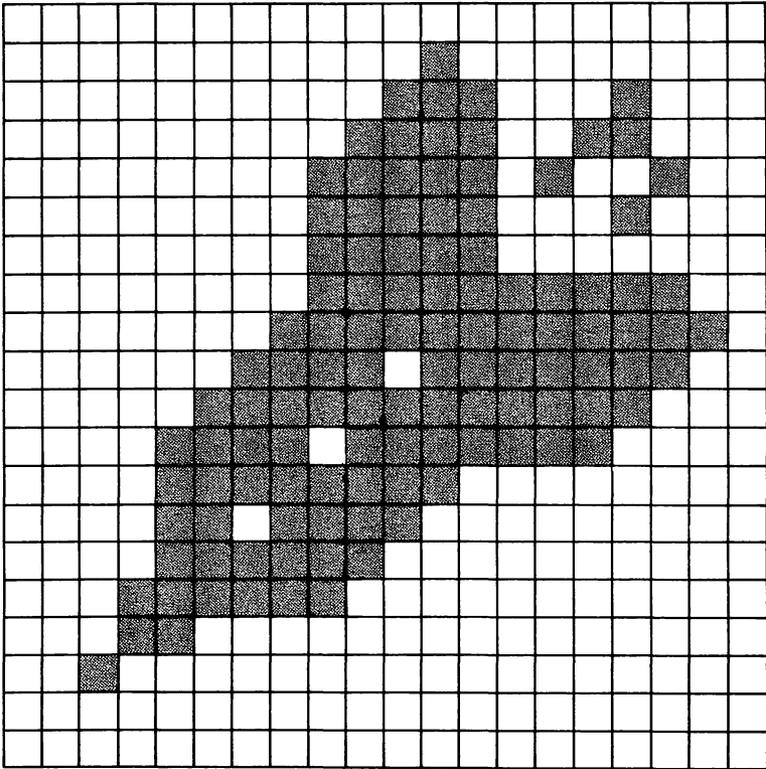
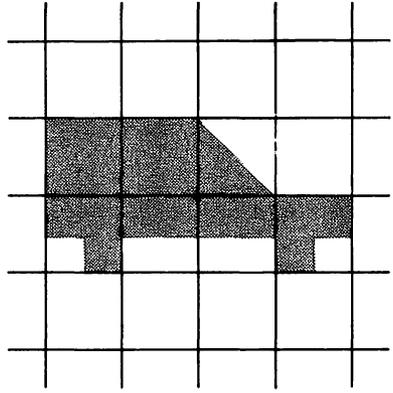
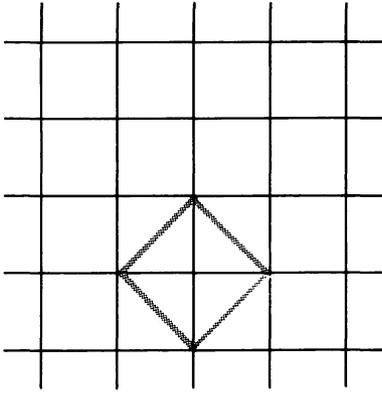
---

# Introduction to graphics

---







## Basic graphics

Let's take a quick tour of the 64's graphics 'factory', the VIC-II chip.

When you switch on your 64 you are in what is known as normal text mode. This means that all of the display is held in a part of RAM called screen memory which is 1,000 locations starting at 1024. The 64 divides this up into 25 lines of 40 giving the text screen resolution of 40 × 25. When typing in direct mode, or using the PRINT command, the VIC chip manages the display automatically. It will PRINT each character, one after the other, until it reaches the end of the line when it 'wraps round' to the left side of the next line. To put characters somewhere else on the screen you can use a number of commands such as TAB(), SPC(), or use the cursor control codes.

In normal text mode you can use the shift and Commodore keys to print capital letters and graphics symbols. By pressing both these keys at once you switch to what is called typewriter mode where all letters are lower case unless you 'shift' them.

In either mode you can display characters in any of 16 colours on a background of one of 16 colours. However, you can use another mode called 'extended background colour' which allows you to use 16 foreground, and 16 background colours. The catch here is that you are restricted to 64 different characters. (You will soon realise, if you haven't done so already, that most graphics modes involve compromises and trade offs – more colours for fewer characters, or a higher resolution for fewer colours.)

Although the graphics characters

are obviously useful for creating forms and charts, they are also an easy way of creating characters for games. Try this for a simple alien fighter:

```
10 PRINT CHR$(147)
20 POKE 1524,98:POKE
   1525,87:POKE 1526,98
30 POKE 55796,2:POKE
   55797,2:POKE 55798,2
```

## The next step

However, as useful as the block graphics are, there will come a time when you want something more. The 64 allows you to get it in the form of user-defined characters. On some machines you are limited to designing a handful of your own graphics but on the 64 you can change the whole character set if you wish. This gives you the opportunity to use another colour mode as well – multicolour mode. There's nothing to stop you using multicolour mode anyway but try this and you'll see the problem: enter directly

```
POKE 53270, PEEK (53270)
OR 16
```

As you can see, things look a little confused. To get back to normal just press RUN/STOP and RESTORE.

Multicolour mode is genuinely useful with user-defined characters because you can design the graphics to take advantage of it. In multicolour, the horizontal resolution is halved (four dots instead of eight) but you can have twice as many colours.

Yet another graphics mode on the 64 allows you to display extremely detailed pictures. This is high resolution mode and gives 320 dots

across the screen and 200 dots down. Combining high resolution and multicolour has the expected effect – dot resolution is halved horizontally, but the colour resolution is doubled.

### A final solution

The final section in the 64 graphics factory is the sprite maker. Sprites are perhaps the most exciting feature, allowing you to create detailed and colourful graphics and move them about with (comparative) ease. Although sprites may appear difficult without Basic commands to help, you only need to be methodical and patient to get the most from them.

All of these graphics and colour modes are under the control of the VIC chip. In the following sections we'll see how you in turn can control VIC and add these tools to your programming toolkit. But first there are two things we need to consider. You already know how to manipulate bytes of information using PEEK and POKE. Now you need to be able to manipulate bits, the eight parts that make up a byte. Secondly, you need to know how the VIC chip 'sees' the memory locations of the 64 and how you can control this.

### Essential tools

You are probably familiar with the logical operators AND and OR and can probably make sense of the following line:

```
IF A = 10 AND B = 1 OR
B = 2 THEN GOTO 500
```

But what about this one:

### POKE 1, PEEK(1) AND 251

To understand how Boolean – or logical – operators work on numbers we have to look at them at the binary level. A single byte number can be in the range 0 to 255. To see why this is so look at this:

|         |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|
| Bits    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Content | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In binary each bit can be either a 0 or a 1. In this example all the bits are off (set to 0) so the value of the byte is 0. If all the bits are on, the byte has the value 255:

|         |     |    |    |    |   |   |   |   |
|---------|-----|----|----|----|---|---|---|---|
| Bits    | 7   | 6  | 5  | 4  | 3 | 2 | 1 | 0 |
| Content | 1   | 1  | 1  | 1  | 1 | 1 | 1 | 1 |
| Value   | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

To get the value 129 we must turn on the first and last bits:

|         |     |   |   |   |   |   |   |   |
|---------|-----|---|---|---|---|---|---|---|
| Bits    | 7   | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Content | 1   | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Value   | 128 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

With this information can you work out the value of the following bytes?

|         |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|
| Bits    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Content | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Bits    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Content | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

The first totals 66 (64 + 2) and the second totals 56 (32 + 16 + 8).

Obviously, then, to turn off all the bits of byte X, we could use POKE X,0. To turn on bits 0 and 1 we POKE X,3. However, this last command also has the effect of turning off bits 2, 3, 4, 5, 6 and 7. This is where AND and OR come in. Using them we can turn individual bits on or off without affecting the rest of the byte.

To do this we take the number or

byte we want to work with, then AND or OR it with another number.

are on or off using the following: IF PEEK(X) AND 2 = 2 then bit 1 is on. IF PEEK(X) AND 2 = 0 then bit 1 is off.

### How it works

**AND works like this: if the first bit is on AND the second bit is on, then the resulting bit will be on. Here are the possibilities:**

|       |       |       |       |
|-------|-------|-------|-------|
| 0     | 1     | 0     | 1     |
| AND 0 | AND 0 | AND 1 | AND 1 |
| = 0   | = 0   | = 0   | = 1   |

**OR works in the opposite way: if the first bit is on OR the second bit is on then the resulting bit will be on.**

|      |      |      |      |
|------|------|------|------|
| 0    | 1    | 0    | 1    |
| OR 0 | OR 0 | OR 1 | OR 1 |
| = 0  | = 1  | = 1  | = 1  |

OR is used to set bits without affecting the rest of the byte. For example, to set bit 1 – regardless of whether it is on or off – we use POKE X, PEEK(X) OR 2. This gets the value of X and whether bit 1 is on OR off, turns it on. To turn bit 1 off we use AND in the following way: POKE X, PEEK(X) AND (255-2). To see why this works let's assume X = 135:

|         |     |   |   |   |   |   |   |   |
|---------|-----|---|---|---|---|---|---|---|
| Bits    | 7   | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Content | 1   | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Value   | 128 | 0 | 0 | 0 | 0 | 4 | 2 | 1 |
|         | 1   | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| AND     | 1   | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| =       | 1   | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

As you can see we have successfully switched off bit 1.

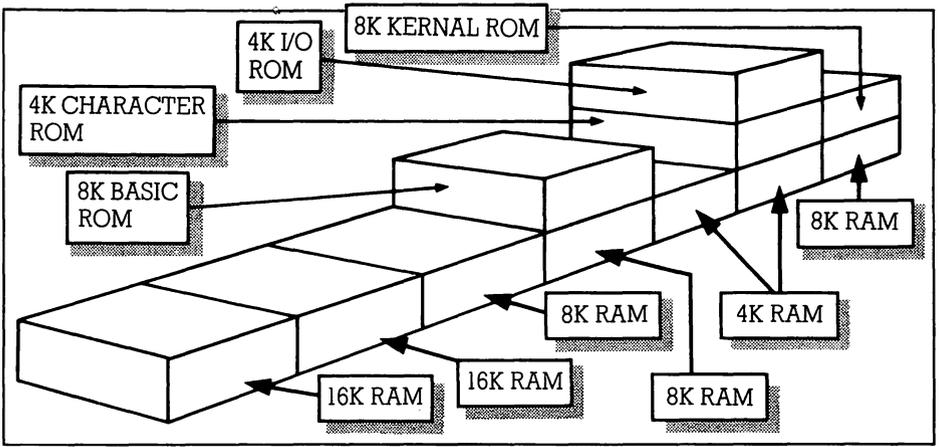
Finally, we can see whether any bits

### Memory maps: limitations

You may be wondering why all of this manipulating of bits is necessary. One reason is the way the designers of the 64 decided to use some of the memory locations. If you have a number of conditions which can be determined by a yes/no or on/off condition it is wasteful to use a whole byte for this. A single bit will do the job. So, many of the conditions in the 64 can be altered by changing a single bit. This means that one byte can control up to eight different functions. If you have 16 possible conditions, then half a byte (called a nybble) is sufficient. This is how the colour screen memory is mapped. If it took one byte for each screen location colour RAM would occupy nearly 1K. Using a nybble for each location means the 1,000 positions can be individually coloured using only 500 bytes of RAM.

In the VIC chip, many of the locations are multi-purpose controllers, using one or more bits as switches. For example, one byte can determine the on or off condition of eight sprites.

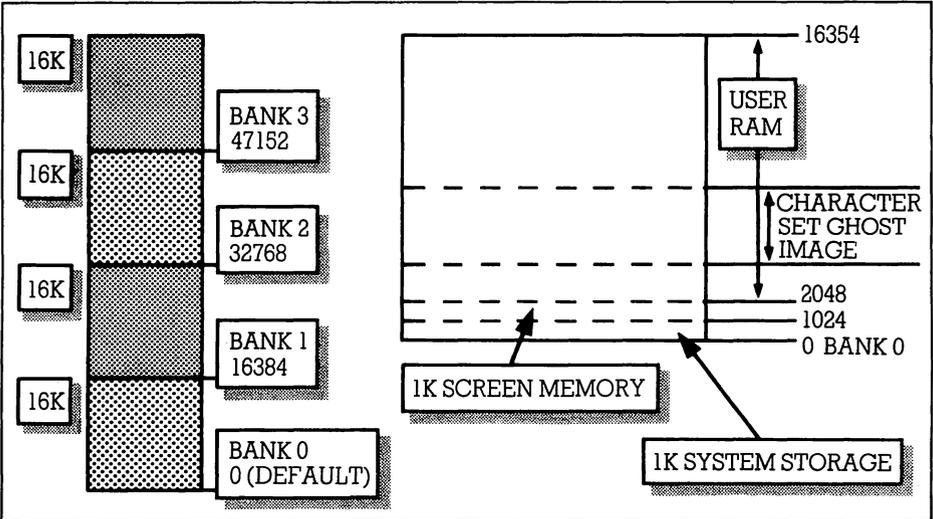
Here's a brief look at the memory map of the 64. There is 64K of RAM memory, the maximum that the 6510 main processor can 'see'. The VIC chip, however, can only see 16K at a time. Most of the parts of RAM are divided up into regular blocks of 4 or 8K: the character generator takes 4K, Basic takes 8K, and a high resolution screen would take 8K.



### Solutions

Because of the 16K limitation on the VIC chip, if you want to use your own graphics, or set up a high resolution screen, it's necessary to manipulate blocks of memory and you can do this in two ways. The first is known as paging and is the way in which the 64 appears to have more than one set of

RAM or ROM in the same place. The overlying blocks can be paged in or out, depending on what you are trying to do. The second is using a window and is the way the VIC chip looks at memory. It is possible to move the VIC's window up and down the memory map, allowing it to see



different areas of RAM for different functions.

These techniques are extremely powerful and allow immense flexibility. For example, the paging technique allows the 64 to have more than 80K of memory despite the 64K limitations on an eight-bit machine. They also mean that it is possible for machine code programmers to use up to 52K of memory with the Kernal

operating system routines present, or 60K if they are prepared to write all of the housekeeping routines themselves. This is why you can run other languages on the 64 without losing any of the free memory space.

Although this may seem complicated, it will become clear as we look at the different VIC functions in the following sections.

## Block graphics and colour

When you turn on the 64 you have more than 60 graphics characters directly available from the keyboard, or through POKEs to screen memory. It is easy to scorn these as useful graphics tools but when combined with colour and reverse images, and a little imagination, they are a quick and easy way to start programming graphics.

Within the limitations of the 64's Basic, there are still several ways of displaying characters. For example, a direct POKE to screen memory, using the PRINT command with quote mode, or by using the CHR\$ codes.

If you have a non-Commodore printer and are having trouble printing program listings containing cursor and screen control codes, remember you can change them to the relevant CHR\$ codes. For example, clear screen is CHR\$(147), CTRL 7 (blue) is CHR\$(31), cursor down is CHR\$(17) and so on. If you have a toolkit with a search and replace function, it is simple to make these changes before printing.

## A simple display

On many machines POKE is not only a more elegant way of using block graphics, it is faster. However, the 64's POKE command is much slower than PRINT, especially where you are using a composite character, i.e. one made up of several individual graphics like our three-part alien spaceship in the previous chapter. It is much quicker to

define the composite character as a string and print it on screen:

```
10 AL$ = CHR$(171) +  
    CHR$(113) + CHR$(179)  
20 PRINT CHR$(147)  
30 FOR I = 1 TO 5: PRINT  
    CHR$(17): NEXT  
40 PRINT TAB(19);AL$
```

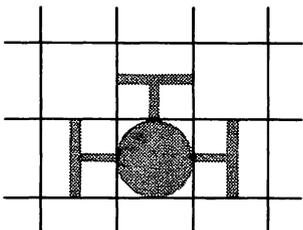
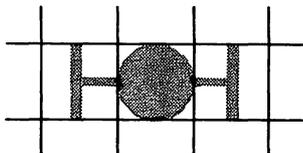
If we want our character to use more than one line, we use cursor controls to position the graphics. To give the ship a tailfin, add this:

```
15 AL$ = AL$ + CHR$(157) +  
    CHR$(157) + CHR$(145) +  
    CHR$(178)
```

Just as there's more than one way to print the character, there are several ways of adding colour. With print the most obvious is the use of the colour CTRL keys. However, POKE 646 has the same effect. If you wanted to change colours in logical order it

would be complicated using PRINT and the colour codes, but using a FOR .NEXT loop and POKES to 646 would do the job easily.

Having to POKE colour values into colour memory is largely responsible for the slowness of POKEing graphics. However, there are occasions when it can be useful to use this technique. By POKEing your display first, you can set up the screen while it is invisible, then bring it into view by either changing the screen colour, or by changing the colour values for a section of the screen.



FOR THIS USE:

```
15 AL$ = AL$ + CHR$(157) + CHR$(157) +  
    CHR$(145) + CHR$(178)
```

## AS A RULE

There is one other technique that you should always use when POKEing to the screen. Set up the key locations as variables at the start of your program. Not only is it easier to remember POKE SC + 40 than POKE 1064, but if

you later change the program so that screen memory is in a different location, you don't have to change every single screen and colour POKE. Simply change the value of your variables.

## TRY THIS

These two short examples illustrate these points.

```

10 SC = 1024: CO = 54272:
   PRINT CHR$(147)
20 FOR I = 0 TO 399
30 POKE SC+I, 160
40 POKE SC+I+CO,6
50 NEXT
60 PRINT "[HOME, 12 CURSOR
   DOWN] PRESS A KEY"
70 GET A$: IF A$ = ""
   THEN 70

```

```

80 POKE 53280, 14: POKE
   53281,14
90 END

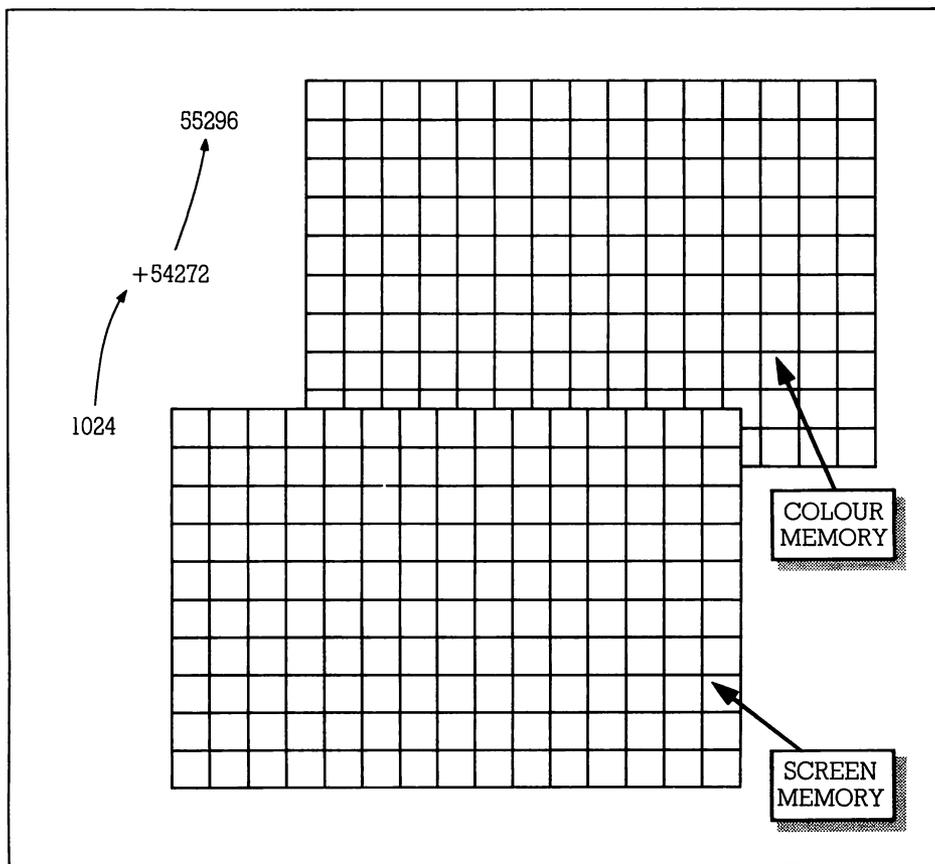
```

Press RUN/STOP and RESTORE and change line 70:

```

70 MD = SC + 199
75 FOR I = 0 TO 199: POKE
   MD + CO + I, 14: POKE
   MD + CO - I, 14: NEXT

```



## User-defined characters

Sooner or later you'll exhaust the possibilities of block graphics and you'll want to be able to display your own characters. The 64 lets you do that. These are the steps to take:

### The theory

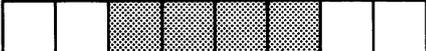
- Tell the VIC chip where to find the new character information.**
- Protect the new characters from being overwritten by Basic.**
- Decide which of the normal character set you want to keep and copy them into RAM.**
- Design your new characters and add them to the character set.**

Depending on your programming needs there may be additional steps such as moving screen memory. The reason for this is the 16K limitation on the VIC chip. First, look at how the VIC chip sees the character information. The character generator is held in RAM starting at 53248 in eight blocks of 512 bytes.

53248 Upper case characters  
 53760 Graphics  
 54272 Reversed upper case characters  
 54784 Reversed graphics  
 55296 Lower case characters  
 55808 Upper case and graphics  
 56320 Reversed lower case characters  
 56832 Reversed upper case and graphics

Since each character is made up of eight rows of eight dots, eight bytes are needed for every letter and graphic. Each of the blocks above contains 64 characters.

A safe place to put your characters is at 12288 (remember that because the VIC chip is normally looking at the

| CHARACTER   | ADDRESS | CONTENT |
|---|---------|---------|
|  | 53248   | 60      |
|  | 53249   | 102     |
|  | 53250   | 110     |
|  | 53251   | 110     |
|  | 53252   | 96      |
|  | 53253   | 98      |
|  | 53254   | 60      |
|  | 53255   | 0       |

first 16K of memory the character set must be in this area. How does the VIC chip see it if it's normally at 53248?

Because of a clever trick of programming, a ghost image appears in the first 16K.)

Before we copy some characters and create a few new ones, there are a couple of other points to note. To read the character generator we need to flip out the I/O ROM which appears above it. We also need to stop the computer being interrupted during the copying. (Interrupts occur every 60th of a second when the 64 does its housekeeping and include checking the keyboard. If the I/O block is not present the micro will crash.)

Enter NEW and type in the following program:

```
10 PRINT CHR$(147)
20 POKE 53272, (PEEK(53272)
  AND 240) OR 12
30 POKE 52,48: POKE
  56,48: CLR
40 POKE 56334, PEEK(56334)
  AND 254
50 POKE 1, PEEK(1) AND 251
60 FOR I = 0 TO 511
70 POKE 12288 + I,
  PEEK(53248 + I)
80 NEXT
90 POKE 1, PEEK(1) OR 4
100 POKE 56334, PEEK(56334)
  OR 1
```

### How it works

This copies the first 64 characters into RAM. Line 20 looks complicated. What it does is tell the VIC chip where in the current 16K of RAM the character memory is stored.

Line 30 lowers the top of Basic

memory to make sure our new character set is not written over by Basic variables. Line 40 switches off the interrupts and line 50 flips the I/O ROM out and the character ROM in.

Lines 60–80 do the copying, taking

The form of the command is `POKE 53272, (PEEK(53272) AND 240) OR A` where `A` has the following values and effects:

| Value of A | Character location |
|------------|--------------------|
| 0          | 0                  |
| 2          | 2048               |
| 4          | 4096               |
| 6          | 6144               |
| 8          | 8192               |
| 10         | 10240              |
| 12         | 12288              |
| 14         | 14336              |

In our example the VIC chip is looking at the first 16K of memory so giving `A` the value 12 tells the VIC chip to look at 12288 for character memory. If the VIC was looking at the second 16K bank of RAM (starting at 16384) then giving `A` the value 12 would require character memory to start at  $16384 + 12288 = 28672$ .

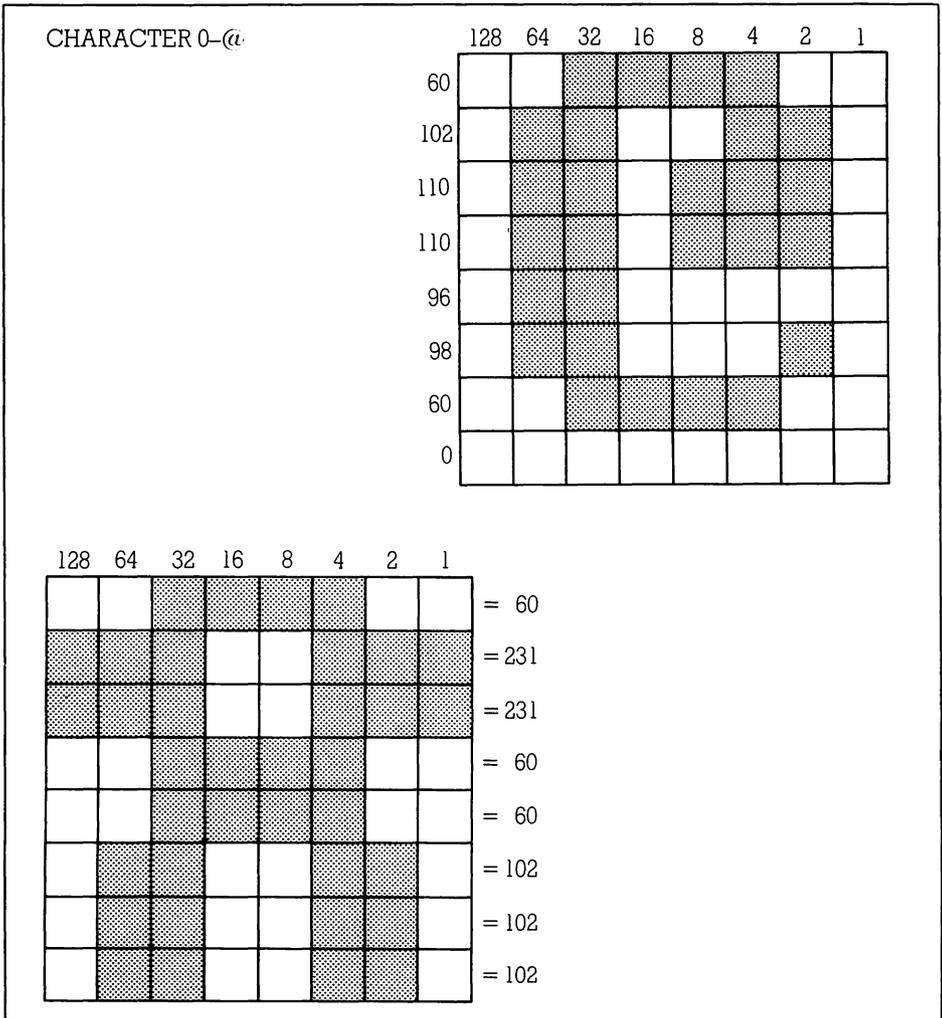
the character values for the first 64 characters and putting them into the new location. The last lines flip the I/O block back in and turn on the interrupts.

To see if this has worked try typing something. If the alphabet appears normal, try pressing the shift and Commodore keys to get lower case. If everything turns to garbage the

program has worked. Press shift and Commodore again.

### The next step

Now the useful part – creating our own characters. First we need to design them on a piece of 8×8 squared paper.



Let's change the @ character to a space invader.

