CPC464

TEACH YOURSELF AMSTRAD BASIC

SOFT 156

A TUTORIAL GUIDE PART 2

AMSTRAD

BASIC

a tutorial guide

AMSTRAD

BASIC

a tutorial guide

http://amstrad.cpc.free.fr

Part 2
MORE BASIC

by Ian Padwick and George Tappenden

Amstrad Consumer Electronics

First edition 1985

Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
Brentwood
Essex

Amstrad BASIC
A Tutorial Guide
Part 2: More BASIC

SOFT 156
ISBN 1 85084 001 6

□ Production by Peter Hill, Ray Smith and Solo

□ Printed in England by Carlton Barclay Limited, Units 5, 6 & 7 Carlton Court Grainger Road Southend-on-Sea Essex SS2 5BZ Telephone Southend (0702) 613944 (5 lines)

CONTENTS

PREFACE

This is Part 2 of a self-study course on programming in BASIC using the Amstrad CPC464 Colour Personal Computer. Whereas Part 1 is intended for the beginner, *More BASIC* is suitable for anyone who is familiar with the principles of programming and already understands the most commonly used BASIC commands.

The two datacassettes that accompany this written text contain computer programs that are an integral part of the course.

Datacassette A contains:

- Programs to help illustrate the techniques described in the text.
- Games for your amusement and to demonstrate the capabilities of the CPC464.

Datacassette B contains:

- Self-assessment tests to make sure you have understood the content of the preceding chapter.

The third part of this series covers the most advanced features of Amstrad BASIC and is intended for experienced programmers.

Chapter 1
HERE WE GO AGAIN

If you read Part 1 of this series, you will have already learnt a certain number of Amstrad BASIC commands and will have a very good idea of the capabilities of the Amstrad CPC464 Colour Personal Computer. Hopefully, you will also have tried out your new-found knowledge by writing a few programs of your own and have had the satisfaction of seeing them run successfully - not the first time, maybe, but still...! This part of the course is intended to consolidate what you learnt in Part 1 and to give you an even more extensive vocabulary of Amstrad BASIC commands.

If you skipped Part 1, it is worth our repeating that Amstrad BASIC will not necessarily work on other computers - and vice versa. This is because it contains many unique commands and functions that are not available on less sophisticated equipment.

Each time a new keyword, or an extension of a previously described keyword, is introduced, it is printed in the outside margin so that you can easily flip back through the pages when you need to refresh your memory.

HOW TO USE THIS BOOK

Although this book has been written with a minimum of computer jargon, it would be very tiresome if we kept calling a spade a 'wooden-handled, straight-metal-bladed garden implement'. Within any specialised field of study it is inevitable that some words acquire special meanings, and computing is no exception. So we will start off with a look at the main terms used to describe the commands of Amstrad BASIC.

Each chapter of this book represents about one or two evenings'

work. Typically it will contain:

- Written explanation
- Practical work on the computer
- Examples for you to program yourself

There are exercises to reinforce what you have learnt, and there is a programmed self-assessment test to go with each chapter.

Don't skip chapters. New information is introduced progressively through the book and is built on the knowledge obtained from previous chapters. If you think a chapter or a section of a chapter looks a bit complicated, just read it quickly once or twice and then work through it slowly. The exercises and the programs that you are expected to enter yourself are intended to help you to understand the principles involved in the subject under discussion, so don't miss them out. Make sure that you understand by means of the self-assessment tests.

After completing this part of the course you should be able to write your own computer games, or hobbies and business software - not terribly sophisticated as yet, but that only comes with experience. Part 3 of this course will instruct you further in the fine detail of Amstrad BASIC.

COMMAND DESCRIPTIONS

In Part 1 we saw that a command is made up of a keyword and (mostly) one or more arguments, as in the line-drawing command:

```
DRAW x,y
```

where the arguments 'x' and 'y' give the graphic co-ordinates of the position on the screen to which the line must be drawn. As you already know, putting in the wrong characters or leaving spaces and characters out when typing in the command may produce the message 'SYNTAX ERROR'! So what is syntax? And why do you need a comma in the middle of the command? All will be clear before the end of this chapter.

Syntax

In ordinary English the sentence, 'The tail was wagged the little dog by', is inept and ridiculous. There are not many of us (we hope) who would dream of breaking the rules of our language in such a way. The syntax, i.e. the order in which words are placed in a sentence, is badly wrong. Even so, it is a measure of the superiority of the human brain over the computer that it is still

possible to understand the sense of the phrase and to compensate for the inadequacy.

A more extreme case is where there is an unclear connection between the parts of the sentence:

The tail/the little dog/was wagged

If the connecting word 'by' is not added at the right place there is considerable doubt as to what wagged which!

In BASIC the rules of the syntax are even more rigid than in ordinary English. There is no question of an IF-THEN-ELSE branch being an equivalent to an IF-ELSE-THEN branch – the second one just doesn't exist in BASIC. In the same way, if you forget to leave a space between one word and another in English, it may make it a little more difficult to read but it is still understandable. If you omit a space at a critical point when writing BASIC, the CPC464 will not be able to work out what you were trying to tell it and will often flatly refuse to run the program until you have corrected the error.

There are a lot of new and sometimes complicated commands to be learnt in this book so, to save time and space, a sort of shorthand will be used to describe the syntax of each command. This involves some unfamiliar terms and punctuation which require some explanation. Study the following glossary carefully. You may have to refer to it continually until the words become familiar.

Command glossary

Brackets Certain commands and functions require that the argument(s) be enclosed in round brackets '()', e.g. CHR$(97). Round brackets are also used to impose the order in which arithmetic operations are carried out.

In this book we will also use two other sorts of brackets for *describing* commands. The first sort, pointed brackets '<>', mean that whatever they enclose is described elsewhere. The second sort, square brackets '[]', show that whatever they enclose is optional.

Defaults When the CPC464 is switched on, or reset, there are a number of things, such as the screen, border and text colours, which are set up automatically. In addition, there are commands like DRAW and PLOT which, once the INK has been specified, can have this argument omitted in following commands. Values such as this are known as defaults and we shall see later that there are many commands where optional arguments have default values.

Expression See Integer Expression, Numeric Expression, and String Expression.

Integer Expression A numeric expression containing real or integer numbers whose result is rounded to the nearest whole number before it is used.

Integer number Whole numbers; that is, they do not have a decimal part.

Keyword A word reserved for use in BASIC commands such as RUN, MOVE and PRINT, or BASIC functions such as RND and INT.

Numeric expression This can be one of four things:

- A number, e.g. '20473.611'
- A numeric variable, e.g. 'total'
- The result of a numeric operation, e.g. 'total/252★percent'
- A function, e.g. 'LOG (x)'

Real numbers Numbers that have a decimal part.

Separators Used in BASIC to define the beginnings and ends of keywords and their arguments. The usual separators are:

comma	(,)
semi-colon	(;)
space	( )

In our command *descriptions* we will also use the colon (:) to show when an argument may consist of a list of items.

String expression Alphanumeric information which may be one of four things:

- A string enclosed by quotation marks, e.g. '"Name"'
- A string variable, e.g. 'type$'
- The result of a string operation, e.g. 'type$ + "Name"'
- A string function, e.g. 'LEFT$(A$,3)'

Example of a typical command

Take a careful look at the following:

```
DRAW (x co-ordinate),(y co-ordinate)[,(ink)]
```

where:

<x co-ordinate> = integer expression
<y co-ordinate> = integer expression
<ink> = integer expression

Because both <x co-ordinate> and <y co-ordinate> are described as *expressions*, we know that we can put into those positions

either a numeric value, or a numeric variable, or a combination of values, operators and variables which will result in a numeric quantity. Both of these integer expressions must be included in the command. If either, both, or the separator (,) are missing when the line is entered, the CPC464 will give the 'syntax error' message.

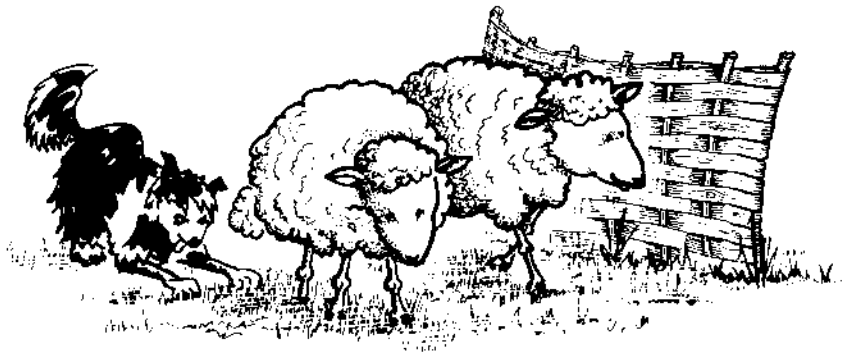
Conversely, '[,<ink>]' is enclosed in square brackets. If you leave out this part of the command the CPC464 assumes that you are happy for the current INK to be used as a default, and doesn't give you an error message. As a general rule, where an argument is optional it is because there is a default value of some sort.

So, every command in Amstrad BASIC can now be described in the same way. By using these brackets and separators, and now that you know what the different sorts of expressions are, we don't need to spend too much time on the subtleties of each command. Notice, for instance, that our DRAW command requires integer expressions for the co-ordinates. This does not mean that only whole numbers can be used here, but that the CPC464 rounds the value in these arguments to the nearest pixel. So you don't have to worry about decimal parts in these expressions.

If you refer to your copy of the *Amstrad CPC464 User Guide*, you will see that we are using exactly the same set of conventions. In this way you can quickly understand the use of *any* command not covered in the following chapters.

PLAYTIME

Load the first program from Datacassette A, SHEP. You can then spend an enjoyable half-hour pretending that you are a hill farmer directing your faithful sheepdog to get two rather stupid sheep into their fold. Have fun!



TESTING

For those of you who didn't go through Part 1 of this course, *First Steps*, it would be a good idea to run the first Self-assessment Test, SAT1, to make sure there are no aspects of Amstrad BASIC that you need to study before going on to Chapter 2. Even if you did complete Part 1, a little revision wouldn't do any harm!

Chapter 2
REHOUSING

We will meet quite a few old friends from Part 1 in this chapter: the house for a start. Our original design was rather long-winded with all those MOVEs, so now is the time to learn how to make it shorter and neater. Then we are going to have a more detailed look at variables.

Before all this though, let us have a look at a group of commands which are aids for program entry.

ABOVE AND BELOW THE LINE

When you have used LIST to display long programs on the monitor screen, it has probably been a bit tedious waiting for the bit you want to come round. In fact the command has an optional argument which speeds things up quite a lot. The structure is:


```
LIST[(line number range)]
```

where <line number range> can be one of four things. The best way of understanding what these are is to try them out as follows. Load the next program from Datacassette A, called CHATEAU, but don't run it. Then enter:

```
LIST 100
```

As you can see, line 100 is put on the screen on its own.

Now try:

```
LIST -100
```

This time the CPC464 gives you all the line numbers up to and including 100. Now you no longer have to be fast on the ESC key to stop the listing where you want to.

For the third example enter:

```
LIST 100-200
```

to get all the lines between and including 100 and 200. If you imagine that you have a niggling bug in that part of a program, you can easily see how useful this is for pulling out just that section onto the screen each time there is a change to be made.

And, finally, try:

```
LIST 700-
```

which gives you all the lines from 800 through to the end of the program. It is especially useful if you follow the normal practice of putting all your subroutines at the end of the program, starting at a particular line number.

DELETE

The next command is DELETE. The syntax is as follows:

```
DELETE (line number range)
```

You will notice that the argument is not optional, but otherwise it can take line number ranges in identical fashion to the LIST command. Entering, for example:

```
DELETE 100-200
```

will wipe those line numbers out of memory, so that the next time you LIST they will not appear on the screen.

Be careful when using this command. If you haven't saved a copy of the program you may accidentally rub out quite a lot of hard work. Where it is useful, though, is when you want to use some routines from an existing program. Just load the program and delete what you don't want.

RENUM

The next useful command in this group is RENUM. You will now see why we need a type of shorthand to explain the syntax:

```
RENUM [(new line number)]
  [,[(old line number)][,(increment)]]
```

These optional arguments allow you to renumber the lines of the current program starting at the position given by <old line number> beginning with <new line number> in steps of <increment>. If you don't specify any of the arguments, the default values are assumed. These defaults are equivalent to:

```
RENUM 10,,10
```

The missing argument merely indicates that renumbering starts at the first line in the program.

If you still have any of CHATEAU left after playing with the previous command, you can try out the various optional arguments before giving a RENUM by itself so that you get the

default arguments. Notice that RENUM automatically updates the GOTO and GOSUB references.

Finally, there is a command that saves you the chore of typing in the line number each time. This is AUTO, and is used as follows:


```
AUTO [(line number)][,(increment)]
```

The argument <line number> is the number at which you want to start and <increment> is the size of the step between numbers. If, for example, you enter:

```
AUTO 100,5
```

the CPC464 will put 100 at the beginning of the next line, leave a character space and leave the cursor in position so that you can complete the line by typing in a command. When you hit the ENTER key it does the same thing again except that the number will be 105 this time.

AUTO numbering is terminated by pressing the ESC key.

If there is already a line in memory with the same number as the one just generated, an asterisk (*) is printed as a warning. If no action is taken before pressing the return key, the new line replaces the old one.

VARIABLE VARIABLES

The way variables were described in Part 1 – labelled storage places for numbers and strings – was only part of the story. There are actually two types of numeric variable, *real* and *integer;* both these and string variables may also be *subscripted.*

Real numeric variables

Real numbers or expressions may have a decimal part or may even have a value less than one. Another name for these is 'floating point' variables since the decimal point is not in a fixed position. The following are examples of real numbers:

1234862.01
0.000000154768119
2.3

When you specify a variable, either by a LET or by implication in an expression, the default is that it will be real. For clarity in your programs, you could make sure you can recognise them by using the suffix '!', for example:

```
LET finallength!=measure*coeff*0.0001
```

Integer numeric variables

If you want to work in integers only, i.e. whole numbers which do not have a decimal part, it is necessary to specify this by adding the suffix '%' to the variable name, for example:

```
LET percent%=100*profit/cost
```

The variable 'percent%' can only ever contain integers, and any decimal part of any value stored in it will be rounded to the nearest integer.

Subscripted variables

Subscripted variables are sometimes called 'list variables' because they can store lists of similar data rather than just a single item. They can be real, integer, or string variables, and are described by:

```
<variable name>(<list of:<subscripts>)
```

where <subscripts> are integer expressions. Whole lists of variables can therefore be stored under a common name but with different subscripts.

For example, take the top six football teams in the first division at the beginning of April 1984:

Team	*Points*
Liverpool	69
Manchester Utd	67
Nottm Forest	60
QPR	57
Southampton	56
West Ham	55

Now suppose that we want to store information so that it can be updated week by week, and the list printed out in the latest order. If we number the league positions from zero to five the current points of the team at that position can be stored in a subscripted variable called 'points(n)' where 'n' is the league position, as follows

Variable	*Contents*
points(0)	69
points(1)	67
points(2)	60
points(3)	57
points(4)	56
points(5)	55

A table of data like the one above is also known as an *array*, and this is the term we shall use from now on. (Strictly speaking, our football points table is a *one-dimensional* array - we shall see why later on in this chapter.)

You would probably also want an array of strings with the names of the teams at each position.

CHATEAU

I don't suppose there are many of us who have not been taught that chateau is French for castle. What you may not know is that it is also the name used for particularly nice mansions in one of the richest parts of France.

Having massacred the program in the first part of this chapter, you would be well advised to rewind the cassette and load the program again. The intention here is to show how a program can be progressively improved, and so our original HOUSE program has had several rebuilds to bring it into the luxury dwelling class.

So, having loaded the program, run it. It looks just like the MANSION program with its window panes and fence, doesn't it? If you look at the listing below, however, you will see that the program has been completely rewritten.

```
100 ' Chateau
110 ' (Improved MANSION)
120 '
130 ' DA 17/9/84
140 '
150 MODE 0 : BORDER 12
160 INK 0,12 : INK 1,3 : INK 2,6 : INK 3,17
170 PAPER 0
180 '
190 ' Borders
200 '
210 READ x,y
220 IF y=-1 THEN 290
230 IF x=0 THEN colour=y : GOTO 210
240 IF x<0 THEN MOVE -x,-y ELSE DRAW x,y,colour
250 GOTO 210
260 '
270 ' Fence
280 '
290 FOR F=0 TO 620 STEP 20
300 MOVE F,0:DRAW F,60
```

```
310 NEXT F
320 MOVE 0,45:DRAW 620,45
330 '
340 ' Window Frames
350 '
360 size=18
370 FOR i=1 TO 16
380 READ x,y:MOVE x,y
390 DRAWR 0,size : DRAWR size,0
400 DRAWR 0,-size: DRAWR -size,0
410 NEXT
420 '
430 END
440 '
450 ' Data for edges
460 ' in pairs : negative = move
470 '          : 0,x      = colour change
480 '          : 0,-1     = end
490 '
500 DATA    0,    1
510 DATA -100,-50,   100,250,  400,250,  400,50,   100,50
520 DATA -400,-250,  600,250,  600,50,   400,50,   400,250
530 DATA 500,350,    600,250,  400,250
540 DATA -100,-250,  200,350,  500,350
550 '
560 ' Door
570 '
580 DATA 0,2
590 DATA -225,-50,  225,140,  275,140,  275,50
600 '
610 ' Large Windows
620 '
630 DATA 0,3
640 DATA -120,-70,   120,130,  180,130,  180,70,   120,70
650 DATA -120,-170,  120,230,  180,230,  180,170,  120,170
660 DATA -320,-170,  320,230,  380,230,  380,170,  320,170
670 DATA -320,-70,   320,130,  380,130,  380,70,   320,70
680 DATA 0,-1
690 '
700 ' Data for little windows
710 '
720 DATA 130,78,   156,78,   130,103,  156,103,  130,178
730 DATA 156,178,  130,203,  156,203,  330,78,   356,78
740 DATA 330,103,  356,103,  330,178,  356,178,  330,203,
356,203
```

The first thing you will notice is the use of the apostrophe (') sign instead of REM. It operates in exactly the same way - anything following an apostrophe is ignored. As you can see, the

advantage is that the layout can be a lot clearer with dummy lines between explanations and remarks.

And here are two important new keywords for you as well, DATA and READ. In our MANSION program in Part 1 there were a lot of successive MOVE commands to get the graphics cursor into the right place before issuing a GOSUB. A neater way of achieving the same end is to put all the positioning co-ordinates in the same place and then use variables in the MOVE and DRAW commands. This is exactly what READ and DATA allow you to do.

The syntax for DATA is:

```
DATA <list of:<constant>
```

A list of numeric or string constants can be used, separated by commas. In fact almost anything is acceptable, since a line of rubbish with no comma in it will be taken as one string constant.

The syntax for READ is:

```
READ <list of:<variable>
```

A <variable> can be either string or numeric, separated by commas.

The following few lines show how these two commands work together:

```
10 FOR x=1 TO 4

20 READ N$

30 PRINT N$

40 NEXT

50 DATA Ann,Dorothy,Jill,Sharon
```

Each time the program executes the READ command, the next string constant in the DATA statement is put into the variable N$. So, when x=1 'Ann' is printed, when x=2 'Dorothy' is printed, when x=3 'Jill' is printed, and when x=4 'Sharon' is printed.

You may have wondered if there was any way of avoiding having to write out long DATA statements all over again when you needed exactly the same values to be read more than once. The keyword that can help you do this is RESTORE. Here is the syntax:

```
RESTORE [<line number>]
```

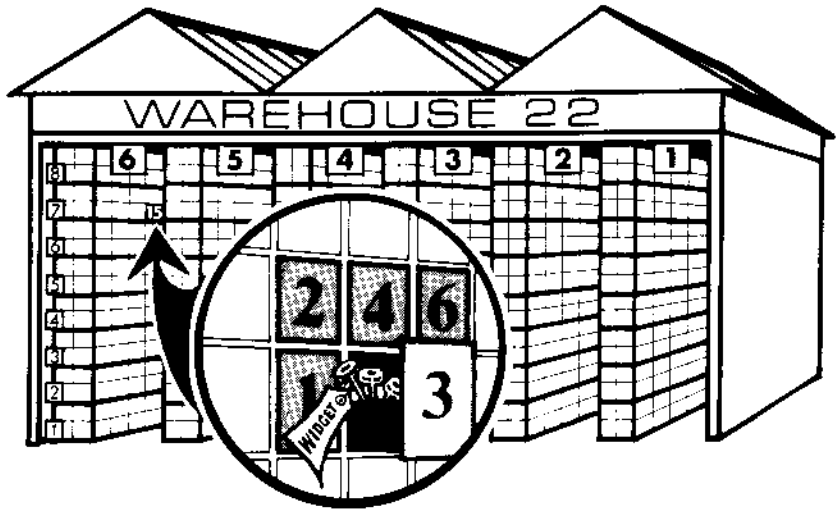
To understand how the argument <line number> works, you have to remember what happens each time the CPC464 executes a READ command. No matter how many DATA statements there are, or where they are in the program, they are read one by one in the order in which they appear. After each READ command is executed the CPC464 sets up a pointer which indicates the next constant to be used when the next READ command is executed.

The optional argument in the RESTORE command allows you to reset this pointer to the first constant of the DATA statement at or following the position in the program identified by <line number>. By this means you can re-use or select data constants dynamically during execution of a program. Omitting the argument will set the pointer to the first constant of the first DATA statement in the program.

ARRAYS

As promised, we are now going to have a look at arrays. Earlier in this chapter we saw that you could make up tables by using subscripted variables. You then get the effect of a certain number of boxes with a common name.

This, then, is a one-dimensional array, 'widgets(a)', where 'a' gives the number of a particular box. Printing 'widgets(3)' gives the number of widgets in box 3. We can subdivide further to give a two-dimensional array, widgets(b,a) where 'a' is the number of our box, within its own carton, and 'b' is the carton number. Printing 'widgets(15,3)' gives the number of widgets in box 3 of carton number 15.



And before you ask the question, the answer is yes, you *can* have three-dimensional arrays, but we won't be bothering with them in this book. In fact, the CPC464 lets you use arrays of as many dimensions as you like (subject to line length and available memory), as you can see from our illustration!