We need eight numbers to define the new character and these are stored as DATA statements so add line 1000 to the program:

```
1000 DATA 60, 231, 231,  
        60, 60, 102, 102, 102
```

(To make sure the program doesn't run into this line also add 999 END.)

Now add the following:

```
110 PRINT "@", "@", "@"  
120 FOR I = 0 TO 7:  
    READ A  
130 POKE 12288 + I, A  
140 NEXT
```

When you RUN this, you should see the three @ characters change to space invaders on screen. Now whenever you press the @ key the space invader will appear.

### A limitation

There is one severe limitation to our example. By placing the character set in the first 16K and lowering memory to protect it, we are left with only about 10K of Basic space. For many programs this might not be a problem but there is little point in buying a 64K computer and ending up with only 10K after a simple program like this.

The answer is to move everything around. For example, instead of lowering the top of Basic, we could raise the bottom so that Basic starts about 16K. This would give us about 20K which is better. The problem here is that any program in the normal Basic space would be lost. The answer is to have a loader program. This would set

up the new characters and raise the bottom of memory before loading the main program.

### The solution

Better still is to move the VIC window higher into memory. The following program does this.

```
5 POKE 51, 255: POKE 52,  
  127  
10 POKE 55, 255: POKE 56,  
  127  
20 OC = 53248: NC = 32768  
30 POKE 56578, PEEK(56578)  
  OR 3  
40 POKE 56576,  
  (PEEK(56576) AND 252)  
  OR 1  
50 POKE 53272,  
  (PEEK(53272) AND 15)  
  OR 48  
60 POKE 53272,  
  (PEEK(53272) AND 240)  
  OR 0  
70 POKE 648, 140  
80 POKE 56334, PEEK(56334)  
  AND 254  
90 POKE 1, PEEK(1) AND  
  251  
100 FOR I = 0 TO 511  
110 POKE NC + I, PEEK  
  (OC + I)  
120 NEXT  
130 POKE 1, PEEK(1) OR 4  
140 POKE 56334, PEEK(56334)  
  OR 1
```

The new memory map looks like this to the VIC chip:

```
32768 - 34815 Character data  
34816 - 35839 Sprite data (16 sprites)  
35840 - 36839 Screen memory  
36856 - 36863 Sprite pointers
```

Because the VIC chip expects to look at 2K of character data at a time, the character set has the 2K from 32768 to 34815 but we only copied down the first 64 characters. This means that there is space in the character area for

192 user-defined characters. The alternatives would be to copy more characters from ROM or to use the free space for more sprite data. The latter course would give space for a total of 40 sets of sprite data.

## How it works

### Moving the VIC window:

In the 64K of RAM in the 64 the VIC chip can look at any of four banks of 16K. The register that controls this is 56578. There are two steps to switching the banks: first set the relevant bits (bits 0 and 1):

```
POKE 56578, PEEK(56578)
OR 3
```

The second step is to actually make the switch:

```
POKE 56578, (PEEK(56578)
AND 252) OR A
```

| Value of A | Bank | Starting location |
|------------|------|-------------------|
| 0          | 3    | 49152             |
| 1          | 2    | 32768             |
| 2          | 4    | 16384             |
| 3          | 0    | 0 (default)       |

Remember that if you move the VIC window you must also move the screen memory. This

is controlled by the upper nybble of location 53272 and is changed with this command:

```
POKE 53272, (PEEK(53272)
AND 15) OR A
```

'A' can be any multiple of 16 and it raises screen memory by 1024 bytes for each multiple above the bottom of the VIC window. For example, if A = 0 then screen memory starts at the bottom of the VIC window. If A = 160 then screen memory starts at 10240 bytes above the bottom of the VIC window.

Finally, in addition to this, the Kernal operating system also has to know where to find screen memory. The pointer for this is location 648 and it should contain the page where screen memory starts. A page is 256 bytes of memory so page = address/256. In our example,  $35840/256 = 140$ .

---

## Checklist

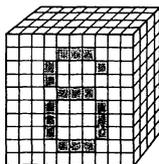
In this chapter you've learned:

- A little about the different display and colour modes.
- How to manipulate individual bits and bytes using AND and OR.
- How the 64 memory map works.
- How to create and display your own graphics characters.
- An alternative way to organise memory to give more space for sprites and user-defined characters.

---

## Projects

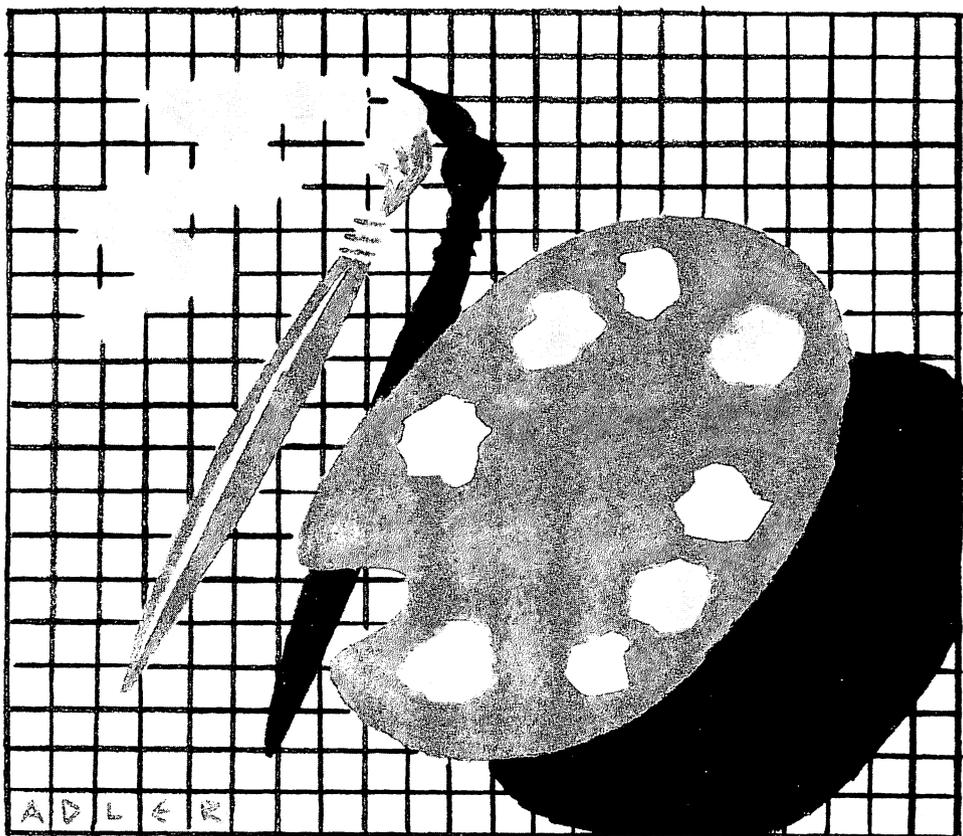
- Create a character set for your favourite game (eg, Pacman, space invaders or Scramble type graphics).
- Make sure you can make it work with the character set at 12288 and at 32768.
- Write a graphics editor program. These are the steps you might take:
  - 1** Set up the memory map for user-defined characters.
  - 2** Display a working grid of  $8 \times 8$  full-sized characters.
  - 3** Using cursor keys allows the user to move about the grid turning blocks on or off.
  - 4** Create the character: if a block in row 1 is on, then set that pixel in byte 1 on, etc.
  - 5** Save the new graphics set to tape or display the data values for the new graphics.



---

# Advanced colour

---



---

Back in chapter 5 the various colour modes of the 64 were mentioned and it was said that user-defined characters were needed to make the most of multicolour mode. This powerful mode

is also very useful with high resolution graphics, so now let's take a look at it in detail, along with the other colour facilities available.

### Extended background colour mode

The first option after the normal colour display is called extended background colour mode. We have already seen that, under normal circumstances, screen memory has 1,000 locations for character information and a matching number for colour. This gives us the choice of any one of 16 colours for the background and any or all of 16 colours for the foreground. However, it means that all characters on screen will have a common background unless printed in reverse. But as the name 'reverse video' implies, all that happens here is that background becomes foreground and vice versa.

Extended background colour gives you a great deal more control over the

colour of individual characters. In effect, you can now have any foreground colour combined with any background colour for every character position on screen. In other words, you could have a green character on a white background next to a blue character on a yellow background.

The mode is turned on by a POKE to location 53265. We need to set bit 6 to a 1 like this:

```
POKE 53265, PEEK(53265) OR 64
```

(Using your new knowledge of Boolean operators, can you work out how to turn off the bit and get back to normal? The answer is given below.)

### TRY THIS

Enter the line above in direct mode and experiment a little. Try printing a few characters, then go into reverse mode with CTRL 9 and print a few more. Unexpectedly you don't get reversed characters. You will see instead that the same characters appear but on a different background colour.

As always there is a trade-off involved. In this case we gain the extra colour flexibility at the cost of the

number of characters. In extended background colour mode you can only display the first 64 characters (the upper case alphabet, numbers and a few graphics) or your first 64 user-defined characters.

This compromise is forced because the extra colour information is held in the high two bits of the character code. To see how it works print first an A, then a shifted A, then the same characters in reverse.

The extra colour information is also stored in locations 53281 to 53284. Try POKEing colour codes in these and see what happens.

To turn off the mode use:

**POKE 53265, PEEK(53265) AND (255 - 64)**

or

**POKE 53265, PEEK(53265) AND 191**

Because of the limitation in the number of characters available extended background colour is not used often but you might use it to create colourful and eye-catching title screens.

## Multicolour mode

We saw in the introduction to graphics that multicolour mode is all but useless with standard graphics because of the change in dot resolution. However, there are two areas in which it becomes a very powerful tool: user-defined characters and high resolution graphics.

Multicolour mode gives us even more flexibility than extended background colour – four colours in every character position. The drawback, as we have seen, is that horizontal resolution is halved.

The four colours are dictated, first by the bit patterns in the character bytes, and second, by the values of the colour registers. Look at the box to see how this works.

## How it works

By halving the horizontal dot resolution we can have four pairs of bits giving the four colour patterns like this:

00 = background (screen) colour

01 = background colour 1

10 = background colour 2

11 = foreground (character) colour

The registers are as follows:

Screen colour at 53281

Background colour 1 at 53282

Background colour 2 at 53283

Foreground colour in colour RAM

Look at this example. Let's set the screen colour to black, foreground colour to yellow, background colour 1 to dark green and background colour 2 to light green:

POKE 53281,0: POKE  
53282,5: POKE 53283,13

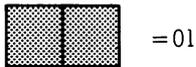
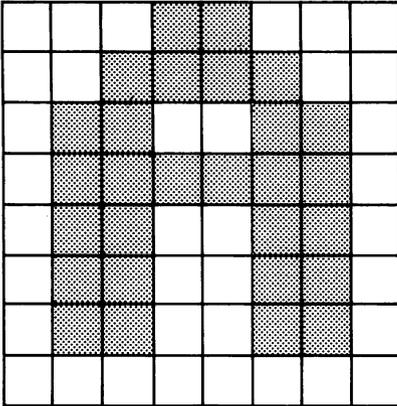
Foreground colour obviously depends where on screen the character will appear but suppose it's at 1024. The colour location is 55296 so for yellow:

POKE 55296, 7

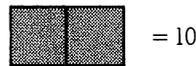
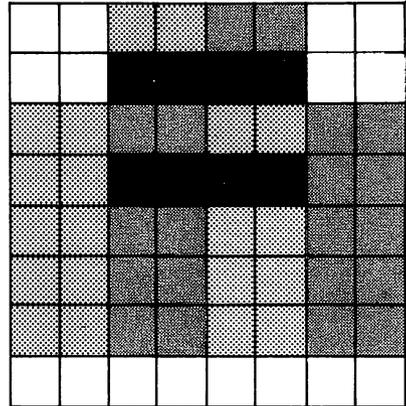


If we put the letter A into 1024 now it will appear like this:

Standard



Multicolour



## TRY THIS

To see what's happening a little better let's design some characters to take advantage of multicolour mode. First, load the program from chapter 5 which sets up the character set for user-defined characters.

Now we'll redesign the first four characters (@, A, B, C) to make a 2 x 2 composite character.

The data statements for our new design work out like this:

```
1000 DATA 15, 63, 53, 53,
      53, 53, 63, 15
1010 DATA 252, 255, 215,
      215, 215, 215, 255,
      252
```

```
1020 DATA 12, 12, 0, 0, 0,
      0, 0, 0
1030 DATA 12, 12, 0, 0, 0,
      0, 0, 0
```

Now we can add a line to put the new design into memory:

```
150 FOR I = 0 TO 31: READ
  A: POKE 12288 + I, A:
  NEXT
```

Now we need to turn on multicolour mode:

```
160 POKE 53270, PEEK(53270)
  OR 16
```

And now set the colour registers:

```
170 POKE 53281, 6: REM
  SCREEN = BLUE
```

```

180 POKE 53282, 1: REM
    COLOUR 1 = WHITE
190 POKE 53283, 7: REM
    COLOUR 2 = YELLOW

```

The next section is to display our new characters:

```

200 POKE 1523, 0: POKE
    1524, 1
210 POKE 1563, 2: POKE
    1564, 3
220 POKE 55795, 8: POKE
    55796, 8
230 POKE 55835, 8: POKE
    55836, 8

```

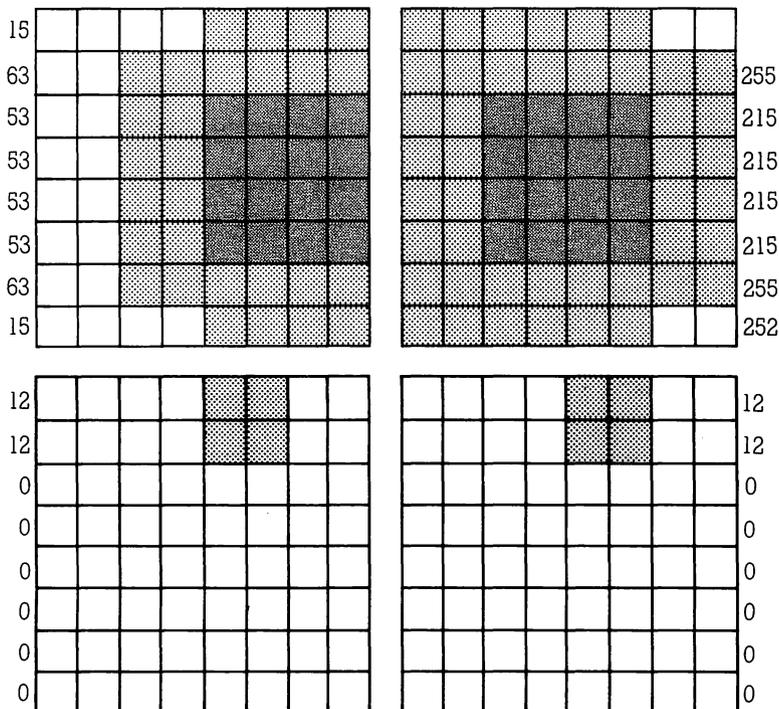
The last step demonstrates one of the most powerful features of multicolour mode. By changing the value in one of the colour registers to a new colour, every dot displayed in that colour will change instantly. Try it:

```

240 FOR I = 1 TO 500: NEXT
250 POKE 53282, 4

```

Multicolour characters are a very powerful tool – most commercial games programs make extensive use of this mode and you can see how effective it can be. Keep experimenting.



## Multicolour bit map mode

As with multicolour characters we can display four colours in the  $8 \times 8$  cells but horizontal resolution is halved from 320 to 160. The advantage here is the extra colour resolution.

In normal high resolution mode if you draw a line in one colour, then draw a second in another colour which enters an  $8 \times 8$  character cell already occupied by the first, the dots in that area of the first line will change colour. This can ruin a carefully created display. Using multicolour each pair of dots can be in any of the 16 possible colours without affecting any other pair.

## The theory

The colour information comes from these sources:

Bit pair 00 = screen colour (53281)

Bit pair 01 = high nybble of screen memory

Bit pair 10 = low nybble of screen memory

Bit pair 11 = colour memory (55296 – 56295)

Obviously the background colour will appear where no bits in the bit pair are on. To get foreground colour in any character cell POKE the colour code into  $55296 + \text{cell number}$  then turn on both bits.

To get background colour 1, turn on the second bit of each pair. Referring to the high resolution program in

chapter 7, set X like this:

```
500 FOR X = 51 TO 100
STEP 2
```

This will draw a line from 50 to 100 turning on bits 51, 53, 55 etc. Similarly to draw in background colour 2, use:

```
500 FOR X = 50 TO 99
STEP 2
```

This turns on bits 50, 52, 54 etc.

There is an extra POKE to get into multicolour bit mapped mode. In addition to POKE 53270, PEEK(53270) OR 16, we need POKE 53265, PEEK(53265) OR 32.

To turn off multicolour use:

```
POKE 53265, PEEK(53265) AND
223: POKE 53270,
PEEK(53270) AND 239.
```

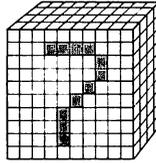
## Checklist

In this chapter you've learned:

- How to use extended background colour mode.
- How to use multicolour mode.

## Project

- Create a full-blown title page for an imaginary game using user-defined characters and the different colour modes. (Save it – you might write the game one day.)



---

# High resolution graphics

---



---

High resolution, or bit map, graphics is perhaps the area of micro programming which provokes most interest among would-be programmers and yet it is one of the least used facilities on most home computers. The reason for this is that a colourful hi-res display can be extremely impressive but actually achieving it can be cumbersome and very slow.

A moment's examination shows the problems. As the name bit mapping indicates, every single dot on the screen is controlled by one bit of RAM. On the 64 with a  $40 \times 25$  character screen, this means there are  $40 \times 8$  dots across and  $25 \times 8$  dots down –  $320 \times 200$  or 64,000 dots, all of which must be controlled by your program.

In addition, the 64 has no commands in Basic to let you use this capability and in any case, Basic is generally much too slow. High resolution displays are really only practical with machine code. The alternative is to buy one of the many Basic extension programs which add new commands to Basic allowing you to create hi-res screens relatively quickly and with considerably greater ease.

However, all of this is not to say that it can't be done from the ordinary machine.

The main question, as with user-defined characters, is where to locate the bit map in memory. The answer is the same. In fact the simplest way to implement high resolution is to think of the screen in terms of an 8K block of user-defined graphics. To put it another way, in user-defined graphics the screen changes but the characters stay the same. In high resolution the screen stays the same but the characters change to produce the

display.

The simplest place to put the hi-res screen for experimentation is at 8192. The memory location controlling the position is the same as controls the character set – 53272 – and the command is:

```
POKE 53272, PEEK(53272)
OR 8
```

This puts the screen at the eighth 1K block above the bottom of the VIC window, or 8192.

Next we tell the 64 that we want bit mapped graphics with this command:

```
POKE 53265, PEEK(53265)
OR 32
```

Enter these two lines as a program then RUN it and see what happens. To get back to normal press RUN/STOP and RESTORE. The reason for the garbage is that the memory locations we're using for the bit map contain random values. Before we do anything we have to clear all these locations by POKEing them with zeroes. The other requirement is to prevent Basic from overwriting our new screen.

With these routines our program now looks like this:

```
10 POKE 52, 32: POKE 56,
   32
20 PRINT CHR$(147): SC =
   8192: CO = 1024
30 POKE 53272, PEEK(53272)
   OR 8
40 POKE 53265, PEEK(53265)
   OR 32
50 FOR I = SC TO SC + 7999
60 POKE I, 0: NEXT
```

The next point to consider is the colour of our hi-res screen. The colour information is stored, not in the normal colour RAM, but in the original low

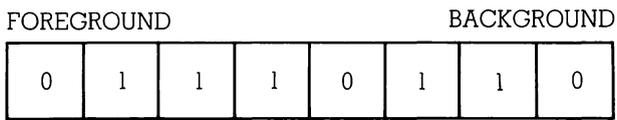
resolution screen starting at 1024.  
 Foreground and background colours are determined by the two nybbles in each byte. So to get a white foreground and black background we must POKE all 1000 locations with 16:

```
70 FOR I = C0 TO C0 + 999
80 POKE I, 16: NEXT
```

Now, at last, we're ready to start the actual drawing and this is where things get complicated.

Colour codes for bit mapped screen

| Colour      | Foreground | Background |
|-------------|------------|------------|
| Black       | 0          | 0          |
| White       | 16         | 1          |
| Red         | 32         | 2          |
| Cyan        | 48         | 3          |
| Purple      | 64         | 4          |
| Green       | 80         | 5          |
| Blue        | 96         | 6          |
| Yellow      | 112        | 7          |
| Orange      | 128        | 8          |
| Brown       | 144        | 9          |
| Light Red   | 160        | 10         |
| Dark Grey   | 176        | 11         |
| Medium Grey | 192        | 12         |
| Light Green | 208        | 13         |
| Light Blue  | 224        | 14         |
| Light Grey  | 240        | 15         |



$$112 + 6 = 118$$

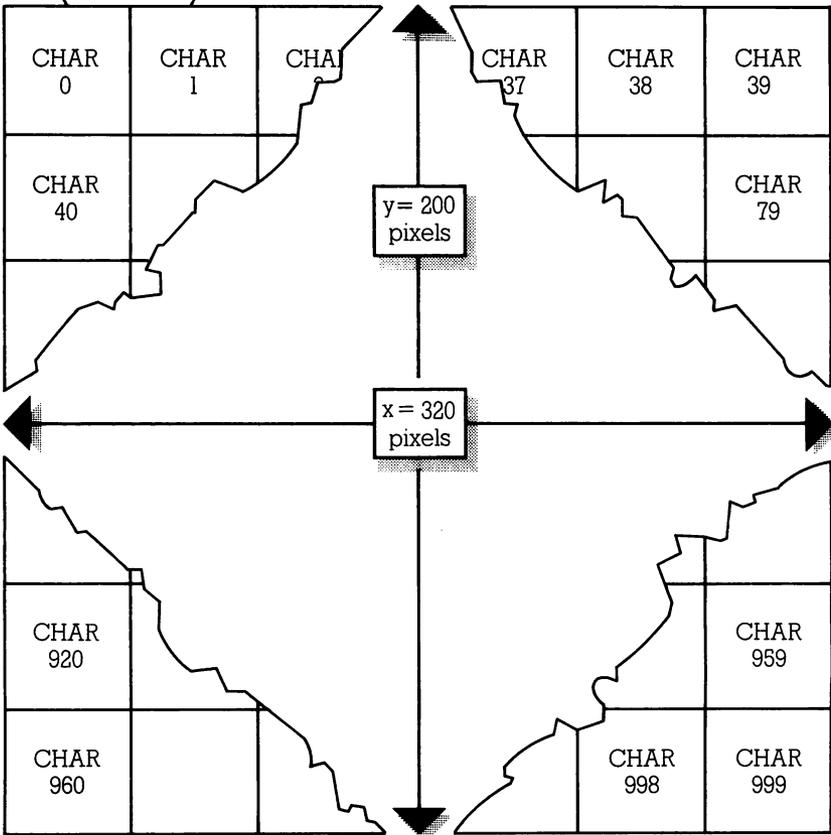
$$= 16 \times 7$$

YELLOW      BLUE

To determine the value to be POKEd into the colour location simply take the value of the background colour and add it to the value of the foreground colour. For example, a

black background and white foreground gives 16. A yellow foreground on a blue background gives 118.

|        |         |
|--------|---------|
| BYTE 0 | BYTE 8  |
| BYTE 1 | BYTE 9  |
| BYTE 2 | BYTE 10 |
| BYTE 3 | BYTE 11 |
| BYTE 4 | BYTE 12 |
| BYTE 5 | BYTE 13 |
| BYTE 6 | BYTE 14 |
| BYTE 7 | BYTE 15 |



If you look at the bit map diagram you'll see that the usual logic of screen memory breaks down here because of the way in which the screen bytes are stored. If we want to draw a line from top to bottom of the screen it is not sufficient to specify a starting point and then increase or decrease the co-ordinates regularly. So first we need a routine to calculate the position of each dot in memory in terms of an X and Y co-ordinate.

If we think of the screen again as being 40 characters across we can determine which character position the target dot is in by  $CH = INT(X/8)$ . Similarly the row will equal  $INT(Y/8)$ . The line in the character position equals  $Y AND 7$ . So we can calculate the byte as follows:

$$BY = SC + RO * 320 + CH * 8 + LN$$

The target bit =  $7 - (X AND 7)$ . To turn it on use this: POKE BY, PEEK (BY) OR 2 ↑ BI.

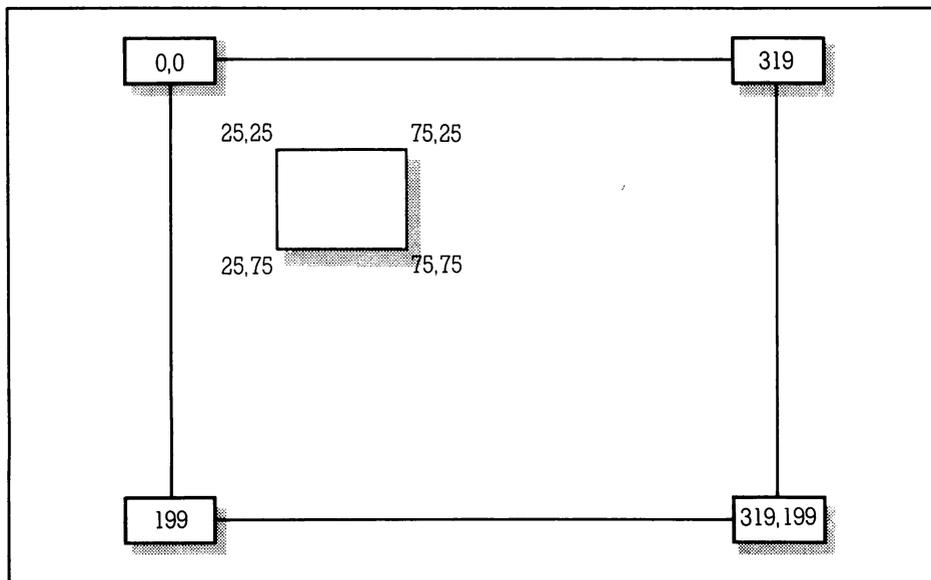
If all of this sounds like mathematical mumbo-jumbo, don't worry. The idea is to be able to use it so just add this subroutine to our program:

```

1000 CH = INT(X/8)
1010 RO = INT(Y/8)
1020 LN = Y AND 7
1030 BY = SC + RO * 320 +
      8 * CH + LN
1040 BI = 7 - (X AND 7)
1050 POKE BY, PEEK(BY) OR
      (2 ↑ BI)
1060 RETURN

```

With this routine incorporated we can now think of the screen in straightforward terms. The top left corner is 0,0; the right is 319,0; bottom left is 0,199 and bottom right is 319,199.



Let's add some lines that will actually draw something (and stop the program running into the subroutine):

```
200 Y = 25
210 FOR X = 25 TO 75
220 GOSUB 1000
230 NEXT
240 FOR Y = 25 TO 75
250 GOSUB 1000
260 NEXT
270 FOR X = 75 TO 25
    STEP -1
280 GOSUB 1000
290 NEXT
300 FOR Y = 75 TO 25
    STEP -1
310 GOSUB 1000
320 NEXT
999 END
```

Finally, let's add a routine to put everything back to normal before the program finishes. This routine will wait for the RETURN key to be pressed and then bring things to a tidy conclusion:

```
700 GET A$: IF A$ <>
    CHR$(13) THEN 700
710 PRINT CHR$(147)
720 POKE 53265, PEEK(53265)
    AND 223
730 POKE 53272, PEEK(53272)
    AND (255-8)
```

If you run this program now you can see a perfect demonstration of the speed of Basic hi-res plotting. Any complex display is going to take at least several minutes. We can gain a slight increase in speed by using predetermined values when turning on the pixel in this line:

```
POKE BY, PEEK(BY) OR
2 ↑ BI
```

The ↑ is the symbol for exponentiation and in this line gives us

2 to the power of BI. Exponentiation is one of the slowest functions on the 64 so by working the values out in advance we save a little time. Add the following subroutine to the program:

```
2000 DIM A(7)
2010 FOR I = 0 TO 7
2020 A(I) = 2 ↑ I
2030 NEXT
2040 RETURN
```

Now amend line 1050 as follows:

```
1050 POKE BY, PEEK(BY) OR
    A(BI)
```

And add:

```
5 GOSUB 2000
```

Even this is not going to speed things up dramatically and it's obvious that you won't be able to use these routines for hi-res animation. However, you could certainly create a hi-res background for a game and use sprites in the foreground.

One last thing remains: how to turn off a pixel. You may recall from the chapter on Boolean operators that we use OR to turn on a bit and AND (255 - BIT) to turn it off. Using the routines given above, the following line will turn off the target pixel:

```
POKE BY, PEEK(BY) AND (255
- A(BI))
```

If you enter and save this program, you'll find a routine in the chapter on interfacing which will allow you to use a joystick to create a high resolution sketchpad.

One of the trickiest high resolution routines is the circle, largely because of the more complex equations needed to plot all the points around the circumference.

There are a number of different

methods which largely involve compromises between speed, simplicity and the quality of the final image. The one that follows is not the fastest or the best, but it is the simplest. You can add it as a subroutine to the high resolution program.

```

400 GOSUB 5000
5000 XC = 160: YC = 100:
      R = 25: REM SET CIRCLE
      CENTRE AND RADIUS
5010 FOR X1 = -R TO R:
      Y1 = SQR(R*R-X1*X1)
5020 X = XC + X1: Y = YC
      + Y1

```

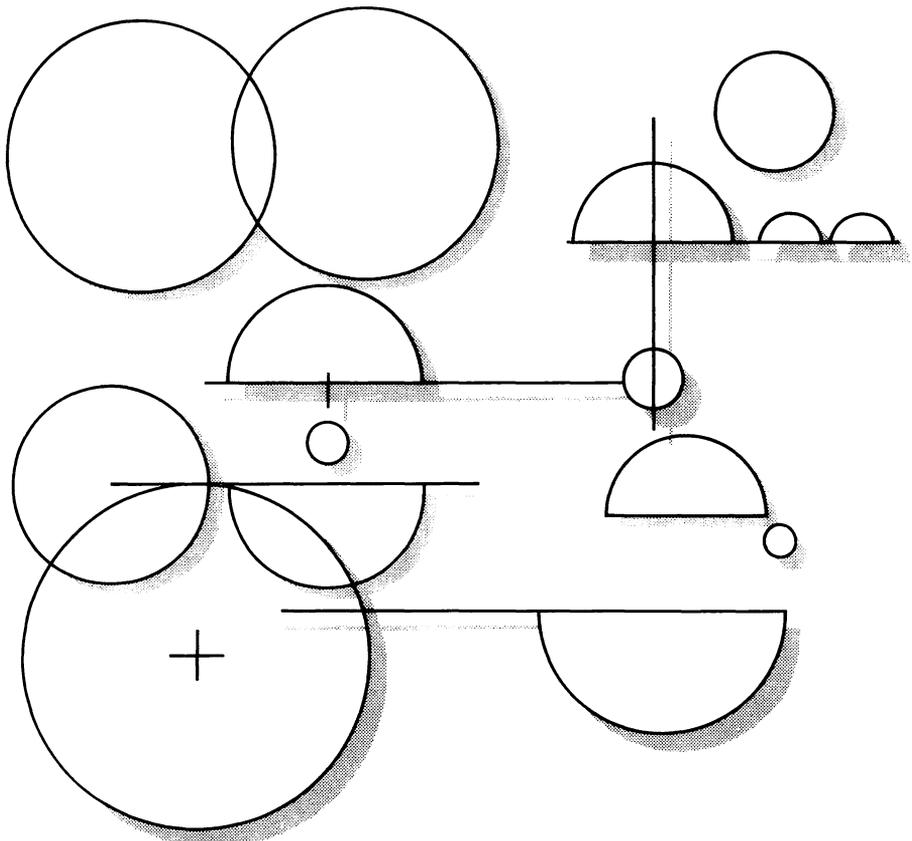
```

5030 GOSUB 1000: REM PLOT
      POINT
5040 Y = YC - Y1
5050 GOSUB 1000
5060 NEXT
5070 RETURN

```

Try changing line 5010 to STEP in larger or smaller increments and see the effects. Smaller steps produce a better-looking circle while larger steps are much faster.

For a better routine, try one of the many books dedicated to computer graphics.



---

## **Checklist**

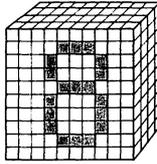
In this chapter you've learned:

- How to set up a high resolution graphics screen.
- How the 64 'sees' the screen in high resolution mode.
- How to use colour with bit-mapped graphics.
- How to turn a pixel on and off.

---

## **Projects**

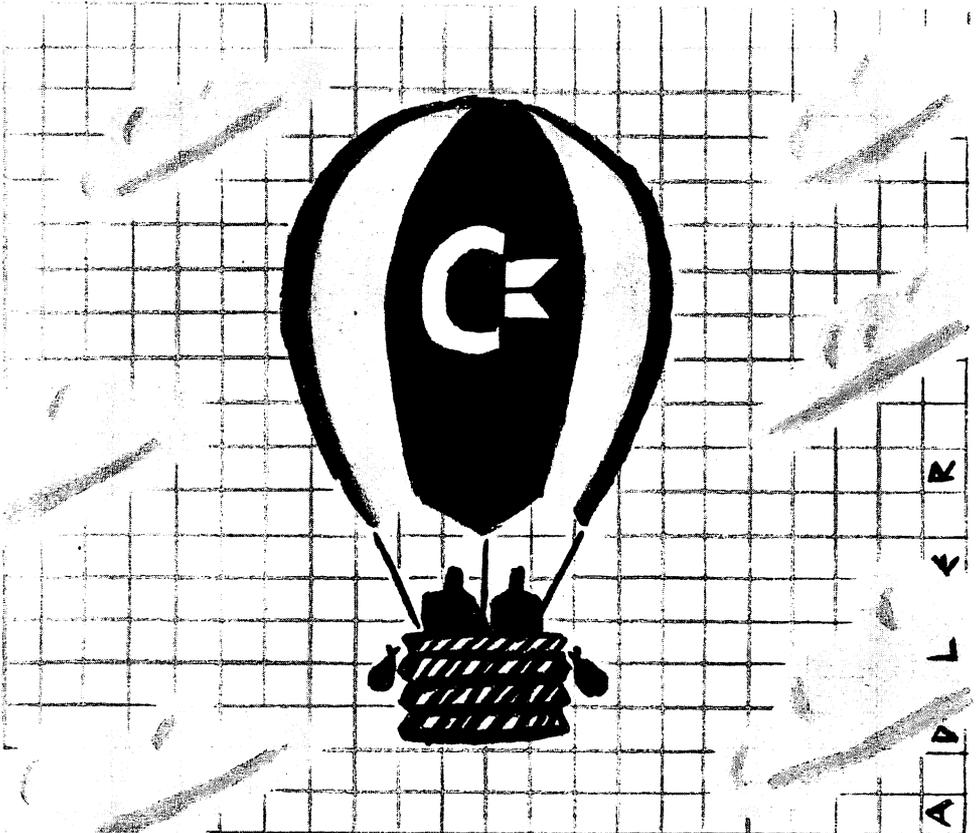
- Get a high resolution screen with the reconfigured memory map from chapter 5.
- Write a program to switch from a normal text screen to a high resolution screen and back again. Hint: the bit-map screen will be untouched if it's protected from Basic. The text screen will have to be stored somewhere (copied into an array?) and then reinstated.



---

# Introduction to sprites

---



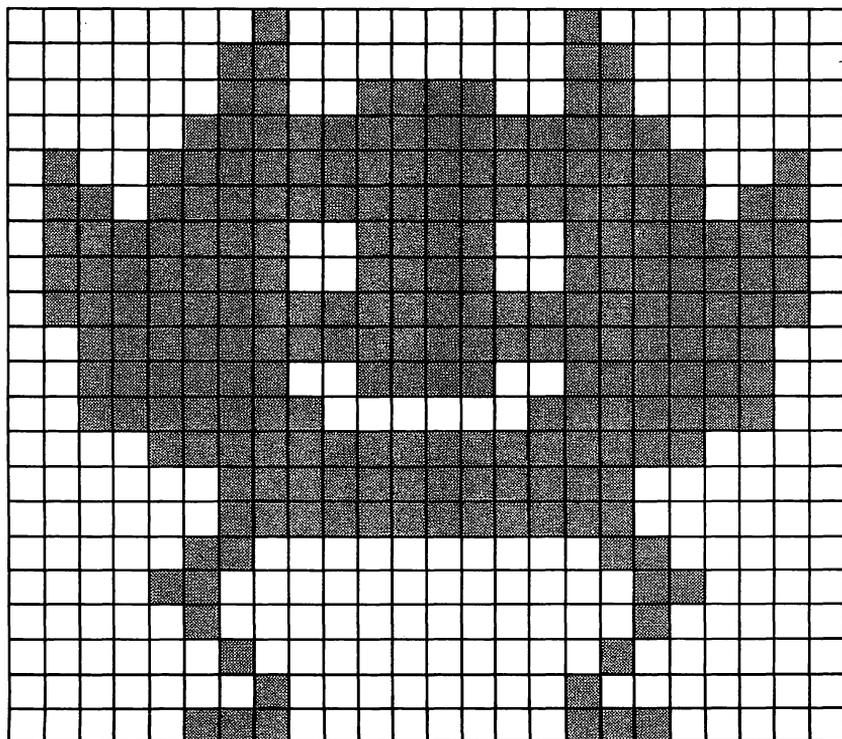
## Sprite graphics

Sprites are perhaps the single most powerful graphics feature of the Commodore 64 – or any home micro come to that. Also sometimes called Moveable Object Blocks (MOBs) they allow you to create high resolution characters which can be moved around the screen with ease.

On many home computers the only way to create moving displays is to print a character, pause, delete it, pause, print it again in another position. If this object is to move over a second, there is a further step in that the background object has to be 'remembered' and replaced after the

moving object has passed on. On the 64 sprites do away with most of these steps. Once the character is defined, it is only necessary to tell it where to be – the computer takes care of all the other details.

Unfortunately, the power of sprites is lost to most programmers because of the simple dialect of Basic on the 64. Commodore, sadly, decided not to add extra commands to allow the simple use of sprites so it is a painstaking business. However, the advantages are well worth the effort so let's see if we can take some of the pain out of the process.



## The theory

First, what exactly is a sprite? On the 64 each sprite has a resolution of 24 horizontal dots by 21 vertical dots. From what you have learned of high resolution graphics you might guess that each sprite needs 63 bytes to define it (3 bytes wide  $\times$  21 bytes deep). The 64 demands that each set of sprite data be separated by a zero, so each data block is 64 bytes long.

For practical purposes you can have eight sprites on screen at any time. It is possible to greatly increase this number but it requires some very advanced machine code programming to do so. However, it is possible to have many more sprites in memory at once – eight is simply the display limit. For example, you could have four versions of each sprite and display them in rotation. The limiting factor is the memory available to the VIC chip (remember the 16K window?). 255 sets of data is the absolute limit, although you would rarely need more than a dozen or so.

So for each sprite you need a minimum of 64 data statements. Everything else is provided in hardware.

## The sprite registers

There are a number of locations in the 64 memory map that control various aspects of the sprites. Not surprisingly, most of these are in the VIC chip.

First are the sprite data pointers. These are a single byte each and are always the last eight bytes of the 1K of screen memory so if you move the

screen, the sprite pointers move as well. Under normal circumstances they are at 2040 to 2047. The content of a pointer register is a number between 0 and 255 which tells the 64 which block of 64 bytes contains the data for the sprite. For example:

**100 POKE 2040, 13**

tells the 64 that the data for sprite 0 begins at  $13 * 64$  or location 832 counting from the bottom of the VIC window.

## A limitation

When you turn on the VIC there are only a few places readily available for your sprite data. Obviously you can't use 1024 to 2023 because the screen memory is here. Other forbidden zones include most of the first 1024 locations because they contain the operating system pointers, and locations 2048 upwards because this is the start of Basic storage. By raising the bottom of Basic more space can be gained but there are three places readily available and for most purposes these will be sufficient. These are in the cassette buffer starting at 832, 896 and 960.

You can probably guess some of the sprite controls – on/off switches for each sprite, X and Y locations and the registers to control the colour of the eight sprites.

However, there are some advanced features too. Sprites have a fixed priority in relation to one another. This means that if more than one sprite appears in the same place, the lowest numbered sprite will appear 'in front' of the others. But you can make sprites

appear either behind or in front of background characters.

There are also registers which allow you to detect collisions between sprites and between sprites and the background display, to set individual sprites to multicolour, and even to expand them horizontally or vertically or both.

As you can see this is a very powerful collection of features. And you might have guessed the time has come to roll up our sleeves and start POKEing.

### TRY THIS

Generally a sprite will be created in precisely the same way as a user-defined character, using READ and DATA statements. But for a quick demonstration let's use a solid square:

```
10 FOR I = 0 TO 62
20 POKE 832 + I, 255
30 NEXT
```

Next, set the sprite pointer for sprite 0 to 832 which is data block 13:

```
40 POKE 2040, 13
```

Finally, put the sprite into the middle of the screen, set the colour to white, and turn it on:

```
50 POKE 53248, 160
60 POKE 53249, 100
70 POKE 53287, 1
80 POKE 53269, 1
```

Notice that when the program ends the sprite is still there on the screen. In direct mode enter:

```
POKE 53248, 50
```

DEFINE  
THE  
SPRITE



SET SPRITE  
POINTER



SET X, Y  
POSITION



COLOUR  
SPRITE



TURN ON



Now switch the sprite to background priority with:

```
POKE 53275, 1
```

To turn off the sprite enter: POKE 53269, 0 or press RUN/STOP and RESTORE.

Let's move the sprite around by adding a few lines:

```
90 FOR X = 0 TO 200  
100 POKE 53248, X  
110 NEXT  
120 GOTO 90
```

Run this and notice the way the sprite scrolls into view from the left. This is because the sprite area is larger than the display area.

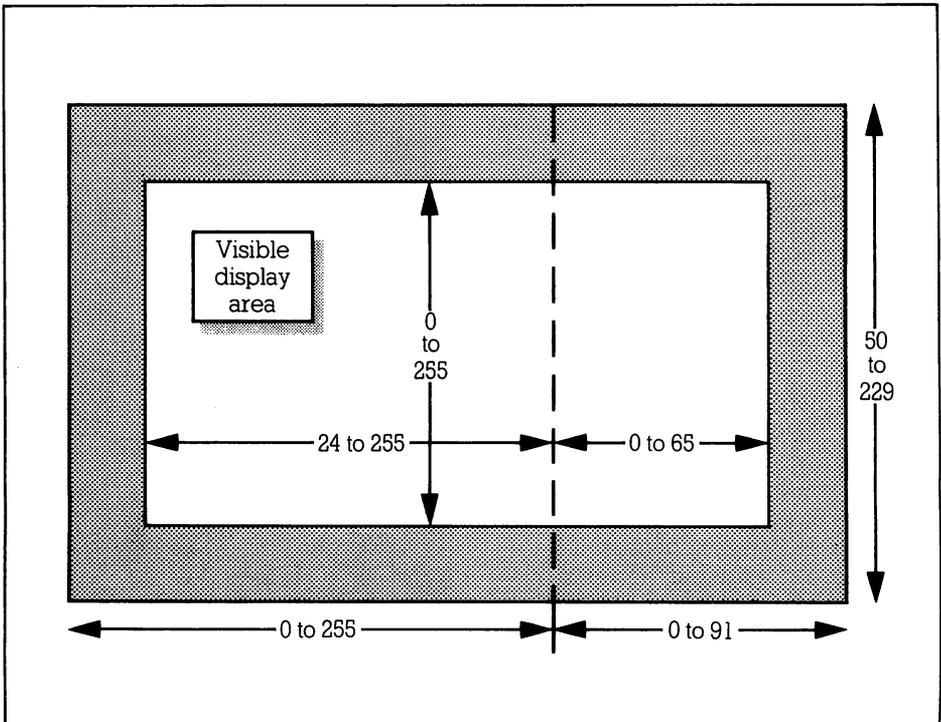
### The next step

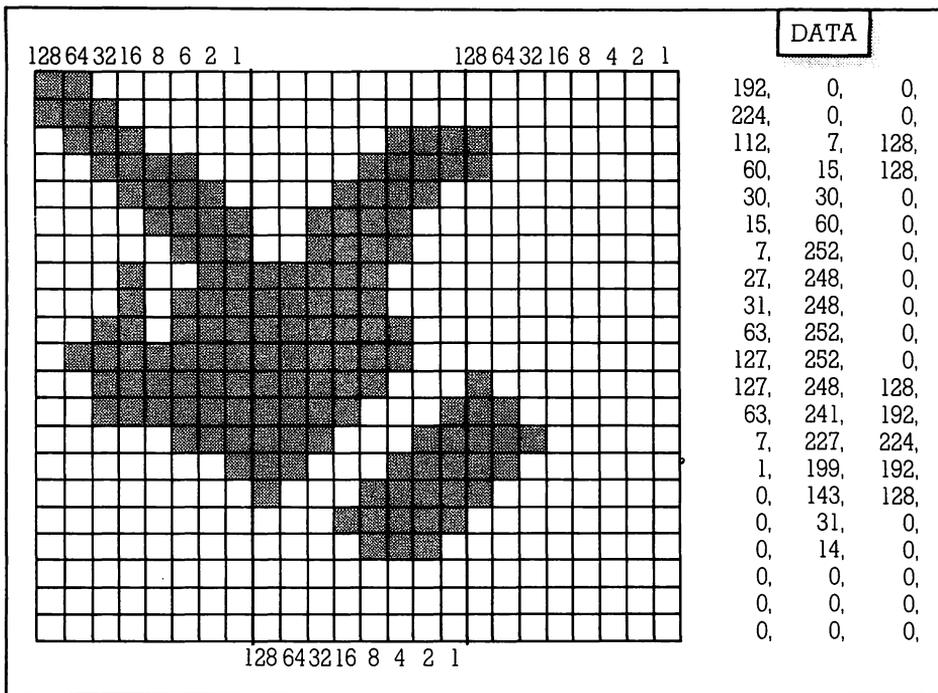
We'll consider sprite positioning in more detail in the next chapter. First let's see how to create a more interesting sprite. The data is laid out in this fashion:

```
Byte 1 Byte 2 Byte 3  
Byte 4 Byte 5 Byte 6  
Byte 7 etc
```

and so on for all 63 bytes.

Just as with user-defined graphics each dot in the sprite is turned on by setting the relevant bit to a 1. We then add up the bits to arrive at the byte values for our data statements. (This is one of the reasons why programming sprites can be such a chore – to create





four sprites involves working out and entering over 250 values.)

A glance at the accompanying table shows that many single registers control all eight sprites. This means that we need to manipulate each of the eight bits individually in order to control one or more sprites without affecting the remainder.

For example, location  $V+21$  contains the bit switches to turn sprites on or off. To turn on sprite 3 we could use `POKE V+21, 8`. However, this would have the side effect of turning off all the other sprites. So use `POKE V+21, PEEK(V+21) OR (2 ↑ 3)`. Alternatively, work out the value of  $2 ↑ 3$  (8) and use that instead. To turn on more than one sprite, calculate the values and add them together. For example, to turn on sprites 0 and 7 use `POKE V+21, PEEK(V+21) OR 129`.

### TRY THIS

Using the 'solid square' example above, try displaying all eight sprites on screen at different places and in different colours.

Remember that you don't need to define all eight sprites. Just set the sprite pointers to the same set of data. Use the table to find the colour and position registers. The values to be POKEd for colour are the same as normal, eg 0 for black, 1 for white, 2 for red.

## Sprite registers

Note: for ease of programming use V as the base of the video chip.

| Register | Location | Function                     |
|----------|----------|------------------------------|
| V        | 53248    | sprite 0 X value             |
| V+1      | 53249    | sprite 0 Y value             |
| V+2      | 53250    | sprite 1 X                   |
| V+3      | 53251    | sprite 1 Y                   |
| V+4      | 53252    | sprite 2 X                   |
| V+5      | 53253    | sprite 2 Y                   |
| V+6      | 53254    | sprite 3 X                   |
| V+7      | 53255    | sprite 3 Y                   |
| V+8      | 53256    | sprite 4 X                   |
| V+9      | 53257    | sprite 4 Y                   |
| V+10     | 53258    | sprite 5 X                   |
| V+11     | 53259    | sprite 5 Y                   |
| V+12     | 53260    | sprite 6 X                   |
| V+13     | 53261    | sprite 6 Y                   |
| V+14     | 53262    | sprite 7 X                   |
| V+15     | 53263    | sprite 7 Y                   |
| V+16     | 53264    | X > 255*                     |
| V+21     | 53269    | sprites on/off               |
| V+23     | 53271    | expand sprites vertically    |
| V+27     | 53275    | sprite/background priorities |
| V+28     | 53276    | multicolour sprites          |
| V+29     | 53277    | expand sprites horizontally  |
| V+30     | 53278    | sprite-sprite collisions     |
| V+31     | 53279    | sprite-background collisions |
| V+37     | 53285    | sprite multicolour 0         |
| V+38     | 53286    | sprite multicolour 1         |
| V+39     | 53287    | sprite 0 colour              |
| V+40     | 53288    | sprite 1 colour              |
| V+41     | 53289    | sprite 2 colour              |
| V+42     | 53290    | sprite 3 colour              |
| V+43     | 53291    | sprite 4 colour              |
| V+44     | 53292    | sprite 5 colour              |
| V+45     | 53293    | sprite 6 colour              |
| V+46     | 53294    | sprite 7 colour              |

Note that there are gaps in this table where registers in the VIC chip are used for facilities other than sprites.

\*V+16 (location 53264) is used to set horizontal positions greater than 255. See chapter 9 for more details.

---

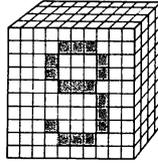
## **Checklist**

In this chapter you've learned:

- The theory of sprites.
- The locations in the VIC chip that control sprite functions.
- How the sprite data is stored.
- How to display sprites in colour.

## **Project**

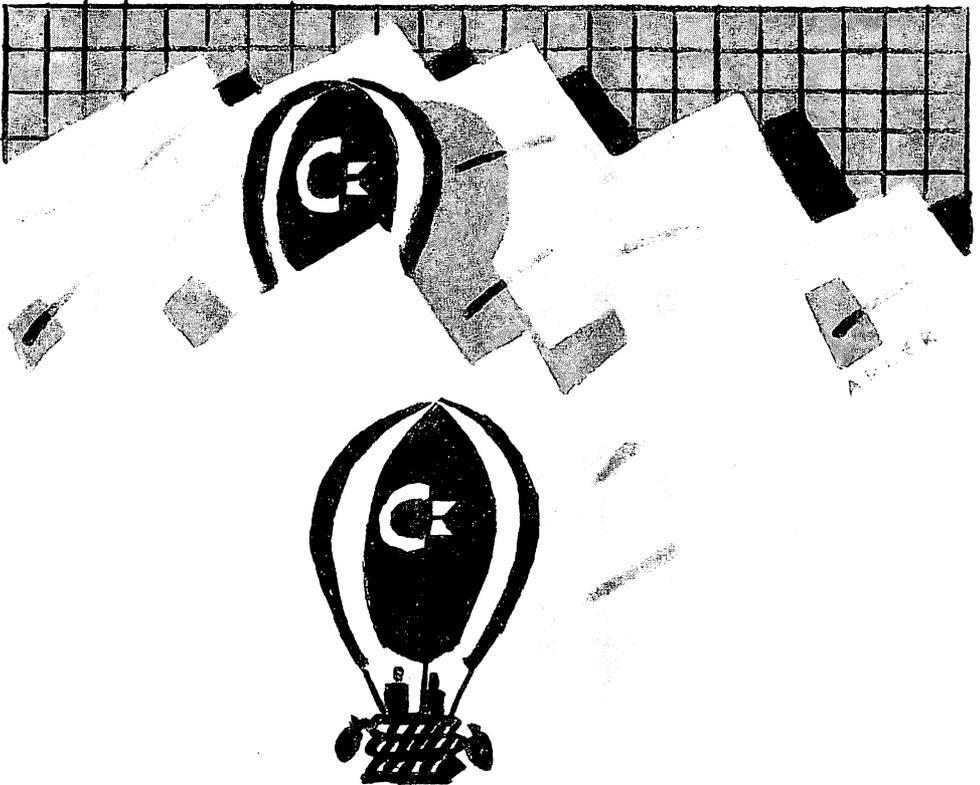
- Using the reconfigured memory map in chapter 5 adapt the 'square sprite' program to work (remember the data pointers and data storage area will move).



---

# Advanced sprites

---



In the last chapter we looked at how easy it can be to get a sprite on screen. Now let's look in more detail at getting a sprite of our own design to move about. If necessary, refer back to the table of sprite registers to see what's happening.

### The program

The first stage is to create the sprite using squared or graph paper. The 'pointing hand' from the last chapter can be reduced to the following data statements:

```
1000 DATA 192, 0, 0, 224,
      0, 0, 112, 7
1010 DATA 128, 60, 15,
      128, 30, 30, 0, 15
1020 DATA 60, 0, 7, 252,
      0, 27, 248, 0
1030 DATA 31, 248, 0, 63,
      252, 0, 127, 252
1040 DATA 0, 127, 248,
      128, 63, 241, 192, 7
1050 DATA 227, 224, 1,
      199, 192, 0, 143, 128
1060 DATA 0, 31, 0, 0, 14,
      0, 0, 0
1070 DATA 0, 0, 0, 0, 0, 0,
      0, 0, 0
```

To put the data into the correct place is a simple job for a FOR...NEXT loop. We'll put the hand into the data space starting at 832 which is block 13 and set V to the base address of the VIC chip:

```
10 PRINT CHR$(147)
20 V = 53248
30 POKE 2040, 13
40 FOR I = 0 TO 62
50 READ A: POKE 832+I, A
60 NEXT
```

Enter the program so far and run it. If you get an **OUT OF DATA** error, carefully check the **DATA** statements and make sure that you have not missed a comma. If all is well, in direct mode enter these lines:

```
POKE V+21, 1
POKE V+39, 1
POKE V, 150
POKE V+1, 100
```

If everything has been done correctly a white pointing hand should appear in the centre of the screen. In fact, it will be slightly off centre. In the last chapter we mentioned that the visible area is smaller than the display area.

This is deliberate and allows you to scroll your sprites into view from behind the border area. The borders are 23 pixels wide at the sides and 49 pixels deep at the top and bottom. So to have the whole of the sprite in view, the X coordinate must be greater than 23 and the Y coordinate must be between 50 and 229.