So, how do you use an array? Well, enter the following and run it:

```
10 widgets(5)=47

20 for n=0 to 20

30 print widgets(n);

40 next
```

Note that each of the positions in the array designated by a subscript is known as an 'element', and the numbering starts at zero. Line 10 assigns the value 47 to subscript 5, i.e. the sixth box in the array, and the remaining lines are just to print the contents of the entire array. What you will get on the screen is:

```
 0  0  0  0  0  47  0  0  0  0  0

Subscript out of range in 30
```

The error message is because there is a new command to learn, but we'll come to that in a minute. You can see that the CPC464 sets the initial value of subscripted variables to zero, so the first five zeros represent the values of elements 0, 1, 2, 3 and 4, and the last five zeros the values of elements 6–20. So what happened to element 11? Well, unless we tell it otherwise, the CPC464 assumed that any array we use has 10 elements or less, which explains why we got an error message.

For arrays with more than 10 elements we have to use the DIM command to specify the maximum size of an array. The syntax is as follows:

```
DIM <list of:<subscripted variable>
```

where <subscripted variable> is, of course:

```
<variable name>(<list of:<integer expression>>)
```

Each <integer expression> sets the maximum value allowed for the corresponding subscript.

We can therefore get rid of the error message in two ways. The simplest is to stop the loop at 10 instead of 20! But if you really insist on a 21-element array, you can add the following line:

```
5 DIM widgets (20)
```

Try it.

PLAYTIME

The next program, VILLAGE, gives you not just one house but lots of houses!

TESTING

Check that you have fully understood all this new information by running SAT2 before going on to the next chapter.

Chapter 3
PAINTING BY NUMBERS

PEN

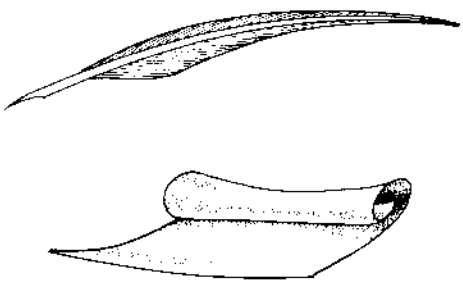
PAPER

INK

If you have been confused by the relationship between PEN, PAPER and INK up till now then read on. If not, then read on anyway since it is probably a lot more subtle than you think!

The CPC464 can display 27 different colours (see table overleaf), which can be put into 16 different ink pots. However, the number of ink pots within range for drawing depends on the MODE.

In MODE 0 you have a range of 16 inkpots, in MODE 1 you have 4, and in MODE 2 you have just 2. Imagine now that whenever you PRINT text you are using a quill and a sheet of paper that have each been dipped in an ink pot.



Switch on or force a restart (SHIFT/CTRL/ESC); you are in MODE 1, and you therefore can have 4 inks on the screen at once. All 16 inkpots have been filled with their default colours. The screen is blue because it has been set to the colour of PAPER, which has defaulted to INK 0 (colour 1). The text is in yellow because it was written by the PEN which has defaulted to INK 1 (colour 24).

Number	*Colour*	*Number*	*Colour*
0	black	14	pastel blue
1	blue	15	orange
2	bright blue	16	pink
3	red	17	pastel magenta
4	magenta	18	bright green
5	mauve	19	sea green
6	bright red	20	bright cyan
7	purple	21	lime green
8	bright magenta	22	pastel green
9	green	23	pastel cyan
10	cyan	24	bright yellow
11	sky blue	25	pastel yellow
12	yellow	26	bright white
13	white		

Type PEN 2. Now new text is written in bright cyan (colour 20). Type PEN 3 and the new text becomes red (colour 6). Type PEN 4, however, and the new text will be blue – the same as the paper. This is the same as typing PEN 0 since although there are 16 ink-pots, you only have 4 colours within range in this mode. In fact the PEN really *has* been set to INK 0. In this mode the CPC464 just refuses to set it outside the range 0 to 3: it has converted your request for INK 4, which is out of range, into INK 0 which *is* in range. Similarly, INK 5 would be converted to INK 1, INK 6 to INK 2, INK 7 to INK 3, INK 8 to INK 0 again, and so on.

You still have blue on blue, so type PAPER 1 carefully, because you won't see any mistakes you might make. If you've got it right you will now see the word 'Ready' in blue on a yellow

background. This is because when the CPC464 puts a character on the screen, it uses a square background, just large enough for one character, which is in the INK of PAPER. The character is then written on this background with the current PEN, so the yellow surrounding 'Ready' is the result of drawing a character with the newly defined background.

Entering CLS will set the whole of the screen (except the border, of course) to the new yellow PAPER.

Now we come to the tricky part. Let's say we don't want yellow, we want white instead. This involves using the INK command. Enter:

```
INK 1,26
```

Lo and behold, everything that was yellow is now white!

This is what happened. To change yellow to white we had to empty inkpot 1 (which contained yellow) and refill it with white, colour 26. So 'INK 1,26' means 'change the ink in pot 1 to colour 26'. It does not mean 'change colour 1 to colour 26'.

You will remember that INK 4 was defaulted to white, even though it is out of range in MODE 1. Now we have white in INK 1 as well. That's perfectly all right – we can have the same colour in as many pots as we like!

Of course, the CPC464 is constantly, dipping into the inkpots to refresh the screen (50 times a second in fact) so everything that was yellow becomes white. Try changing the colour of another ink, say blue into orange. The blue is in inkpot 0 and orange is colour 15 so we type:

```
INK 0,15
```

Now change it back to blue:

```
INK 0,1
```

COLOUR AND MODE

In MODEs 1 and 2 you can put any number in the range 0–15 after the PEN and PAPER commands, but the CPC464 will change the value it uses to refer to the lowest numbered inks, i.e.

in MODE 2 whatever INK you try to set PEN and PAPER to, only INKs 0 and 1 are within range, and so on. However, the same is not true of the INK command. The colours of all 16 INKs can be affected, although you won't see this unless you change modes.

Force a restart, then type:

```
MODE 0:PEN 13
```

Note that the colour of the text is pastel green. Now type

```
MODE 1
```

and then

```
INK 13,26
```

Nothing changes since INK 13 is far out of range in MODE 1. Now type MODE 0:PEN 13 again. The text is now white, not pastel green. What we have proved is that you can change the colour in an inkpot even when it is out of range in the current screen mode.

COLOURED DRAWINGS

PLOT

DRAW

Here are the syntax descriptions of our old friends the PLOT and DRAW commands:

```
PLOT (x co-ordinate),(y co-ordinate)[,(ink)]
DRAW (x co-ordinate),(y co-ordinate)[,(ink)]
```

INK DEFAULTS

Ink	*Colour*
0	1
1	24
2	20
3	6
4	26
5	0
6	2
7	8
8	10
9	12
10	14
11	16
12	18
13	22
14	1,24
15	16,11

The <x co-ordinate> and <y co-ordinate> must be numeric expressions. If the INK argument is not included, the line or pixel is drawn in the current ink of the graphics pen. This is set to INK 1 by default, but every time a command including the INK argument is used, the line is drawn in the new ink and the graphics pen is set to it.

Changing the text PEN INK does not affect the INK with which you PLOT and DRAW. Enter:

```
DRAW 640,400
```

You should get a line across the screen the same colour as the text. Now enter:

```
MOVE 0,0:DRAW 640,400,3
```

The line will be redrawn in INK 3.

Type in the following program after forcing a restart.

```
 10 CLS

 20 r=40:y=200:c=1

 30 FOR x=40 TO 600 STEP 40

 40 FOR i=-r TO r STEP 2

 50 h=SQR(r*r-i*i)

 60 PLOT x-h,i+y,c

 70 DRAW x+h,i+y

 80 NEXT i

 90 c=c+1

100 NEXT x
```

Type MODE 0 then RUN. You will see a series of variously coloured disks drawn across the screen, each partly covered by the next. Don't worry how the disks are drawn, it is the colours we are concerned with.

First, remember that in MODE 0 you have all 16 inkpots within range and filled with the colours you see. Secondly, you change the colour you PLOT with by putting the INK number after the co-ordinates. Now look at the listing. In line 60 we have a third variable, 'c', which specifies the INK to use. In line 90 we see that 'c' is increased by one each time the 'i' loop is completed. Variable 'c' has an initial value of one, line 20. RUN the program in each mode to confirm what has been discussed so far in this chapter.

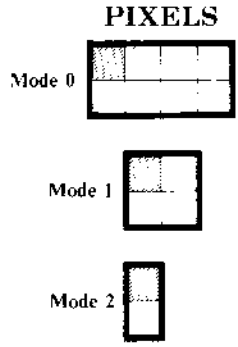
Finally, RUN the program in MODE 0 and then experiment by typing some INK commands like INK 6,12 and INK 2,15.

WINDOW DRESSING

Once you have mastered the manipulation of PEN, PAPER and INKs you should be able to create some interesting effects. Even more elaborate screen displays can be achieved once you have learnt a few more details about the CPC464's ability to mix text and graphics.

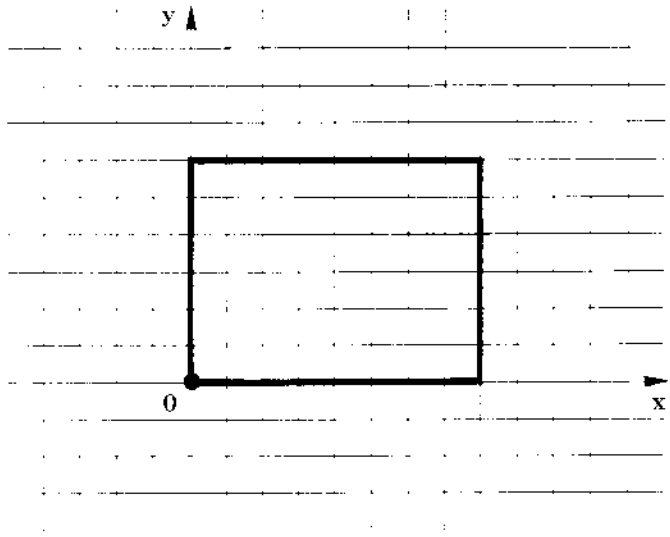
First, let us study the ORIGIN command. Those of you familiar with mathematics will know that the origin of a graph is the point where the horizontal and vertical axes meet, usually x=0, y=0.

The CPC464's display is made up of 256,000 individual points. There are 640 across the screen (numbered 0 to 639 from left to right), and 400 up the screen (numbered 0 to 399 from bottom to top). These points are not the same as pixels, which are made up of a different number of points depending on the mode. Each pixel can be designated by the co-ordinates of any of the points of which it is comprised.



Each of these points can be referred to individually for the purposes of drawing graphs, etc. Plotting 0,0 will cause point 0,0 (and all the other points in the same pixel) to be set to the graphics pen.

Plotting values of x and y that are outside the limits above (x = 0 to 639 and y = 0 to 399) will not cause an error. This is because the CPC464 remembers your position off the screen and continues to execute PLOT and DRAW commands, even though you can't see anything on the screen.



ORIGIN

The ORIGIN command allows you to 'move' the co-ordinate 0,0 to anywhere on the screen, or off it! Being able to do this is very convenient for plotting graphs.

Force a restart, then type in the following program:

```
 5 CLG
10 PLOT 0,0
20 DRAW 200,200
30 DRAW 400,200
40 DRAW 200,350
50 DRAW -100,350
60 DRAW -100,100
70 DRAW 0,100
80 DRAW 0,0
```

RUN the program and you will see that an irregular polygon of no particular significance has been drawn. Note how part of it cannot be seen because it is off the left edge of the screen. Now type ORIGIN 150,0 and then RUN. Clever isn't it! You have just told the CPC464 to consider the point 150,0 relative to the bottom left of the actual screen as the co-ordinate 0,0 and to do all PLOTing and DRAWing relative to this new origin. The bottom left corner of the screen is now referenced by PLOT -150,0.

Don't get confused about what is happening. Imagine a piece of paper on an enormous drawing table: you can draw anything you like on the whole of the table, but the paper only remembers what actually passes over it. The rest might as well never have happened.

Try putting in some different values for the origin such as -50,0 or 200,-50. Once you have done that add some more lines to the program:

```
5 FOR o=0 TO 100 STEP 10

6 ORIGIN 100+o,o

90 NEXT o
```

and RUN.

You will have come across the CLG command before - it has an effect similar to CLS. Here is the syntax:

```
CLG [(ink)]
```

CLG clears the graphics window and moves the graphics cursor to wherever the co-ordinates 0,0 are. The optional argument <ink> will fill the graphics window with the ink specified. But just a minute, you may say, what is the graphics window? Well, it's the whole screen until you tell the CPC464 that it isn't. You do this by an addition to the ORIGIN command. Type in:

```
ORIGIN 100,0,0,300,200,0
```

This tells the CPC464 to move the origin to co-ordinate 100,0 and also to limit the graphics window to all pixels 0 to 300 horizontally and 0 to 200 vertically. Doing this will chop off all but the bottom left quarter of the screen.

It's about time we looked at the syntax of ORIGIN:

```
ORIGIN (x),(y)[,(left),(right),(top),(bottom)]
```

All the arguments are called 'absolute screen co-ordinates' and must be numeric expressions. An absolute screen co-ordinate is just like the numbers you give to DRAW or PLOT, except that it is always relative to the actual bottom left corner of the screen. It

doesn't matter whether a previous ORIGIN command has moved co-ordinate 0,0; the next one always has the same effect. In other words a new ORIGIN or window is not relative to the last setting of co-ordinate 0,0; but to the bottom left corner of the screen.

Enter CLS and then RUN the program. You will see that the graphics display is now limited to the lower left quarter of the screen. LIST the program two or three times then type CLG. Just the lower left quarter of the screen is cleared. Enter the following:

```
CLS: PAPER 3: CLS
```

You should have a red screen; now type CLG. Our graphics window is blue. Now RUN the program to complete the effect.

Try some different values for the window, e.g.

```
ORIGIN 100,0,301,600,400,201
```

will move the window to the upper right of the screen. To return to normal you would type:

```
ORIGIN 0,0,0,639,399,0
```

WORDS AND PICTURES

The following listing is of the next program on Datacassette A called SINCOS which we are going to use to illustrate some of the new keywords and to introduce a couple more.

```
100 '
110 ' Sine and Cosine Graph
120 '
130 ' By Ian Packwick
140 ' Amended by DA 21/9/84
150 '
160 ' Set up screen
170 '
180 MODE 1:BORDER 20
190 INK 0,23:INK 1,20:INK 2,0:INK 3,26
200 ORIGIN 144,200,108,534,310,56 : CLG 2
210 DEG ' use degrees in sin/cos
220 PEN 2:LOCATE 4,4
230 PRINT"Graphs of Sine and Cosine functions"
240 '
250 PLOT 0,-100,3 : DRAW 0,100
260 PLOT 360,-100 : DRAW 360,100
270 PLOT 0,0      : DRAW 360,0
280 PLOT 90,7     : DRAWR 0,-14
```

```
290 PLOT 180,7     : DRAWR 0,-14
300 PLOT 270,7     : DRAWR 0,-14
310 TAG
320 MOVE -20,104   : PRINT "1";
330 MOVE -20,4     : PRINT "0";
340 MOVE -36,-94   : PRINT "-1";
350 MOVE -4,-110   : PRINT "0";
360 MOVE 78,-110   : PRINT "90";
370 MOVE 158,-110  : PRINT "180";
380 MOVE 248,-110  : PRINT "270";
390 MOVE 338,-110  : PRINT "360";
400 MOVE 50,-128   : PRINT "Angle in degrees";
410 '
420 ' Draw sine curve
430 '
440 FOR x=0 TO 360
450 PLOT x,100*SIN(x),1
460 NEXT
470 MOVE 132,90
480 PRINT"SINE";
490 '
500 ' Draw cosine curve
510 '
520 FOR x=0 TO 360
530 PLOT x,100*COS(x),3
540 NEXT
550 MOVE 40,-80:PRINT"COSINE";
560 '
570 ' Tidy up
580 '
590 TAGOFF : LOCATE 1,1
600 WHILE INKEY$="" : WEND
610 END
```

After running the program, study the listing above. First, line 180. The border is set to colour 20. In line 200 we see an ORIGIN command that sets a graphics window in the centre of the screen. Our new origin is 144,200 and the window is limited to all points from 108 to 534 horizontally and from 56 to 310 vertically.

The command DEG in line 210 tells the CPC464 to do all its calculations of sines and cosines in degrees instead of radians (see Chapter 6) and lines 250 to 300 draw the axes for the graph.

The keyword in line 310, TAG, allows you to do some clever things with PRINTed text. When you PRINT ordinarily, the characters are assigned to the 'cells' into which the screen is

divided and to which we can refer using the LOCATE command. The text cursor points to the cell where the next character will be PRINTed.

TAG stands for Text At Graphics cursor and when it is in operation all PRINTed text commences at the graphics cursor instead of the text cursor. Since we know that the graphics cursor can be anywhere on or off the screen, this allows great flexibility when PRINTing. In this exercise we are going to use TAG to label the axes and curves of the graph.

The reason we have a special command to achieve this is that it is much slower for the CPC464 to draw a character on the screen if it straddles character cells.

The graphics cursor is moved around by the MOVE or MOVER command. Type LIST 310-400 and you will see how the PRINTing is done. Note that the semi-colon after each set of quotes is used to suppress a pair of arrows that would otherwise be PRINTed.

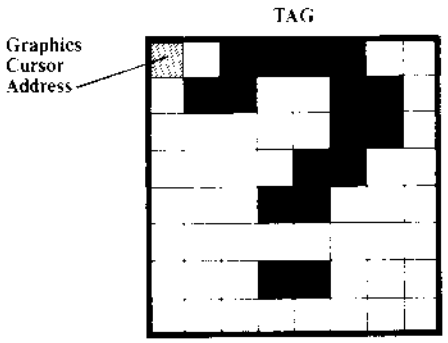
Enter the following:

```
CLS:TAG:MOVE 50,50:PRINT"HELLO THERE"
```

and you will see the arrows. If we weren't using TAG, these characters would move the cursor to the start of the next line. Since we are using TAG we have no use for them.

For every character the graphics cursor takes the top left position within each 8-pixel-by-8-pixel character cell, and is moved 8 pixels right for the next character. Note that if you now type PRINT"LOLO", the characters are PRINTed at the text cursor once more. TAG is switched off after ENTER is used; that is why the first line above was typed in as one line. Type TAG:PRINT"LOLO"; once more the characters are PRINTed at the graphics cursor.

Note also that the ink for the TAG text is the graphics and not the text PEN ink. To change the ink you have to use the third



argument of PLOT or DRAW since there is no command like PEN to change the graphics pen directly. Similarly, the background colour for the characters is that of the graphics window, *not* PAPER.

Lines 440–550 carry out the calculations and PLOTing for the graphs (which may or may not be familiar to you). In Chapter 6 the uses of SIN and COS will be fully explained and reference will be made to this program. Line 590 has the keyword TAGOFF which does exactly what you think it does - it returns PRINTing to the text cursor. Note that when TAG is on, LOCATE still moves the text cursor around the screen, even though nothing happens there until you issue a TAGOFF.

TAGOFF

PIE CHART

The next program on Datacassette A is PIE CHART. It is a further example of a program that uses both words and graphics on the same screen.

Pie charts are even more popular with sales managers, market researchers and sociologists than bar charts. The reason is they can show percentages as a portion of a total cake, rather than just relative to one another. This program only asks you to enter the numbers you want displayed - it even calculates the percentages for you! Try it out before studying the following listing.

```
100 ' Pie chart
110 '
120 ' DA 30/9/84
130 '
140 ' Screen
150 '
160 CLEAR
170 DIM name$(9),amount(9) ' optional
180 MODE 2   : BORDER 13
190 INK 0,23 : INK 1,0
200 PAPER 0  : PEN 1
210 PRINT CHR$(23);CHR$(0);
220 '
230 GOSUB 500 ' heading
240 '
250 a$="Y"
260 WHILE UPPER$(a$)="Y"
270 '
280 GOSUB 650 ' get data
290 CLS
```

```
300 GOSUB 930 ' Draw pie
310 '
320 LOCATE 1,19
330 INPUT "Again (Y/N) ",a$
340 WEND
350 END
360 '
370 ' SUBROUTINE : Draw a circle
380 '      Centre : x,y
390 '      Radius : r
400 '
410 MOVE x,y+r
420 s=0.2 : 'Fineness of PLOT
430 FOR i=0 TO 2*PI+s STEP s
440 DRAW x+SIN(i)*r,y+COS(i)*r,1
450 NEXT
460 RETURN
470 '
480 ' SUBROUTINE : Print heading page
490 '
500 CLS
510 LOCATE 35,2 : PRINT "Pie Chart"
520 MOVE 260,360:DRAW 356,360:DRAW 356,390:DRAW 260,390:
DRAW 260,360
530 WINDOW 5,80,5,25
540 PRINT "This program demonstrates a simple subroutine
which will draw pie charts"
550 PRINT "from  a  set  of  information  supplied.  You
should enter the data as a"
560 PRINT "numeric  list  giving  if you  wish  a name to
each item. The program is"
570 PRINT "designed  to  accept ten items only although
the pie routine within will"
580 PRINT "work with as many items as you wish,  should
you alter the input program"
590 LOCATE 25,7:PRINT"Press SPACE to continue"
600 WHILE INKEY$()" " : WEND
610 RETURN
620 '
630 ' SUBROUTINE : Get 10 data items
640 '
650 CLS
660 PRINT"Please enter the amount and the the description
of each item. The list will"
670 PRINT"terminate after 10 entries or if you give a
negative value for an amount"
680 PRINT
690 LOCATE 18,4 : PRINT " Amount" : LOCATE 35,4 : PRINT
"Description"
700 entry=0:count=0
710 WHILE ( entry)=0 )  AND ( count(10 )
720 LOCATE   9,5+count : PRINT "Item";count+1
```

```
730 LOCATE  19,5+count : INPUT "",entry ;
amount(count)=entry
740 LOCATE  35,5+count : INPUT "",name$(count)
750 count = count + 1
760 WEND
770 IF entry)=0 THEN count=count+1
780 count = count - 2 ' now=no of valid items
790 sum=0
800 FOR i=0 TO count
810 sum=sum+amount(i)
820 NEXT i
830 PRINT : PRINT TAB(15);"Thank you. Now press SPACE
to see the chart"
840 WHILE INKEY$ () " " : WEND
850 RETURN
860 '
870 ' SUBROUTINE : Draw the pie
880 ' Entry variables : x,y,r - centre and radius of circle
890 '                 : count - number of items
900 '                 : amount(0) to amount(count) - items
910 '                 :
920 '
930 x=320 : y=180 : r=130
940 GOSUB 410 ' Draw circle
950 '
960 p = 0 : oldp = 0
970 '
980 FOR i=0 TO count ' main loop
990 cum=0 ' zero cumulator
1000 '
1010 ' Count up to current bar
1020 '
1030 FOR j=0 TO i : cum=cum+amount(j) : NEXT
1040 '
1050 oldp = p ' store last bar
1060 p = ( cum/sum )*2*PI  ' calculate next one
1070 MOVE x,y ' centre of circle
1080 DRAWR SIN(p)*r,COS(p)*r,1 ' draw to edge
1090 GOSUB 1160 ' print description
1100 NEXT i
1110 RETURN
1130 '
1140 ' SUBROUTINE : Print pie description
1150 '
1160 middle=(p+oldp)/2  ' middle of segment
1170 outer=r+25         ' outside circle
1180 MOVE x+SIN(middle)*outer,y+COS(middle)*outer
1190 wid = LEN(name$(i))
1200 IF middle ) PI THEN MOVER -8*wid,0 'if on left
move left
1210 TAG : PRINT name$(i); : TAGOFF
1220 RETURN
```

TESTING

When you run SAT3, keep this book to hand so that you can look up the answers. You aren't sitting an exam - these SATs are meant only as a help in assessing how well you have understood before going on to the next chapter.

Chapter 4
TIME ON YOUR HANDS

LOOPS WITHIN LOOPS

In Part 1 the keywords FOR, NEXT and STEP were introduced to show how a row of fence posts could be drawn across the screen. To save you looking it up, the routine was:

FOR-NEXT

```
650 FOR f=0 TO 620 STEP 20

660 MOVE f,0:DRAW f,60,15

670 NEXT f
```

Type it in now if you like and RUN it. It was explained that the CPC464 allocates successive values of 'f', increased by the value of the STEP, each time it comes across the NEXT, beginning with 0 and ending with 620. So the first value of 'f' is 0, the next is 20, the next 40 and so on. It should be mentioned that the end-of-range value is not necessarily the last value to be taken; that depends on the STEP size. For example, if you type in a new line 650:

```
650 FOR f=0 TO 635 STEP 20
```

The last value of 'f' will still be 620 because the next value, 640, falls outside the FOR-NEXT range. The STEP size can be virtually any amount including fractions and negative numbers, for example:

```
FOR f=0 TO 10 STEP 0.15
```

gives values 0, 0.15, 0.3, 0.45, 0.6, etc., up to 9.9. 10 is not the last value.

```
FOR f=10 TO -7 STEP -2
```

gives values 10, 8, 6, 4, 2, 0, -2, -4 and -6. Again, -7 is not the last value.

In the last chapter, the program which drew a row of disks across the screen had two FOR-NEXT loops, one inside the other. The inner loop (variable i) drew the disks, while the outer loop (variable x) positioned the centre of each disk.

FOR-NEXT loops can be used in three ways:

- As in the example above, if we want particular successive values of the FOR variables that are to be used for PRINTing, PLOTing, or running through an array.
- If we want to repeat a set of commands a particular number of times.
- If we want to introduce a delay to slow down the speed at which things happen.

The following program illustrates all three principles:

```
10 FOR 1=1 TO 3

20 FOR n=10 TO 100 STEP 5

30 PLOT 0,n:DRAWR 100,100

40 FOR d=1 to 500:NEXT d

50 NEXT n

60 CLS

70 NEXT 1
```

The innermost loop on line 40 is the delay loop which causes the CPC464 to count from 1 to 500 to itself before carrying on. Try changing the value 500 to, say, 50 or 5000 to get the idea, then delete that line completely to see the maximum speed.

Nesting

The next loop is from lines 20 to 50 and uses the variable 'n' to position the graphics cursor. The outer loop, lines 10 to 70, makes sure the whole process is repeated three times. The loops in the program are said to be 'nested', which means stacked one inside the other. Any number of loops can be nested so long as you remember that they must never overlap. Swop lines 50 and 70 and then RUN.... You will get a message: Unexpected NEXT in 70. The CPC464 found that loops 'l' and 'n' were overlapped and so failed to execute the program. If you leave the variable name out after the NEXT command the CPC464 will match a NEXT to the last FOR. But that means that if your program is wrong, it

will simply run on a bit further before collapsing, so don't do it! *Always* add the variable name.

Note that when the CPC464 exits from a loop, the variable used for the loop will have a value equal to the first value that was OUTSIDE the range and not the last value inside the range.

RUN the program again and when it has finished type in:

```
PRINT n;1
```

You will see the numbers 105 and 4 on the screen: *not* 100 and 3 as you may have expected. The CPC464 will also allow you to jump out of a loop before it is completed and, of course, any of the range and step numbers can be replaced by variables:

```
10 LET a=7:LET b=73: LET s=1.2

20 FOR n=a TO b STEP s

30 IF n>50 THEN GOTO 60

40 PLOT 5,n

50 NEXT n

60 PRINT n
```

ONCE IN A WHILE

The FOR-NEXT loop is not the only sort of loop you can use on the CPC464. There is also the WHILE-WEND loop. The syntax of WHILE is as follows:

```
WHILE (logical expression)
```

WHILE-WEND

These loops can be nested just like FOR-NEXT loops and the same rules apply; every WHILE must have its WEND. The WHILE-WEND loop is one that depends upon a specified condition being true or false. All the program that is between the WHILE and its matching WEND will be executed so long as the condition specified in the WHILE is true. For example:

```
10 WHILE s<1000

20 n=n+1

30 s=n*n*n

40 PRINT s;
```

```
50 WEND

60 PRINT
```

RUN this and you should see on your screen ten numbers which are the cubes of the numbers 1 to 10. The program was prevented from continuing beyond this by the WHILE, but only after the condition had been tested. When the CPC464 then came across the WEND again, 's' was not less than 1000 so it jumped out of the loop to line 60.

What, you may be thinking, are the advantages of using a WHILE-WEND loop instead of the following?

```
10 IF s>=1000 GOTO 60

20 n=n+1

30 s=n*n*n

40 PRINT s;

50 GOTO 10

60 REM end of loop
```

After all, it does the same thing!

The answer is that with WHILE-WEND you do not have to concern yourself with which line numbers to GOTO, and it is therefore more convenient, especially if you have a large block of program in the loop or you are liable to move and change lines and forget where you had to jump back to.

You may exit a WHILE-WEND loop before the condition becomes false if you wish; the CPC464 won't get confused!

DIGITAL CLOCK

If you now look at the listing of DIGITALC you will see that there are seven FOR-NEXT loops, six of which are nested. There is also a very simple WHILE-WEND loop at line 170. There are several aspects of programming in this program that may be new to you, but one very important one is the conditional multi-statement line as seen in lines 1000 and 1180 of DIGITALC.

```
10 INK 1,26:PEN 1:MODE 1
20 FOR n=48 TO 57
30 n$(n-48)=CHR$(n)
40 NEXT n
50  PRINT TAB(14);"DIGITAL CLOCK"
60 PRINT :PRINT
70 INPUT "ENTER starting hour:-";h
80 ht=INT(h/10)
90 x=h-ht*10
100 PRINT
110 INPUT "ENTER starting minutes:-";m
120 y=INT(m/10)
130 z=m-y*10
140 LOCATE 6,12:PRINT"Clock set to start at";CHR$(ht*10+x);
":";n$(y);n$(z);":00"
150 PEN 2:INK 2,1:INK 1,24:SPEED INK 12,35
160 LOCATE 2,22:PRINT "  Press any key to start the clock
170 WHILE INKEY$="":WEND
180 MODE 0:PEN 2:INK 2,24
200 PRINT "    DIGITAL  CLOCK"
210 LOCATE 9,10:PRINT ":  :"
1000 IF ht=0 THEN v=9 ELSE v=2:LOCATE 7,10:PRINT n$(ht)
1010 FOR hu=x TO v
1020 LOCATE 8,10:PRINT n$(hu)
1030 FOR mt=y TO 5
1040 LOCATE 10,10:PRINT n$(mt)
1050 FOR mu=z TO 9
1060 LOCATE 11,10:PRINT n$(mu)
1070 FOR st =0 TO 5
1080 LOCATE 13,10:PRINT n$(st)
1090 FOR su=0 TO 9
1100 LOCATE 14,10:PRINT n$(su)
1110 FOR p=1 TO 929:NEXT p
1120 NEXT su
1130 NEXT st
1140 NEXT mu:z=0
1150 NEXT mt:y=0
1160 NEXT hu:x=0
1170 LOCATE 7,10
1180 IF ht=1 THEN ht=0:x=1:PRINT " ":ELSE ht=1:x=0:
PRINT n$(ht)
1190 GOTO 1000
```

In Chapter 7 of Part 1, you were introduced to the IF-THEN-ELSE command. Remember this?

```
IF money)0 THEN PRINT "Rich"
  ELSE PRINT "Hard up"
```

IF-THEN-ELSE

In this command the word 'Rich' would be PRINTed if the variable 'money' had a value greater than zero, or 'Hard up' would be PRINTed if 'money' had a value of zero or less. In Chapter 6 of Part 1 you were told how to put several commands on a single line so long as they were separated by colons. We did this earlier in this chapter.

Until now you may well have thought that there is no difference between putting lots of commands on a single line or on separate lines. This is correct so long as one of those commands is not an IF command. Whether or not any commands following an IF statement on a multistatement line are executed depends on whether the condition was true or false.

Consider

```
IF money>0 THEN PRINT "Rich" ELSE PRINT
"Hard up":PRINT"Lend me some money please"
```

and

```
IF money>0 THEN PRINT "Rich" ELSE PRINT
"Hard up"
PRINT "Lend me some money please"
```

You may think there is no difference between these two arrangements, but you would be wrong. In the lower example, 'Lend me some money please' would be PRINTed whatever the value of the variable 'money'. *But* in the upper example this is not the case. 'Lend me some money please' will only be PRINTed if 'money' is zero or less, just as 'Hard up' will be. This is a very important point to remember and can be put like this: *When the IF condition is true, only the statements after the THEN and up to the ELSE are executed; when the condition isn't true, only the statements after the ELSE are executed.*

Referring to line 1180 of DIGITALC we can see that if the variable 'ht' has a value of 1 it will be changed to 0, the variable 'x' will be set to 1, and a space will be printed at column 7, line 10. However, if the value of 'ht' is not 1 then it will be set to 1, 'x' will be set to 0 and the subscripted variable nS(ht) will be printed at column 7, line 10.

Conditional multistatement lines are very useful when used correctly and save a lot of extra lines in the program.

Another point to make concerning DIGITALC is the use of subscripted string variables for the printing of the numbers that make up the clock. Why not use the values of the FOR variables itself? The answer is that when the CPC464 prints a number it always puts a space in front and a space after it. Thus you cannot PRINT one number directly after another as the first

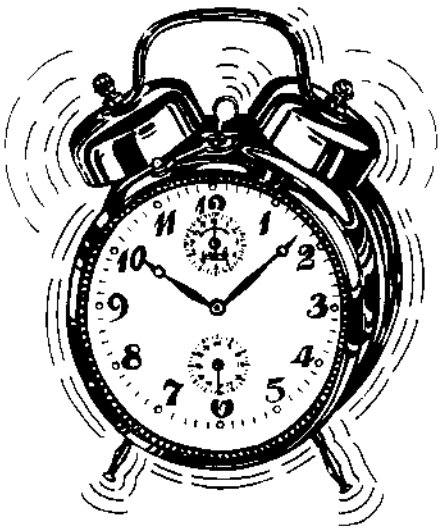
will be partly erased by the second one's leading space. When a string variable is PRINTed, however, there are no leading or trailing spaces. It is therefore much easier to set up the string array and use those characters for the numbers than chopping the spaces off the actual numbers (which can be done, see Chapter 7).

Note that the program DIGITALC does not make any checks on the input you make. It will accept silly times like 10:76 or 35:15. In a later chapter we will explain how to trap errors by inserting lines that check the inputs. Finally, you may be wondering why this clock does not make use of the CPC464's own built-in clock (which can be accessed by the TIME keyword). The answer is that the use of TIME calls for some fancy string handling and chopping which will be explained in Chapter 7. Suffice it to say that if you type PRINT TIME, the number you get is the number of 1/300ths of a second that have elapsed since the computer was switched on or reset (not including cassette use).

PRINT TIME/300 will give you seconds and PRINT TIME/300/60 will give you minutes, of course. Using TIME will make the clock more accurate, but for now find out how accurate the present version is.

ALARM

ALARM is the next program on Datacassette A. It makes use of a lot of commands we have looked at in this chapter, and quite a few others besides. You really can set it to wake you up in the morning, or you can use it to teach your little sister to tell the time!



Although this program contains some rather advanced programming, you may like to list it on your screen to see how it is put together.

TESTING

Yet another chance to check up on your progress through this course. Don't worry if you keep having to go back over things – that's what learning is all about, so load up SAT4 and away you go.

Chapter 5
ALL WORK AND NO PLAY

In the early days of computers, they were nearly all bought by commercial organisations and used almost exclusively for processing such things as stock levels, invoices and payrolls. There was one company that tried to make all its personnel aware of this by putting the following message on the screen every time someone signed on at the computer:

The company computers are for business purposes only

The idea was to stop people from playing computer games in company time. It didn't work!

Since computers have now become available at a price ordinary people can afford, the problem hardly arises any more. Few commercial computers have anything like the colour graphics capability of the CPC464, let alone the sound commands.

The real irony is that a CPC464 which has been bought for playing and programming computer games can also be used for running business software. It could well be the only computer that a small businessman will ever need!

BUSINESS PROGRAMMING

Very few business users of computers know, or indeed want to know, anything about programming. For them the computer is just another part of their business equipment and not the main object. Professional decorators don't need to know the chemical composition of paints they use to produce a good job, any more than they have to make their own brushes! For this reason the design of such programs is considerably different from those which hobbyists produce for their own pleasure.

The first difference is in the design of the screen instructions to

the user. For a business user they must be clear and straightforward, and any INPUT statements should give the nature and range of the required data.

The other difference is the need for a 'User Guide' for the program. Even though the screen design will help users through the data entry stages, users would find it tedious if they had to sit through a long explanation of what was required each time the program was run. Many people also prefer to have a quick skim through the program description before actually sitting down at the keyboard. In addition, we all find it helpful to have some form of reference material at our elbow whatever we are doing.

Most of the rest of this chapter is about a program called ESTIM; this will give you a taste of what business software is like. Don't be put off by the idea. Although Amstrad have never made the inflated claim that the CPC464 could run a nuclear power station, there are many ways in which it could help you at home, or at school, or at work.

SCREEN DESIGN

Getting the facts in

Here is another old friend from Part 1:

```
INPUT [<quoted string>;]
  [<list of variable>]
```

As you can see, it is possible to have more than one numeric or string variable to a single INPUT command. This is done by using a comma as a separator as in the following example:

```
INPUT "Enter four numbers";a,b,c,d
```

In this form the command puts a question mark at the end of the prompt string and leaves the cursor one space to the right of it. When entries are now made on the keyboard, separated by commas, the CPC464 will attempt to assign them to the successive variables in the list. If the entries are not of the same type, e.g. string instead of numeric, or if insufficient entries have been made, it will give the message

```
Redo from start
```

and repeat the prompt.

If you replace the semi-colon after the prompt string with a comma, the question mark and space will be suppressed.

PRINT

An even older friend is PRINT. Whereas INPUT is the *main* way we have for the user to talk to the program, PRINT is the *only* method for the program to talk to user. Apart from the cursor controls of LOCATE and TAG, the use of the PRINT command is the thing that can make or mar the screen presentation.

The usual syntax is:

```
PRINT (print item)[(separator)(print item)
  ...][separator]
```

As you will remember from Part 1, the separator may be either a comma, which will cause the cursor to skip to the beginning of the next print zone before handling the next <print item> (we'll come to the ZONE command later on!), or a semi-colon, which will append <print item>s. When there is a separator at the end of the command it also stops the next PRINT command from being started on a new line.

SPC

TAB

The <print item> can be one of three things:

```
(expression)

SPC((integer expression))

TAB((integer expression))
```

SPC and TAB may appear at first sight to carry out the same function, since they are both used to automatically generate spaces between <print item>s. The difference is that SPC will insert the number of spaces given by the <integer expression> starting from the current position of the text cursor, whereas TAB counts the number of print positions from the beginning of the line.

If the number of spaces specified by SPC is greater than the line width, the CPC464 automatically subtracts the number of print positions per line (a suitable number of times if the value is greater than one line width) and then gives the resulting number of spaces. SPC need not be followed by a semi-colon as a separator since one is automatically assumed, even when SPC is the last item in the PRINT list.

When the print position given by a TAB is greater than the current cursor position, spaces are generated until the required position is reached. (Nothing happens if the cursor is already at the required position.) Should the required position have already been passed, a new line is started and spaces generated on it until the required position is reached. A semi-colon separator is assumed automatically in the same way as SPC.

SPACE$

STRING$

There are also two string functions which can be used as the <expression> in a print command: SPACE$ and STRING$.

SPACE$ is nearly, but not quite, the same as SPC since, if the number specified in its argument is greater than the number of remaining print positions on the current line, a new line is generated and the spaces continued along it. The maximum number of spaces is 255.

Strings of spaces are often used for erasing text without using CLS. It is easier to type

```
PRINT SPACE$(40)
```

than

```
PRINT "                                        "
```

Finally, there is STRING$ that will give you a string full of any one character:

```
STRING$ (integer expression)
```

The <integer expression> is the number of characters you want in the string. The <expression> is either an integer giving the ASCII value of the character you want, or you can put in a string directly:

```
PRINT STRING$(10,90)          gives    ZZZZZZZZZZ

PRINT STRING$(16,"=")         gives    ================

PRINT STRING$(7,"CPC464")     gives    CCCCCCC
```

ZONE

Here is the ZONE command as promised:

```
ZONE(integer expression),(expression)
```

Not very complicated really - it resets the width of the print zones across the screen to the value given by <integer expression> (the default is 13 - remember?). This is a useful command which makes tables of numbers easier to handle.

Laying the facts out

The above new facets of the PRINT command are all to do with positioning words and numbers on the screen. Even so, they still can't handle all aspects of screen design without a lot of extra programming. This is why Amstrad BASIC has a very powerful option known as PRINT USING. Here is the syntax:

```
PRINT USING (string expression);(print list)
```

The <print list> is the list of <print item> with separators as for

the ordinary PRINT command, but the <string expression> is known as the 'Format template', and is used, for example, as follows:

PRINT USING

```
PRINT USING "###.##";finaltotal
```

If 'finaltotal' was calculated to be 87.4473, the number that appeared on the screen would be:

```
87.45
```

What happens here is that as long as the expression to be printed is of the right sort (numeric in this case), the CPC464 will print it with the decimal point in the correct position on the line and rounded to the number of digits printed.

The '.' and '#' are known as 'format field specifiers'. There are eleven of these, but by now you are probably getting impatient to get back to the keyboard so we'll leave the explanation of these until later (there is a list of them on page 55, Chapter 8 of the User Guide).

ESTIMATING WITH ESTIM

ESTIM is an example of a simple estimating program that could be used by a professional painter and decorator, or a DIY enthusiast. It is actually a lot more sophisticated than the usual method used in the trade, which is to give a fixed price per room (based on experience, of course) and hope that it all balances out, but is still not overkill for a small business. The principle is that *all* materials and times are related to the average size of the rooms in a house, and to the number of rooms – even the outside work.

To use computer jargon, this program is 'menu driven'; this means that the user chooses between different procedures by selecting them from a screen display known as a 'menu'. The menu routine can be seen starting at line 640 following the initialisation, screen and heading routines. This is the main loop of the program and it directs the user to the routines for five options. Apart from 'End program', these options are either input procedures which ask the user to enter a series of values, or output procedures, which give a display of standard data or results.

Note that the default values used for calculating estimates are contained in the data list starting at line 3740. These values are suggestions and *not* immutable facts. Please check these figures against current prices and wage rates before committing yourself to decorating a 57-bedroom palace in Monte Carlo!