### Caution

One point to be careful of is that X and Y coordinates refer to the top left pixel of the sprite. If your sprite is smaller than  $24 \times 21$  and does not start at the top and left edges you must make allowances for this.

Let's add some lines to our program and put the sprite into action.

```

70 X = 24: Y = 60
80 POKE V+21, 1
90 POKE V+39, 1
100 IF X+7 > 255 THEN X =
    24: Y = Y+8: PRINT
    CHR$(13);
105 IF Y>255 THEN END
110 POKE V,X: POKE V+1, Y
120 GETAS$: IFA$ = "" THEN
    120
130 FOR X = X TO X+7: POKE
    V,X: NEXT
140 PRINTAS$;
150 GOTO 100

```

### How it works

Referring back to the table of sprite registers you'll see that lines 80 and 90 turn on the sprite and set the colour to white. Line 100 looks odd here because it's checking for a condition before anything has happened, specifically to see whether the sprite is

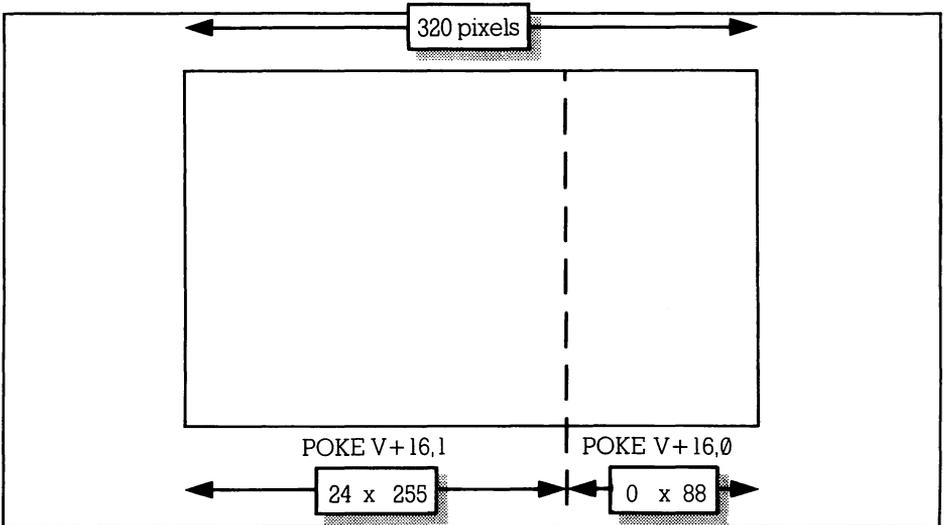
moving too far across the screen (more on this in a minute). If X would become too great the sprite is moved down and back to the left of the screen.

Line 110 displays the sprite and then the program waits for you to press a key. Run the amended program and when the sprite appears, type something on the keyboard and see what happens.

This is a simple example of how to relate what the sprite is doing to what happens on the normal text screen. Provided that the sprite position is set to the first print position it's easy to keep track of things. Simply move the sprite eight pixels across the screen to the next print position, or eight pixels down to the next screen line.

### The wide screen

But what if you don't know in advance where the sprite will be? We can work out the nearest print position



quite simply. If the sprite is at X position 86 and Y position 140 it works out like this:

screen column = INT(X-24)/8

screen row = INT(Y-50)/8

Remember that we have to deduct 24 and 50 for the border areas. In this case we get 86-24 = 62, divided by 8 = 7 and 6 left over; and 140-50 = 90 divided by 8 = 11 and 2 left over so the top left corner of the sprite is in the seventh column across and the eleventh row down.

reverse video is easy with the following line. It simply turns off all the dots that were on and vice versa. In this case assume that the sprite data is in the block starting at location 832:

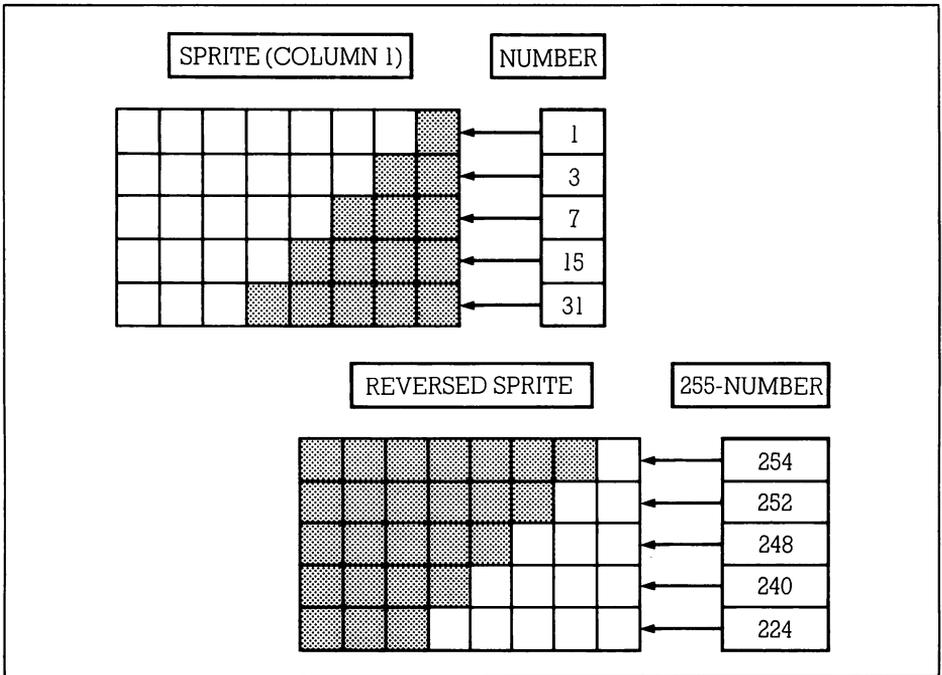
```
FOR I = 832 TO 832 + 62:
POKE I, 255 - PEEK(I):
NEXT
```

### A complication

So far our sprite movement has been confined to the left of the screen with X values less than 255. This is because we cannot POKE a value larger than this into any memory location. However, we will often want to use the full screen width and there is a register that allows us to do this. In the sprite register table this is location

### TRY THIS

With this information, change our program so that you can print a message in the middle of the screen. Hint: Making a sprite appear in



V+16 (53264).

Normally this holds the value 0. By changing it we can tell the 64 that a sprite has crossed the line and that the VIC chip should now put it on the right of the screen. Each of the eight bits controls one of the eight sprites and it works like this:

Imagine sprite 0 moving across the screen. When the X coordinate in location V reaches 255 it cannot go any higher (or an ILLEGAL QUANTITY ERROR will result). We now POKE V+16, 1 and this resets X to 0 but the X coordinate now starts at 256.

For example, on the high resolution screen we have positions 0 to 319 across the screen. If sprite 0 is at 200 the relevant registers will look like this:

```
PEEK(V) = 200
PEEK(V+16) = 0
```

When sprite 0 'crosses the line' and is now at position 256, the registers must look like this:

```
PEEK(V) = 0
PEEK(V+16) = 1
```

At position 319 the sprite will vanish behind the righthand border but the registers will look like this:

```
PEEK(V) = 91
PEEK(V+16) = 1
```

Making use of Boolean operators once more we can work out the values for V+16 if more than one sprite has crossed the line. For example, if sprite 7 is at position 300 then V+16 will be set to 128. If sprites 1 and 2 are over the line then V+16 will be set to 6.

If you expect your sprite to use the full screen you need to include the following check:

```
500 IF X + X1 > 255 THEN
    POKE V+16, 1: POKE V,
    0
```

where X1 is the increment for the horizontal position. If you want your sprite to disappear on the right and reappear on the left then this line will do the trick:

```
500 IF PEEK (V+16) = 1 AND
    X > 91 THEN POKE V, 0:
    POKE V+16, 0
```

Both of these examples apply to sprite 0 only.

### A dodge

Remember that it is not essential that you use the full screen width for your sprites. From Basic where speed is of the essence in action games, the necessary checks and actions to cross the line can slow things down unacceptably. In this case use the right side of the screen for the game information like scores, number of lives, fuel remaining and so on, and confine the sprite action to the remainder of the screen. You will find that many commercial games use this dodge.

## Expanded sprites

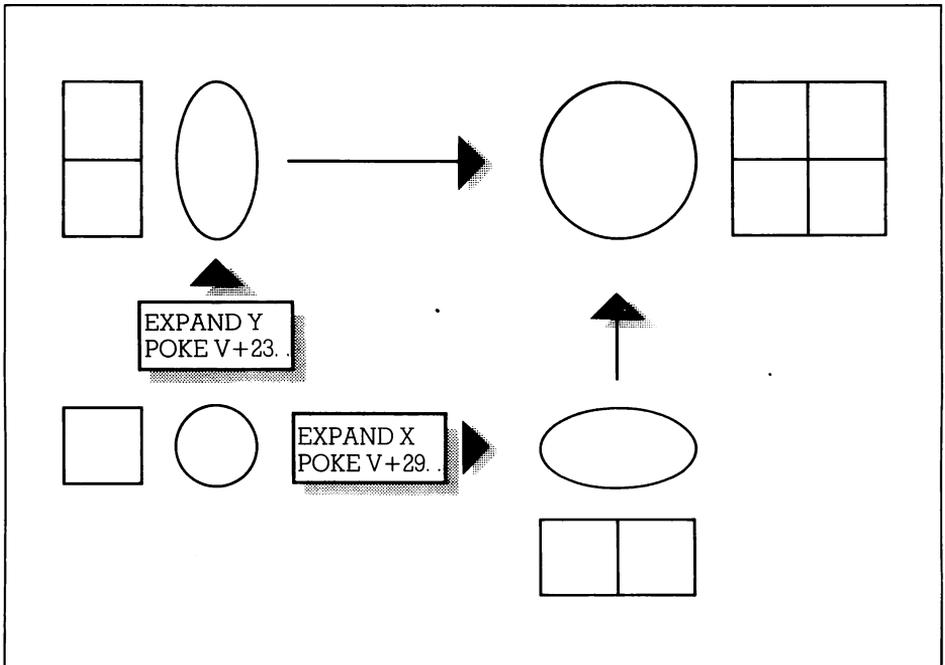
Another feature of sprites on the 64 is the ability to double their size both horizontally and vertically with a single POKE. The relevant registers are

V+29 and V+23 respectively, and once more these two locations use their eight bits as on/off switches for the eight sprites.

### Caution

There are two things to note about expanded sprites. First, the resolution does not increase – your sprite remains a  $24 \times 21$  picture but the horizontal or vertical dots double in size. The second point is this: expanding an ugly sprite will not make it any better. Generally it will look

worse. In addition, an attractive unexpanded sprite will suffer for being expanded. If you need expanded sprites in your program, design them with this point in mind and try them out before you include them in your program.



To expand sprite N, use the following line:

```
POKE V+29, PEEK(V+29) OR  
2↑N
```

and to unexpand it use:

```
POKE V+29, PEEK(V+29) AND  
(255-(2↑N))
```

### Sprite collisions

One of the keys to computer animation is the ability to detect and act upon collisions. Think of almost any computer game and the chances are it will depend in some way on this ability – missiles hitting targets, balls rebounding from obstacles, cars running off the road.

Collision detection with sprites is made easy by the presence of two registers in the VIC chip: V+30 and V+31. These record collisions between sprites and between a sprite

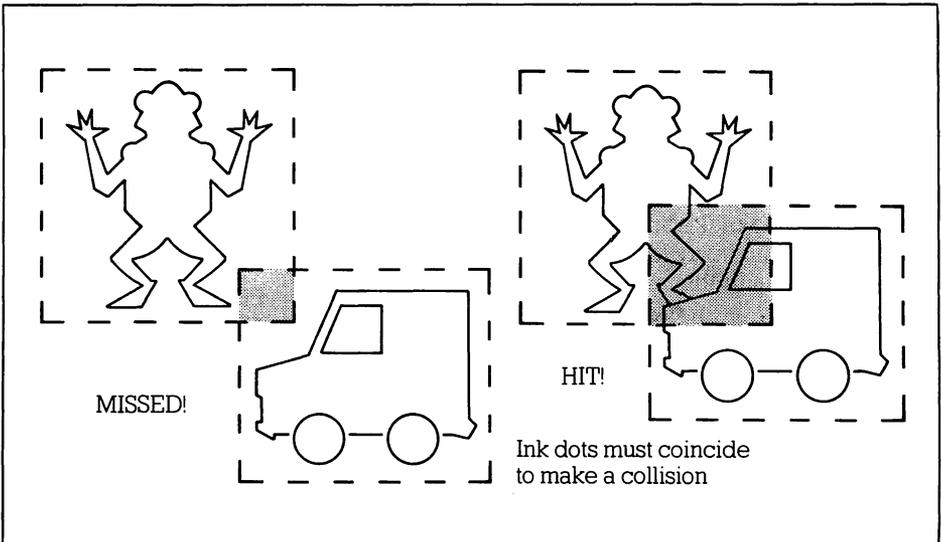
and the background characters respectively. Each bit is set when the corresponding sprite is involved in a collision. For example, if  $PEEK(V+31) = 16$  then sprite 4 has hit something.

### Caution

An important point to bear in mind when programming for sprite collisions is that the registers are cleared as soon as they are PEEKed, so you should always use the following construction:

```
100 A = PEEK(V+31): IF  
A = etc
```

You can then carry out the necessary checks on the variable A.



## TRY THIS

The following program will demonstrate the points using two simple sprites. It will move them about and produce a suitable message on collision.

```
10 PRINT CHR$(147)
20 V = 53248
30 FOR I = 832 TO 832
  + 64
40 POKE I, 255: NEXT
50 POKE 2040, 13: POKE
  2041, 13: POKE V+21, 3
60 POKE V+39, 1: POKE
  V+40, 7
70 POKE V, 75
80 POKE V+3, 200
90 FOR I=1 TO 200
100 POKE V+1, I
110 POKE V+2, 250-I
120 REM: POKE V+30, 0
130 A = PEEK(V+30)
140 IF A <> 0 THEN 160
150 NEXT: GOTO 90
160 PRINT "CRASH !"
170 END
```

Note line 120 which would set the collision register to zero. This may seem strange but in some programs it is necessary to prevent an old collision from messing things up. Its purpose here would be to make sure that we only detect collisions when we are looking for them. In this program it isn't really necessary, hence the REM statement which makes it ineffective. Edit the line to remove the REM then run the program two or three times in a row and see what difference it makes.

In this case the job is easy because we are dealing with only two sprites so if V+30 is set at all our two sprites have collided. How do we detect collisions

where there are more than two sprites? The bit values give us the answer.

For example, if  $\text{PEEK}(V+30) = 128$  then sprite 7 (bit value 127) and sprite 0 (bit value 1) have collided. If the value of V+30 is 255 then all the sprites have collided.

The problem here, of course, is that all the sprites may be in one place, or they may have hit in groups of two or three, so a further check of each sprite's X and Y positions is necessary.

Similarly in detecting sprite to background collisions, there is no way of telling what background character is involved. You need to use the technique given earlier to convert the sprite's position into a screen position, and see what's there.

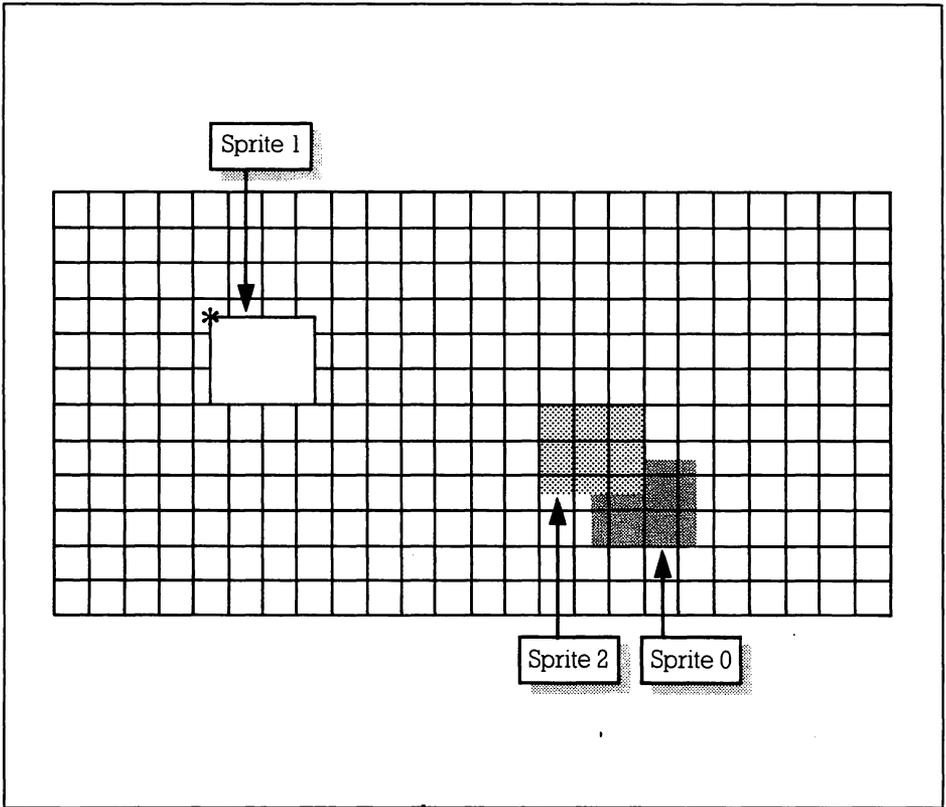
## Sprite collision detection

In this example we have three sprites on screen, sprites 0, 1 and 2. If at this point we were to PEEK (V+30) and PEEK (V+31) we would get the values 5 and 2 respectively indicating that sprites 0 and 2 have collided with each other, and sprite 1 has hit a background character.

At this point there's no way of telling which character has been hit so we need the following kind of routine:

```
100 ROW = PEEK(V+3) - 50
110 ROW = INT(ROW/8)
120 COL = PEEK(V+2) - 24
130 COL = INT(COL/8)
140 HIT = 1024 + COL * ROW
```

Now we can find the screen code of the character in the collision by PEEK (HIT) which, in this case, will give us 42 – an asterisk.



The conclusion to be drawn from all this is: don't be too ambitious in your programs. If you have eight sprites whizzing about on screen and the game depends on sprite to sprite

collisions, or sprite to background collisions, you have to allow for the time it will take the 64 to find out what's hitting what. The action will likely slow down to a crawl.

### Multicolour sprites

In the same way that we can use multicolour mode with user-defined characters and in high resolution bit map mode, we can have multicolour sprites.

Multicolour sprites have the usual disadvantage of horizontal resolution being halved – from 24 to 12 dots – and

each dot becomes two pixels wide. The four colours available are contained in four locations: the screen colour (background); sprite multicolour 1 (register  $V+37$ ); sprite multicolour 2 (register  $V+38$ ); and the normal sprite colour registers between  $V+39$  and  $V+46$ . However,

just to keep things interesting Commodore decided that multicolour sprites would work a little differently from the other multicolour modes. Whereas user-defined graphics and bit map mode have 11 as the bit pair for the main colour, sprites use bit pair

10. Bit pair 11 takes its colour from multi-colour 2 (V+38).

If all of this sounds a little confusing, it's only because there are a number of locations to keep track of. If you approach things methodically, you shouldn't go far wrong.

## The theory

The registers and bit pairs in your design work like this:

| Bit pair | Colour        | Location     |
|----------|---------------|--------------|
| 00       | screen colour | 53281        |
| 01       | multicolour 1 | V+37         |
| 10       | sprite colour | V+39 to V+46 |
| 11       | multicolour 2 | V+38         |

From this you should be able to see that when you use more than one sprite in multicolour mode they will have three colours in common: screen colour, multicolour 1 and

multicolour 2. Therefore, if you want multicolour sprites in substantially different colours you should use bit pair 10 for the largest part of the design.

One of the advantages of multicolour sprites is that you can have some sprites in normal mode and others in multicolour. The choice is yours and is controlled by register V+28, another of those single byte locations that controls eight sprites. To turn a sprite to multicolour, you have to set the relevant bit of V+28. Use this line to

switch sprite N without affecting any other sprite:

```
POKE V+28, PEEK(V+28) OR
2 ↑ N
```

and to turn it back to normal:

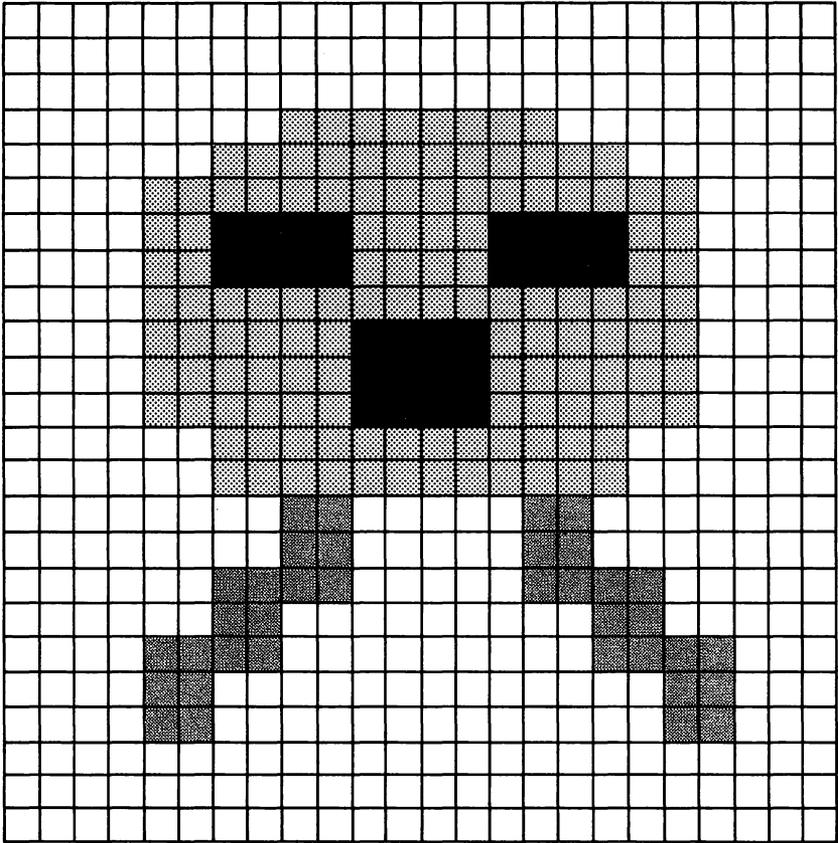
```
POKE V+28, PEEK(V+28) AND
(255 - 2 ↑ N)
```

## Caution

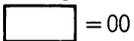
Remember what happened when we turned on multicolour mode with normal text? They became a colourful mess. The same will happen with your

sprites unless you design them specifically to take advantage of multicolour mode. If you need to have two versions of the same sprite

Multicolour space invader – colour map



Background



Multicolour 1



Sprite colour



Multicolour 2



A space invader design marked for colour in bit pairs. The actual bit design is overleaf.

alternating between normal and multicolour modes you must design it twice and then change the sprite pointer to look at the required version.

Another point to beware while using multicolour mode concerns collision detection. In normal mode a collision will be flagged if any part of the sprite which is on hits another non-transparent object. In multicolour mode, however, the 64 considers bit pair 01 to be transparent too. So if you design a multicolour sprite which largely uses multicolour 1, you may have problems detecting a collision.

On the other hand this can be useful if you want a sprite on screen which should not cause collisions. Just make it all multicolour 1.

### How it works

Now let's go through the process of creating a multicolour sprite step by step – in this case, our old friend the

space invader.

The first stage is to design the sprite on squared paper. It is a help to use graph paper marked off into a 24 × 21 grid, then mark each pair of squares going across for the bit pairs.

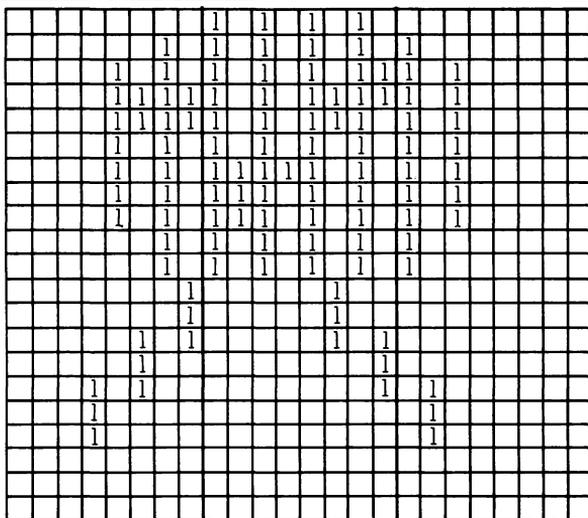
Use different pens, or three kinds of shading, for the different colours otherwise it is easy to get confused.

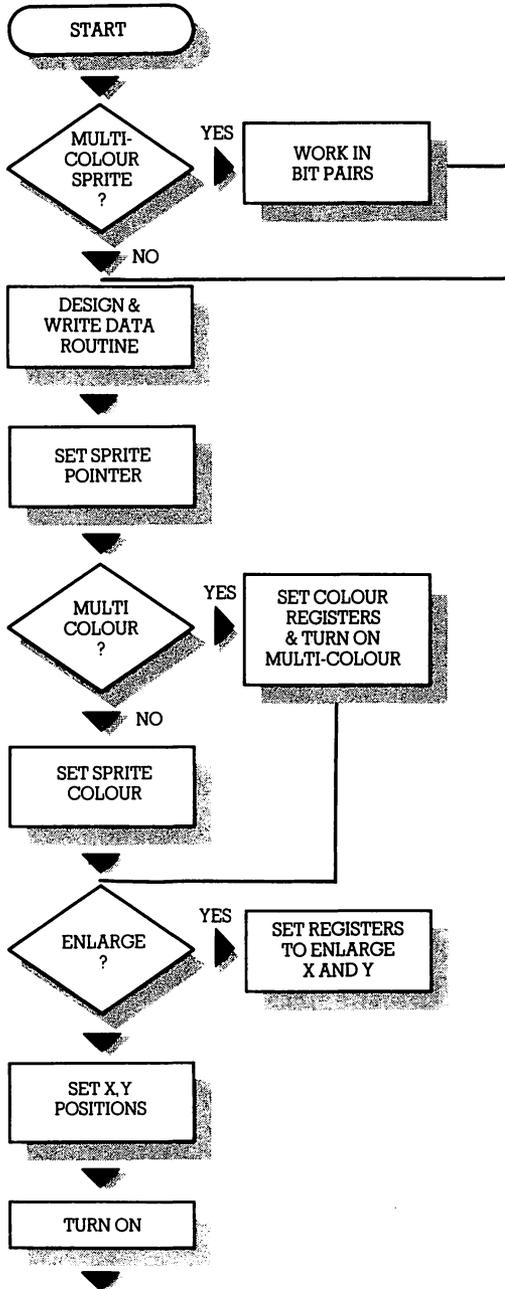
Next, work out the values of the 63 bytes. If necessary, transfer your colour design onto another grid marked in pixel pairs.

Third, write the Basic routine to POKE the data values into the sprite data block.

Finally, set the colour registers to the values you want, like this:

```
POKE V+29, 1: REM turn on
multicolour
POKE V+39, 1: REM sprite
colour white
POKE V+37, 5: REM
multicolour 1 green
POKE V+38, 2: REM
multicolour 2 red
```





---

It's worth saying again at this point: sprites take time to set up but if you proceed logically and carefully you should not have much trouble and the results are well worthwhile.