Here is the listing for your perusal.

```
100 '
110 ' Estim
120 '
130 ' DA 29/9/84  DGC 28/11/84
140 '
150 ' Initialise
160 '
170 CLEAR
180 tb%=45 ' column for input of replies
190 true=-1:false=0
200 beep$=CHR$(7):bs$=CHR$(8):lf$=CHR$(10):cr$=CHR$(13)
210 x$=" "+bs$+CHR$(24)+" " : x1$=" "+CHR$(24)
220 s1$=CR$+lf$+SPACE$(5) : s2$=SPACE$(10)
230 cp.roll=12 'a roll covers 12 sq m
240 DEF FNcentre$(a$)=SPACE$(40-LEN(a$)/2)+a$
250 DEF FNeven(x,y)=ROUND( x/y ) * y  'round x to nearest y
260 '
270 ' Initialise defaults
280 '
290 DIM m$(7),m(7),mu$(7),c$(4),c(4),a$(6),a(6),l$(1),
l(1) ' optional
300 RESTORE 3790
310 FOR i=0 TO 7 : READ m$(i),m(i),mu$(i) : NEXT
320 FOR i=0 TO 4 : READ c$(i),c(i) : NEXT
330 FOR i=0 TO 6 : READ a$(i),a(i) : NEXT
340 FOR i=0 TO 1 : READ l$(i),l(i) : NEXT
350 READ rph$,rph
360 '
370 ' Screen
380 '
390 MODE 2  : BORDER 0
400 INK 0,0 : INK 1,22
410 PAPER 0 : PEN 1
420 '
430 h$="Decorating Estimator"
440 GOSUB 1860 ' heading
450 '
460 ' Menu
470 '
480 RESTORE 3760
490 FOR i=1 TO 5
500 READ t$
510 LOCATE 28,i*2+5
520 PRINT i;". ";t$
530 NEXT
540 LOCATE 32,20 : PRINT CHR$(18);
550 INPUT "Select (1-5) : ",i$ : i=VAL(i$)
560 IF i(1 OR i)5 THEN PRINT beep$; : GOTO 540
570 ON i GOSUB 670,2360,2660,1670,630
580 '
```

```
590 GOTO 430
600 '
610 ' End program
620 '
630 LOCATE 1,22 : END
640 '
650 ' SUBROUTINE : Input job details
660 '
670 h$="Inout details"
680 GOSUB 1860 ' heading
690 LOCATE 1,5
700 '
710 ' General
720 '
730 hd$="Name of client":F%=79-tb%:GOSUB 2140:client$=v$
740 hd$="Address (no commas please)":F%=79-tb%:GOSUB 2140:
address$=v$
750 GOSUB 1810 : total.rooms = rooms
760 PRINT
770 '
780 ' Capture data
790 '
800 hd$=x$+"INSIDE DECORATION"+x1$:GOSUB 2240:id.flag=v%
810 IF id.flag=false GOTO 1150 ' no inside dec
820 '
830 ' Ceilings
840 '
850 hd$=s1$+x$+"Ceilings"+x1$:GOSUB 2240:ce.flag=v%
860 IF ce.flag=false GOTO 930
870 hd$=s2$+"Ceilings papered+emulsioned":GOSUB 2040:
ce.pe=v%
880 hd$=s2$+"Ceilings emulsioned only":GOSUB 2040:ce.e=v%
890 ce.number=ce.pe+ce.e
900 ce.area.pap=ce.pe*a(0) : ce.area.emul=(ce.pe+ce.e)*a(0)
910 ce.cost=(ce.area.pap * ( m(1)/cp.roll ) ) +
(ce.area.emul * (m(2)/c(2)) )
920 ce.time=(ce.area.pap / l(0)) + (ce.area.emul / l(1))
930 '
940 ' Walls
950 '
960 hd$=s1$+x$+"Walls"+x1$:GOSUB 2240:wa.flag=v%
970 IF wa.flag=false GOTO 1040
980 hd$=s2$+"Rooms papered only":GOSUB 2040:wa.p=v%
990 hd$=s2$+"Rooms emulsioned only":GOSUB 2040:wa.e=v%
1000 wa.number=wa.p+wa.e
1010 wa.area.pap=wa.p*a(1) : wa.area.emul=wa.e*a(1)
1020 wa.cost=(wa.area.pap *  (m(0)/cp.roll ) ) +
(wa.area.emul * (m(2)/c(2)) )
1030 wa.time=(wa.area.pap / l(0)) + (wa.area.emul / l(1))
1040 '
1050 ' Woodwork
```

```
1060
1070 hd$=s1$+x$+"Woodwork"+x1$:GOSUB 2240:ww.flag=v%
1080 IF ww.flag=false GOTO 1150
1090 hd$=s2$+"Rooms undercoated+topcoated":GOSUB 2040:
ww.ut=v%
1100 hd$=s2$+"Rooms topcoated only":GOSUB 2040:ww.t=v%
1110 ww.number=ww.ut+ww.t
1120 ww.area.under=ww.ut*a(2) : ww.area.top=(ww.ut+ww.t)
*a(2)
1130 ww.cost=(ww.area.under * m(3)/c(0) ) + (ww.area.top *
m(4)/c(1) )
1140 ww.time=(ww.area.under+ww.area.top) / l(1)
1150 '
1160 ' Outside decoration
1170 '
1180 PRINT : PRINT
1190 hd$=x$+"OUTSIDE DECORATION"+x1$:GOSUB 2240:od.flag=v%
1200 IF od.flag=false GOTO 1520 ' no outside dec
1210 '
1220 ' Windows and Doors
1230 '
1240 hd$=s1$+x$+"Windows and doors"+x1$:GOSUB 2240:
wd.flag=v%
1250 IF wd.flag=false GOTO 1340
1260 wd.cost=0:wd.time=0
1270 doors.and.wind=total.rooms * a(3)
1280 hd$=s2$+"All Varnished only?":GOSUB 2240
1290 IF v% THEN wd.cost=doors.and.wind * m(6)/c(3) :
wd.time=doors.and.wind / l(1):GOTO 1340
1300 hd$=s2$+"All Undercoated+topcoated?":GOSUB 2240
1310 IF v% THEN wd.cost=doors.and.wind *( m(3)/c(0)
+m(5)/c(1) ):wd.time=doors.and.wind*2 / l(1):GOTO 1340
1320 hd$=s2$+"All Topcoated only?":GOSUB 2240
1330 IF v% THEN wd.cost=doors.and.wind * m(5)/c(1) :
wd.time=doors.and.wind / l(i)
1340 '
1350 ' Guttering & Downpipes
1360 '
1370 hd$=s1$+x$+"Guttering and downpipes"+x1$:GOSUB 2240:
gd.flag=v%
1380 IF gd.flag=false GOTO 1450
1390 gd.cost=0:gd.time=0
1400 gd.length=total.rooms * a(4)
1410 hd$=s2$+"All Undercoated+topcoated?" : GOSUB 2240
1420 IF v% THEN gd.cost=gd.length *( m(3)/c(0) + m(5)/
c(1) ):gd.time=gd.length*2 / l(i) :GOTO 1450
1430 hd$=s2$+"All Topcoated only?" : GOSUB 2240
1440 IF v% THEN gd.cost= gd.length * m(5)/c(1) :gd.time
=gd.length / l(i)
1450 '
1460 ' Outside Walls
```

```
1470 '
1480 hd$=s1$+x$+"Outside walls"+x1$:GOSUB 2240:ow.flag=v%
1490 IF ow.flag=false THEN  1520
1500 ow.cost = total.rooms * a(5) * ( m(7)/c(4) )
1510 ow.time = total.rooms * a(5) / l(1)
1520 '
1530 ' Rounding of variables
1540 '
1550 ce.cost=FNeven(ce.cost,0.01) : ce.time=FNeven(ce.tim
0.25)
1560 wa.cost=FNeven(wa.cost,0.01) : wa.time=FNeven(wa.tim
0.25)
1570 ww.cost=FNeven(ww.cost,0.01) : ww.time=FNeven(ww.tim
0.25)
1580 wd.cost=FNeven(wd.cost,0.01) : wd.time=FNeven(wd.tim
0.25)
1590 gd.cost=FNeven(gd.cost,0.01) : gd.time=FNeven(gd.tim
0.25)
1600 ow.cost=FNeven(ow.cost,0.01) : ow.time=FNeven(ow.tim
0.25)
1610 '
1620 PRINT : PRINT : GOSUB 1950 ' spacebar
1630 RETURN
1640 '
1650 ' SUBROUTINE : Prepare for estimate output
1660 '
1670 h$="Print Estimate" : GOSUB 1860 ' heading
1680 ' send to screen
1690 '
1700 str = 0 : GOSUB 3100
1710 '
1720 ' Send to printer
1730 '
1740 REM:  str = 8 : GOSUB 3100  (disabled)
1750 '
1760 RETURN
1770 '
1780 ' SUBROUTINE : Ask the number of rooms
1790 '                return answer in 'rooms'
1800 '
1810 hd$="Total number of rooms" : F% = 2
1820 GOSUB 2040 : rooms=v% : RETURN
1830 '
1840 ' SUBROUTINE : Print heading
1850 '
1860 CLS
1870 temp1=LEN(h$):temp2=(80-temp1)/2
1880 PRINT TAB(temp2);CHR$(24);STRING$(temp1+2,131);CHR$(
1890 PRINT TAB(temp2);CHR$(24);" ";h$;" ";CHR$(24)
1900 PRINT TAB(temp2);CHR$(24);STRING$(temp1+2,140);CHR$(
1910 RETURN
```

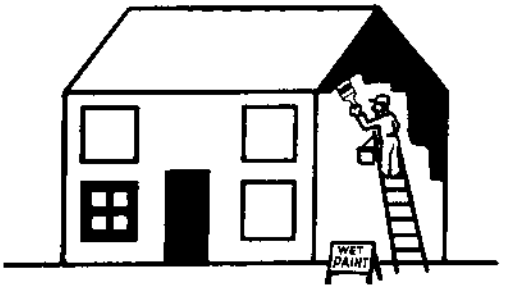
```
1920 '
1930 ' SUBROUTINE : Wait for the spacebar (debounced)
1940 '
1950 LOCATE 1,25
1960 PRINT FNcentre$("Press "+CHR$(24)+" SPACE "+CHR$(24)
+" to continue");
1970 WHILE INKEY$ > "" : WEND
1980 WHILE INKEY$()" " : WEND
1990 RETURN
2000 '
2010 ' SUBROUTINE : Input routine (number)
2020 '               returns number in V%
2030 '
2040 PRINT hd$;TAB(tb%);
2050 PRINT"[";
2060 PRINT STRING$(F%," ");"] number";STRING$(F%+8,bs$);
2070 INPUT "",V%
2080 RETURN
2090 '
2100 ' SUBROUTINE : Input routine (string)
2110 '               returns string in V$
2120 '               Field length in F%
2130 '
2140 PRINT hd$;TAB(tb%);
2150 PRINT "[";
2160 LOCATE tb%+F%,VPOS(#0):PRINT"]";
2170 LOCATE tb%+1 ,VPOS(#0)
2180 INPUT "",V$
2190 RETURN
2200 '
2210 ' SUBROUTINE : Yes/No routine
2220 '               if yes then v%=true else v%=false
2230 '
2240 PRINT hd$;TAB(tb%);
2250 v%=1
2260 PRINT"[ ]  Y - yes  or N - no"STRING$(35,8);
2270 g$=INKEY$:IF g$="" GOTO 2270
2280 IF UPPER$(g$)="Y" THEN v%=true : PRINT"Y";STRING$
(1,bs$);:GOTO 2320
2290 IF UPPER$(g$)="N" THEN v%=false: PRINT"N";STRING$
(1,bs$);:GOTO 2320
2300 IF v%(1 AND g$=cr$ THEN PRINT:RETURN
2310 PRINT beep$;
2320 GOTO 2270
2330 '
2340 ' SUBROUTINE : Display defaults
2350 '
2360 GOSUB 2610
2370 PRINT : PRINT"MATERIALS";TAB(40);"COST";TAB(55);
"UNIT" : PRINT
2380 FOR i=0 TO 7
2390 PRINT " ";m$(i);
```

```
2400 PRINT TAB(40);USING "##.##";m(i);
2410 PRINT TAB(55);mu$(i)
2420 NEXT
2430 PRINT : PRINT"COVERING POWER";TAB(40);"SQ METRES/
LITRE": PRINT
2440 FOR i=0 TO 4
2450 PRINT " ";c$(i);TAB(40);USING "###.#";c(i)
2460 NEXT
2470 GOSUB 1950 ' Next screen
2480 GOSUB 2610
2490 PRINT : PRINT"AREAS";TAB(40);"SQ METRES/ROOM" : PRINT
2500 FOR i=0 TO 5
2510 PRINT " ";a$(i);TAB(40);USING "###.#";a(i)
2520 NEXT
2530 PRINT : PRINT "MAN-HOURS";TAB(40);"SQ METRES/HOUR" :
PRINT
2540 FOR i=0 TO 1
2550 PRINT " ";l$(i);TAB(40);USING "###.#";l(i)
2560 NEXT
2570 PRINT : PRINT "RATE / HOUR";TAB(40);"POUNDS/HOUR" :
PRINT
2580 PRINT " ";rph$;TAB(40);USING "##.##";rph
2590 GOSUB 1950 ' space
2600 RETURN
2610 h$="Costings" : GOSUB 1860 ' heading
2620 RETURN
2630 '
2640 ' SUBROUTINE : Amend Material Costs
2650 '
2660 GOSUB 3000
2670 PRINT : PRINT"MATERIALS";TAB(40);"COST";TAB(50);
"NEW COST";TAB(65);"UNIT" : PRINT
2680 FOR i=0 TO 7
2690 PRINT " ";m$(i);TAB(40);USING "##.##";m(i);
2700 PRINT TAB(65);mu$(i)
2710 LOCATE 55,VPOS(#0)-1 : INPUT "",n
2720 IF n>0 THEN m(i)=n
2730 NEXT
2740 PRINT : PRINT"COVERING POWER";TAB(40);"SQ METRES/
LITRE";TAB(60);"NEW VALUE": PRINT
2750 FOR i=0 TO 4
2760 PRINT " ";c$(i);TAB(40);USING "###.#";c(i);
2770 PRINT TAB(60);:INPUT "",n
2780 IF n>0 THEN c(i)=n
2790 NEXT
2800 GOSUB 1950 ' Next screen
2810 GOSUB 3000
2820 PRINT : PRINT"AREAS";TAB(40);"SQ METRES/ROOM";TAB(60
"NEW VALUE" : PRINT
2830 FOR i=0 TO 5
2840 PRINT " ";a$(i);TAB(40);USING "###.#";a(i);
2850 PRINT TAB(60); : INPUT "",n
```

```
2860 IF n)0 THEN a(i)=n
2870 NEXT
2880 PRINT : PRINT "WORK-RATES";TAB(40);"SQ METRES/HOUR";
TAB(60);"NEW VALUE"
2890 PRINT
2900 FOR i=0 TO 1
2910 PRINT " ";1$(i);TAB(40);USING "###.#";l(i);
2920 PRINT TAB(60); : INPUT "",n
2925 IF n)0 THEN l(i)=n
2930 NEXT
2940 PRINT : PRINT "RATE / HOUR";TAB(40);"POUNDS/HOUR";
TAB(60);"NEW RATE"
2950 PRINT
2960 PRINT " ";rph$;TAB(40);USING "##.##";rph;
2970 PRINT TAB(60); : INPUT "",n
2975 IF n)0 THEN rph=n
2980 GOSUB 1950 ' space
2990 RETURN
3000 h$="Amend Costings" : GOSUB 1860 ' heading
3010 PRINT"      Use ENTER to skip"
3020 RETURN
3030 '
3040 ' SUBROUTINE : Print out estimate
3050 '            : str=stream to print to
3060 '            : ta - te = tab settings
3070 '
3080 ' Main heading
3090 '
3100 ta=0 : tb=5 : tc=25 : td=45 : te=65 ' column positions
on report
3110 IF (id.flag = false) AND (od.flag = false) THEN PRINT
#str,"" : RETURN ' no figures
3120 PRINT #str,FNcentre$("Estimate of time and materials
for redecorating work at:")
3130 PRINT #str
3140 PRINT #str,FNcentre$(address$) : PRINT #str
3150 PRINT #str,FNcentre$("for "+client$) : PRINT #str
3160 '
3170 ' Inside work
3180 '
3190 id.cost=0 : id.time=0
3200 IF id.flag=false GOTO 3350
3210 PRINT #str,TAB(ta);"Inside work :" : PRINT #str
3220 PRINT #str,TAB(tb);"WORK";TAB(tc);"NO OF ROOMS";
TAB(td);"MATERIAL COSTS";
3230 PRINT #str,TAB(te+1);"TIME"
3240 PRINT #str,TAB(tb);"----";TAB(tc);"----------";
TAB(td);"---------------";
3250 PRINT #str,TAB(te+1);"----" : PRINT #str
3260 IF ce.flag THEN n=0 : n1=ce.number : c1=ce.cost :
t1=ce.time : GOSUB 3670 : id.cost=id.cost+ce.cost : id.time
```

```
=id.time+ce.time
3270 IF wa.flag THEN n=1 : n1=wa.number : c1=wa.cost : t1=
wa.time : GOSUB 3670 : id.cost=id.cost+wa.cost : id.time=
id.time+wa.time
3280 IF ww.flag THEN n=2 : n1=ww.number : c1=ww.cost : t1=
ww.time : GOSUB 3670 : id.cost=id.cost+ww.cost : id.time=
id.time+ww.time
3290 PRINT #str
3300 PRINT #str,TAB(td);"-------"; TAB(te-2);"--------"
3310 PRINT #str,TAB(tc);"Totals:";
3320 n=6 : n1=0 : c1=id.cost : t1=id.time : GOSUB 3670
3330 PRINT #str,TAB(td);"======="; TAB(te-2);"========"
3340 PRINT #str : PRINT #str
3350 '
3360 ' Outside
3370 '
3380 od.cost=0 : od.time=0
3390 IF od.flag=false GOTO 3520
3400 PRINT #str,TAB(ta);"Outside work :" : PRINT #str
3410 PRINT #str,TAB(tb);"WORK"; TAB(td);"MATERIAL COSTS";
TAB(te);"TIME"
3420 PRINT #str,TAB(tb);"----"; TAB(td);"--------------";
TAB(te);"----"
3430 PRINT #str
3440 IF wd.flag THEN n=3 : n1=0 : c1=wd.cost : t1=wd.time
GOSUB 3670 : od.cost=od.cost+wd.cost : od.time=od.time+
wd.time
3450 IF gd.flag THEN n=4 : n1=0 : c1=gd.cost : t1=gd.time
GOSUB 3670 : od.cost=od.cost+gd.cost : od.time=od.time+
gd.time
3460 IF ow.flag THEN n=5 : n1=0 : c1=ow.cost : t1=ow.time
GOSUB 3670 : od.cost=od.cost+ow.cost : od.time=od.time+
ow.time
3470 PRINT #str,TAB(td);"-------"; TAB(te-2);"--------"
3480 PRINT #str,TAB(tc);"Totals:";
3490 n=6 : n1=0 : c1=od.cost : t1=od.time : GOSUB 3670
3500 PRINT #str,TAB(td);"======="; TAB(te-2);"========"
3510 PRINT #str : PRINT #str
3520 '
3530 t.cost=id.cost+od.cost : t.time=id.time+od.time
3540 PRINT#str,TAB(tb);"Grand Totals";
3550 n=6 : n1=0 : c1=t.cost : t1=t.time :GOSUB 3670
3560 PRINT #str,TAB(tb);"Labour Cost";TAB(td);USING
"####.##";t1*rph;
3570 PRINT #str
3580 PRINT #STR,TAB(td-2);"==========";
3590 PRINT #str,TAB(tb);"Total Estimated";TAB(td);USING
"####.##";t.time*rph+t.cost
3600 PRINT #str : PRINT #str
3610 '
3620 GOSUB 1950 : RETURN
```

```
3630 '
3640 ' SUBROUTINE : Output line for use with est. printer
3650 '               needs n for a$,n1,c1,t1 : if n1=0 then
                     don't print
3660 '
3670 IF a$(n))"" THEN PRINT #str,TAB(tb);a$(n);
3680 IF n1>0 THEN PRINT #str,TAB(tc);USING "##";n1;
3690 PRINT #str,TAB(td);USING "####.##";c1;
3700 PRINT #str,TAB(te);USING "###.##";t1;
3710 PRINT #str," hrs."
3720 RETURN
3730 '
3740 ' Data list
3750 '
3760 DATA Input job details,Display costings,Amend costings
3770 DATA Print customers estimate,End program
3780 '
3790 DATA wallpaper,3.00,roll (12x1m)
3800 DATA ceiling paper,3.00,roll (12x1m)
3810 DATA emulsion,2.72,litre
3820 DATA undercoat,2.81,litre
3830 DATA topcoat (inside),3.05,litre
3840 DATA topcoat (outside),3.05,litre
3850 DATA varnish,11.20,litre
3860 DATA wall paint,2.72,litre
3870 '
3880 DATA undercoat,14.5,topcoat,15,emulsion,14
3890 DATA varnish,22,wall paint,17
3900 '
3910 DATA ceilings,9,inside walls,34,inside woodwork,3
3920 DATA outside windows and doors,3,guttering etc,0.5
3930 DATA outside walls,16,"",0
3940 '
3950 DATA papering,6,painting,1.5
3960 '
3970 DATA Rate per hour,5.50
```



ESTIM USER'S GUIDE

Well, we did say at the beginning of this chapter that the guide book was an essential part of a business program, didn't we? Without labouring the point too much, the following few paragraphs are the sort of thing that could help a non-programmer to use the ESTIM program without getting tied into knots.

Starting up

When you load and run the ESTIM program a menu is put up on the screen from which you may select the procedure you require. You are returned to this screen at the end of procedures 1-4 so that you may make a new selection.

Note that this program only stores information for the time that the system is running. Be careful before selecting procedure 5, 'End program', that there is nothing you want to record, since running the program again will reset everything.

Entering the job details

First you enter the customer's name and address as requested by the screen prompts. You are then asked for the number of the rooms in the house for which an estimate is required. Include the kitchen, bathroom, toilet and large hallways, but you can exclude cupboards and pantries.

After answering yes to any of the 'yes-no' questions, enter the number of rooms that require each particular type of work. You can either enter zero or press the ENTER key if that type of work doesn't apply to any room.

Costings

Option 2 displays the quantities, prices, hours, etc., on which the estimate will be based.

Amending the costings

This option (No. 3) puts each costing on the screen, one after the other, so that you can enter a new value. You can skip over the ones you don't want to change by pressing the ENTER key.

These changes are only retained for the time the program is being run.

Printing the estimate

You print the estimate on the screen by selecting option 4. This is a summary of the estimated costs, calculated from the information you entered as job details and the current costings. Note that only one estimate can be handled at a time, and that putting in a new set of job details will completely erase the previous estimate.

It is possible, however, to go back to option 3 (*without* ending the program) and change some of the costings – a cheaper grade of wallpaper, for example – and print a new version of the same estimate.

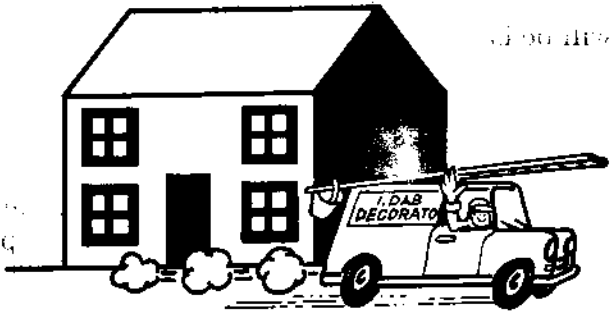
FURTHER IMPROVEMENTS

There isn't a program yet written that can't be improved or extended in some way. ESTIM is no exception. In addition, business programs are constantly being modified and adapted to the changing needs of their users. No doubt you will want to do the same thing to this one.

Here are a few things you may like to consider as further improvements to ESTIM:

- ☐ Allow for type of house (semi-detached, bungalow, etc.)
- ☐ Record paper and colour scheme per room
- ☐ Calculate total paint and paper requirements
- ☐ Add paint and paper stripping
- ☐ Add cost of hiring ladders or other equipment

You can probably think up another half-dozen for yourself, although the first priority is to write a new option that will give you printed output on your Amstrad DMP-1 dot matrix printer. Some guidance on this is given in Chapter 12.



Don't be too ambitious though. Computers have often acquired a bad name simply because an over-enthusiastic programmer has made the system so complicated to use that it is much quicker to work out estimates on the back of an envelope!

TESTING

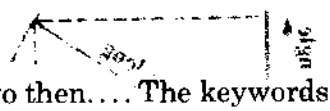
This really *has* been all work and no play, hasn't it? For this reason we are being kind and not giving you a SAT for this chapter.

Chapter 6
BACK TO SKOOL

Those of you lucky enough to have done mathematics to 'O' Level or CSE level and beyond will have heard all about things like sines, cosines, tangents, radians, logarithms and exponentials. The rest of you have that pleasure to come!

This chapter is not intended to be a maths lesson - after all you can buy some very good maths books - but it will be necessary to indulge a little in order to explain the maths functions the CPC464 supports. The CPC464 is a clever beast and can do some pretty fast number-crunching, but a lot of computer programs do not use actual maths functions at all, just +, -, ★ and /. However, you will need to know about things like SIN, LOG and EXP in order to know when to use them (or when not to).

TRIG



So here we go then.... The keywords SIN, COS, TAN and ATN all deal with angles and are trigonometric (trig for short) functions. In any calculations to do with angles, the CPC464 assumes, by default, that your angles are measured in radians and not degrees. A radian is a special angle and is equal to 57.295 degrees; there are 2★PI of them (6.2838) in a circle. Strangely enough, working in radians sometimes makes things easier. It is most likely that you will want to work in degrees, °, however, and so to save you lots of tedious conversion, the CPC464 has the keyword DEG which we met in Chapter 3. It tells the CPC464 to work in degrees. There is another keyword, RAD, that tells it to work in radians again.

Type in the following program:

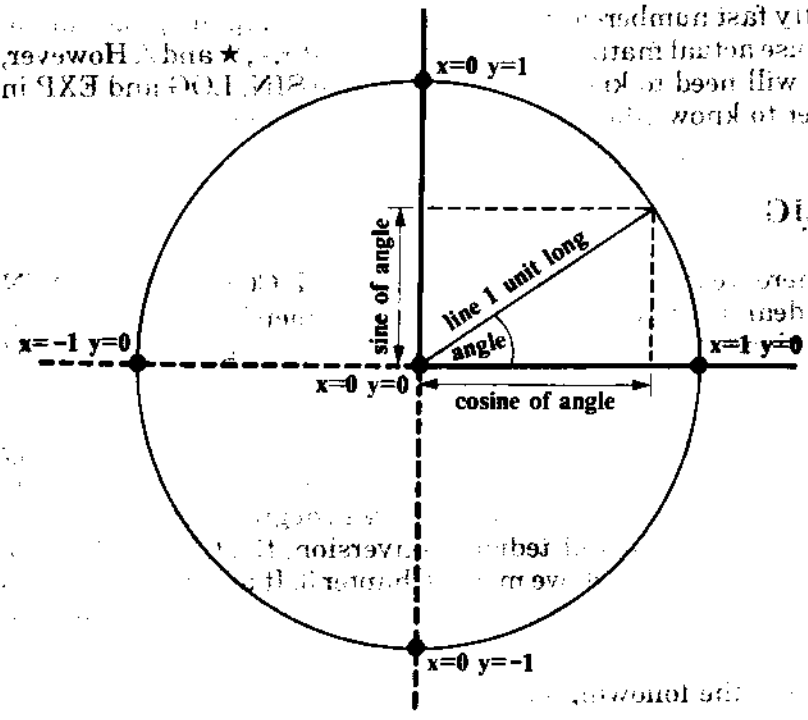
```
5 DEG
10 ORIGIN 300,200
20 FOR n=1 TO 360
30 x=SIN (n)
40 y=COS (n)
50 PLOT 100*x,100*y
60 NEXT n
```

Let us now use this program to demonstrate a few points.

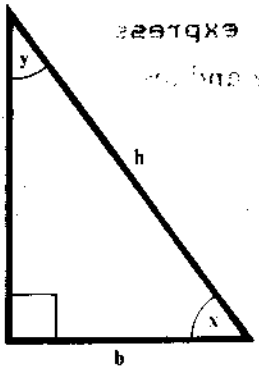
The ORIGIN command moves the point 0,0 to the centre of the screen more or less (as was explained in Chapter 3). The variable 'n' is going to be incremented by one, starting at 1 and finishing with 360. The sines and cosines of each value of 'n' (remember SINCOS) are multiplied by 100 and PLOTted in turn.

Now RUN and you will see the circle PLOTted degree by degree.

For those of you who are wondering how our circle program works, the following diagram will help.



SINes, COSines and TANgents are most often used for the solution of triangles, i.e. given some information about a triangle you can find out all the rest. These trig functions represent the ratios between the lengths of the different sides. In a right-angled triangle (one angle = 90°), the sine of either of the other two angles is found by dividing the length of the side opposite the angle by the length of the longest side, which is called the hypotenuse.



In Amstrad BASIC, as in fact in maths books, instead of 'The sine of the angle x', we write SIN(x), just as we write 'The square root of x', as SQR(x).

Referring to the diagram,

sin(x)=a/h

and, similarly,

sin(y)=b/h.

The cosine of the angle is found by dividing the length of the side adjacent (next to) the angle by the length of the hypotenuse:

cos(x)=b/h and cos(y)=a/h

Notice that sin(x)=cos(y), and vice versa. This is a special feature of right-angled triangles and makes them easier to solve.

The tangent is found by dividing the opposite side by the adjacent side:

tan(x)=a/b and tan(y)=b/a

To solve any right-angled triangle we need know only two sides or one side and one other angle. So, if a=20 and h=50, we can find sin(x) and cos(y). If you type:

```
LET a=20: LET h=50:PRINT a/h
```

you get the answer 0.4; 0.4 is both sin(x) and cos(y):

```
SIN(x)=COS(y)=0.4
```

Now we know this, we can easily find out how big the angles 'x' and 'y' are in degrees. On a scientific hand-calculator you would probably press an INV key and then press SIN or COS to get the answers. What you are actually doing is using the inverse functions of SIN and COS. These are called arcsine and arccosine. There is also arctangent.

Of these arc functions, the CPC464 has only arctangent:

```
ATN ((numeric expression))
```

So finding the arcsine and arccosine is a little tricky but not impossible since the functions are all related. We simply convert the SINes and COSines into TANgents then use the ATN to find the angle. To save you wading through pages and pages of maths notes or slaving over a hot pencil for hours trying to work it out, here is the solution:

$$\tan(x) = \frac{\sin(x)}{\sqrt{1 - \sin^2(x)}}$$

$$\tan(x) = \frac{\sqrt{1 - \cos^2(x)}}{\cos(x)}$$

SQUARE ROOTS

The CPC464 has a function for square root. It is SQR and it is used thus: PRINT SQR(64) gives the answer 8, since 8 squared (in BASIC the notation for this is 8^2*) is 64. Type the following:

```
LET tangent=0.4/SQR(1-(0.4^2))
PRINT ATN(tangent)
```

You should get the answer 23.578, which is about 23½ degrees. Now do the same for the COSine:

```
PRINT ATN(SQR(1-(0.4^2))/0.4)
```

You should get 66.421, or just under 66½ degrees. If we now add these two answers together we should get 90 because the angles

**Note:* the character '^', meaning 'raise to the power of', appears on the keyboard and screen of the CPC464 as '↑'.

of any triangle always add up to 180°, and 90° of them are used up by the right-angled corner:

```
23.578+66.421+90=180     (close enough!)
```

So now we know the angles, all that is left is the side 'b'. We know for example that sin(y)=b/h. We can easily change this around to give b=h★sin(y). We can do the same thing to the other functions as well to produce a table:

sin(y)=b/h becomes b=h★sin(y)
cos(x)=b/h becomes b=h★cos(x)
tan(x)=a/b becomes b=a/tan(x)
tan(y)=b/a becomes b=a★tan(y)

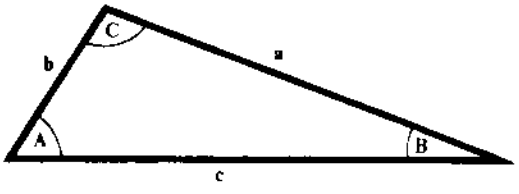
So we have four ways of finding the length of side 'b' and each should give the same answer. Try it and see! Type the following:

```
PRINT 50*SIN(66.421)     (gives 45.825...)

PRINT 50*COS(23.578)     (gives 45.825...)

PRINT 20/TAN(23.578)     (gives 45.826...)

PRINT 20*TAN(66.421)     (gives 45.824...)
```

We do indeed get the same answers, and a good thing too! That was an example of solving a right-angled triangle. As we said before, two sides, or one angle (apart from the right-angle) and one side are enough for us to figure out the rest.

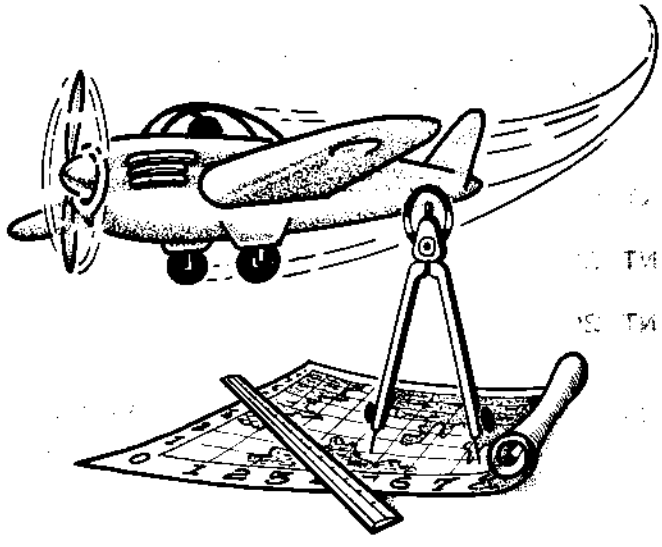
If instead we have *any* old triangle, then the situation is a little more complicated, but, given enough information we can find out all we need. However, the calculations are a bit more complicated. They make use of two rules:

- *The sine rule:* for any triangle
  $a/\sin(A) = b/\sin(B) = c/\sin(C)$
- *The cosine rule:* for any triangle
  $a^2 = b^2 + c^2 - 2bc\cos(A)$



TAKING OFF

By this stage you may well be thinking 'This is all very well but how does it apply to me and the real world?'.... OK then. Suppose you were the pilot of an aircraft wanting to fly due north to another airport. On a day with no wind you would simply fly due north to get there. However, on a windy day, which is most days, if you flew due north you would, depending on the wind direction, be likely to miss the airport because the wind would blow you off course.



Getting your bearings

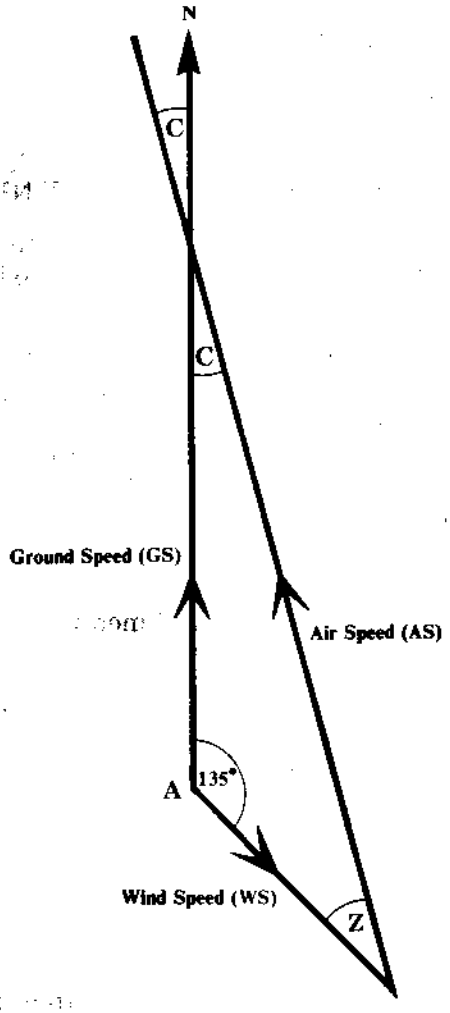
What you need is some way of calculating the course to steer in order to compensate for wind drift. The answer? Triangles!

Ever heard of vectors? A vector is a line that has a direction and magnitude. The wind can be represented by a vector; it has direction, e.g. southerly, and magnitude, e.g. force 6. An aircraft course is also a vector: direction, e.g. course 156 degrees, and magnitude, e.g. speed 315 knots.

What we do is set up a triangle of vectors whose lengths represent speed, not distance.

Refer to the diagram.... The first vector then is the direction we want to fly, due north or 0 degrees. We can draw a line straight up from our starting point A but we don't know how long the line is as we don't know what speed we can achieve. For instance, if we

are flying at 200 knots straight into the teeth of a 50 knot wind, we will only manage 150 knots compared to the ground. This speed across the ground is called, logically enough, the groundspeed to distinguish it from airspeed, which only depends on how hard our engine works.



The second vector is the wind, drawn from point A. When we say that the wind is coming from the north-west, it is equivalent to saying that it is going to the south-east. Since we are more interested in where it is going, that is what we will use. Assume a wind speed of 20 knots. The third vector to complete the triangle is going to represent the direction in which we point the aircraft, but we don't know that yet. All we do know is its length (our airspeed), say 200 knots.

So we have a triangle and we want to find two values: first our course given by the angle C, and the length of the groundspeed vector so that we can calculate how long the journey will take. So, to calculate angle C we can use the sine rule. We can put in some values:

```
SIN(C)/20 = SIN(135)/200
```

from which we can see that:

```
SIN(C) = 20*SIN(135)/200
```

Now type DEG followed by:

```
LET sinc = 20*SIN(135)/200: PRINT sinc
```

You should get the answer - 7.07107E-02, whatever that might mean! Maybe it's time you learnt how the CPC464 displays large and small numbers.

When numbers get very big or very small they become very tedious to write out so a shorthand called scientific notation is used. In this, numbers are represented as being raised to a certain power of ten. e.g. 1,000,000 (1 million) becomes 10^6 (because 10★10★10★10★10★10 = 1,000,000), 3000 becomes 3★10^3. A negative power of ten means dividing by ten that number of times. So 10^-6 is 1/10/10/10/10/10 or 0.000001, and 0.000002 becomes 2★10^-6. The CPC464 would display these as 1E6, 3E3 and 2E-06. The E means 'multiplied by ten raised to the power of'.

So our number, 7.07E-02 means 7.07★10^-2 which is 7.07★0.01 or 0.0707; we will still refer to it as 'sinc' however. Now we must convert 'sinc' into an angle. Remember the magic formula we gave you earlier? Type:

```
LET tanc = sinc/SQR(1-sinc^2):PRINT tanc
```

The value of 'tanc' is 7.08881E-02 or 0.0708881. We can now type:

```
PRINT ATN(tanc)
```

and get the answer 4.05480723. Angle C is therefore about 4.05 degrees so our course must be 4.05 degrees west of north, or 355.95 degrees as it is conventionally stated (courses are measured in degrees from north going clockwise).

Flight plan

The wind is blowing against the direction of the aircraft's forward movement to some extent, so you can expect to be slowed down a bit. How much? We must work out how long the groundspeed vector is, side GS of our triangle. Again we can use

the sine rule.

gs/sin(Z) = 200/sin(135)

so we can change that to

gs = sin(Z)★200/sin(135)

Hang on though, we don't know angle 'Z'! We can easily find it because we know the other two, and all three angles add up to 180 degrees:

Z = 180 - (135 + 4.05)

Therefore

Z = 40.95

PRINT SIN(40.95)★200/SIN(135) gives 185.375162 so our speed over the ground is a little over 185 knots. The wind slows the aircraft by nearly 15 knots. (One knot is one nautical mile per hour, which is just a bit faster than one ordinary, statute, mile per hour.) We could use the cosine rule to find the groundspeed:

$$gs = \sqrt{as^2 + ws^2 - 2as.ws.\cos(Z)}$$

Or in BASIC:

```
LET GS=SQR(200^2+20^2-(2*200*20*COS(40.95)))
```

gives 185.358424, which is just about the same as before. One point about the line above is it looks very complex. It could be split up into several shorter operations. However, the rule to remember is that brackets must be nested, like FOR-NEXT loops. Every opening bracket must have a closing bracket. The innermost brackets are operated on first and the CPC464 works its way outwards. Our angle C could be calculated like this:

$$c = \arctan \frac{\left(\frac{ws.\sin(Z)}{as}\right)}{\sqrt{\left[1 - \left(\frac{ws.\sin(Z)}{as}\right)^2\right]}}$$

Or in BASIC:

```
LET C = ATN((20*SIN(135)/200)/SQR(1-
  (20*SIN(135)/200)^2)
```

This line combines all the operations we went through. Test it to see that it works. You should get the same answer, 4.05480723.

If you now load the program called FLIGHTPL, the next one on Datacassette A, and run it you will see that all that has been

discussed so far is incorporated within it. The listing is shown below.

```
100 '
110 ' Flight Plan
120 '
130 ' by Ian Padwick
140 ' amended by DA 21/9/84
150 '
160 ' Screen
170 '
180 MODE 2:BORDER 20
190 INK 0,20:INK 1,0
200 LOCATE 30,3:PRINT CHR$(24) ; STRING$(15,32)     ; CHR$(24)
210 LOCATE 30,4:PRINT CHR$(24) ; "  Flight Plan  " ; CHR$(24)
220 LOCATE 30,5:PRINT CHR$(24) ; STRING$(15,32)     ; CHR$(24)
230 WINDOW 5,79,8,24
240 '
250 DEG
260 CLS
270 '
280 ' Random speed and direction
290 '
300 ws=ROUND(RND*50)
310 wd=ROUND(RND*359)
320 '
330 ' Get speed
340 '
350 PRINT:PRINT"Wind speed:";ws;"knots"
360 PRINT:PRINT"Wind direction:";wd;"degrees"
370 PRINT: INPUT"Enter the speed of your aircraft:",as
380 IF as(12 THEN PRINT "Too low!" : GOTO 370
390 '
400 ' Do calculations
410 '
420 IF wd)=180 THEN wd=wd-180 ELSE wd=wd+180
430 sinc=ws*SIN(wd)/as
440 c=ATN(sinc/SQR(1-sinc^2))
450 IF c(0 THEN co=ABS(c) ELSE co=360-c
460 co=ROUND(co,2)
470 IF wd=180 OR wd=0 THEN gs=as+ws*COS(wd):GOTO 510
480 IF wd)180 THEN z=180-((360-wd)+co)ELSE z=180-(wd+c)
490 gs=ABS(SIN(z)*as/SIN(wd))
500 gs=ROUND (gs,1)
510 t=100/(gs/60)
520 t=ROUND (t,1)
530 '
540 ' Print results
550 '
```

```
560 PRINT:PRINT"Course to fly is";co;"degrees"
570 PRINT:PRINT"Your groundspeed will be";gs;"knots"
580 PRINT:PRINT"The flight will take";t;"minutes"
590 '
600 ' Do it again
610 '
620 PRINT:PRINT:PRINT "Press any key....."
630 WHILE INKEY$="":WEND
640 GOTO 260
```

Once again there are some new keywords. The program generates a random wind and asks you to input your aircraft's flying speed. It then tells you the course to fly and how long a 100-mile journey will take. *Warning...!* Do not enter a speed lower than the wind speed, you may never get there if you're flying against it! One new keyword is ABS. ABS returns the ABSolute value of a number which means it makes negative numbers positive but doesn't change positive numbers.

The only lines that need some explanation are 470 and 480. Line 470 deals with two special cases. These are when the wind direction is due north or due south (0 or 180 degrees). We cannot allow the calculation to continue in these cases since a division by zero will occur: sin(0) = sin(180) = 0. Instead we add or subtract the wind speed accordingly. COS(wd) in line 470 can only be 1 or -1 and provides a rather neat way of doing this.

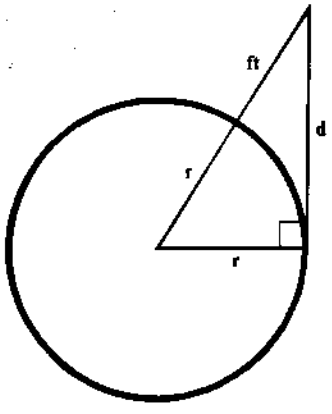
Line 480 calculates the angle 'Z'. If 'wd' is greater than 180, i.e. a reflex angle, then we must subtract that angle from 360 to get the complementary angle (the one inside).

As far as the eye can see

Let's come on to another problem now. Suppose that having taken off we want to know how far we can see, i.e. the distance to the horizon. For example, how high would we have to go to be able to see our destination 100 miles away? Type the following program and RUN it. Study the listing in conjunction with the diagram and figure out how it works.

```
10 r=3960     'radius of the earth in miles

20 INPUT "Enter altitude in feet:",ft

30 mi=ft/5280

40 d=(mi+r)^2-r^2
```

```
50 d=SQR(d)

60 d=ROUND (d,2)

70 PRINT:PRINT"Distance to horizon is";d;"miles"

80 PRINT:PRINT:PRINT

90 GOTO 20
```



Sines and cosines turn up in the maths of many games. In snooker, for instance, they govern the directions in which two balls go after they strike one another, and the direction of a ball after bouncing off the cushion. They also turn up whenever things are wobbling about - the speed of a piston in a car engine, a swinging pendulum, or a weight on the end of a piece of elastic can all be described by sines or cosines.

If you want to use a computer to model what happens in reality (and that is behind the best games programs), trig functions are a great ally.

Of course, there are a few other maths functions we could explain to you like LOG, LOG10, EXP, SGN but these are explained well in the manual.

PRIME TIME

And finally, here is a program which does absolutely nothing useful at all. Reset the CPC464 and enter the following:

```
10 MODE 0

20 DIM a(2000)

30 a(0)=2: x=0

40 FOR n=3 TO 10001 STEP 2

50 p=0

60 IF a(p)^2)n THEN x=x+1: a(x)=n: PEN 1: GOTO 90

70 IF n/a(p)=INT(n/a(p)) THEN PEN 3: GOTO 90

80 p=p+1: GOTO 60

90 PRINT TAB(8);n

100 NEXT n
```

It prints out what could be an endless stream of odd numbers. Those that are prime numbers it prints in yellow, the others are red. A prime number is one that cannot be divided exactly by anything other than itself and 1. There are quite a lot of them as you will see, but they get harder to find the bigger they get.

Although the program prints out all values of 'n', only those that are prime are loaded into the array 'a'. The program then uses these numbers to divide into further values of 'n'. After all, if a number is not divisible by a prime number it must be a prime number itself.

When the program has finished, or before if you get bored, you can print out the values in the array by typing the following:

```
FOR n=0 TO 2000: PRINT a(n): NEXT n
```

It does not need a line number as it is self-contained. How many prime numbers are there between 1 and 10,000? To what value of 'n' would you have to go before the array was full? If the array filled the CPC464's memory, how big a prime number would you have at the end?

TESTING

That was pretty tough wasn't it! Go back over the main points of this chapter again before running SAT6.

Chapter 7
PLAYING WITH WORDS

The CPC464 is not a word processor in itself but it can process words and has several keywords designed for doing all sorts of clever things. You need to remember that any variable you use which began life enclosed in quotes ("") or anything that you PRINT that is enclosed in quotes is called a 'string'.

A string is not necessarily a word. It is *any* list of characters enclosed in quotes. When you assign a variable name to a string it must have a dollar ($) suffix to tell the CPC464 that it is dealing with a string. Here are some strings; enter them on your CPC464:

```
LET a$="Amstrad"

LET c$="CPC464"

LET r$="45^%) lko+-CF#!} "
```

Now type:

```
PRINT a$, c$, r$
```

Within a string you can put any of the CPC464's 256 characters. We cannot do mathematical operations on strings directly except to add them together, joining them end to end (a process called concatenation), or compare them.