---

### **Checklist**

---

In this chapter you have learned:

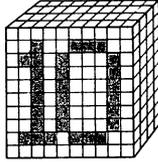
- How to design your own sprites.
- How to relate the sprite display area to the normal display screens.
- How to use the full width of the screen for sprite displays.
- How to expand sprites.
- How to detect sprite and background collisions.
- How to define a multicolour sprite and set the multicolour registers.

---

### **Projects**

---

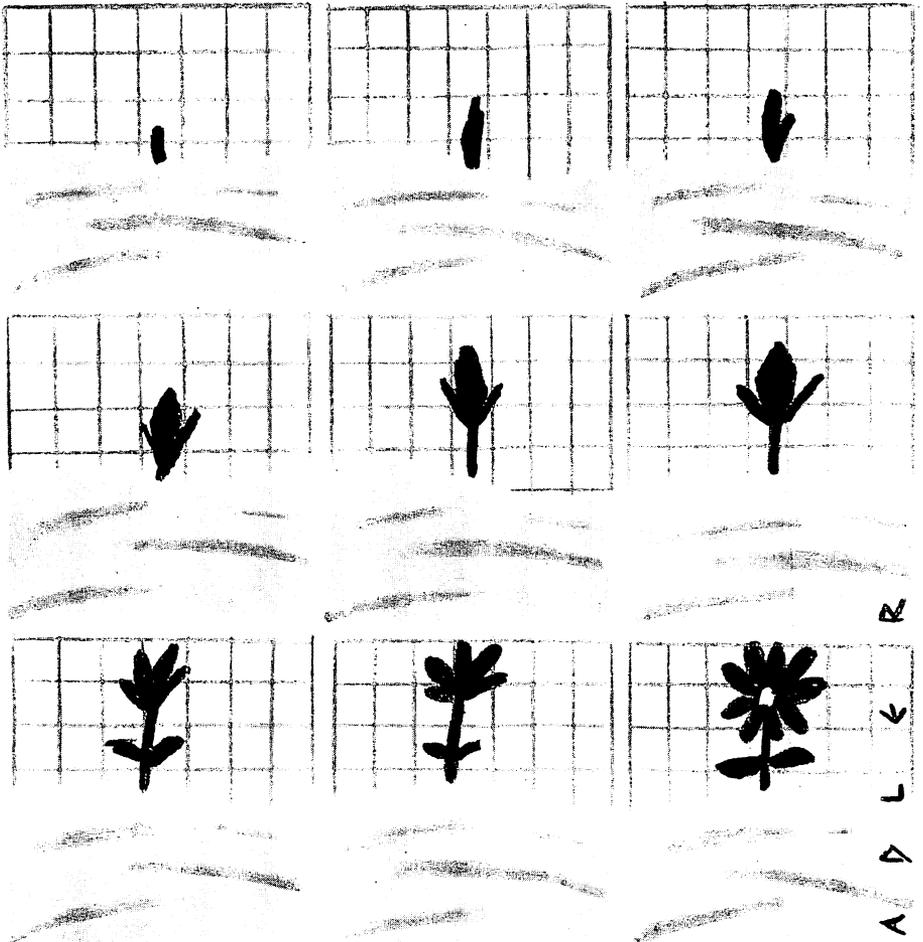
- Amend the 'writing hand' program to use the full-width screen.
- Write a program to move a sprite of your own design around the screen. Position random blocks on screen and delete them when your sprite comes into contact.



---

# Animation

---



Some micro owners use their computers for word processing, some for storing information, some for simply learning about computers but the one use almost every owner has in common is computer games – lone spaceships fighting off the alien hordes, heroic humanoids leaping past falling hazards, racing cars hurtling round grand prix circuits.

And the thing that makes a computer game unique is graphic animation. Only computers have the power to keep large numbers of objects moving on screen in response to the player and in the Commodore 64 you have a powerful tool to help you explore the techniques of animation.

In this chapter we'll look at the different ways of animating objects using block graphics and sprites, and at ways of speeding things up when Basic starts to run out of steam.

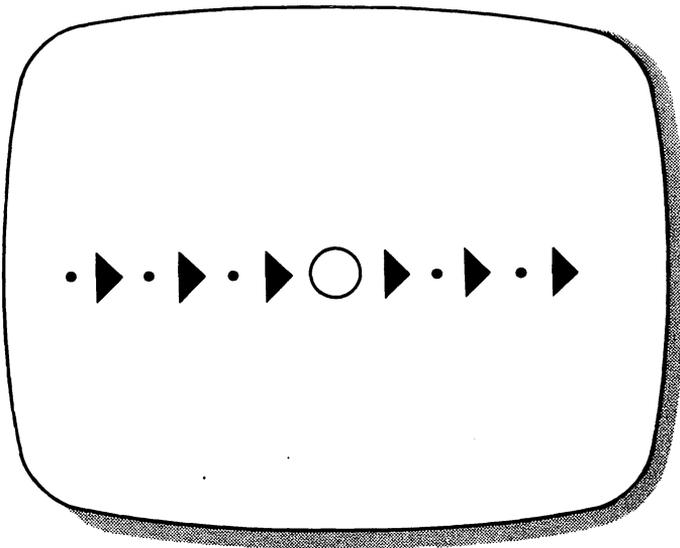
## The theory

At the simplest level animation involves displaying an object, pausing, blanking it out then displaying it in a slightly different position. The easiest example is a moving ball since it doesn't change when moving, unlike people and animals.

## TRY THIS

Try the following example:

```
10 PRINT CHR$(147)
20 POKE 53280, 0: POKE
   53281, 0
30 X = 0: Y = 13
40 SC = 1024: CO = 54272
50 L = SC + X + Y * 40
```



```

60 POKE L, 88: POKE L +
   CO, 1
70 FOR D = 1 TO 50: NEXT
80 POKE L, 32
90 X = X + 1
100 IF X > 39 THEN X = 0
110 GOTO 50

```

### How it works

This program simply moves a ball from one side of the screen to the other and then starts again but it involves most of the principles of animation.

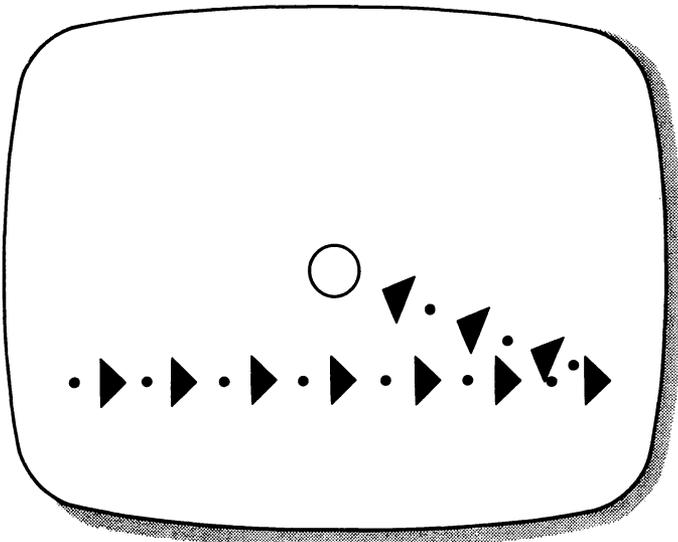
There are several points to note:

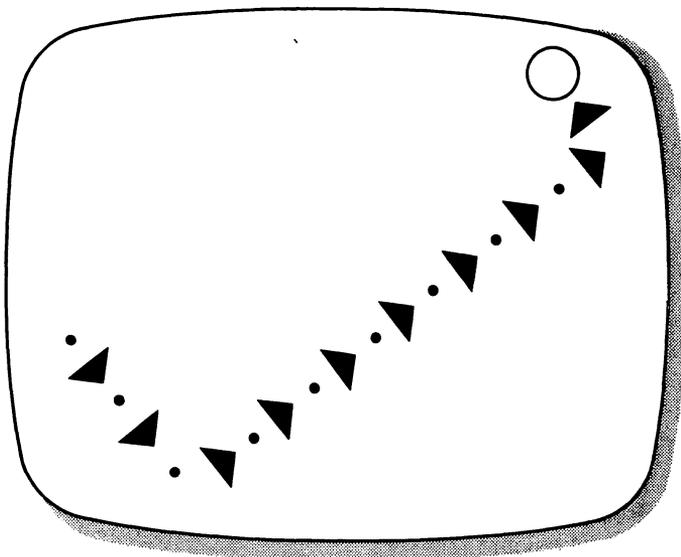
- The use of variables for the start of screen memory (SC) and the value to be added to get colour memory (CO).

- The use of X and Y coordinates to determine the screen position.
- The simple calculation to work out the screen location of the object (line 50). X tells the 64 how far across the screen the object is, Y\*40 tells it how far down the screen.
- The delay loop in line 70. Try changing the value here and then delete it altogether and see the difference.

### The next step

Suppose we want to confine the object within the screen as though the screen border were physically solid. This involves another important factor in animation – response to boundaries.





Edit line 20 like this:

```
20 POKE 53280, 6: POKE
   53281, 0
```

Now we need to add two new variables to control the change in direction: DX and DY:

```
35 DX = 1: DY = 0
```

Finally make the following changes:

```
90 X = X + DX: Y = Y +
   DY
```

```
100 IF X = 39 THEN DX =
    -1
```

```
110 IF X = 0 THEN DX = 1
```

```
120 GOTO 50
```

Run the program now and the ball will appear to bounce from one side of the screen to the other.

The introduction of DY allows us to move the ball up and down as well as across the screen. Change the value of DY in line 35 to 1 and add these lines:

```
105 IF Y = 24 THEN DY =
    -1
```

```
115 IF Y = 0 THEN DY = 1
```

### The final step

So far so good. But our response depends on the edges of the screen. What if we wanted to introduce another obstacle? This is a little more complicated but hardly difficult. First let's put the obstacle on screen: add this subroutine and line 45:

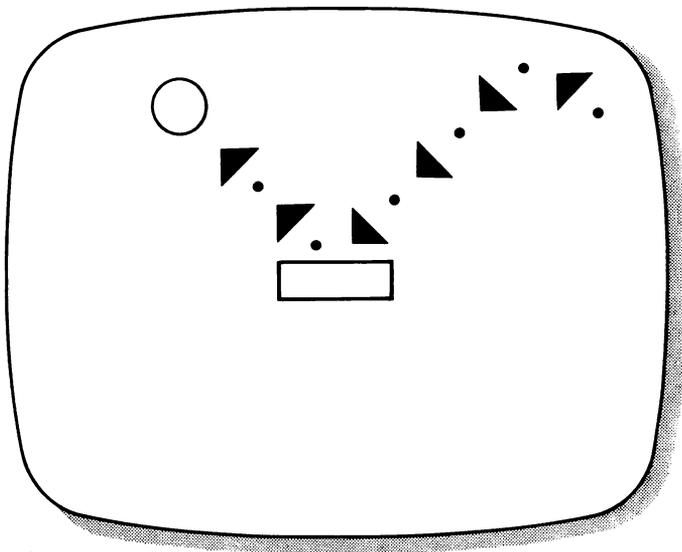
```
45 GOSUB 500
```

```
500 OB = SC + 19 + Y * 12
510 POKE OB, 160: POKE OB
    + C0, 7
```

```
520 POKE OB+1, 160: POKE
    OB+1 + C0, 7
```

```
530 RETURN
```

Next the crucial part – to determine



if our ball has hit the blocks. Actually, we check to see if it would hit the blocks next move. Because the ball has four possible directions we should check four possible locations but we can take a shortcut by saying in effect, 'if the new location is not a space then  $DX = -DX$ '.

Make the following additions:

```

55 IFPEEK(L) <> 32 THEN
    DX = -DX: GOTO 90
85 GOSUB 200
200 IFPEEK(L+DX) <> 32
    THEN DX = -DX
210 IFPEEK(L+DY) <> 32
    THEN DY = -DY
220 RETURN

```

To abbreviate the program a little we can change lines 100 to 115 like this:

```

100 IF X = 39 OR X = 0
    THEN DX = -DX

```

```

110 IF Y = 24 OR Y = 0
    THEN DY = -DY

```

thus making the necessary checks in two lines not four.

The above program is not the most efficient or elegant way of doing things but if you have followed things so far you should have a good basis for understanding the way simple animation works.

### Other techniques

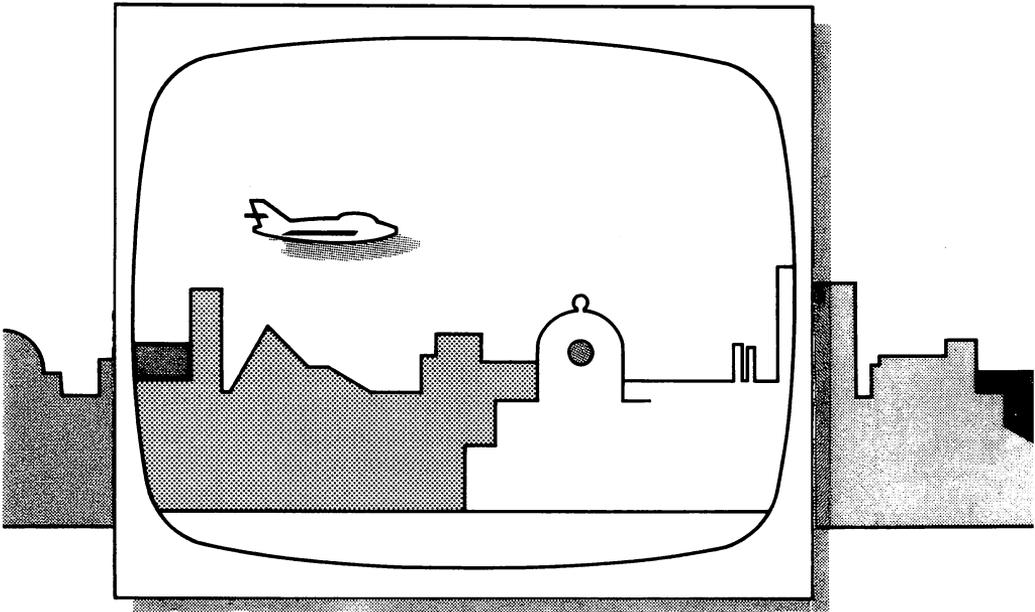
Let's look at another way of giving the impression of movement. Instead of moving the foreground object around the screen, this time we'll move a background while the object remains stationary.

The following program displays a crude plane flying above a random landscape.

```

5 DIM S(3): S(1)=162:          CURSOR DOWNS]" +
  S(2)=175: S(3)=185          CHR$(182)
10 PRINT CHR$(147)           110 OB$ = OB$ + CHR$(162)+
20 POKE 53280, 0: POKE      CHR$(185)+CHR$(175)+
  53281, 0                   CHR$(175)
30 SC = 1024: CO = 54272    120 POS$ = "[HOME + 15
40 GOSUB 100                CURSOR DOWNS]"
50 FOR I = 1 TO 160         130 FOR I = 1 TO 60: SS$ =
60 PRINTPOS; MID$(S$,I,40)  SS$ + CHR$(164): NEXT
70 PRINT OB$                140 FOR I = 1 TO 100: S =
80 FOR D = 1 TO 50: NEXT    INT(RND(0)*3+1)
90 NEXT I                   150 S$ = S$ + CHR$(S(S))
95 GOTO 50                  160 NEXT
99 END                       170 S$ = SS$ + S$ + SS$
100 OB$ = "[HOME + 5        180 RETURN

```



### How it works

Although some of this may look complex, it is quite a simple program. Lines 100 and 110 create the plane, lines 130 to 170 create the random landscape using the CHR\$ values set

in the array S(3). The action is carried out in lines 50-95 which repeatedly print a slightly different part of the landscape string S\$.

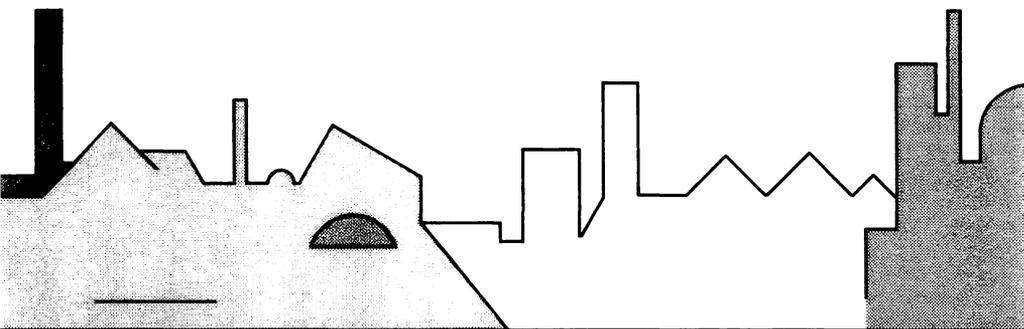
## TRY THIS

If you run this you may see the potential for a simple game. As an exercise, try to introduce a bombing routine. This would be easy since the position of the plane doesn't change and only one keypress is needed. Once you get this to work, introduce targets into the landscape string and check for hits using the same technique as in our bouncing ball program.

In this case we used only strings which are printed onto the screen. The method you choose will depend on the kind of animation you are using. For lots of small objects, POKE is probably the best technique while for large, composite characters, PRINTing a string will be quicker. If you use the

SYS AT command from chapter 3 you can have the speed of PRINT plus the convenience of POKE.

However, there is one circumstance which demands that you use POKE and PEEK for animation and that is where you need to know what other objects are on screen and where they are. Try to imagine a routine using PRINT only which would determine whether your object has hit anything else, and if it is to pass over the background, what colour it should be after your foreground object is gone. It could be done but it would be very time consuming and wasteful of memory. PEEK solves all of these problems simply and (relatively) efficiently.

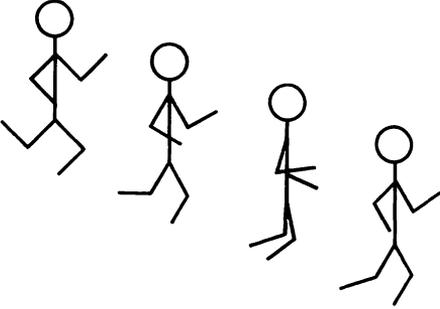


## Sprite animation

As we have already seen in the chapters on sprites, many of the problems of animation, such as

restoring background displays and detecting collisions, are made easier using sprites.

## Advantages



But sprites also take the hard work out of a new technique – true animation. So far we have looked at simple characters that remain the same throughout the program. But things in the real world are seldom like that. They change with motion or when we see them from different angles.

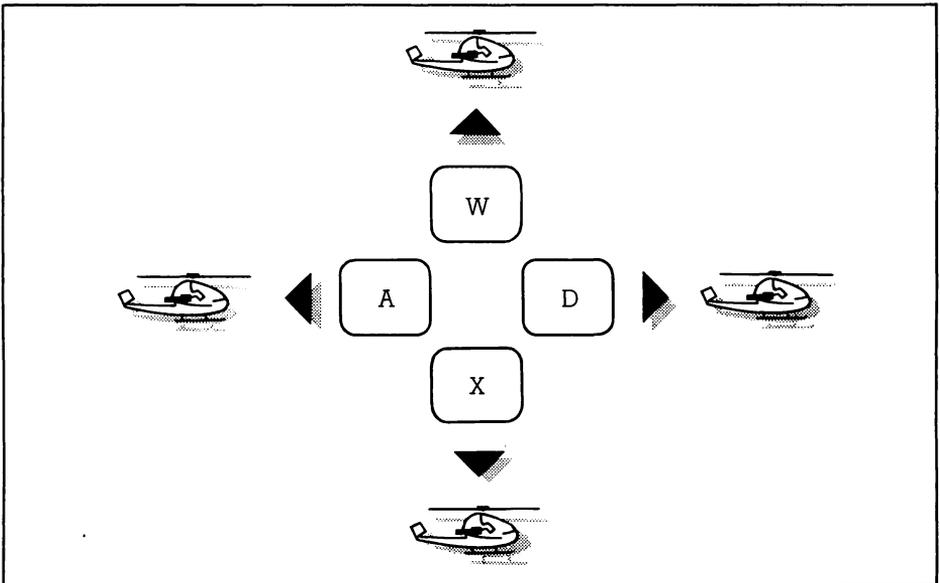
By manipulating the sprite data pointers we can display many

consecutive images, each slightly different, to create the impression of motion.

The other major advantage is that block graphics always look slightly jerky because we can only move them eight pixels at a time. Sprites can be moved a single pixel in any direction giving a very smooth motion that makes the difference between professional and amateur software.

## TRY THIS

The following program uses three pictures of a helicopter with the rotor blades in different positions and displays them successively to create the impression of rotor movement. You can move the chopper up down and sideways using the keys A, D, W and X. (Note that the program does not check for illegal values of X and Y.)



```

10 PRINTCHR$(147):GOSUB
   1000
20 V=53248:S=13:X=50:Y=75
30 POKEV+21,1:
   POKEV+39,1:POKEV+29,1
40 POKEV,X:POKEV+1,Y
50 POKE2040,S
60 Q=PEEK(197):IFQ=64
   THEN120
70 IFQ=9THENY=Y-1
80 IFQ=23THENY=Y+1
90 IFQ=10THENX=X-1
100 IFQ=18THENX=X+1
110 GOTO130
120 FORD=1TO15:NEXT
130 S=S+1:IFS=16THENS=13
140 GOTO40
150 END
199 REM *** SPRITE 1 DATA
   ***
200 DATA63,255,255,0,12,0,
   0,12,0,128
210 DATA63,224,192,96,48,
   224,224,24
220 DATA255,255,252,255,
   255,254,127
230 DATA255,252,48,7,216,0,
   6,24,0,31
240 DATA254,0,0,0,0,0,0,0,
   0,0,0,0,0,0
250 DATA0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0
255 REM *** SPRITE 2 DATA
   ***
260 DATA3,255,240,0,
   12,0,0,12,0,128,63
270 DATA224,192,96,48,224,
   224,24,255
280 DATA255,252,255,255,
   254,127,255
290 DATA252,48,7,216,0,6,
   24,0,31,254,0
300 DATA0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0
310 DATA0,0,0,0,0,0,0,0,0,
   0,0,0
315 REM *** SPRITE 3 DATA
   ***
320 DATA0,20,0,0,12,
   0,0,12,0,128,63,224,192
330 DATA96,48,224,224,24,
   255,255,252
340 DATA255,255,254,127,
   255,252,48,7
350 DATA216,0,6,24,0,31,
   254,0,0,0,0,0
360 DATA0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0
370 DATA0,0,0,0,0,0,0,0,0
999 REM *** DEFINE
   SPRITES ***
1000 FORI=0TO191:READA
1010 POKE832+I,A:NEXT
1020 RETURN

```

## How it works

This method is the simplest way of displaying successive images. The sprite pointer S is first set to the lowest data block, 13. The program then PEEKs the keyboard for a control key. If none is found (Q=64) it jumps to line 120 which is a delay loop and then increments the data pointer to the next sprite image before looping back.

If a control key is pressed the X or Y coordinate is updated and the delay loop is skipped since an equivalent amount of time has been consumed in the IF...THEN sequence. If the delay loop was performed every time, the rotors would spin slower every time a key was pressed.

## Speeding things up

In all our animation programs so far we have used delay loops when displaying the foreground objects. However, these programs have been very simple. When your own programs get more complicated, using sound and moving many more objects, you will find that you must not only eliminate the delays, but find ways of speeding up the program to keep the animation as smooth and fast as possible.

We have already discussed some of the methods in earlier chapters. For example, using the machine code SYS AT routine instead of POKE or cursor strings; putting your most used and time crucial subroutines at the start of the program etc.

A new technique, particularly useful in animation, is to work out as many as

possible of the calculations used by the program in advance. You can do this either when you write the program, or get the computer to do it during the initialisation phase before the action begins.

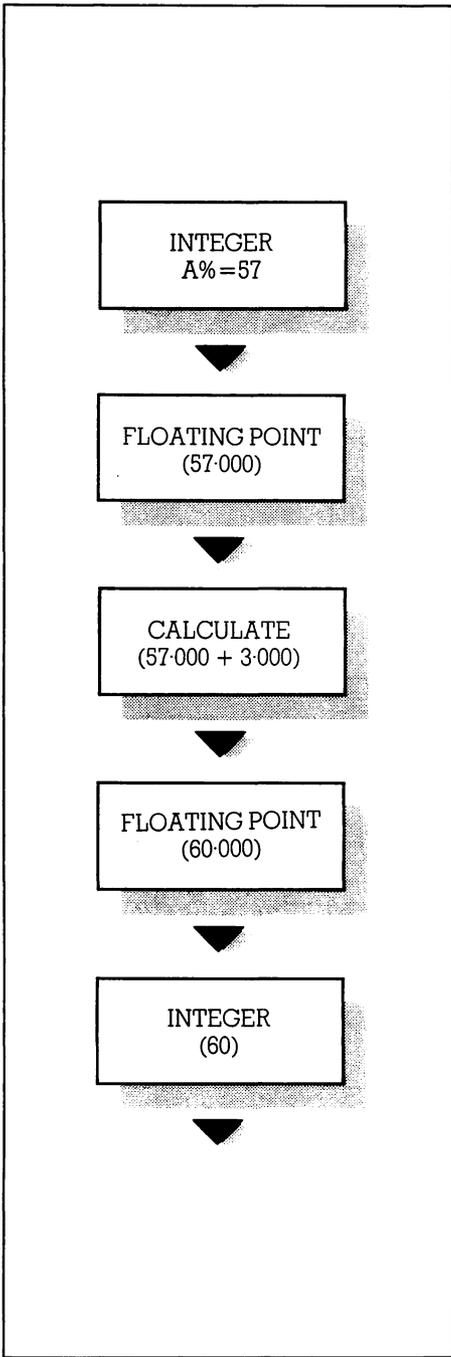
As a general rule, if it's easy for the computer it's harder or slower for you. Conversely, if you do as much of the computer's work as possible before the program is run, it will run much faster.

Having worked out the information you can use arrays to store it and the computer can use these as 'look-up' tables. For example: if you want to display more than two or three successive animated images, and the image to be displayed depends on the command received, you could hold them in an array and have the command simply change a pointer.

There are a number of other simple techniques you can use to speed up Basic programs but as usual there is a trade off. This generally comes in the form of memory taken versus speed of execution – the fastest techniques will take slightly more of your free RAM but in a 64K computer that shouldn't matter too much.

### **Tip 1: unless every byte is crucial use floating point numbers.**

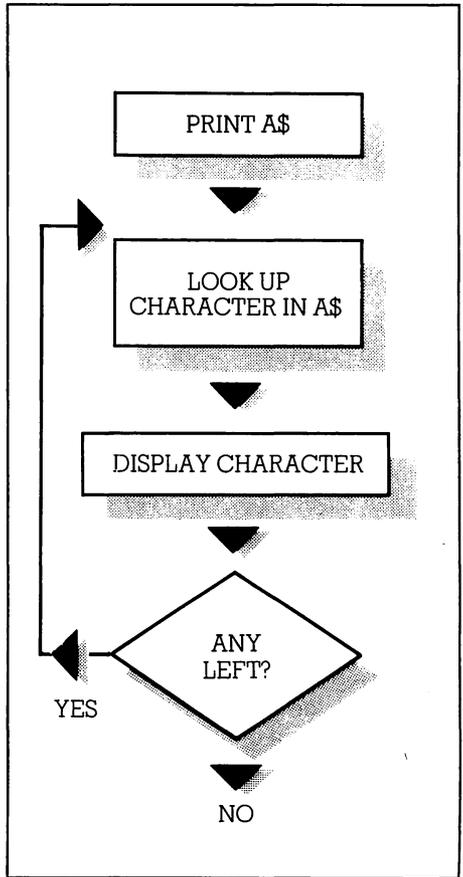
The first go-faster method involves integer variables (A% etc). On many micros these not only take less space – the computer doesn't need to leave space for numbers after the decimal point – but run faster too. However, Commodore computers handle them in an odd way. If you want to do any arithmetic with them – addition, multiplication or anything else – the 64 will convert them back to floating point, do the arithmetic, then convert



them to integer again. That conversion takes time.