Try this:

```
LET z$=a$+" "+c$: PRINT z$
```

You should get 'Amstrad CPC464' printed on the screen. The contents of a$ and c$ which have been copied into z$ are now *substrings* of z$, i.e. strings within a larger string.

HOW LONG IS A STRING?

So, what can we do with strings? Well, using the keywords in this chapter you will be able to chop them up into smaller strings, swop them, reverse them, search them for particular characters and much more.

The first thing we can do with a string is find its length, i.e. the number of characters within it. We do this using the command LEN which has the structure:

```
LEN ((string expression))
```

For example, PRINT LEN(a$) will give 7 because there are seven letters in 'Amstrad'. PRINT LEN(c$) also gives 7. PRINT LEN(z$) gives 15. We might want to know the length of a word for a variety of reasons, to enable a program to print a string centrally on the screen for instance, for example:

```
10 INPUT "Enter your name: ",name$

20 IF LEN(name$))40 THEN PRINT "Name too long":GOTO 10

30 l=LEN(name$)

40 PRINT TAB(20-l/2);name$: PRINT

50 GOTO 10
```

You will have noticed that when you print a list of numbers that have varying numbers of digits, the numbers are printed from the left so that all the first digits are lined up. However, when we write lists of numbers we usually do it so that all the units, tens, hundreds, etc., are in line. For example, you get

```
1
12
97
4032
255
```

but you want

```
   1
  12
  97
4032
 255
```

A neat way of doing this would be to find the length of the number and then calculate a TAB position as before. But if you

try to PRINT LEN (n), where 'n' is our number, you will get the message:

```
Type mismatch
```

This is because numbers are not stored in the same way as strings and you do not have individual digits occupying one position as is the case for string information. To get over this we must convert our number into a string. We can do this using the keyword STR$:

```
STR$((numeric expression))
```

We can type n$=STR$(n) and then PRINT LEN(n$) or we can do it in one go, PRINT LEN(STR$(n)). Remember that when the CPC464 prints a number it always puts a space in front and behind. The space in front is actually the sign position, and is there instead of a + sign. Negative numbers have a - sign of course. Using STR$ automatically chops off the following space but still leaves the sign space; thus when you find the LENgth of a number the answer is always one more than the number of digits. For example:

```
PRINT LEN( STR$(464))
```

gives the answer 4 not 3.

This need not cause any problems so long as you take account of it. (Later you will find out how to chop off that leading sign space.)

So now we can write a short program to achieve our aim:

```
10 INPUT "Enter a number: ",n

20 l=LEN( STR$(n))

30 PRINT TAB(20-l);n

40 GOTO 10
```

STRINGY NUMBERS

All right then, we can turn numbers into strings. How about strings into numbers? Of course!

VAL

The function VAL allows us to find the numeric value of a string. The structure is:

```
VAL((string expression))
```

Let's use our variable a$ ('Amstrad'). What VALue could that have? Try it by entering:

```
PRINT VAL(a$)
```

It has no value! This is because the string expression must be formed from numeric characters or have numeric characters at the beginning of the string if it is to have a value.

Type

```
n$="65536"
```

then

```
PRINT n$:PRINT VAL(n$)
```

You will get 65536 printed at the edge of the screen and underneath 65536 printed with a space in front. VAL(n$) becomes a number that can be operated on:

```
PRINT VAL(n$)/256*4
```

will give 1024 as the answer.

Try out VAL on these strings: 'CPC464', '464 CPC', '-4 64Am'. You should get 0, 464, -464.

How could we use this function? Well suppose we had a program that asked for a number to be entered from the keyboard and someone put in a letter by mistake (or perhaps deliberately). The message:

```
?Redo from start
```

would appear and mess up the display:

```
10 INPUT "Enter a number (1 to 9): ",n

20 IF n<1 OR n>9 GOTO 10

30 PRINT n

40 GOTO 10
```

RUN this and enter a letter. See what happens? Now, if the INPUT took a string variable instead, the error could be trapped using VAL. Change the variable 'n' in lines 10 and 30 to n$ then type:

```
20 IF VAL(n$)<1 OR VAL(n$)>9 GOTO 10
```

Entering a letter now will not upset the program.

Suppose that the number range was not 1 to 9 but 0 to 9. Entering a letter would give VAL(n$) the value 0, which would be valid. How can we then tell whether 0 or a letter was entered? Here is another keyword: ASC. It is short for ASCII which is itself short for American Standard Code for Information Interchange.

This is one of the few things that most micros have in common, an ASCII character set. Every character the CPC464 can generate has an ASCII value. From numbers 32 to 126 the characters are the same as on other computers that use ASCII.

ASCII AND ASCII

Using ASC then, returns the ASCII value of the first character in the string using the syntax:

```
ASC(<string expression>)
```

Type

```
PRINT ASC("0")
```

and you get 48, the ASCII value for 0. Type

```
PRINT ASC("Amstrad")
```

and you get 65, the ASCII value for A (upper case, or capital 'A'; lower case 'a' has value 97). So we can change the 1 in line 20 to 0 and add:

```
25 IF VAL(n$)=0 AND ASC(n$)<>48 GOTO 10
```

If you now RUN the program you will find that the numbers 0 to 9 are accepted but everything else is not. This is not quite true though.... If you enter '7Amstrad' it will still accept it! How can we prevent this? Well, we only want a single character entry so we could put a different line 20:

```
20 IF LEN(n$)>1 THEN GOTO 10
```

Or, we could use INKEY$ to get our character instead of INPUT. You can work out how to do that!

Two more keywords we have used here are AND and OR. Their use is really self-explanatory, but there is just one point about OR. In English we may say 'if a=10 or b=20 then c=30' meaning that c=30 when a=10 or b=20 *but not* when a=10 and b=20. Such a condition is known as an exclusive or (XOR). XOR allows us to say: 'if one or the other but not both then...'. The line:

```
IF n>50 XOR n<73 THEN PRINT n
```

will print the value of 'n' if one and only one of the conditions is true.

OR allows us to say: 'if one or the other or both then...'. The line:

```
IF n>50 OR n<73 THEN PRINT n
```

will print the value of 'n' if one or both conditions are true.

Finally on this point, suppose we only want numbers 1 to 5 to be entered (like a menu choice). The next four lines of program all do the same test in different ways...

```
20 IF VAL(n$)<1 OR VAL(n$)>5 THEN...incorrect

20 IF VAL(n$)>0 AND VAL(n$)<6 THEN...  correct

20 IF ASC(n$)<49 OR ASC(n$)>53 THEN...incorrect

20 IF ASC(n$)>48 AND ASC(n$)<54 THEN...  correct
```

OPPOSITE NUMBERS

The opposite function of ASC is CHR$ which you have met before. CHR$ takes a number after it and returns the character whose ASCII code is that number:

```
CHR$(<numeric expression>)
```

The next program prints out the entire character set:

```
10 FOR n=0 TO 255

20 PRINT CHR$(1);CHR$(n);

30 NEXT n
```

Printing CHR$(1) on its own has no effect. It is called a control character and allows the characters whose codes are 0 to 31 to be printed. They are inaccessible otherwise. A table of these control characters can be found on page 2 of Chapter 9 in the *Amstrad CPC464 User Guide*.

LEFT, RIGHT AND CENTRE

In Chapter 4 of this book is a program called DIGITALC which prints the tens and units of the hours and minutes separately, but does it so that they look like two-digit numbers. Remember that ordinarily one cannot print one number right next to another because the sign space of the second causes a space to be put between them. We know we can use STR$ to lose the trailing space, but how do we lose that sign space?

Here are three commands that Amstrad BASIC uses for sorting out such things: LEFT$, MID$ and RIGHT$. All three are used for extracting parts of a string. LEFT$ does this from the left

end, RIGHT$ does this from the right end and MID$ does it anywhere. The syntax for RIGHT$ is the same as for LEFT$:

```
LEFT$(<string expression>,<integer expression>)
```

With LEFT$ and RIGHT$ the <integer expression> defines the number of characters to include, starting from the left or right end of the string expression:

```
MID$(<string expression>,<int exp>[,<int exp>])
```

With MID$, the first integer expression defines the first character of the substring and the second (optional) integer expression defines the number of characters to include from that first character. Here are some examples of this method of definition (z$="Amstrad CPC464"):

```
PRINT RIGHT$(z$,3)      gives 464
  "   RIGHT$(z$,6)      gives CPC464
  "   RIGHT$(z$,14)     gives Amstrad CPC464
  "   LEFT$(z$,3)       gives Ams
  "   LEFT$(z$,9)       gives Amstrad C
  "   LEFT$(z$,14)      gives Amstrad CPC464
  "   MID$(z$,4)        gives trad CPC464
  "   MID$(z$,4,4)      gives trad
  "   MID$(z$,9,3)      gives CPC
```

Any part of any string can be extracted using these functions. Why should we? Well that depends on what you want to do. Let's get rid of that leading sign space. If 'n' is a single-digit number you could use either of these:

```
PRINT RIGHT$(STR$(n),1)
```

or

```
PRINT MID$(STR$(n),2)
```

If 'n' is to be a multidigit number, however, you need to employ LEN as well. Try this program:

```
10 CLS: PAPER 3

20 INPUT "Enter a number: ",n

30 PRINT n
```

```
40 PRINT RIGHT$(STR$(n),LEN(STR$(n))-1)

50 GOTO 20
```

The purpose of changing the paper is that you can see the spaces more easily. In line 40, STR$(n) is the string expression and LEN(STR$(n))-1 is the integer expression. We subtract 1 because LEN(STR$(n)) will be one more than the number of digits in 'n' - remember?

The following lines of program can replace those of the same numbers already in DIGITALC. So, load that program, then type:

```
180 t=TIME

1070 LOCATE 13,10:PRINT "0"

1080 WHILE s<60

1090 s=INT((TIME-t)/300)

1100 s$=STR$(s)

1110 LOCATE 14+(s)9),10

1120 PRINT RIGHT$(s$,LEN(s$)-1)

1130 WEND:s=0:t=TIME
```

These lines replace the seconds loop and make use of the CPC464's own internal clock which is accessible through the keyword TIME.

You can see that line 1120 chops off the leading space and that s$ has been previously defined as being equal to STR$(s) in line 1100. The variable 's' represents the difference between the actual TIME and the value of 't' which was the TIME when the clock was started (line 180). The difference increases by 300 every second, hence line 1090.

The WHILE-WEND loop ensures that 't' is updated every 60 seconds to the new value of TIME.

Look at line 1110. An interesting concept is used here. The condition in the brackets, s>9 is evaluated as being either true or false: either 's' is greater than 9 or it is not. If true the value of the condition is -1, if false the value is 0. The value is then used to move the cursor to column 14 if 's' is a single-digit number, or column 13 if it is a two-digit number. All conditions are given values of 0 or -1, depending on whether they are false or true.

To prove it type the following:

```
v=17
```

then:

```
PRINT v)4;v(20;v(2;v)50;v=17;v*2)50;SQR(v))
  4;v/2(v/3;v+3=20
```

You should get

```
-1 -1 0 0 -1 0 -1 0 -1
```

This logical value could be very useful to you when you get better at programming, but don't worry too much about it now.

SEARCHING FOR WORDS

Here is another new function for you: INSTR. This enables you to search a string for substrings. The number returned is the position of the first character of the substring within the main string:

```
INSTR([(integer expression),](string exp),
  (string exp))
```

The integer expression is optional and gives the position within the string to start searching. If omitted it starts at the beginning. The first string expression is the string in which to search while the second is the string to search for. So, if z$="Amstrad CPC 464" then PRINT INSTR(z$,"d") returns 7 because 'd' is the seventh character of the string.

```
PRINT INSTR(z$,"CPC")
```

This returns 9 because the first character of CPC occurs at character 9 of the searched string.

```
PRINT INSTR(z$,"4")
```

This returns 12, the position of the first '4'. To get the second one you type:

```
PRINT INSTR(13,z$,"4")
```

You now get the answer 14 because the search was begun from the thirteenth character of z$.

This command is very useful in the program HANGMAN which we saw in Part 1 of this Tutorial Guide. The following program illustrates the way INSTR was used:

```
10 CLS

20 LOCATE 1,1: INPUT "Type a word: ",w$
```

```
30 w$=UPPER$(w$)

40 LOCATE 1,3: PRINT "Guess a letter: "

50 l$=INKEY$

60 IF l$="" GOTO 50

70 l$=UPPER$(l$)

80 LOCATE 1,3: PRINT SPACE$(16)

90 p=1

100 p=INSTR(p,w$,l$)

110 IF p=0 GOTO 40

120 LOCATE 10+p,10: PRINT MID$(w$,p,1)

130 p=p+1: GOTO 100
```

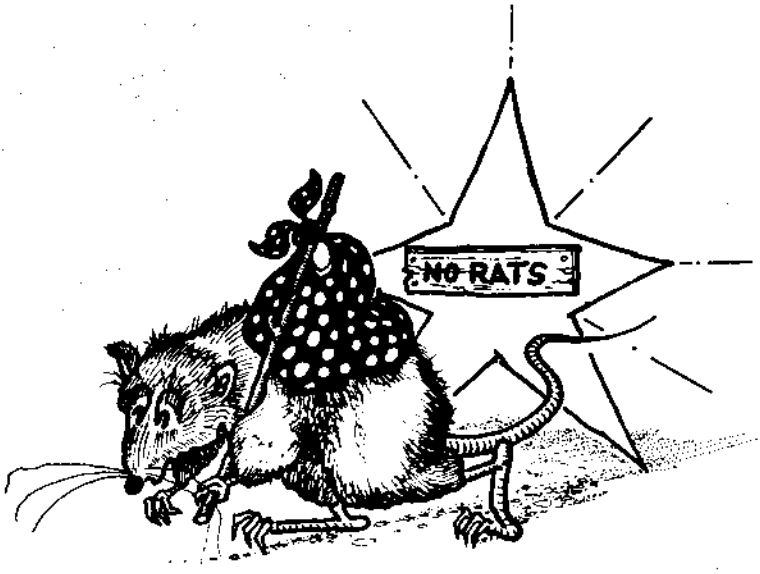
Note that this program will not stop even when you have got all the letters. You can work out how to get it to stop. There is another new keyword here - UPPER$ - which converts all lower case alpha charactrs (a-z) within the following string expression to upper case (capitals).

Back to front

The final program below demonstrates some more string chopping. This time, whatever you type in is printed in reverse. You should be able to work out how it is done.

```
10 PRINT "Type a string: "

20 INPUT w$

30 FOR n=LEN(w$) TO 1 STEP -1

40 b$=b$+MID$(w$,n,1)

50 NEXT

60 PRINT "  ";b$: PRINT: b$=""

70 GOTO 10
```

Try entering 'evil rats on no star live'.



PLAYTIME

Our game this time is very appropriate. It is an aid to those newspaper word games where you have to make up lots of different words from the letters of one long word. So load up WORDPUZL and enjoy yourself for a while.

TESTING

Now run SAT7. You never know, but you might be able to write a program for solving *The Times* crossword puzzle.

Chapter 8
MOVING PICTURES

The principle of making 'moving pictures' has been around for a long time. One displays a 'still' picture, then quickly replaces it with another that has some slight differences, then replaces that with another and so on. If this is done rapidly and evenly the differences between each of the 'frames' appears continuous and animation is achieved.

FLASHY PROGRAMMING

One of the simplest ways of adding action to a screen message or graphic display is to use an INK command which specifies *two* colours. The syntax is as follows:

```
INK (ink number), (colour)[, (colour)]
```

When this second <colour> is included, the CPC464 will then alternate the INK between the two colours.

Try entering this directly after a general reset or power on:

```
INK 1,24,1
```

It gives you a flashing display, doesn't it? This is because the second colour is the same as that used in the default INK specified for the PAPER. It isn't that the display is 'turned off' for half the time - it's just that you are writing with a blue PEN on blue PAPER! Any two colours can be specified so if you try entering different pairs of your own choice it will be obvious what is happening.

The speed at which the two colours alternate can be changed by a new keyword: SPEED INK. Here is the syntax:

```
SPEED INK (integer expression), (integer
  expression)
```

The first argument specifies the length of time that the first colour of the INK should appear on the screen, and the second argument that of the second colour. These <integer expressions> are in steps of 0.02 second.

Here is a short program that illustrates the use of flashing INKs: INKs:

```
10 rem Flashy Message program

20 cls

30 mode 0

40 ink 1,24,1

50 for a=50 to 1 step -1

60 locate 3,12

70 speed ink a,a

80 print "FLASHY MESSAGE"

90 for x=1 to 200:next

100 next a

110 goto 10
```

Flashing text or graphics are immensely useful when you want to draw attention to a particular part of the screen display.

In MODE 0, INKs 14 and 15 flash by default; 14 alternates between blue and bright yellow and 15 alternates between sky blue and pink.

ANIMATION

When you are writing games programs, the really essential ingredient is movement. Flashing graphics and text may also be used but the whole essence of most arcade-type programs is the battle between graphically displayed elements that the player can control and those that he can't. So we need to know how to get our graphics to move around on the screen.

To animate a picture, one need not replace the entire picture every time. Only the parts that are moving need to be changed between frames. Computer animation works in this way. The entire picture is not redrawn every time part of it changes, rather

only the moving part is changed. Movement is achieved by erasing the old position of whatever is moving and then redrawing it in a new position.

Another way of achieving the impression of movement is to draw the moving character in all its positions but do it in such a way that it cannot be seen, e.g. make it the same colour as the background. Then, when the movement is required, each position can be 'switched on' and then 'off' in turn, not by redrawing it, but by changing the colour of the ink used to draw it in the first place. This can be demonstrated if you now type in the program that appears on page 32 of Chapter 3.

Make sure that it works in MODE 0 then type in the following extra lines (easier in MODE 1 or 2):

```
110 FOR n=2 TO 15
120 INK n,1
130 NEXT n
140 FOR n=1 TO 15
150 INK n,24
160 INK n-1,1
170 FOR x=1 TO 80: NEXT x
180 NEXT n
190 INK 15,1
200 GOTO 140
```

Make sure you are in MODE 0 when you RUN the program. You should see the disks drawn as usual, then all but the first will disappear and then the first will appear to move across the screen rapidly over and over.

When you get fed up and press ESC twice you will more than likely have blue text on blue paper. Carefully type INK 1,24 and the text should appear. Now study the listing. You should be able to understand how it works but if you don't, just change a value here and there and see what happens! Doing this should make things clear.

Because of the way the CPC464 organises its colours, you can do this with any PRINTed and PLOTed character or picture.

Twinkletoes

The only problem with the colour switching method of animation is that you are limited to some extent by the number of INKs available. Also you must be careful not to draw over an unseen character that you will want to use later. For these reasons PRINTing characters and erasing them again is the preferred method of getting things to move.

Let's start with the simplest sort of movement, a little man dancing across the screen.

The CPC464 is supplied with four characters which can be used directly for this purpose. They are given by ASCII codes 248-251, inclusive, as shown in the *Amstrad CPC464 User Guide*, Appendix III, pages 12 and 13. Here is an extremely simple example of how they can be used:

```
10 rem Dancer program

20 cls:mode 0

30 data 248,250,249,251

40 for a=1 to 4:read dance(a):next

50 b=1

60 for c=1 to 4

70 locate b,12

80 print chr$(dance(c));

90 for x=1 to 500:next

100 locate b,12:print " ";

110 b=b+1

120 next c

130 if b<20 then goto 60

140 goto 50
```

The first thing we do is to load an array, 'dance(a)', with the ASCII value of the characters. This is done in the order in which they are to be used. Then, within the loop of 'b' going from 1 to 20, we have a nested FOR-NEXT loop which provides the next value

of the array. This is printed at the next 'x' co-ordinate of the screen by using the current value of 'b' in the LOCATE command.

After printing the characters given by the current value provided by the array, there is a delay loop of x=1 to 500. A blank is then printed in the same position to wipe the character out before moving on to the next one.

When you have entered and run the program, remove line 100 and run it again. This will show you why we had to include it in the first place. You could also experiment with the duration of the delay loop in line 90.

Where did they all come from?

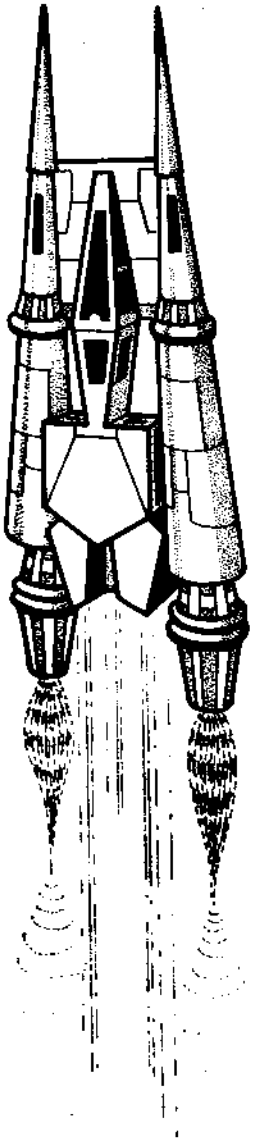
Part 1 of this course had a zapping game called BOMBER which was designed to teach you the basics of co-ordinate geometry by getting you to estimate the position of an alien invader. If you remember, the little creature kept appearing at different places on the screen by means of the RND command. This command is essential for providing an element of unpredictability in the movements of characters and graphics.

The following few lines will make the biggest space fleet you have ever seen appear on your screen.

```
10 rem Space Invasion

20 cls:mode 0

30 locate 19*rnd+1,24*rnd+1

40 print chr$(239);

50 for delay=1 to 999*rnd

60 next

70 goto 30
```

The interesting thing here is that the spaceships also appear at random intervals thanks to the inclusion of RND in the delay loop of line 50.

Note that you have to add '1' to both arguments of the LOCATE command. Otherwise, the product of 19★rnd or 24★rnd could be less than 1 and the CPC464 would stop the program, giving an 'Improper Argument' error message, since LOCATE can only accept arguments greater or equal to 1.



One over the eight

Suppose our little dancer had been a bit over-indulgent, but is still trying to keep a straight line across the screen. We can simulate this by adding to the program:

```
65 d=int(3*rnd+5)
```

INT

As you will remember, the INT keyword ensures that the result of the numeric expression in its argument is a whole number by truncating any decimal part. The manipulations of RND by

multiplying by 3 and adding 5 are only there to get the resulting expression within the desired range.

This new variable, 'd', can now be added to the program to give the second argument for the two LOCATE commands as follows:

```
10 rem Drunkard program

20 cls:mode 0

30 data 248,250,249,251

40 for a=1 to 4:read dance(a):next

50 b=1

60 for c=1 to 4

65 d=int(3*rnd+5)

70 locate b,d

80 print chr$(dance(c));

90 for x=1 to 500:next

100 locate b,d:print " ";

110 b=b+1

120 next c

130 if b<20 then goto 60

140 goto 50
```

If you run this modified program you will find that our dancer is now a little unsteady on its feet!

ANOTHER BRICK IN THE WALL

One of the original computer arcade games was called 'Brickout'. The player had to move a 'bat' back and forth across the bottom of the screen to deflect a 'bouncing ball' upwards against a 'wall' to knock out the 'bricks'. We will now develop this program to some extent to demonstrate some principles of programming an arcade game and introduce some new keywords as we do so.

Make sure you have reset your CPC464 and have some blank tape to save the program on.

The first thing to do is draw the boundary to our screen. We could just use the border, but for reasons that will become clear later and for effect, we won't. The line numbers are deliberate, so don't change them yet:

```
10 MODE 1:CLS:GOTO 200

200 LOCATE 1,1: PRINT CHR$(136);

210 FOR c=2 TO 39

220 LOCATE c,1: PRINT CHR$(140);

230 NEXT

240 LOCATE 40,1: PRINT CHR$(132);

250 FOR  1=2 TO 24

260 LOCATE 1,1: PRINT CHR$(138);

270 LOCATE 40,1: PRINT CHR$(133);

280 NEXT

300 LOCATE 1,25: PRINT CHR$(130);

310 FOR c=2 TO 39

320 LOCATE c,25: PRINT CHR$(131);

330 NEXT

340 LOCATE 40,25: PRINT CHR$(129);

350 LOCATE 1,1
```

Line 350 has been put there as a temporary measure to stop the screen scrolling.

You should have no problems with this. At the moment the Ready message gets in the way but it won't later on. What we want now is a bouncing ball...:

```
20 x=1: y=-1

30 c=INT(RND*37+2)
```

```
40 l=INT(RND*22+2)

50 IF c+x)39 OR c+x(2 THEN x=-x

60 IF l+y)24 OR l+y(2 THEN y=-y

80 LOCATE c,l: PRINT " ";

90 c=c+x:l=l+y

100 LOCATE c,l: PRINT CHR$(224);

110 GOTO 50

350 GOTO 20
```

When you RUN the program, you should get a 'ball' bouncing around inside the boundary. Quite fast isn't it? If you want to slow it down a bit you can insert a temporary line:

```
105 FOR d=1 TO 50: NEXT
```

Lines 20 to 40 set up initial values for variables 'x' and 'y' (used to move the ball) and 'c' and 'l' (the ball's position). Lines 50 and 60 find out whether or not the next position of the ball will be outside the boundary. (We are using columns 2 to 39 and lines 2 to 24.) If it will be, then the relevant increment, 'x' or 'y', is inverted.

Line 80 erases the old position of the ball by PRINTing a space over it, line 90 calculates its new position and line 100 PRINTs the new ball. Line 110 completes the loop. Quite simple isn't it?

If you are able you should now SAVE this program because we are going to change it now, but change it back again later....

Smoothing the way

Each new position of the ball is necessarily a whole character square from the old position. While the speed of the ball helps the movement to appear smooth, it is not as smooth as one might like. To get smoother movement we would need to PRINT the ball at half-character intervals or less. This can be done using TAG.

Remember that TAG allows you to PRINT at the graphics cursor instead of the text cursor. The graphics cursor can be moved to any of 640 positions horizontally and 400 positions vertically. (It

can be moved further but it will be off the screen!) Remember also that in MODE 1 each pixel can be referred to by four sets of co-ordinates (two vertically and two horizontally, see Chapter 3). This means that to move a character the equivalent of one character square we must move the graphics cursor 16 positions one way or the other.

Let's change the program now. The first thing to do is switch on TAG, so insert this line:

```
45 TAG
```

The range of numbers you can PRINT is now much greater so you must change lines 50 and 60 appropriately.

```
50 IF c+x)609 OR c+x(16 THEN x=-x

60 IF 1+y)383 OR 1+y(30 THEN y=-y
```

Lines 30 and 40 must also be changed to give a valid initial position for the ball:

```
30 c=INT(RND*592+16)

40 1=INT(RND*337+42)
```

When TAG is in force LOCATE has no effect so we must change LOCATE to MOVE:

```
80 MOVE c,1: PRINT " ";

100 MOVE c,1: PRINT CHR$(224);
```

If you *did* put line 105 in, now is the time to take it out.

The first thing that strikes you is how slow it is. This is because the program references every pixel position twice. The speed can be doubled with no change to the smoothness by doubling the increments 'x' and 'y':

```
20 x=2: y=-2
```

RUN the program again to see the improvement. If you are not sure what a particular line or variable does, change it and try to understand the effect this has.

Bat'n'ball

When you have finished experimenting with version two of the ball program, if you SAVEd the original, load it. If you didn't, change the program back to what it was. Now is the time

to include a 'bat' (not the flying kind) routine in our program. The bat will move back and forth across the screen where the lower edge of the boundary is at present.

First, then, delete the lines that draw the lower edge of the boundary, 300-340. (The ball will still bounce off, of course.) Now what we want is a bat. We will use three CHR$(131)s for this purpose. The PRINTing of the bat must occur within the main loop of the ball program so we will do this after the ball has been PRINTed in line 100. Only the horizontal position of the bat will change, so only one variable is needed.

To PRINT three CHR$(131)s you could type:

```
PRINT CHR$(131);CHR$(131);CHR$(131)
```

But remember the STRING$ function from Chapter 7:

```
STRING$(3,131)
```

Now insert lines:

```
4 bat$=" "+STRING$(3,131)+" "

109 LOCATE b,25: PRINT bat$;
```

The spaces before and after the bat are to erase the 'old' bat as it moves. We now need to change the value of variable 'b' in the range 1 (left) to 36 (right). It has to be 36 because the bat, including spaces, is 5 characters long and column 36 is as far right as it will go without going off the edge.

We can test whether or not a key is being pressed by using the keyword INKEY$. Let's choose a key for left, say Q, and one for right, say P. You will have to make sure that CAPS LOCK is off to get 'q' and 'p' and not 'Q' and 'P'. You can type the following:

```
101 ky$=INKEY$

102 IF ky$="q" THEN b=b-1

103 IF ky$="p" THEN b=b+1
```

Clearly, 'b' could increase or decrease too much and cause the program to crash, so you should add:

```
104 IF b>36 THEN b=36

105 IF b<1 THEN b=1
```

And give 'b' an initial value at line 20:

```
20 x=1: y=-1: b=18
```

If you now RUN the program you will find that the ball bounces as usual and you can move the bat by pressing key Q or key P (but not both). The bat does not respond particularly well to the INKEY$ command so there is another way.

INKEY

The keyword INKEY is very similar to INKEY$ but reads the keyboard differently and doesn't wait for the keyboard to repeat. The syntax is

```
INKEY ((integer variable))
```

where the variable is a key number as on page 16 of Appendix 3 in the User Guide. You will see that the Q key is number 67 and the P key is number 27.

Type in these changed lines:

```
101      (this deletes it!)

102 IF INKEY(67)=0 THEN b=b-1

103 IF INKEY(27)=0 THEN b=b+1
```

You will now find that the bat responds better than before. The following line can replace lines 102, 103, 104 and 105 and is a much neater way of doing the same thing:

```
102 b=b+(INKEY(67)=0 AND b)1)-(INKEY(27)=0
  AND b(36)
```

It works by making use of the logical value of the condition within the brackets. The value is 0 if the condition is false and -1 if it is true (see Chapter 7).

Now the bat moves more quickly we can insert a line or two to detect when the ball hits it. The ball will be touching the bat when the next line value of the ball is 25 AND when the next column value is equal to b+1, b+2 or b+3. (The bat is three characters long.)

You could type the line:

```
IF l+y=25 AND (c=b+1 OR c=b+2 OR c=b+3)
  THEN.... a hit
```

Instead type these lines:

```
60 IF l+y(2 THEN y=-y

65 IF l+y)24 THEN y=-y:IF c)b AND c(b+4 THEN
  s=s+1 ELSE s=s-5
```

The variable 's' is to be used to keep the score. If you hit the ball

then your score is increased. Every time you miss, your score drops by 5 points!

Insert this line to display the score:

```
106 LOCATE 16,1: PRINT "SCORE =";s;
```

The program is getting quite complex now, and the ball is slowing down a bit. That's why using TAG is impractical where speed is required.

Collisions, collisions

So far, we have used our knowledge of where the boundaries and bat are to detect when the ball hits them. In fact, the boundary and bat need not be there at all and the ball would still bounce off, because the conditions dictate the collision, not what is actually there on the screen. What if some random character were introduced into the arena? How could we detect a collision without knowing where it was before hand?

Clearly we must know what we are looking for, or at least know something about it; its shape or colour for example. The CPC464 does not have any keyword facility for testing the conditions within a particular whole character square, i.e. testing for the character or colour. Instead there is a keyword for testing the colour of a specific co-ordinate. This is, appropriately, TEST, which has the structure:

```
TEST ((integer expression),(integer
  expression))
```

where the first <integer expression> is the 'x' co-ordinate and the second <integer expression> is the 'y' co-ordinate, PRINT TEST (100,100) will return a number whose value is that of the INK colour at 100,100.

So, if you did PLOT 150,300,3 and then PRINT TEST (150,300) you would get the answer 3. In order to detect the presence of a character within a character square one would first have to convert the column and line values into 'x' and 'y' co-ordinate values, and then do the test. Doing the conversion takes a little thinking about, especially in the vertical direction, since lines are numbered top to bottom and 'y' co-ordinates numbered bottom to top. The more mathematical minded you are the easier it will be for you.

One way out of the dilemma is to use a two-dimensional array that is dimensioned to the column and line values. For example you could type DIM a(40,25). Each subscripted variable would then be a 'shadow' of the screen. If, for instance, you did:

```
LOCATE 15,7: PRINT"*": LOCATE 34,13:
  PRINT "O"
```

then you would also fill the array thus:

```
a(15,7)=1: a(34,13)=2
```

where the numbers 1 and 2 signify a particular character.

You could use the ASCII value if you wished:

```
a(15,7)=ASC("*"):a(34,13)=ASC("o")
```

To detect a collision on screen you would constantly check the contents of the array. We will now add and change some more lines as follows:

```
5 DIM a(40,25)

40 l=INT(RND*10+13)

107 IF a(c,l)=1 THEN a(c,l)=0:s=s+1:x=-x:y=-y

300 PEN 3

310 FOR l=8 TO 12

320 FOR c=2 TO 39

330 LOCATE c,l: PRINT CHR$(143);

340 a(c,l)=1

350 NEXT c

360 NEXT l

370 PEN 1

380 GOTO 20
```

Lines 300 to 360 set up the wall and the 'shadow' array which is dimensioned in line 5. Line 40 ensures the ball always starts below the wall. If a collision does occur, the ball's direction is reversed and your score is increased by 1.

DIY

The game, as it is, can hardly be said to be 'exciting' but then the intention was not to write an exciting game. The intention was to show how a simple game can be developed and expanded upon. Also it is hoped you have a greater understanding of some of the processes and problems involved in writing a program. Now it's up to you to have a go. Here are some suggestions for improvements:

- ☐ Make the ball disappear when it misses the bat and replace it with a new one.
- ☐ Change the direction variables to allow the ball to move at angles other than 45 degrees.
- ☐ Get the ball to bounce off the corner of the bat in the manner you would expect it to.
- ☐ Make the ball bounce off the wall according to the angle at which it strikes.
- ☐ Add some beeps and burps.
- ☐ Add some more obstructions in the field of play.

PLAYTIME

The next program on Datacassette A is BLITZ. Here is a game which will let you release all those pent-up destructive impulses. Your aircraft is steadily losing height over the city, and is likely to crash into one of the buildings. Your only way of getting down safely is to bomb the whole lot flat!

Happy landings.

TESTING

You have probably tired yourself out with all that bombing so it might be a good idea to have a short rest before running SAT8.

Chapter 9
SOUND FX

When we explored the sound commands in Part 1 of this course, it was necessary to restrict the information to simple forms only. As you will have gained a lot more knowledge and confidence by now, we can go into things in much more detail and study a lot of aspects that were previously left out. Using our new notation, the SOUND command is described by:

```
SOUND <channel status>,<tone period>
  [,<duration>[,<volume>[,<volume envelope>
  [,<tone envelope>[,<noise>]]]]]
```

All the arguments are integer expressions. Don't be confused by the number of brackets. What they mean is that you can leave off arguments, starting at the right-hand end, but you can't leave any out in the middle.

CHANNELS

The <channel status> argument is the most complicated part of the SOUND command. In the User Guide you will see that it can be an integer between 1 and 255! Don't be frightened though, for the moment we will just look at how the CPC464's three channels are selected.

These channels are called A, B and C, and can be entirely independent of one another, with their own tone and volume envelopes. However, more than one channel can be specified in a SOUND command, as shown in the following table:

Channel status	*Channel(s)*
1	A only
2	B only
3	A and B
4	C only
5	A and C
6	B and C
7	A, B and C

This might seem a little odd but it works!

You can also plug your Amstrad Hi-Fi Tower System into the back of your CPC464 to get stereo sound. Channel A is left, channel C is right, and channel B gives you a bit of both left *and* right.

ENVELOPES

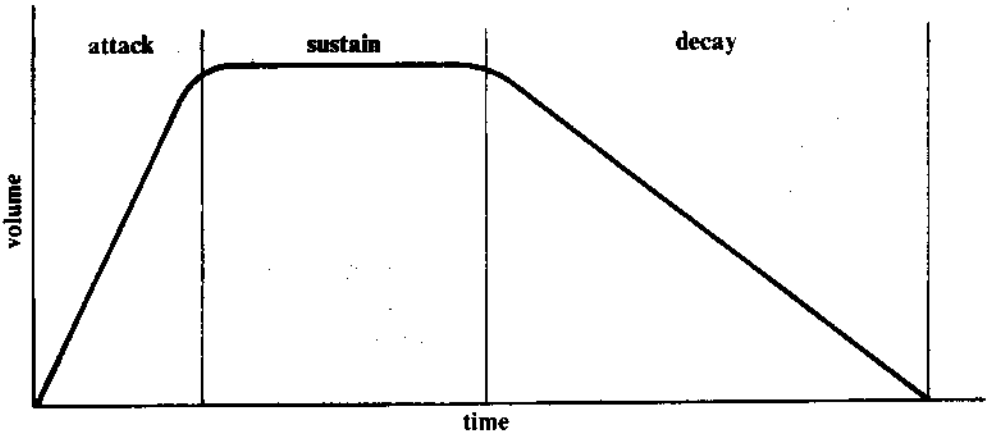
Volume envelopes change the level of sound specified by <volume> in the SOUND command, and tone envelopes change the note or *pitch* specified by <period>. These changes can be dynamic during the time the sound lasts, as specified by <duration>. You already know that you can have up to 15 volume envelopes and 15 tone envelopes in one program. What you may not know is that each envelope can have up to five sections! Read on!

Volume envelopes

A five-section envelope is a pretty daunting prospect at first sight, so we'll have a look at some three-section ones to start with. Load the program, ZOUNDS, from Datacassette A, and RUN it. Ignoring the tone envelope for the moment, you can see that you are given a choice of three things called 'attack', 'sustain' and 'decay'. The diagram shows what these words mean.

It's a bit like a story which has a beginning, a middle and an end. Unlike a story, however, it is often the beginning and the end which are the most interesting. If you keep using the ZOUNDS program you can compare the differences in the sounds produced by different types of attack, sustain and decay.

A typical sound with a fast attack is that of *percussive* musical instruments such as a guitar. Other examples of sounds with a fast attack are thunderclaps, drum beats and explosions. A slow



attack, however, is more characteristic of a large organ pipe, or a brass instrument, such as a tuba. Conversely, once you stop blowing a tuba the sound stops immediately (fast decay), whereas the sound from a guitar dies away gradually (slow decay).

So you can see how a multisection volume envelope enables us to mimic the natural sounds we hear around us. Now we can have a look at the ENV statement again:

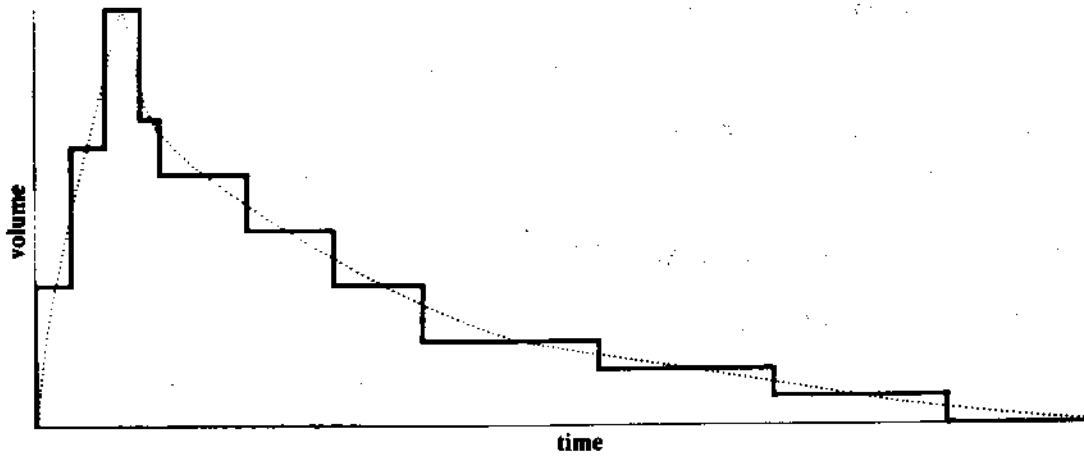
```
ENV (envelope number)[,(envelope sections)]
```

As you already know, each <envelope section> comprises

```
(step count),(step size),(pause time)
```

and as you have recently found out, you can have from one to five of them.

Here is an example:



The coding for this is:

```
ENV 1, 3,5,2, 1,-4,1, 4,-2,5, 1,0,5, 3,-1,10
```

If this envelope is used with a SOUND command whose initial volume is zero, the effect on the volume will be as follows:

Section number	*Effect*
1	Medium attack to maximum volume
2	Fast decay to level 11
3	Medium decay to level 3
4	Short sustain
5	Slow decay to level 0

You can see that the sustain is produced by specifying a single step of zero change in volume - the <period> giving the length of the sustain.

Now is the time for you to experiment with volume envelopes of your own. If your SOUND command is given a <duration> of zero, it means that the duration of the sound is controlled entirely by the ENV statement. You can see this from the following listing of ZOUNDS. Try adding your own envelopes for ENV 13, ENV 14 and ENV 15.

```
100 ' Zounds : Envelope demonstrations
110 '
120 ' by George Tappenden
130 ' amended by DA 25/9/84
140 '
150 CLEAR
160 DEF FNt(x$)=INSTR("FSLMH",x$)-1
170 FOR i=0 TO 4:READ n$(i):NEXT
180 READ e$,a$,s$,d$,t$
190 '
200 ' Screen
210 '
220 MODE 2   : BORDER 26
230 INK 0,26 : INK 1,0
240 PAPER 0  : PEN 1
250 '
260 ' Descriptive Layout
270 '
280 PRINT STRING$(13,"*");" VOLUME ENVELOPES ";STRING$
(13,"*");
290 PRINT SPACE$(8);STRING$(4,"*");" TONE ENVELOPES ";
STRING$(4,"*")
300 PRINT : PRINT
310 PRINT e$,a$,s$,d$,e$,t$
```

```
320 PRINT
330 FOR i=1 TO 12
340 READ a$,b$,c$
350 a=FNt(a$):b=FNt(b$):c=FNt(c$)
360 PRINT i,n$(a),n$(b),n$(c)
370 NEXT
380 FOR i=1 TO 4
390 READ tn$
400 LOCATE 53,4+i*2 : PRINT i
410 LOCATE 66,4+i*2 : PRINT tn$
420 NEXT
430 '
440 ' Volume Envelopes
450 '
460 ENV 1,1,15,1,1,0,40,1,-15,1
470 ENV 2,1,15,1,1,0,40,15,-1,4
480 ENV 3,1,15,1,1,0,5,15,-1,4
490 ENV 4,1,15,1,1,0,5,1,-15,1
500 ENV 5,5,3,1,1,0,40,1,-15,1
510 ENV 6,5,3,1,1,0,40,15,-1,4
520 ENV 7,5,3,1,1,0,5,1,-15,1
530 ENV 8,5,3,1,1,0,5,15,-1,4
540 ENV 9,15,1,2,1,0,40,1,-15,1
550 ENV 10,15,1,2,1,0,40,15,-1,4
560 ENV 11,15,1,2,1,0,5,1,-15,1
570 ENV 12,15,1,2,1,0,5,15,-1,4
580 '
590 ' Tone Envelopes
600 '
605 ENT -1,1,0,1
610 ENT -2,2,1,2,2,-1,2
620 ENT 3,100,1,2
630 ENT 4,100,-1,2
650 '
660 ' User choice
670 '
680 WHILE ( n)=0 ) AND ( t)=0 )
690 '
700 LOCATE 10,22 : PRINT "  "
710 LOCATE 10,23 : PRINT "  "
720 LOCATE  1,20 : PRINT "Type in the envelope number"
730 PRINT
740 INPUT "volume ";n
750 INPUT "tone   ";t
760 '
770 ' Play it
780 '
790 FOR times=1 TO 3
800 LOCATE 6,(n+5)    : PRINT "*****"
810 LOCATE 58,(2*t+4): PRINT "*****"
820 SOUND 7,200,0,0,n,t
```

```
830 FOR delay = 1 TO 1500 : NEXT
840 LOCATE 6,(n+5)    : PRINT SPACE$(5)
850 LOCATE 58,(2*t+4): PRINT SPACE$(5)
860 FOR delay = 1 TO 500 : NEXT
870 NEXT
880 '
890 WEND
900 END ' of program
910 '
920 ' Data for descriptive layout
930 '
940 DATA Fast,Slow,Long,Medium,Short
950 DATA Envelope,Attack,Sustain,Decay,Type
960 DATA F,L,F,F,L,S,F,H,S,F,H,F
970 DATA M,L,F,M,L,S,M,H,F,M,H,S
980 DATA S,L,F,S,L,S,S,H,F,S,H,S
990 DATA steady,vibrato,falling,rising
```

Tone envelopes

Much of what is said about the structure of volume envelopes also applies to tone envelopes. This time, though, it is the pitch of the sound that can be changed during the execution of a SOUND command. Their main use in sound effects is to simulate the noise made by fast-moving craft or vehicles.