**Tip 2: Use literal strings for faster execution.**

The second tip also has to do with the way the 64 manipulates its data in order to work with it – this time on strings. It is faster to use literal strings than defined strings. For example, if you define `A$ = "THE QUICK BROWN FOX"`, it will take the 64 longer to `PRINT A$`, than it will to `PRINT "THE QUICK BROWN FOX"`. In fact, the second version is almost twice as fast.



### Tip 3: Use defined variables for numbers.

This is exactly the opposite of tip 2. Here your programs will run faster if you use POKE SC, SP than if you use POKE 1024, 32. It's nearly two-and-a-half times faster actually. This is because the 64 reads those numbers as ASCII codes. It then converts them to integers and then further converts them to floating point and back again.

This is particularly important in FOR...NEXT loops where number routines can be handled dozens, perhaps hundreds of times. Look at these times:

```
10 FOR I = 1 TO 100
20 POKE 1024 + I, 32
30 NEXT
```

will take about a second to run.

```
10 SC = 1024: SP = 32
20 FOR I = 1 TO 100
30 POKE SC+ I, SP
40 NEXT
```

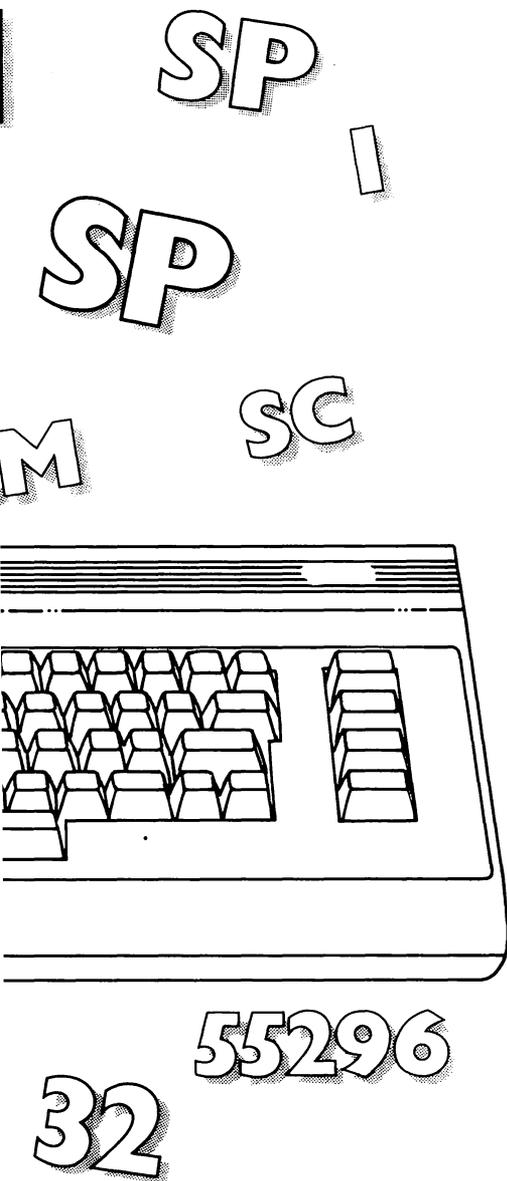
would take just over half a second. A small difference perhaps, but in the course of a program it could be sufficient time to add an extra moving graphic, or a sound routine.

Animation is probably the most testing of tasks for a computer running under Basic. Although all of the strictures about structured programming hold true there is only one hard and fast rule:

If it works in the way you want it to work it's a good program. In arcade-type games the action is 90 per cent of everything so if a convoluted, unstructured program results in quick, slick animation who cares what rules you've broken. Just make sure you know how it works.



1024  
100



---

## Checklist

---

In this chapter you've learned:

- The theory of animation and how to implement it simply using block graphics.
- An alternative method of simulating motion by moving the background.
- How sprites make moving graphics very simple.
- How to create true animation by displaying different images of the same object.
- How to detect collisions and have your program react to them.
- How to squeeze a little extra speed out of Basic.

---

## Projects

---

- Using the bouncing ball routines write a Breakout game. Breakout is old hat by today's standards of computer games but it is surprisingly addictive and is a good training exercise for more complicated programs. All of the necessary routines are given: simply put them together and add a scoring system.
- Combine the two animation techniques given (moving an object and moving the background) to produce a bomber game where the plane can move up and down over a scrolling landscape.

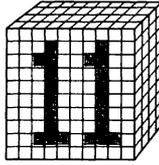
---

Add extra sprite data to the helicopter program so that the helicopter can move both ways. Write a program to make use of this (perhaps a change from death and destruction – make it a rescue helicopter picking up stranded humanoids. You could use user-defined graphics for the humanoids).

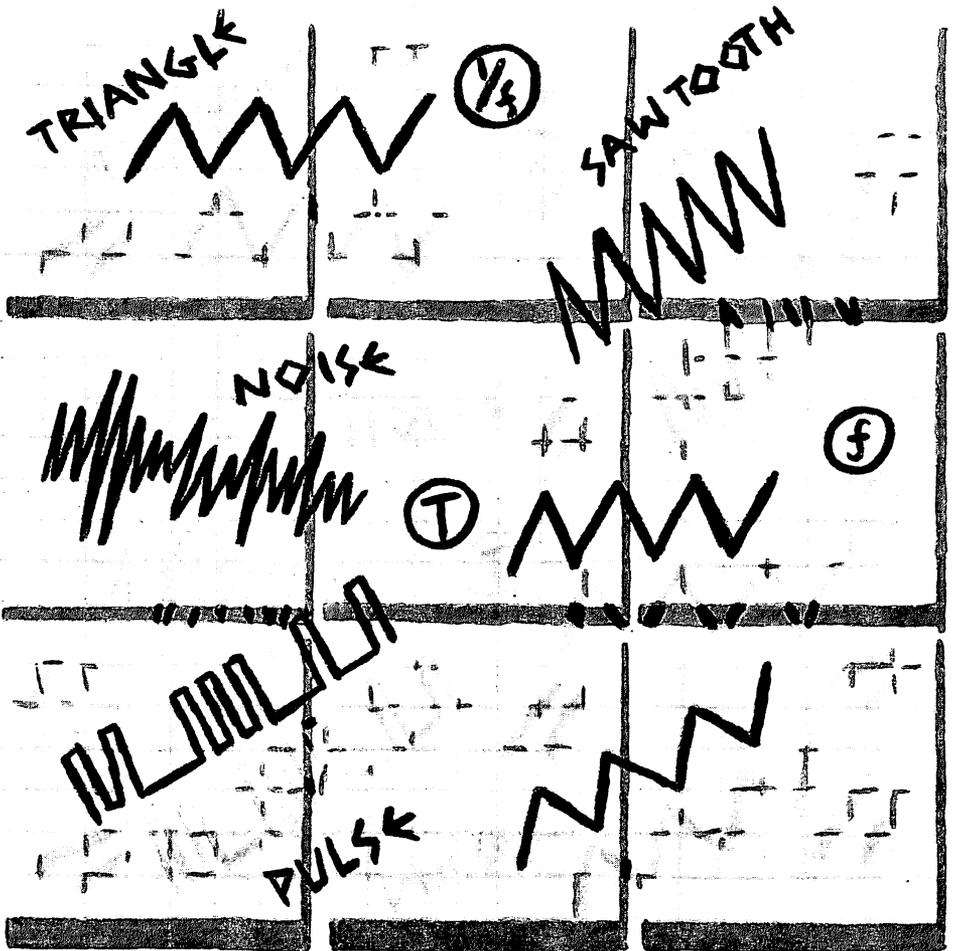
---

When you master these techniques, write an ambitious animated 'cartoon' using the full eight sprites for eight versions of the same object – perhaps a person running or a butterfly flying.

---



# Sound



Your Commodore 64 has arguably the most powerful sound synthesiser on any personal computer. The chip which gives the 64 its extraordinary abilities is the Sound Interface Device – SID – which has three voices, four programmable waveforms including noise, the ability to vary the 'shape' of a sound, filters and a multitude of special effects. What all of the jargon means is that you can create music, mimic a range of instruments and sounds from violins to aircraft engines and even create sounds that have never been heard before.

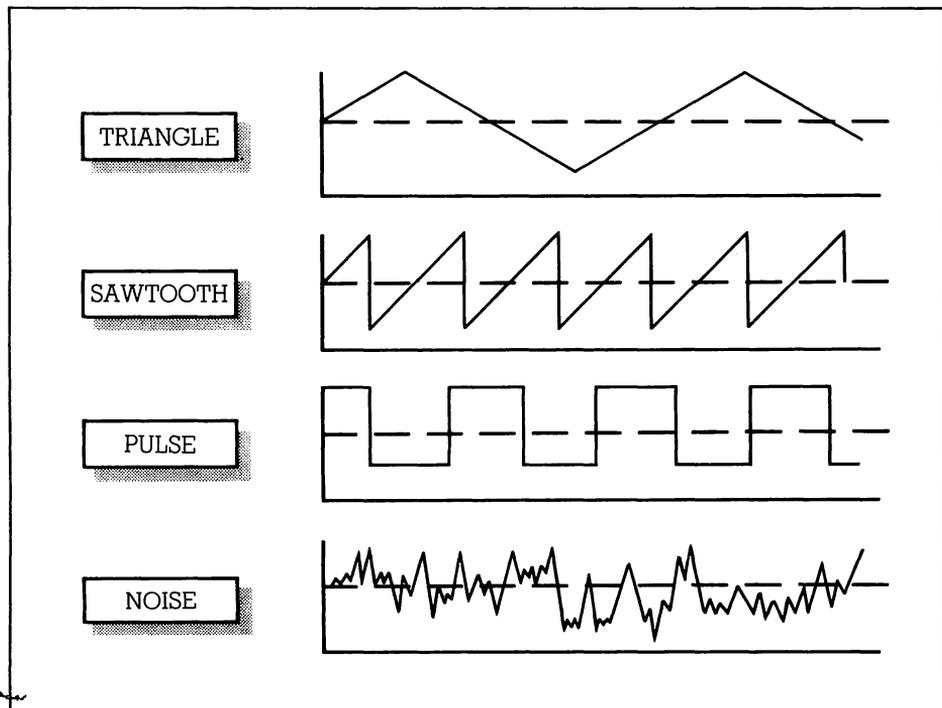
On first sight, programming sound looks to be a frightening task since there are so many registers to be set and so many possible values to choose from. However, it really isn't difficult. As with graphics, the process is simply

complicated by the fact that there are no Basic statements to allow you to control sound. We have to use POKE every time. This chapter will give you a simple step-by-step guide to producing sound, but first we need to learn some of the technical terms involved.

### Waveforms

The waveform of a sound describes the way it would look if you could see it. The 64 has four waveforms you can choose from: triangle, sawtooth, pulse and noise.

Triangle waveforms produce smooth, mellow sounds like flutes and oboes. Sawtooth is a much rougher



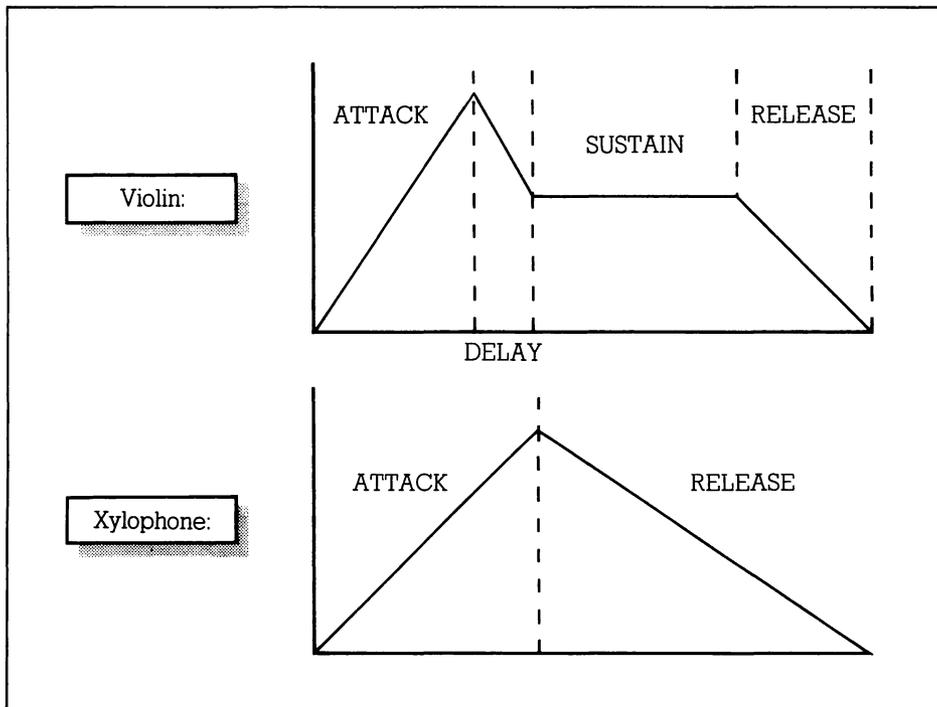
sound, brassy like trumpets or organs. Noise is a random waveform that can make sounds from winds to waves rushing on a beach to percussion and gunshots.

The fourth waveform, pulse, is a little different. If you select a pulse waveform you also have to set a value for the width of the pulse. This can range from 0 to 4096 and the middle value, 2048, will produce a square wave. Low values make very narrow waves and can be difficult to hear, high values are a much wider wave and generally produce better results. Pulse waves give a range of sounds that can be similar to both triangle and sawtooth or very different from either.

## Envelopes

The envelope of a sound describes the way it develops – how quickly it reaches maximum volume (attack), how it descends to the average volume (decay), how long it will hold the average (sustain), and how it fades to silence (release). These are the terms which give envelopes their common name: ADSR envelopes.

The 64 allows you to set all these things so that you can produce a range of sounds. For example, an organ is almost all sustain with a very short attack and decay and no release, a xylophone is all attack and release, and a guitar sound would be short attack and decay, medium sustain and quite a long release.



## Pitch

The pitch of a note governs whether it is high or low and pitch is organised in notes (A, C sharp, E flat etc) and octaves. Pitch on the 64 has a range of 0 to 65,536 which covers about eight octaves and some notes can be so high or low that you may not hear them. Because of the range of values the 64 needs two registers to hold the pitch (also called the frequency). The two values are obtained by dividing the number by 256 to get the high value, then subtracting the high value from the whole number to get the low value. Fortunately page 152 of the user manual that comes free with your 64 has these values already worked out for you.

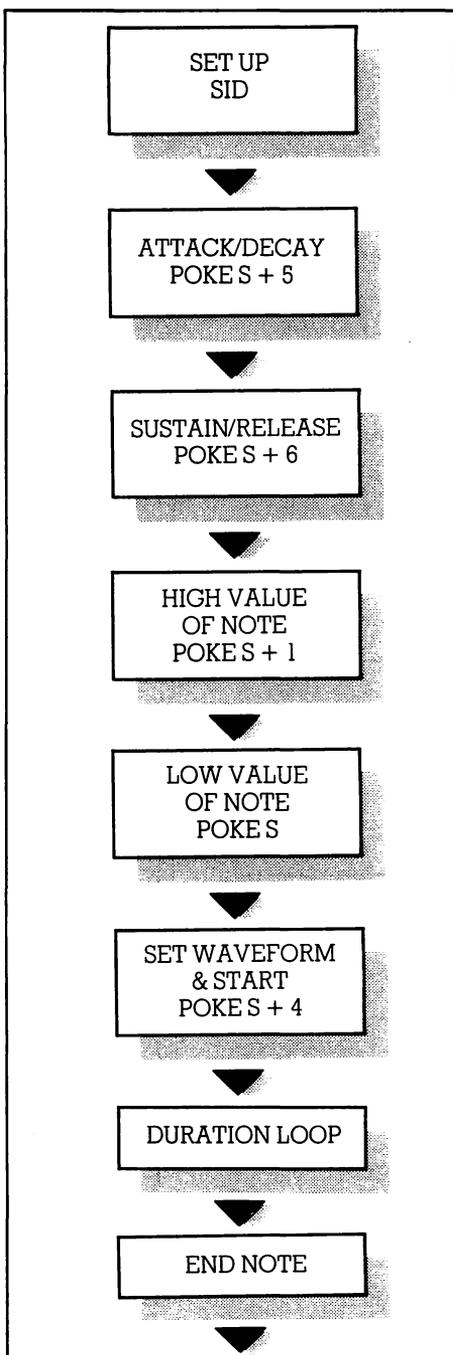
## Making sound

Now let's see how to make sounds on the 64. There are two separate stages:

- Set up the SID chip for use.
- Set up the registers for the sound.

Setting up the SID chip should be done as a subroutine and executed first in any program you write that uses sound. Enter this and save it for use later.

```
10000 S = 54272: REM START  
      OF SID  
10010 FOR I = 0 TO 24:  
      POKE S+I, 0  
10020 NEXT: REM CLEAR ALL  
      REGISTERS  
10030 POKE S+24, 15: REM  
      MAXIMUM VOLUME  
10040 RETURN
```



After this routine SID is ready for use. Now we can set things up to play a note.

## TRY THIS

There are a number of steps necessary here:

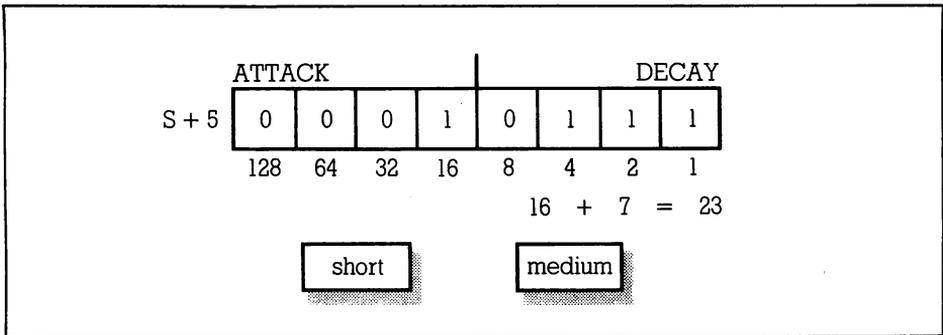
- 1 Set attack/decay rates.
- 2 Set sustain/release rates.
- 3 Set the high frequency of the note.
- 4 Set the low frequency.
- 5 Select the waveform.
- 6 Start the note.
- 7 Let the note sound using a delay loop
- 8 Turn off the sound.

```

10 GOSUB 10000
20 POKE S+5, 130: REM
   ATTACK/DECAY
30 POKE S+6, 72: REM
   SUSTAIN/RELEASE
40 POKE S+1, 8: REM HIGH
   FREQUENCY
50 POKE S, 147: REM LOW
   FREQUENCY
60 POKE S+4, 33: REM
   SAWTOOTH WAVEFORM AND
   START NOTE
70 FOR D = 1 TO 250: NEXT
80 POKE S+4, 32: REM TURN
   OFF SOUND
90 END
  
```

| Register | Description   | Register | Description   |
|----------|---|----------|---|
| 0        | voice 1, low frequency  | 18       | voice 3, control register for gate, ring mod, sync and waveform |
| 1        | voice 1, high frequency   | 19       | voice 3, attack/decay rate                                      |
| 2        | voice 1, low pulse rate   | 20       | voice 3, sustain/release rate                                   |
| 3        | voice 1, high pulse rate  | 21       | filter high frequency cut-off                                   |
| 4        | voice 1, control register for gate, ring mod, sync and waveform | 22       | filter low frequency cut-off                                    |
| 5        | voice 1, attack/decay rate                                      | 23       | filter control for voices plus resonance                        |
| 6        | voice 1, sustain/release rate                                   | 24       | volume control plus filter type                                 |
| 7        | voice 2, low frequency  | 25       | read value generated by game paddle x                           |
| 8        | voice 2, high frequency   | 26       | read value generated by game paddle y                           |
| 9        | voice 2, low pulse rate   | 27       | digitised output of voice 3 high frequency                      |
| 10       | voice 2, high pulse rate  | 28       | digitised output of voice 3 waveform                            |
| 11       | voice 2, control register for gate, ring mod, sync and waveform |          |   |
| 12       | voice 2, attack/decay rate                                      |          |   |
| 13       | voice 2, sustain/release rate                                   |          |   |
| 14       | voice 3, low frequency  |          |   |
| 15       | voice 3, high frequency   |          |   |
| 16       | voice 3, low pulse rate   |          |   |
| 17       | voice 3, high pulse rate  |          |   |

Note: registers 27 and 28 can be used for modulating the output of other voices under software control.



This seems a lot of steps just to play one note but remember that once they have been taken you can play music of any length and the only thing to change will be the note frequency.

The accompanying chart breaks down the SID registers in the same way we looked at the sprite registers in the VIC chip. Now let's look at some of them in detail.

Attack/Decay: register S+5 controls these rates for voice 1 depending on the values you enter:

|         |      |        |       |          |
|---------|------|--------|-------|----------|
|         | long | medium | short | shortest |
| attack: | 128  | 64     | 32    | 16       |
| decay:  | 8    | 4      | 2     | 1        |

As you might guess we are dealing with bit manipulation here and by combining these values (setting different bits) we can fine-tune the attack/decay rates. For example, we can get an extremely long attack rate by setting all of the high bits to get a value of 240 (128+64+32+16). If we put 240 into S+5 we would get the longest possible attack rate and no decay. To get decay we need to add in values between 1 and 15. 15 would give us the longest possible decay rate (8+4+2+1).

Try varying the values for S+5 in the program above and see the difference they can make.

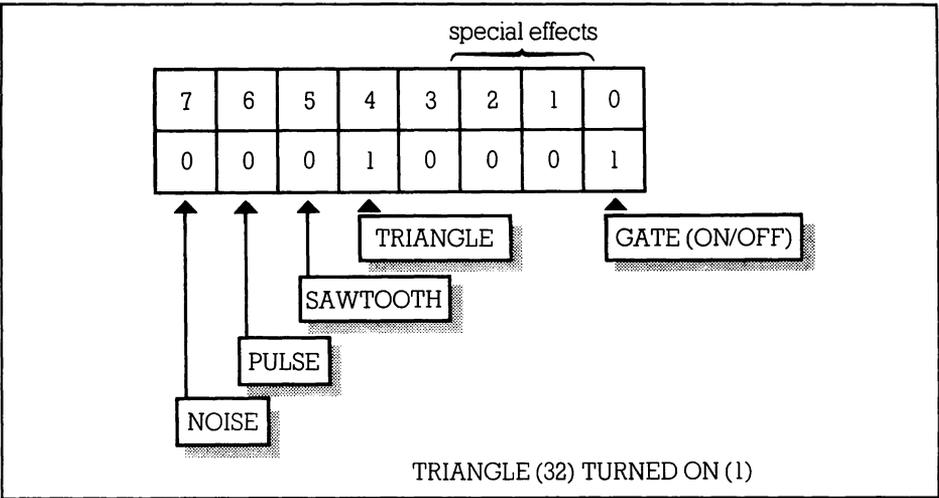
Sustain/Release: these registers (S+6 for voice 1) work in the same way as attack/decay with the four high bits controlling sustain and the low four bits controlling release, and again the longest sustain rate is given by the value 240 and the longest release by 15.

Try altering the values in S+6 in the program for different effects.

Waveform: register S+4 does a lot more than govern the waveform for voice 1. It is actually the control register for that voice. As usual different bits handle different functions. The waveforms are handled by bits 4 to 7 like this:

| Bit | Value | Waveform |
|-----|-------|----------|
| 4   | 16    | triangle |
| 5   | 32    | sawtooth |
| 6   | 64    | pulse    |
| 7   | 128   | noise    |

Bit 0 is called the gate bit and when set to 1 starts the sound (which is why we POKE S+4, 33 in the program). The envelope generator swings into action and begins the attack/decay/sustain part of the envelope. When this cycle is complete or the gate bit is set to



zero, the release part of the envelope begins. If the gate bit is set before that has finished the envelope cycle starts over immediately.

Bits 1 and 2 control two of the most advanced features of SID, ring modulation and sync effects. We'll look at those later.

Bit 3 is called the test bit and doesn't really do anything helpful.

## TRY THIS

To make it easier to try out the different combinations of ADSR and envelopes here's a program that does it for you – the SID Laboratory.

```

10 PRINTCHR$(147) :
   POKE53280,0:POKE53281,0
20 POKE646,7:GOSUB5000
29 REM *** SET UP SCREEN
   DISPLAY ***
30 SYSAT,11,1,"[rev] SID
   CHIP LABORATORY [off]"
40 SYSAT,8,3,"SET RATES
   USING 0 (LOWEST)"

```

```

50 SYSAT,15,4,"OR 15
   (HIGHEST)"
60 SYSAT,0,5," - - - - -
   - - - - -
   - - - - -
   - - - - -
   - - - - -"
70 SYSAT,1,6,A$
80 SYSAT,2,7,D$
90 SYSAT,0,8,S$
100 SYSAT,0,9,R$
110 SYSAT,0,10," - - - - -
   - - - - -
   - - - - -
   - - - - -
   - - - - -"
120 SYSAT,4,11,W$
130 SYSAT,15,11,"1 "W1$:
   SYSAT,27,11,"2 "W2$:
140 SYSAT,15,13,"3 "W3$:
   SYSAT,27,13,"4 "W4$:
150 SYSAT,1,15,"PULSE
   WIDTH:"
160 SYSAT,0,16," - - - - -
   - - - - -
   - - - - -
   - - - - -
   - - - - -"

```

This first section calls the set-up routines and creates the display. /

```

199 REM *** GET SOUND
VALUES ***
200 FORI=0TO3
210 SYSAT,15,6+I,"";
220 INPUT IN$
230 IN(I)=VAL(IN$):IF
IN(I)>15 OR IN(I)<0
THEN210
240 SYSAT,14,6+I,IN(I)
250 NEXT
260 SYSAT,4,11,"[rev]
W$"[off]"
270 GETIN$:W=VAL(IN$):
IFW<10RW>4THEN270
280 SYSAT,4,11,W$
290 IFW=1THENSYSAT,17,11,
"[rev]"W1$:WF=17
300 IFW=2THENSYSAT,29,11,
"[rev]"W2$:WF=33
310 IFW=4THENSYSAT,29,13,
"[rev]"W4$:WF=129
320 IFW<>3THEN370
330 SYSAT,17,13,"[rev]"W3$:
WF=65
340 SYSAT,1,15,"[rev]PULSE
WIDTH:[off] "
350 SYSAT,15,15,"";:
INPUTPW$
360 PW=VAL(PW$):IFPW>4096
OR PW<0 THEN350
370 SYSAT,1,15,"PULSE
WIDTH:"
380 SYSAT,7,23,"PRESS F1
TO PLAY NOTE";
390 GETIN$:IFIN$<>CHR$
(133)THEN380
400 GOSUB500

```

The main control section. This allows you to set the various rates for attack, decay, sustain and release and to choose a waveform. If you select pulse, it then asks for the pulse width. The pulse width is broken into high

and low bytes in the subroutine at 500.

```

415 SYSAT,7,23,"PRESS
RETURN TO CONTINUE";
420 SYSAT,7,24,"OR SPACE
TO REPEAT NOTE";
425 GETA$
430 IFA$=" "THENGOSUB500:
GOTO425
435 IFA$<>CHR$(13)THEN425
440 FORI=0TO3
445 SYSAT,15,6+I," ":REM 2
SPACES
450 NEXT
455 SYSAT,15,15,
" "":REM 7
SPACES
460 SYSAT,7,24,"
":REM 24
SPACES
465 SYSAT,7,23,"
":REM 24
SPACES
470 GOTO120

```

This short section offers you the choice of hearing the note again or of setting up a different sound. The space bar was chosen because it auto-repeats. In other words you can hold it down to hear the sound several times in rapid succession. You get a better idea of how well it works that way.

```

499 REM *** PLAY THE
SOUND ***
500 POKES+24,15
510 POKES+5,IN(0)*16+IN(1)
520 POKES+6,IN(2)*16+IN(3)
530 PH=INT(PW/256):PL=
PW-PH*256
540 POKES+3,PH:POKES+2,PL
550 POKES+1,34:POKES,75
560 POKES+4,WF
570 FORD=1TO500:NEXT
580 POKES+4,WF-1
590 POKES+1,0:POKES,0
600 RETURN

```

This is where the business is done. Line 500 sets the volume. Although it has already been set by the early subroutine, it is included here so that if you develop the program further you can control the volume as well as the other variables. 510 and 520 set attack (IN(0)), decay (IN(1)), and sustain (IN(2)), and release (IN(3)) rates. Notice how the high bits in the registers are set by multiplying the input value by 16.

```

4999 REM *** SYS AT
      ROUTINE ***
5000 FORI=0T038:READA:
      POKE49152+I,A:NEXT
5010 AT=49152
5020 DATA32,241,183,134,87,
      32,241
5030 DATA183,134,88,165,87,
      201,40
5040 DATA176,6,165,88,201,
      25,144
5050 DATA3,76,72,178,166,
      88,164

```

```

5060 DATA87,24,32,240,255,
      32,253
5070 DATA174,76,160,170
5099 REM *** INITIALISE
      VARIABLES ***
5100 S=54272:PW=0:DIMIN(3)
5110 A$="ATTACK RATE:"
5120 D$="DECAY RATE:"
5130 S$="SUSTAIN RATE:"
5140 R$="RELEASE RATE:"
5150 W$="WAVEFORM:"
5160 W1$="TRIANGLE":W2$=
      "SAWTOOTH"
5170 W3$="PULSE":W4$="NOISE"
5199 REM *** CLEAR SID
      CHIP ***
5200 FORI=0T024:POKES+I,0:
      NEXT
5210 POKES+24,15
5220 RETURN

```

The subroutines to set up the SYS AT command, program variables and initialise the SID chip.

## Advanced sound

If you experiment with the SID Laboratory program you will see how much flexibility SID gives you using only the basic abilities of the sound chip. Now let's look at some of the more advanced techniques.

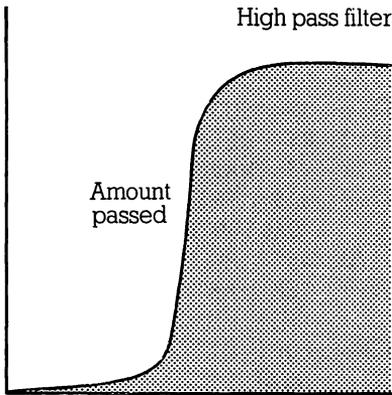
The first technique here is filtering. Filtering lets you select which parts of the frequency will be output and works in much the same way as the tone, or bass and treble controls work on a hi-fi system. There are three kinds of filters built into SID: high pass, low pass and bandpass. High pass, as the name suggests, filters out the low

frequencies and lets through the high ones. Low pass works the other way around, while bandpass cuts off high and low frequencies while passing those in the middle.

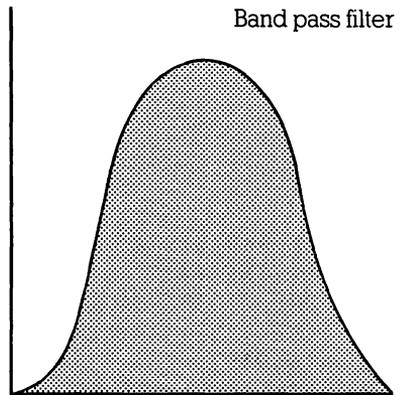
You can also combine high and low pass filters to create a fourth, the notch reject filter. This works in the opposite way to a bandpass, cutting out middle frequencies and passing high and low.

Filters are controlled by registers S+21, 22, 23 and 24. 21 and 22 combine to give the cut off frequency – the point at which the filter becomes active.

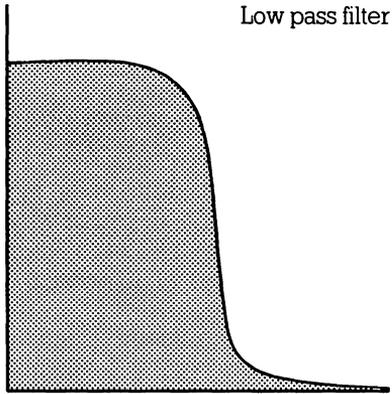
S+23 is another of the 64's multi-



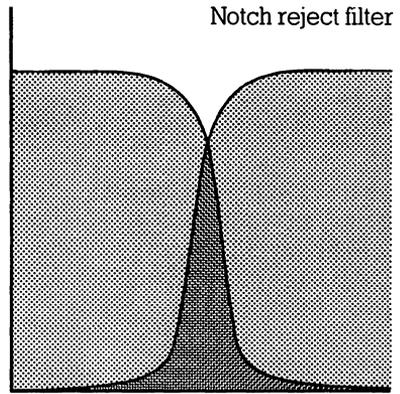
Low Frequency High



Low Frequency High



Low Frequency High



Low Frequency High

switch registers. Bits 0 to 2 turn on filtering for voices 1 to 3 respectively. Bits 4 to 6 control the resonance of the filters. Resonance causes a sharper sound to be produced and playing around with this can create some interesting sounds such as wah-wah effects.

S+24, as well as being the volume control, allows you to choose the filter types. Bit 4 sets low pass and produces richer sounds. Bit 5 selects bandpass which makes the sound thinner and bit 6 is high pass producing 'tinny' sounds.

Bit 7 disconnects voice 3 from the output and is used in modulation effects to prevent voice 3 from producing odd, unwanted noises.

### Demonstration

The following program is a very simple demonstration of just two of the many effects you can get using filters. It allows you to hear one note unfiltered or affected by a high or low pass filter.

```

10 GOSUB 10000
20 POKES+1,40:POKES,50
30 POKES+5,17:POKES+6,36
40 PRINTCHR$(147)
50 PRINT"SELECT FILTER
   TYPE: ";
60 PRINT"1 HIGH PASS"
70 PRINTTAB(20)"2 LOW
   PASS"
80 PRINTTAB(20)"3 NO
   FILTER"
90 GETAS:IFAS<>"1"ANDAS
   <>"2"ANDAS<>"3"THEN90
100 F=VAL(A$):PRINT:
   PRINTF
110 POKES+24,F(F)+15

```

```

112 POKES+23,1:IFF=3THEN
   POKES+23,0
115 POKES+21,18:POKES+22,
   128
120 POKES+4,33
130 PRINT:PRINT"PRESS
   SPACE TO CONTINUE OR
   RETURN TO END"
140 GETAS:IFAS="" THEN140
150 IFAS="" THENPOKES+4,
   32:GOTO40
160 POKES+4,32:END
10000 S=54272
10010 FORI=0TO24:POKES+I,0
10020 NEXT
10030 POKES+24,15
10040 F(1)=64:F(2)=16:
   F(3)=0
10050 RETURN

```

The SID filters are possibly its most important feature since they give you so much control over the noise output.

Ring modulation and synchronisation are similar in that both let you affect the output of a voice with the output of the lower voice. For example, you can sync voice 1 with voice 3, or voice 2 with voice 1. By setting bit 1 of the control register for a voice, the frequency of the voice is synchronised with the frequency of the lower voice. Obviously the lower voice must be set to some frequency, preferably lower than the higher voice.

Ring modulation is controlled by bit 2 of the control register and the lower voice must be set to triangular waveform at some frequency other than zero.

## Demonstration

The following program demonstrates these two effects by affecting voice 1 with voice 3.

```
10 GOSUB 10000
20 PRINTCHR$(147)
30 PRINT:PRINT"THIS IS
THE PURE SOUND"
40 FORD=1T01000:NEXT:
GOSUB 200
50 PRINT:PRINT"THIS IS
WITH SYNC"
60 FORD=1T01000:NEXT:
GOSUB 300
70 PRINT:PRINT"THIS IS
RING MOD"
80 FORD=1T01000:NEXT:
GOSUB 400
90 PRINT"PRESS RETURN TO
PLAY IT AGAIN"
100 GETA$:IFAS<>CHR$(13)
THEN100
110 GOT010
200 POKES+1,130:POKES,100
210 POKES+5,8:POKES+6,72
220 POKES+4,33
230 FORD=1T0500:NEXT
240 POKES+4,32
250 RETURN
300 POKES+15,6:POKES+14,
100
310 POKES+19,72:POKES+20,
72
320 POKES+18,33:POKES+4,
35
330 FORD=1T0500:NEXT
340 POKES+18,0:POKES+4,0
350 RETURN
400 POKES+15,6:POKES+14,
100
410 POKES+19,72:POKES+20,
72
420 POKES+4,21
```

```
430 FORD=1T0500:NEXT
440 POKES+18,0:POKES+4,0
450 RETURN
10000 S=54272
10010 FORI=0T024:POKES+I,0
10020 NEXT
10030 POKES+24,15
10040 RETURN
```

## Making music

Now that you know how to control the various registers in SID, what about putting them to use? Music is relatively easy to program since once the registers are set, it is simply a matter of playing notes in sequence.

## TRY THIS

The common way of producing music is to hold the note values in DATA statements and READ them, like this:

```
10 GOSUB 10000
20 POKES+5,17
30 POKES+6,65
40 READH,L,D
50 IFH=-1THEN200
60 POKES+1,H:POKES,L
70 POKES+4,33
80 FORI=1TOD*250:NEXT
90 POKES+4,32
100 GOT040
200 END
300 DATA7,53,2,7,53,1,10,
205,1,10,205,2
310 DATA8,23,1,8,147,1,8,
23,1
320 DATA7,53,5
330 DATA10,205,1,12,216,
1,14,107,2
```

```

340 DATA12,216,1,10,205,
    1,12,32,1,9,159,1
350 DATA10,205,5,-1,-1,-1
10000 S=54272
10010 FORI=0T024:POKES+I,0
10020 NEXT
10030 POKES+24,15
10040 RETURN

```

S and S+1 and the note is played for D\*250 before the gate bit is unset and the next values are read. Obviously the burden is working out the DATA statements.

No book – least of all this one – can teach you music but if you are already familiar with simple musical theory you should be able to work out simple tunes. A variety of time signatures can be played by varying the delays, and harmony, chords and bass lines can all be incorporated using all three voices. Simply add the extra DATA for each voice to play.

If you don't know music, there are three courses of action open: learn it, get a musical friend to work out the DATA, or leave music alone and stick to sound effects.

### How it works

This simple arrangement demonstrates all that's necessary to produce music on the 64. The DATA statements break down into groups of three: high frequency, low frequency and time delay. The frequency values for the notes are read and POKEd into

MINIM 2 CROTCHETS 4 QUAVERS



8 SEMI-QUAVERS



DOTTED CROTCHET



DOTTED QUAVER



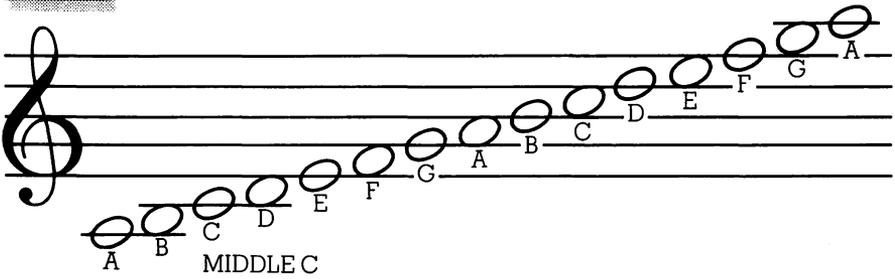
SET TIME VALUES:

|         |   |             |
|---------|---|-------------|
| MIN     | = | 2000        |
| CROT    | = | MIN/2       |
| SEMI    | = | CROT/2      |
| QUAV    | = | SEMI /2     |
| DOTCROT | = | CROT + QUAV |
| DQUAV   | = | QUAV + SEMI |

To change timing now, you only need to reset the MIN value.

**PITCH**

2 octaves in the key of C



|   | HIGH | LOW |
|---|------|-----|
| A | 14   | 107 |
| B | 16   | 47  |
| 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  |
| F | 45   | 198 |
| G | 51   | 97  |
| A | 57   | 172 |

## Sound effects

Happily in this department no prior training is necessary since, if you choose your games carefully, you can make up the rules. After all who's to say that Epsilon space fighters don't sound like this on take-off.

```

10 GOSUB 10000
20 POKES+5,16:POKES+19,
  16
30 POKES+6,190:POKES+20,
  190
40 FORT=5T025STEP.5

```

```

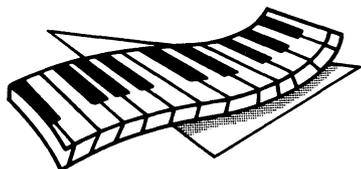
50 POKES+4,33:POKES+18,
  129
60 POKES+1,9+T:POKES+
  15,T
70 POKES,150:POKES+14,50
80 POKES+4,0:POKES+18,0
90 NEXT
99 END
10000 S=54272
10010 FORI=0T024:POKES+I,0
10020 NEXT
10030 POKES+24,15
10040 RETURN

```

## Take it from here

As you can see, the SID chip is capable of producing a near infinite variety of sounds. As was pointed out at the start of this chapter, there are even sounds that have never been heard before. Now that you know which registers control which effects it's all up to you. Experimentation is the

key to sound on the 64 and the programs here could form the basis of many others to let you explore the SID chip in full. However, so that you can start putting sound into your programs straightaway, here is the beginning of a 'sound library' which gives the settings for a few useful sounds.

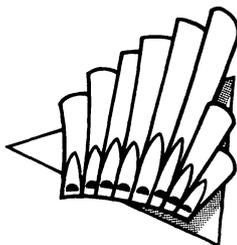


Piano:

```

POKE S+5,9: REM ATTACK/
DECAY
POKE S+6,16: REM SUSTAIN/
RELEASE
POKE S+3,100: REM LOW PULSE
POKE S+2,1: REM HIGH PULSE
POKE S+4,65: REM PULSE
WAVEFORM

```

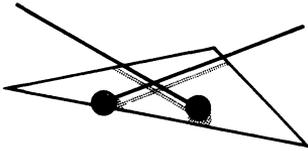


Organ:

```

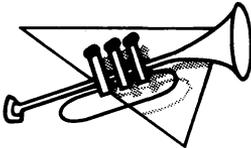
POKE S+5,0: REM ATTACK/
DECAY
POKE S+6,242: REM SUSTAIN/
RELEASE
POKE S+4,33: REM SAWTOOTH
WAVEFORM

```



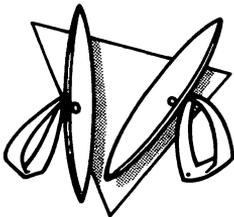
Xylophone:

POKE S+5,9: REM ATTACK/  
DECAY  
POKE S+6,0: REM SUSTAIN/  
RELEASE  
POKE S+4,17: REM TRIANGLE  
WAVEFORM



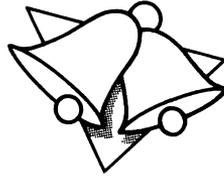
Trumpet:

POKE S+5,96: REM ATTACK/  
DECAY  
POKE S+1,16: REM SUSTAIN/  
RELEASE  
POKE S+4,33: REM SAWTOOTH  
WAVEFORM



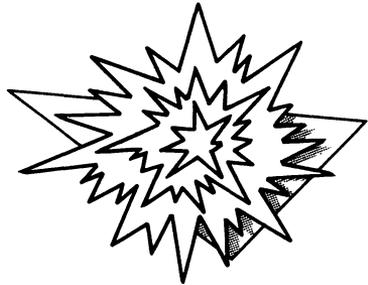
Cymbal:

POKE S+5,9: REM ATTACK/  
DECAY  
POKE S+6,0: REM SUSTAIN/  
RELEASE  
POKE S+4,129: REM NOISE  
WAVEFORM



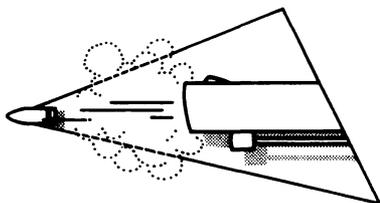
Bell:

POKE S+0,0: REM LOW  
FREQUENCY  
POKE S+1,125: REM HIGH  
FREQUENCY  
POKE S+5,9: REM ATTACK/  
DECAY  
POKE S+6,9: REM SUSTAIN/  
RELEASE  
POKE S+15,30: REM VOICE 3  
HIGH FREQUENCY  
POKE S+4,21: REM TRIANGLE  
WAVEFORM WITH RING  
MODULATION



Explosion:

POKE S+0,180: REM LOW  
FREQUENCY  
POKE S+1,2: REM HIGH  
FREQUENCY  
POKE S+5,9: REM ATTACK/  
DECAY  
POKE S+6,224: REM SUSTAIN/  
RELEASE  
POKE S+4,129: REM NOISE  
WAVEFORM  
POKE S+24,5-15: REM VARY  
VOLUME



Gunshot:

POKE S+0,200: REM LOW  
 FREQUENCY  
 POKE S+1,40: REM HIGH  
 FREQUENCY  
 POKE S+5,15: REM ATTACK/  
 DECAY  
 POKE S+6,2: REM SUSTAIN/  
 RELEASE  
 POKE S+4,129: REM NOISE  
 WAVEFORM  
 POKE S+24,15-1: REM  
 DECREASE VOLUME

Keep a notebook handy as you try new combinations of SID settings and jot down the interesting values. And remember, all of the above are basic sounds: try adding filters and other effects.

## Checklist

In this chapter you've learned:

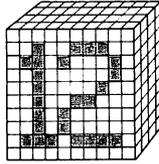
- The functions of the different SID chip registers.
- What waveforms are and how to program them.
- What envelopes are and how to program them.
- How to create a simple sound.
- How to define a piece of music to be played by the 64.

- How to use the filters in the SID chip.
- A little of the advanced features of SID including ring modulation and synchronisation.

## Projects

- Write a program to play a single note then add these stages:
  - 1 Filter the note to the best effect.
  - 2 Vary the frequency as the note is played.
  - 3 Vary the resonance as the note is played.
  - 4 Vary the volume as the note is played.
  - 5 Add a second voice in harmony (try an octave lower – halve the frequency – or an octave higher – double the frequency).
- Add a subroutine to the SID Laboratory program to allow you to try different filter settings.
- Write a 'synthesiser' program to let you play the 64 as a musical instrument. These are the steps you might take:
  - 1 Hold the note values in an array.
  - 2 Use the bottom row of the keyboard (Z to /) for whole notes and the second row (A to =) for sharps and flats. Look up the note values in the User Manual.
  - 3 Use PEEK (197) to get keypresses and use the values generated as pointers to the note array.
  - 4 Set up the SID chip using a subroutine for envelope and waveform so that only the note pitch will change.

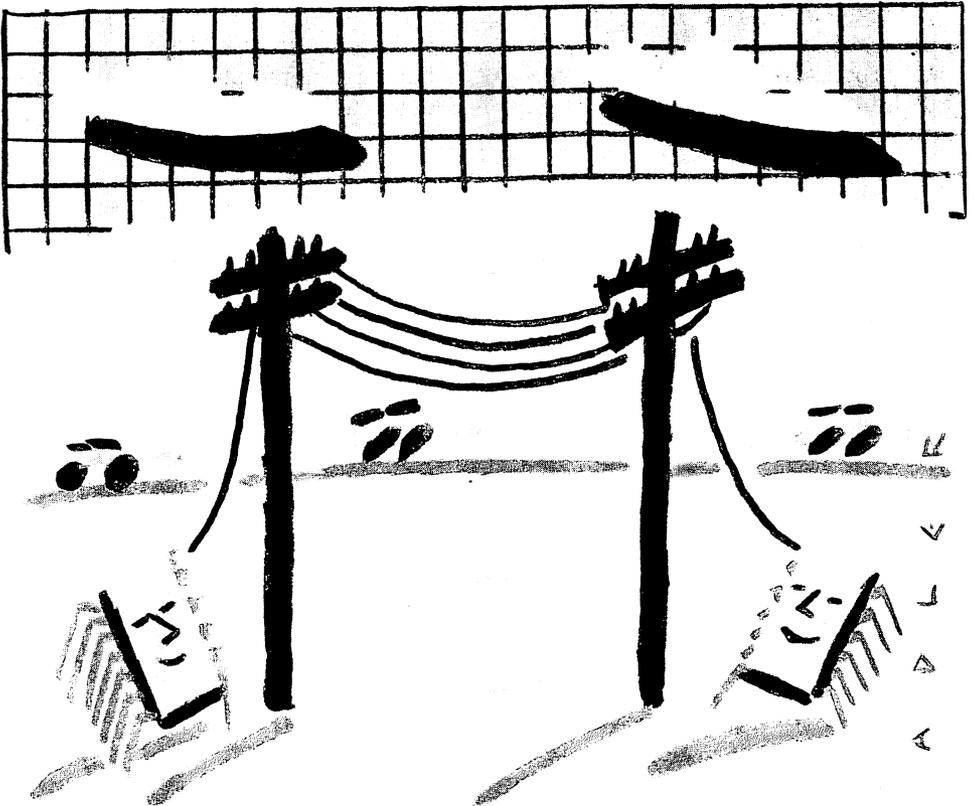
- 
- 5 When this works add more routines to allow different waveforms and envelopes. These could be selected by the function keys (again use PEEK(197).
  - 6 Add more of the features you want like wah-wah effects, or single-key chords.
-



---

# Interfacing

---



---

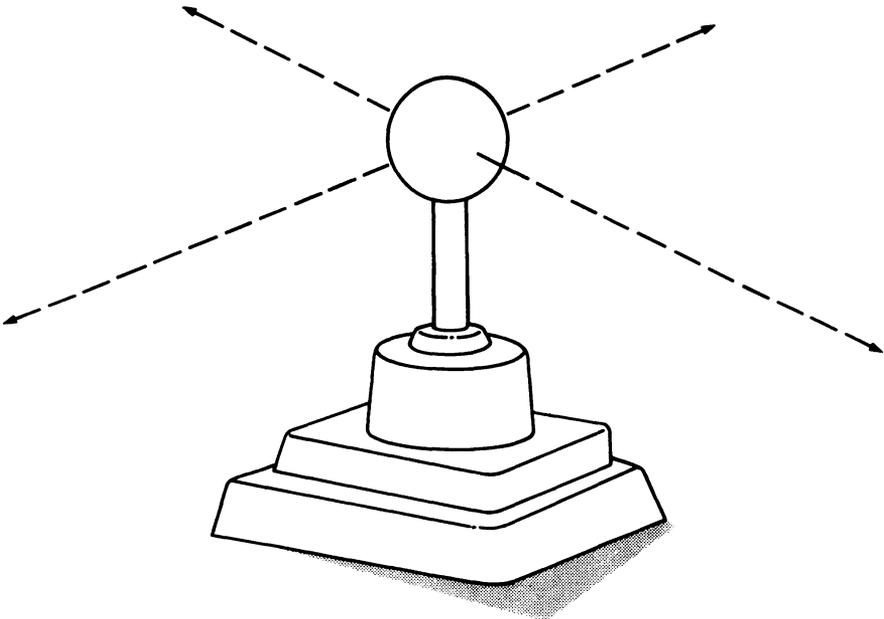
With the exception of joysticks all input/output operations on the 64 depend on the concept of files, device numbers and addresses. Don't worry if the technical terms sound off-putting – I/O is really quite simple and the jargon easy to understand. Joysticks are different, partly because they are input only, but largely because they don't use the file system. Instead we're back with our old friend PEEK so we'll deal with them first.

### Joysticks

The 64 has two joystick ports at the right of the machine marked port 1 and port 2. Perhaps surprisingly, most games use port 2 for joystick control but there is a good reason for this. All input/output is handled by two

dedicated I/O chips one of which deals with port 2 and things like serial devices (printers, disk drives, etc) while the other looks after port 1, the keyboard and a couple of other things. It is the fact that the second chip copes with port 1 and the keyboard that causes problems. If you have a joystick try this simple experiment: plug it into port 1, turn on the 64 then move the joystick around rapidly. You should see random characters appearing on screen. This is because the operating system isn't sure whether the signals are coming from the keyboard or the joystick port. Under program control the problem is worse and if the joystick is moved at the same time as a key is pressed the 64 will often crash, forcing you to switch off.

So as a general rule, always use port 2 for joysticks.



## TRY THIS

Actually reading the values generated by the joystick is fairly simple if you have understood the section on bit manipulation in chapter 5. Plug in a joystick to port 2 and enter and run this short program:

```
10 JY = PEEK(56320)
20 PRINT CHR$(147);
  ABS(JY-127)
30 GOTO 10
```

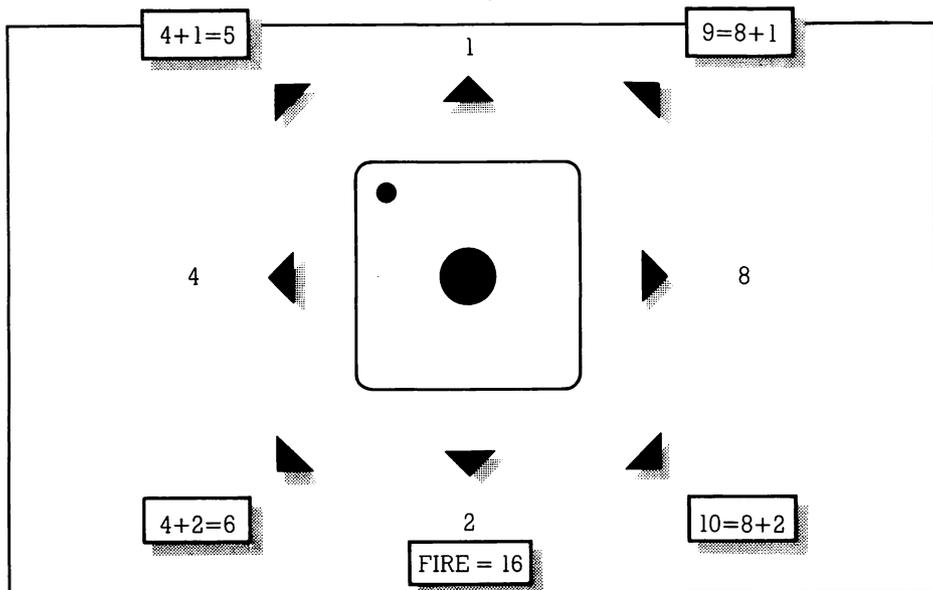
If everything is working properly a zero will appear at the top left of the screen. Move the joystick and watch the number change. Now press the fire button. Finally, hold down the fire button and move the joystick.

## How it works

The joystick is simply a collection of five switches. When the fire button is pressed or the joystick is moved, one or more of the switches is closed and the value appears in location 56320. Diagonal movement is sensed as two switches closing and their values are added together. It all decodes like this:

| JY value | Direction  |
|----------|------------|
| 0        | none       |
| 1        | up         |
| 2        | down       |
| 4        | left       |
| 8        | right      |
| 5        | up/left    |
| 6        | down/left  |
| 9        | up/right   |
| 10       | down/right |
| 16       | fire       |

Values greater than 16 give direction plus fire. Subtract 16 for direction.



## Further steps

Knowing this, joystick control becomes very simple. You could either have a series of IF...THEN statements to work out the correct action or use an array to hold the values. Here's a simple program using arrays to draw a trail of asterisks around the screen under joystick control.

```

10 DIM M(10): JS = 56320
20 GOSUB 1000
30 SC = 1024: CO = 54272
40 L = 1523
50 POKE L, 42: POKE L +
  CO, 1
60 JY = ABS(PEEK(JS) -
  127): IF JY > 10
  THEN 50
70 L = L + M(JY)
80 IF L < SC OR L > SC
  + 1023 THEN L = L -
  M(JY)
90 GOTO 50
1000 PRINT CHR$(147)
1010 M(0) = 0: M(1) = -40
1020 M(2) = 40: M(4) = -1
1030 M(5) = -41: M(6) = 39
1040 M(8) = 1: M(9) = -39
1050 M(10) = 41
1060 RETURN

```

The array M() holds the directions generated by the joystick which gives a simple means of updating screen location L. Line 80 is a check to make sure the asterisk stays on-screen although you will be able to wrap-around from one line to the next.

Using the principles of this program and some of the ideas from chapter 7 we can now create a high resolution sketchpad. This one, however, offers some time-saving machine code routines. Machine code is beyond the scope of this book but suffice it to say

that these routines do exactly what the Basic routines in chapter 7 do but several times faster.

```

10 PRINTCHR$(147):GOSUB
  4000
20 HI=49152:CL=49180:
  CO=49210
30 SC=8192:JS=56320
40 SYSHI:SYSCL:POKE255,
  100:SYSCO
50 X=160:Y=100
60 JY=ABS(PEEK(JS)-127)
70 X=X+M(JY,1):Y=Y+M
  (JY,0)
80 GOSUB1000
90 GOTO60
100 END
999 REM *** PLOT X AND Y
  ***
1000 CH=INT(X/8)
1010 RO=INT(Y/8)
1020 LN=YAND7
1030 BY=SC+RO*320+8*CH+LN
1040 BI=7-(XAND7)
1050 POKEBY,PEEK(BY)OR
  (2↑BI)
1060 RETURN
3999 REM *** SET UP
  MACHINE CODE ***
4000 FORI=0TO98:READA:
  POKE49152+I,A:
4010 NEXT:SYS49152
4019 REM *** SET UP
  JOYSTICK VALUES ***
4020 DIM M(10,1)
4030 FORI=1TO10:READM(I,0):
  READM(I,1)
4040 NEXT
4050 RETURN
5000 REM *** M/C DATA ***
5010 DATA169,32,133,52,133,
  56,169,147

```

|              |              |               |
|--------------|--------------|---------------|
| L-41<br>M(5) | L-40<br>M(1) | L-39<br>M(9)  |
| L-1<br>M(4)  | L            | L+1<br>M(8)   |
| L+39<br>M(6) | L+40<br>M(2) | L+41<br>M(10) |

```

5020 DATA32,210,255,173,24,
      208,9,8,141
5030 DATA24,208,173,17,208,
      9,32,141,17
5040 DATA208,96,169,0,133,
      251,169,32
5050 DATA133,252,166,252,
      160,0,169,0
5060 DATA145,251,200,208,
      251,232,224,64
5070 DATA240,5,134,252,76,
      36,192,96,169
5080 DATA0,133,253,169,4,
      133,254,165
5090 DATA255,166,254,160,
      0,145,253,200
5100 DATA208,251,232,224,
      7,240,5,134
5110 DATA254,76,70,192,
      160,0,230,254
5120 DATA145,253,200,192,
      233,208,249,96
5200 REM *** JOYSTICK DATA
      ***

```

```

5210 DATA -1,0,1,0,0,0,0,
      -1,-1,-1
5220 DATA 1,1,0,0,0,1,-1,
      1,-1,1

```

There are several points to note here. The machine code is set up so that you can call the routines by variable names: HI lowers the top of memory and creates the bit-map screen; CL clears the screen; and CO sets the colours. The colour value is POKEd into location 255 in line 40. By consulting the colour chart in chapter 7 you can change the colours by putting the new value into line 40.

We also need a two-dimensional array for M() this time because we need X and Y values, Y in M(JY,0) and X in M(JY,1).

See the Checklist at the end of this chapter for ideas on extending this program.

## I/O programs

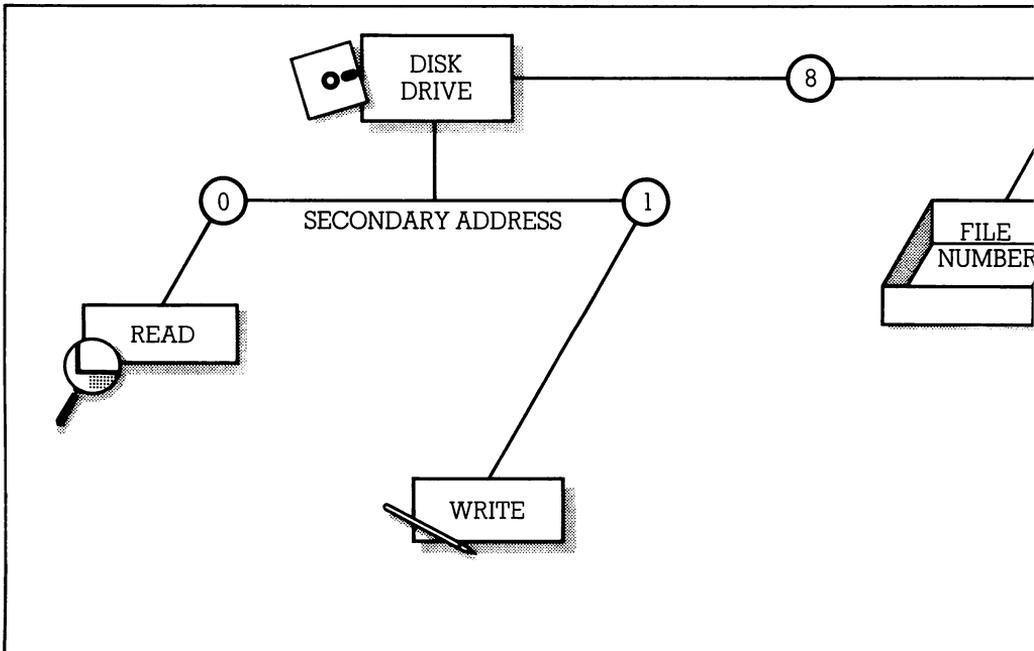
Now let's take a look at other input/output programming, specifically the cassette recorder, disk drive and printers.

At the start of this chapter we mentioned files, device numbers and addresses. On some computers there are special commands for communicating with peripheral devices such as LPRINT and LLIST for printers. The 64 Basic does not have these. Instead it treats all external devices as files although you have to tell it which device is being used by declaring the device number. These are the device numbers:

| Device   | Number |
|----------|--------|
| Cassette | 1      |
| Modem    | 2      |
| Screen   | 3      |
| Printer  | 4/5    |
| Plotter  | 6      |
| Disk     | 8-11   |

There are further rules about device numbers which depend on precisely what kind of operation is to be performed but we'll discuss those as they arise.

In connection with I/O, addresses have a different meaning from normal. They do not refer to memory locations but tell the 64 what kind of operation is involved. They are called secondary addresses (although there are no primary addresses - confusing isn't it?).



## Commands

There are only half a dozen Basic commands and statements associated with I/O and you will be familiar with some of them already. The first of them is OPEN which works like this:

```
10 OPEN 2,1
```

The first number is the file number. The 64, as already noted, handles I/O in terms of files. You can have several files open at once but each must have a unique number otherwise the 64 doesn't know which you want.

The second number is the device number, in this case the cassette recorder. It is at this point that secondary addresses enter the picture. There are several default addresses (which means the computer assumes you want them unless you say

otherwise). With the cassette recorder the default is 0 which means a read operation. So our line above could look like this:

```
10 OPEN 2,1,0
```

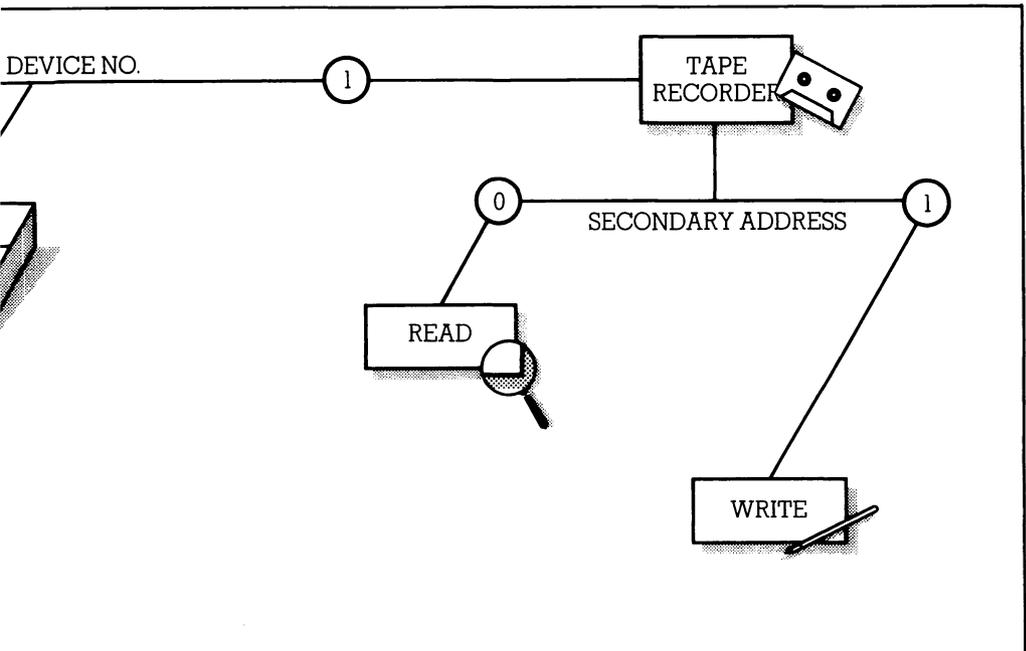
which means the same thing, or:

```
10 OPEN 2,1,1
```

which means a write operation. Read simply means that the 64 is to get information from the device. Write means it will send information to the device.

We can also give our files a name as well as a number:

```
10 OPEN 2,1,1,"ADDRESS BOOK"
```



## Caution

Note that some secondary addresses have specific meanings. In addition to the 0 and 1 associated with the cassette, here are some more:

OPEN 1,1,2 writes an end-of-file mark to the cassette.

OPEN 15,8,15 opens the command channel on the disk drive.

OPEN 1,4,0 opens a printer file in upper case and graphics.

OPEN 1,4,7 opens a printer file in lower/upper case.

On the whole the disk drive works in much the same way as the cassette unit but in some cases has a longer syntax. See the disk drive user's manual for details.

## Further steps

Having opened the file, how do we use it? There are three commands: GET# and INPUT# for reading data, and PRINT# for writing data. These are used in conjunction with the file number specified when the file was opened so to write a string to the cassette we could use this:

```
10 OPEN 1,1,1
20 PRINT# 1,"THIS IS A
   TEST"
30 PRINT# 1,CHR$(13)
40 CLOSE 1
```

Another I/O command sneaked in there: CLOSE. Logically it does exactly the opposite of OPEN and requires only the file number – the 64 will know what the device number is. Also note that items must be separated by a carriage return (CHR\$(13)).

To get our string back again (having first rewound the tape) we open the file, and read the strings:

```
10 OPEN 1,1,0
20 INPUT# 1,A$
30 CLOSE 1
40 PRINTA$
```

GET# and INPUT# work in much the same way as GET and INPUT from the keyboard: GET# will retrieve single characters. INPUT# requires a carriage return at the end.

## Caution

An important point to note is that commas and semicolons have odd effects on tape and disk files, formatting the data in the way they affect screen formats. In other words if you print a comma to a file it will insert 10 spaces in the data. To get round this you must either eliminate them from your files altogether or do a string search and replace them with some other character, reversing the process when they are read back from tape.

The last I/O statement is CMD. This commands the 64 to send all output to the file specified. It is most often used to list a program to a printer:

```
OPEN 4,4:CMD4:LIST
```

followed by:

```
PRINT# 4:CLOSE4
```

## How it works

The most common form of storage is in saving and loading programs. This is identical for tape and disk except that

loading and saving to disk must specify device number 8:

**TRY THIS**

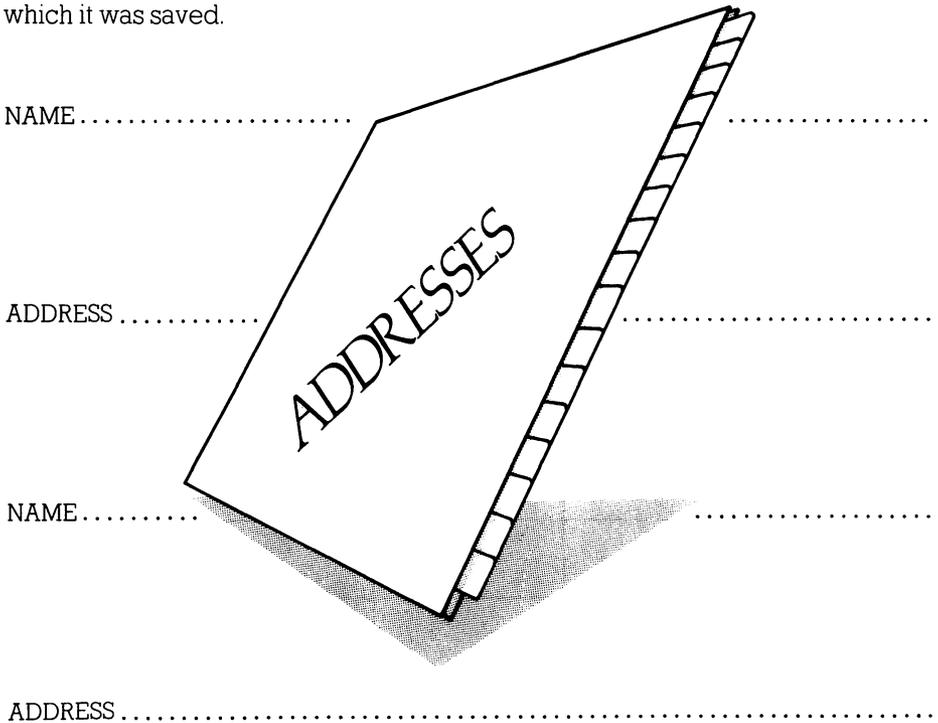
```
LOAD "PROGRAM",8: REM FROM DISK
LOAD "PROGRAM" : REM FROM TAPE
```

The most used 'extra' with loading and saving is in this form:

```
LOAD "PROGRAM",1,1: REM TAPE
LOAD "PROGRAM",8,1: REM DISK
```

This tells the 64 to load the program into the same part of memory from which it was saved.

Loading and saving information is not so simple. However, provided you approach things logically, it isn't very difficult. Let's look at how to create a simple file which can be stored on tape or disk and read back again. In this case it will be a very short address book holding three names and telephone numbers.



```

10 PRINTCHR$(147)
20 PRINT:PRINTTAB(10)
   "1 CREATE A FILE"
30 PRINT:PRINTTAB(10)
   "2 SAVE A FILE"
40 PRINT:PRINTTAB(10)
   "3 LOAD A FILE"
50 PRINT:PRINTTAB(10)
   "4 EXIT"
60 GETIN$:IFIN$="" THEN60
70 ONVAL(IN$)GOSUB100,200,
   300,450
80 GOTO10
100 DIMA$(3,2):PRINTCHR$
   (147)
110 FORI=1TO3
120 INPUT"NAME ";A$(I,0)
130 PRINT:INPUT"TOWN
   ";A$(I,1)

```

```

140 PRINT:INPUT"PHONE
   NUMBER ";A$(I,2)
150 NEXT
160 RETURN
200 PRINTCHR$(147)
210 PRINT:INPUT"TAPE OR
   DISK (T/D)";D$
220 IFD$="T"THENOPEN1,1,1,
   "ADDRESS BOOK"
230 IFD$="D"THENOPEN1,8,1,
   "ADDRESS BOOK,S,W"
240 FORI=1TO3:FORJ=0TO2
250 PRINT# 1,A$(I,J)
260 NEXT:NEXT
270 CLOSE1
280 RETURN
300 PRINTCHR$(147)
310 PRINT:INPUT"TAPE OR
   DISK (T/D)";D$

```

A\$(I,J)

|          | A\$(.,0) | A\$(.,1) | A\$(.,2) |
|----------|----------|----------|----------|
| A\$(1,.) | NAME     | TOWN     | TEL      |
| A\$(2,.) |          |          |          |
| A\$(3,.) |          |          |          |

```

320 IFDS="T"THENOPEN1,1,0,
"ADDRESS BOOK"
330 IFDS="D"THENOPEN1,8,0
"ADDRESS BOOK,S,R"
340 FORI=1TO3:FORJ=0TO2
350 INPUT# 1,A$(I,J)
360 NEXT:NEXT
370 CLOSE1
380 FORI=1TO3:FORJ=0TO2
390 PRINTA$(I,J),
400 NEXT:PRINT:NEXT
410 PRINT:PRINT"PRESS
RETURN"
420 GETINS$:IFINS$<>CHR$(13)
THEN420
430 RETURN
450 PRINTCHR$(147):END

```

The main thing to note here is that the information is written to the tape or disk in a straightforward manner and it must be retrieved in exactly the same order otherwise problems will result.

Although this looks like a long way of saving three names and telephone numbers, it wouldn't be very much longer to hold a hundred or more. Simply extend the loop and the size of A\$( ) and add some kind of routine to handle the printing out of names and numbers.

This kind of file is known as sequential because the information is stored in the order in which it is written, and must be loaded and searched in the same way. This is the only kind of file that can be used with tape storage. Disks can use more powerful files called relative and random access files but these are much more complicated to write. There are good examples in the disk drive user's manual.

---

## Checklist

---

In this chapter you've learned:

- How to read the joystick port for better program control.

---

- How to open files to cassette and disk.

---

- How to write information to tape or disk.

---

- How to recover information from tape or disk files.

---

- How to use other I/O commands and their syntax.

---

---

## Projects

---

- The sketchpad program leaves a lot to be desired.

Add some of the following features:

- 1** A sprite cursor so that you can see where the drawing position is. Try the pointing hand, or a pen.
- 2** Allow the rubbing out of points. What about using the fire button as a toggle switch between draw and erase? Have the sprite cursor change colour to signify which is in effect.
- 3** Add a routine to allow the user to choose the colours on screen and change the drawing colour at will.
- 4** Although it would be very slow, try a routine to save the design to tape or disk. Use a loop to PEEK the screen location then PRINT the values to a file.

- Extend the address book program or incorporate a similar routine into a simple database as suggested in chapter 4.
-



---

# Appendix: design aids

---

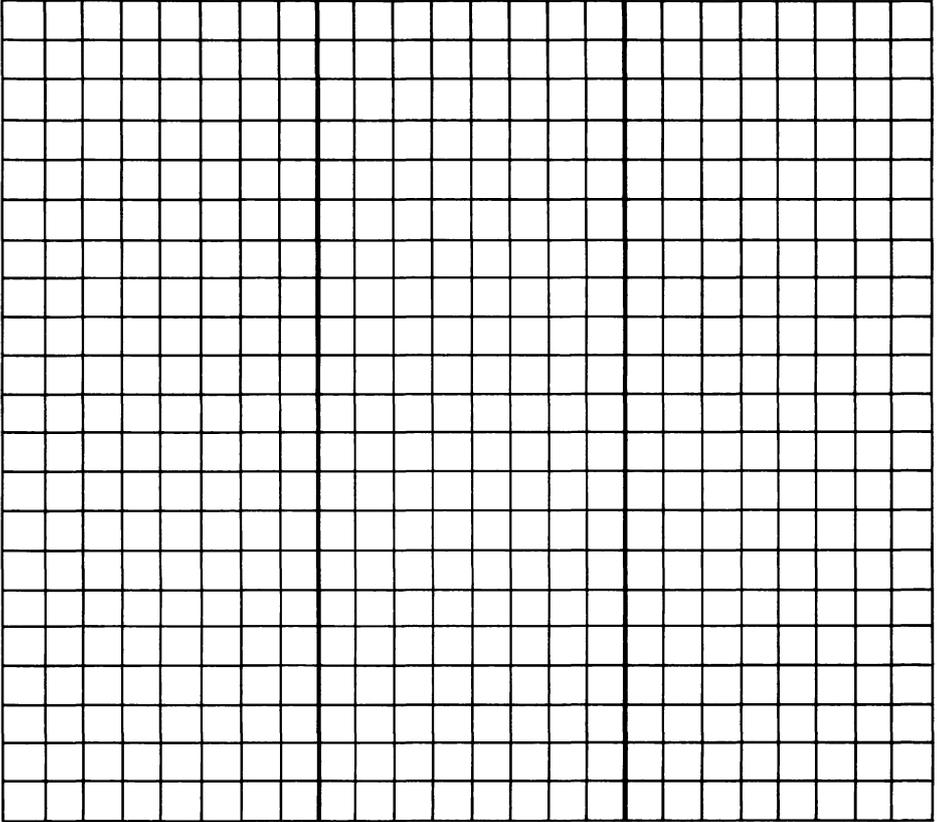
---

## Character design grid

| Bit value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|-----------|-----|----|----|----|---|---|---|---|
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |
|           |     |    |    |    |   |   |   |   |

# Sprite design grid

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



## Screen design grid: low resolution

|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| 1024 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1064 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1104 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1144 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1184 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1224 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1264 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1304 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1344 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1384 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1424 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1464 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1504 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1544 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1584 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1624 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1664 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1704 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1744 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1784 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1824 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1864 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1904 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1944 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |
| 1984 |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |





---

# Index

---

---

## **A**

ABS 22, 23  
ADSR 121, 122, 123, 124  
adventures 46, 47, 48, 52  
AND 11, 57, 58, 64  
animation 106–118  
arrays 41, 42, 43, 44, 47, 50  
ATN 23  
attack 121, 122, 124

## **B**

bit map 76–82  
block graphics 56, 60  
Boolean operators 57, 58  
branches 8–13

## **C**

cassette recorder 138, 142–146  
character generator 63, 64, 65  
CHR\$ 60, 61, 109, 110  
CLOSE 144, 146  
CMD 144  
colour 70, 77  
COS 23  
cursor controls 32, 33, 36

## **D**

DATA 40, 46, 50, 130, 131  
decay 121, 122, 124  
DEF FN 25  
delay loops 15, 114, 115  
device numbers 142  
DIM 41, 42, 43  
disk drive 138, 142, 144, 147

## **E**

envelopes 121, 122, 124  
error trapping 28, 29, 30, 31, 35  
expanded sprites 89, 96  
extended background mode 56, 70

---

## **F**

FOR .NEXT 13, 14, 15  
filters 127, 128, 129  
frequency 122

## **G**

gate 124, 125  
GET 29, 30, 31, 32  
GET# 144, 146  
GOSUB 8-13  
GOTO 8, 9, 10

## **H**

high resolution graphics 56, 57, 76-82, 140, 141

## **I**

IF .THEN 11, 12, 13  
INPUT 29, 30, 31, 32  
INPUT# 144, 146  
input/output 138-147  
INT 21, 22  
internal clock 15, 16, 17

## **J**

joysticks 138, 139, 140, 141

## **L**

LEFT\$ 43, 44, 48, 50  
Len 43, 44, 48, 50  
LOAD 145, 146  
logical files 142  
logical operators 11  
loops 13, 14, 15

## **M**

MID\$ 43, 44, 109, 110  
multi-dimensional arrays 42, 43  
multicolour bit map 74  
multicolour mode 56, 70, 71, 72, 73, 99, 100, 101

---

## **N**

nesting 14

## **O**

ON .GOSUB 12, 13

OPEN 143, 144, 146, 147

OR 11, 57, 58, 64

## **P**

PEEK 106, 107, 109, 110, 111

pitch 122

POKE 106, 107, 109, 110, 111

printer 138, 144

PRINT# 144, 146

PRINT AT 36, 37, 38

## **R**

radians 24

random numbers 20, 21, 22

READ 40, 47, 50, 130, 131

release 121, 122, 124

repeating keys 32, 33

RESTORE 40, 47

RETURN 10, 11

RIGHT\$ 43, 44

ring modulation 128, 129, 130

RND 20, 21, 22

## **S**

SAVE 145, 146

screen colour 29, 33, 34

screen memory 106, 107, 108

secondary address 142

sequential files 147

SGN 22

SID chip 120, 122, 123, 125, 126

SIN 23

sound 35, 120-136

SPC 36

sprites 84-104, 111, 112, 113

sprite collisions 89, 97, 98, 99

sprite colour 89

---

sprite coordinates 89, 92  
sprite data 87, 88, 92  
sprite pointers 85, 86  
sprite registers 89  
stack 10, 11, 15  
STEP 14  
strings 43, 44, 48, 50, 109, 110  
structured programs 8, 9  
sustain 121, 122, 124  
synchronisation 128, 129, 130

## **T**

TAB 36  
TAN 23  
text mode 56  
TI 16  
TI\$ 16  
trigonometry 23, 24

## **U**

user-defined functions 25  
user-defined graphics 63, 64, 65

## **V**

VAL 44  
VIC chip 56–60, 63, 64, 76, 85, 89

## **W**

waveforms 120, 121, 124





# Programming with added power

## Turbocharge your Commodore 64

### **BETTER PROGRAMMING**

Turbocharge your Commodore 64 tells you how the professionals do it. It concentrates on putting more power where it matters - in your hands.

It shows how you can exploit your Commodore 64 to the full and how to approach programming problems the right way.

### **IN-DEPTH EXPLORATION**

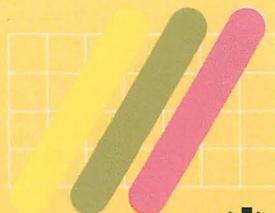
The Commodore 64 is still unexplored territory for many people. This book shows some of the ways in which it can be stretched, some of the ways in which its strengths and weaknesses can be exploited. You could do it for yourself - but not in a hurry. That's where Turbocharge your Commodore 64 comes in.

### **THE RIGHT STUFF**

Turbocharge your Commodore 64 gives expert insights into the full power of your Commodore 64. There are powerful graphics and sound routines and many K's worth of listings for you to explore and develop.

### **FOR THE PROFESSIONAL TOUCH IN YOUR PROGRAMS.**

TURBOCHARGE YOUR COMMODORE 64



**Longman**   
Computer  
Books

ISBN 0-582-91605-4



9 780582 916050

\*