When a jet fighter passes overhead, the noise it makes gets louder and louder until it is directly above, and then it gets softer as it goes away. The pitch also changes because of something called the Doppler effect - it falls continually from the time the aircraft is directly overhead until it is completely out of earshot. You can hear a speeded-up version of this in the ZOUNDS program if you use ENV 5 and ENT 3. This type of effect is much used in arcade games to provide realism and to add excitement.

The opposite effect is necessary when you simulate an aircraft taking off, when the pilot has to gun the engines to give maximum power. The pitch of the sound then increases continuously until the turbines reach their maximum speed. You can get an idea of a rising pitch by using ENT 4 with any of the longer volume envelopes provided by the ZOUNDS program. Note that the ENT statement has no effect on the length of the sound (unlike ENV), only on the pitch.

As we saw for the volume envelope, the structure is:

```
ENT (envelope number),(envelope sections)
```

where <envelope number> is from one to fifteen, and <envelope

section> can be from one to five. Each <envelope section> comprises:

```
(step count),(step size),(pause time)
```

All of these are integer expressions. The <step size> is the required change in the tone period of the pitch you require, and is positive if you want the pitch to go down, and negative if you want the pitch to go up. If you think about it, this *is* the right way round!

If you put a minus (-) sign in front of the <envelope number> it will keep repeating until the end of duration given in the SOUND command. This is useful for things like vibrato.

PLAYTIME

If you would like to hear some of the possibilities for sound effects on the CPC464, load the next program from Datacassette A, ZAPPOW2, and run it. This program is not listed here, but you can put it up on the screen if you wish.

Our favourite is the train!

TESTING

Many aspects of the SOUND commands will already be familiar to you from the work we did on them in Part 1. The really clever bit about them, however, is the way you can fit the different elements together to produce the sound that you want. Run SAT9 to check how well you have grasped the principles of multisection envelopes.

Chapter 10
MUSIC

Before you start reading this chapter it is worth reflecting that there are four sorts of people in this world. There are those who can program computers *and* read music; those who can't program computers but can read music; those who can program computers but can't read music; and those who can neither program computers nor read music.

It is only fair to tell you that if you really can't read musical notation, it is going to be a bit difficult to understand what follows. So, it would be entirely understandable if you skipped to the next chapter.

BOGEY MAN

In Part 1 we gave you a dozen lines of programming which you could enter to make the CPC464 play 'A Familiar Tune'. The tune, as you will all have found out, was *Colonel Bogey*. Since then you may well have tried out a few more tunes yourself. To remind you of the original, however, here it is again:

```
10 REM A Familiar Tune

20 SOUND 1,213

30 SOUND 1,253,60

40 SOUND 1,0,40

50 SOUND 1,253

60 SOUND 1,239

70 SOUND 1,213
```

```
80 SOUND 1,127,40

90 SOUND 1,0,1

100 SOUND 1,127,40

110 SOUND 1,159,60

120 END
```

But this is a very clumsy way of programming music - especially if it is a fairly long tune. Now that you know how to use the DATA and READ commands you can learn the more elegant and economical methods for programming your CPC464 to play music.

Rather than having a series of SOUND commands, it is much better to use variables as arguments. For example:

```
SOUND ch,per,dur,vol,ev,et,nse
```

You can now put all the values into DATA statements and use the READ command to update the variables before executing the sound command.

Let's go back to *Colonel Bogey* again. For simplicity's sake we will leave off the volume, envelope and noise arguments, and use channel 1 only. Our main loop then becomes:

```
10 FOR count =1 to 10

20 READ per,dur

30 SOUND 1,per,dur

40 NEXT count

50 END
```

As you can see, this routine will play any tune that comprises 10 notes since all the music is contained in the constants in the DATA statements. To finish off our new-look tune program you must therefore enter:

```
60 DATA 213,20,253,60,0,40,253,20,239,20,

213,20,127,40,0,1,127,40,159,60
```

If you run this program it will sound (if you haven't made any mistakes) exactly like our original version of *Colonel Bogey*. But why, you ask, go to all this bother? Well, there are many advantages. One is that it is much easier to change individual

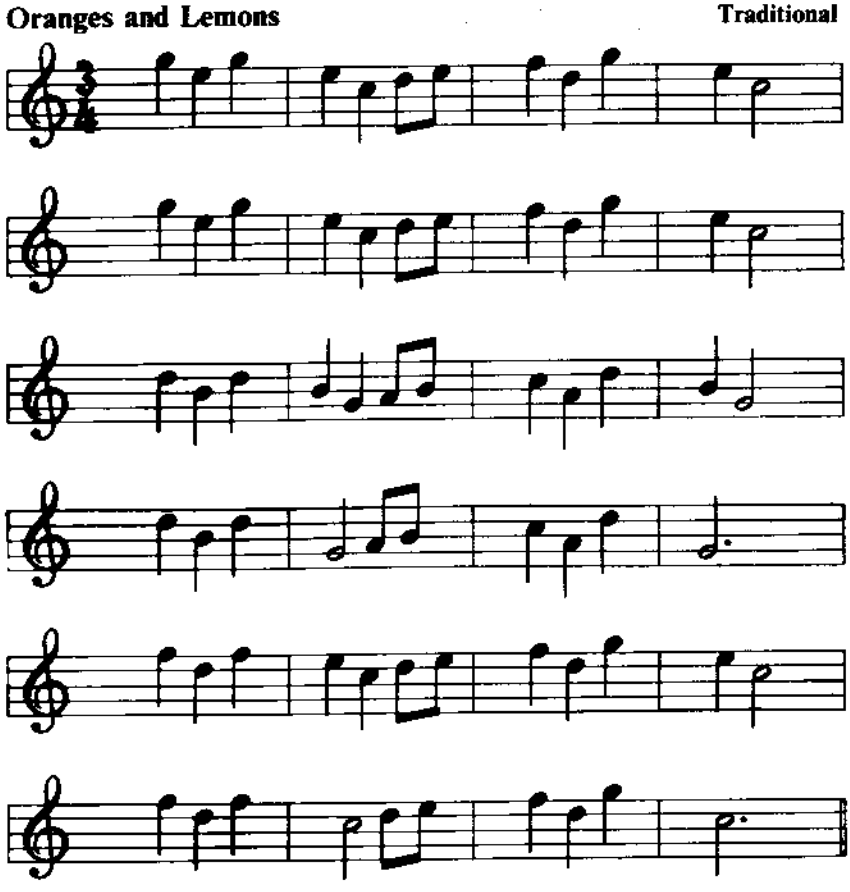
constants in the DATA statements than wading through endless SOUND commands, let alone having to enter them in the first place. Another is that you can easily change certain arguments for all or part of the tune. Try changing, for example, the SOUND command in the new tune program as follows:

```
30 SOUND 1,per*2,dur
```

There couldn't be a simpler way of shifting a sound by one octave.

ORANGES AND LEMONS

Now let's get back to music and have a look at a new tune – a waltz this time:



Like many tunes this is made up of a series of four-bar phrases, but the interesting thing is that there are only two basic patterns of notes in all six phrases. The first two phrases are, of course, identical. But the third phrase is also the same only it is

transposed down by three semitones. The fourth phrase uses the second pattern although it is only slightly different to the first one, but the fifth phrase is the first pattern again. Finally, the sixth phrase is the same as the fourth one but transposed *up* five semitones.

The tune in BASIC

```
100 'oranges and lemons
110 '
120 FOR phrase=1 to 6
130'
140 IF phrase=1 then restore 360
150 IF phrase=2 then restore 360
160 IF phrase=3 then restore 360
170 IF phrase=4 then restore 410
180 IF phrase=5 then restore 360
190 IF phrase=6 then restore 410
200 '
210 FOR note=1 to 12
220 READ pitch,duration
230 '
240 IF phrase=1 then period=pitch
250 IF phrase=2 then period=pitch
260 IF phrase=3 then period=pitch*1.335
270 IF phrase=4 then period=pitch
280 IF phrase=5 then period=pitch
290 IF phrase=6 then period=pitch/1.335
300 '
```

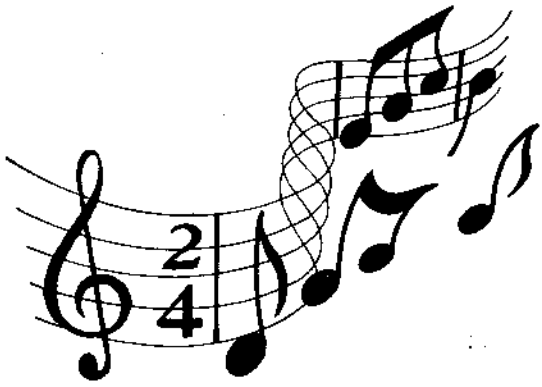
```
310 SOUND 7,period,duration
320 NEXT note
330 NEXT phrase
340 END
350 '
360 'first pattern
370 '
380 DATA 159,40,190,40,159,40,190,40,239,40,213,20
390 DATA 190,20,179,40,213,40,159,40,190,40,239,80
400 '
410 'second pattern
420 '
430 DATA 213,40,253,40,213,40,319,40,319,40,284,20
440 DATA 253,20,239,40,284,40,213,40,319,80,319,40
```

As you can see, this program consists of one loop that plays the notes in a phrase nested inside a second loop that selects the phrase to be played. The phrase selector uses the RESTORE command to set the pointer to the appropriate block of data statements, so that the READ command in line 220 can put the appropriate data into the variables 'pitch' and 'duration'. The correlation between musical notes and the figures in the data statements read into 'pitch', is given in Appendix VII of the User Guide.

The intermediate variable, 'pitch', is then used to generate the variable 'period' according to a second phrase test. An approximation of the five semitone transpositions are achieved by multiplying and dividing 'pitch' by 1.335.

We now arrive at our solitary SOUND command in line 320. You will notice that the channel value is 7, so that all three channels function simultaneously.

If you look at the music a few pages back, you will notice that the second pattern doesn't have 12 notes at all; it only has 10. So how come we have 12 constants in the DATA statements? If you look closely, you will see that what we have done is to break both the two long Gs into two.



Further development

Our melody above was kept deliberately simple so that you could understand the general principles involved. No attempt was made to vary the volume between phrases, and the tone and volume envelopes were ignored completely. So why not try modifying the program to include these things? You could also nest the main loop inside a further one which repeated the tune a certain number of times. How about a different volume envelope for each time round the loop?

Another thing would be to plug your hi-fi into the stereo connection at the back of the CPC464 (the mini-jack socket marked 'I/O'). You could then change the channel value between phrases so that you get a 'call and answer' effect between the two stereo channels.

Anyway, you should now know enough to program almost any single-part melody into the CPC464 and are probably quite capable of making good use of the creative effects possible with the SOUND commands in Amstrad BASIC.

SERIOUS STUFF

The next program on Datacassette A is MINUET by J. S. Bach. It is unlikely that any other composer has had his music transposed for so many different instruments. Even when he originally wrote a piece for harpsichord, it may well have since been adapted for piano, organ, guitar, orchestra, or brass band. So here we have a continuation of the trend – a version for the Amstrad CPC464 Colour Personal Computer.

Apart from being a very good rendering of the piece, there are

some details of this program that are worth discussing - before or after you have played it!

Our intention is to show you quite how good a piece of serious music can sound on the CPC464. However, it was necessary to use commands and techniques that you won't find explained in this part of the course. But there is nothing to stop you having a look at it on the screen if you wish.

The first thing you will notice about it is that three voices are used to give melody, accompaniment and bass lines. This requires a fairly detailed description of the more subtle operations of the three sound channels, and will have to wait until Part 3 of this course. You may also find unfamiliar commands used here and there. Again, unless you want to teach yourself from the User Guide, these will have to wait.

Having said all this, it is worth listing the program on the screen to see how it was put together.

TESTING

Don't worry, this isn't a test of musical knowledge. Run SAT10 to check up on your understanding of the keywords and techniques used in this chapter.

Chapter 11
ADVENTURE

One of the most popular sorts of computer game is the adventure, in which you enter into a world of fantasy that only exists in the mind of the programmer. By typing in commands and information you become an integral part of a saga that the computer creates by telling (through the screen) where you are in time and space, what your surroundings are and the latest event in the story. The fascination lies in the fact that you can influence the course of events and that the game is played out differently every time.

Most adventure games are based on the idea of a maze, or a building with lots of rooms, which you have to find your way around in search of various useful or valuable objects. There may also be traps and hazards or even (in the story) mortal danger. The ultimate aim is usually the retrieval of a particular object and successful escape.

Load the next program from Datacassette A, ADVENTUR. It is the example we are going to use for teaching you how to write your adventure games.

ROLAND IN THE HOUSE

ADVENTUR is very typical of this sort of program. The story is that you want to spend a quiet afternoon programming on your CPC464, but someone has left all the bits you need in different rooms of the house. The object is to find them all and set up your system somewhere without interruptions.

There are three sorts of command:

- *Direction commands:*
  n = north
  e = east
  s = south
  w = west
  u = up
  d = down
- *One-word commands:*
  help
  look
  list
  quit
- *Two-word commands:*
  get . . . .
  drop . . . .
  open . . . .
  close . . . .
  pull . . . .
  plug . . . .

Whenever you get the prompt sign (>), enter one of the above commands. You can always type 'help' if you get really confused.

PLAYTIME

This is where you take some time off and just enjoy playing the game. Even a simple adventure like this can be quite addictive, so don't be surprised if you don't get back to this book for a while.

JUST TO RECAP

When you have played this adventure a few times and have found out where everything is and what you need to complete the game, you may have several ideas about how you would like to change it. Perhaps you would like to add some more rooms and objects, or perhaps you would like to rename all the rooms to make them into caves. Whatever your ideas, you will be unable to make any changes until you understand how the program has been put together.

When you study the listing later in this chapter you may be totally put off by its apparent complexity. Please don't be; you will soon see how simple it really is and should be able to recognise the commands used, all of which have been explained in previous chapters.

There are, however, a few points that you need to know about before you begin.

A few points

You will have noticed some variables with a percent sign, %, after them. The first of these occurs on line 500:

```
FOR i% = 0 TO 5
```

This percent sign tells the CPC464 to handle the variable in a special way.

When the CPC464 is dealing with numbers it uses what is called 'floating point' arithmetic when performing calculations and storing the numbers. This is very sensible really since it allows you to use numbers that have a fractional part, i.e. lots of decimal places, like pi, (3.1415926). Numbers such as these are called 'real', because they are!

It was explained in Part 1 of this course that despite the cleverness of the CPC464 it is possible to end up with an answer that is not quite what you expected. For example, suppose you had performed a calculation which you have worked out should give the answer 5. You have put in a program line:

```
IF x=5 THEN GOTO 2000
```

However, you find that the CPC464 refuses to go to line 2000 so you get it to print the value of 'x' and you get the answer 4.9999999. What happened is that somewhere in the working a very small inaccuracy got multiplied enough for it to matter. You don't believe it can happen? Try this:

```
10 FOR n=1 TO 10 STEP .15

20 PRINT n

30 NEXT n
```

Bearing such things in mind, you know that you can overcome such problems by using keywords like ROUND or INT.

Prevention is better than cure though and you can prevent decimal errors by telling the CPC464 to treat a number as an integer. That is what the percent sign is for. You see, even when calculating 5+4, the CPC464 still uses floating point like this:

5.0000000 + 4.0000000 = 9.0000000

You or I would not do that. When we see an integer number we use integer arithmetic, when we see a fractional number we use

floating point arithmetic. The CPC464 has to be told everything. In an adventure game there is little or no need of fractional numbers so we can work in integers, thus ensuring that errors do not occur and sometimes making calculations faster.

True or false?

As you looked through the listing you may have thought that some of the IF statements looked incomplete, for instance, line 1010:

```
IF INSTR(inv$,command$) THEN GOSUB 1730
```

The INSTR function is not compared to anything, so how can a decision be made? In Chapter 7 brief mention was made about the logical value of a condition. It will have a value of -1 if true and 0 if false. In the line:

```
IF x=5 THEN...
```

the condition is x=5. You can find out whether or not it is true by entering:

```
PRINT x=5
```

You can do this for any condition because the answer can only be true or false. Fair enough you might say, but what about this line:

```
IF x THEN...
```

If 'x' what? The logical value of the condition 'x' depends on the actual value of 'x'. The condition 'x' will be true if 'x' is not zero and false if it is. It's as simple as that. The keyword INSTR returns a number that gives the position of the first character of the searched-for string in the search string. If the searched-for string is found, the result will be non-zero (true); if not it will be zero (false).

Now you should be able to understand how line 1010 (and all the others like it) are able to work. It is a very useful concept.

DESIGN OF 'ADVENTUR'

Imagine you had to design an adventure game from scratch (don't worry, you won't have to here). In Part 1 of this tutorial guide (Chapter 9) a programming technique called Programming Development Language, or PDL for short, was introduced as a method of training you to think in a logical, structured way. We gave an example of a postman delivering mail to a street of houses.

The rest of this chapter will now show you:

- ☐ How the PDL translates into a working program
- ☐ How to modify the program to change the story

A little PDL

There are many ways of writing PDL, and no-one ought to say that their way is better than anyone else's. Apart from the limited number of keywords used, the main object is to make sure that the routines and subroutines are nested in logical structures - hence the term (which you may have already heard) *structured programming.*

Below is the PDL of our adventure game, and you can see that although the BASIC takes 10K of memory, the PDL outline is very compact.

```
If command entered
Then make sense of command
     In case:
          move command-
               If direction not allowed
               Then give message
               Else change location
          action command-
               If action not possible
               Then give message
               Else carry out action
          instruction (help, list, quit, look)
               carry out instruction
     If end of game conditions satisfied
     Then congratulations
Else wait for command
```

Obviously, this is a broad outline and does not include the detailed subroutines necessary to carry out the detailed tasks. But it gives a nice, clear overview of the program and can be used as an 'index' to the different tasks to be carried out.

```
100 ' Arnold Adventure
110 '
120 ' DA  30/9/84
130 '
140 ' Initialise
150 '
160 CLEAR
170 scrwidth=40 ' or change to 80 and use MODE 2
180 '
190 MODE scrwidth/40-0.25
200 BORDER 24
210 INK 0,3: INK 1,24
220 PAPER 0: PEN 1
230 LOCATE 12,5
240 PRINT"Arnold Adventure"
250 LOCATE 1,8
260 PRINT" In  this  adventure  you are trying to"
270 PRINT" have a quiet afternoon playing on your"
280 PRINT" CPC464 but someone has  left  all  the"
290 PRINT" things you need lying around the house"
300 PRINT
310 PRINT" The object is to find them all and set"
320 PRINT" up your system somewhere  without  any"
330 PRINT" interruptions. Beware of the cat!"
340 PRINT
350 PRINT"         Press any key to start"
360 WHILE INKEY$="" : WEND
370 CLS
380 BORDER 23:INK 0,23
390 INK 1,1:INK 2,5:INK 3,8
400 '
410 ' Read map
420 '
430 rooms=11 : items=11  'number of rooms and items(-1)
440 fixed=7 ' the first (fixed-1) items are movable, rest
are not
450 held=0:heldmax=4  ' max number of objects holdable
460 goes=0 ' number of goes
470 '
480 '
490 DIM ex$(5) ' optional
500 FOR i%=0 TO 5
510 READ ex$(i%) : NEXT
520 '
```

```
530 ' Read rooms
540 '
550 DIM loc$(rooms),dir$(rooms),dest$(rooms)
560 FOR i%=1 TO rooms
570 READ loc$(i%),dir$(i%),dest$(i%)
580 NEXT
590 '
600 ' Read objects
610 '
620 DIM object$(items),objloc(items)
630 FOR i%=0 TO items
640 READ object$(i%),objloc(i%)
650 NEXT
660 '
670 closed=-1:open=0:onn=-1:off=0
680 backdoor=closed:frontdoor=closed
690 switch=off:plugged=off
700 DIM sw$(1) : sw$(0)="off" : sw$(1)="on"
710 position=3 'initial position
720 direction$="NSEWDU"
730 get$="GETPICKTAKELIFT"
740 put$="DROPUTLEAVE"
750 pul$="PULLPUSHPRESSTURN"
760 inv$="INVENTORYLIST"
770 clo$="SHUTCLOSE"
780 ope$="OPEN"
790 loo$="LOOKWHEREEXAMINE"
800 qui$="QUITENDSTOPFINISH"
810 plu$="PLUGCONNECT"
820 hel$="HELP"
830 nul$=CHR$(0) ' for blanking
840 '
850 GOSUB 2140 : REM look (for starters)
860 '
870 ' Main loop
880 '
890 PRINT
900 INPUT ")",command$
910 IF command$="" THEN 890
920 IF LEFT$(command$,1)=" " THEN command$=MID$(command$,
2) : GOTO 910
930 command$=UPPER$(command$)
940 goes=goes+1
950 '
960 ' Now parse the string
970 '
980 ' one word
990 '
1000 IF LEN(command$)=1 THEN GOSUB 1270 : GOTO 1190 :REM
move
1010 IF INSTR(inv$,command$) THEN GOSUB 1730 : GOTO 1190
REM inventory
```

```
1020 IF INSTR(loo$,command$) THEN GOSUB 2120 : GOTO 1190 :
REM look
1030 IF INSTR(qui$,command$) THEN GOSUB 2070 : GOTO 1190 :
REM quit
1040 IF INSTR(hel$,command$) THEN GOSUB 2380 : GOTO 1190 :
REM help
1050 '
1060 ' two words
1070 '
1080 d%=INSTR(command$," ")
1090 IF d%=0 THEN PRINT" I don't understand that":GOTO 1210
1100 noun$=MID$(command$,d%+1) : command$=LEFT$(command$,
d%-1)
1110 IF INSTR(noun$," ") THEN P$="Only two words at a time
please":GOSUB 2680: GOTO 890
1120 IF INSTR(get$,command$) THEN GOSUB 1380 : GOTO 1190 :
REM get
1130 IF INSTR(put$,command$) THEN GOSUB 1520 : GOTO 1190 :
REM put
1140 IF INSTR(pul$,command$) THEN GOSUB 1650 : GOTO 1190 :
REM pull
1150 IF INSTR(clo$,command$) THEN GOSUB 1840 : GOTO 1190 :
REM close
1160 IF INSTR(ope$,command$) THEN GOSUB 1950 : GOTO 1190 :
REM open
1170 IF INSTR(plu$,command$) THEN GOSUB 2270 : GOTO 1190 :
REM plug
1180 PRINT "I don't know how to do that";
1190 '
1200 GOSUB 2470 ' is adventure complete
1210 '
1220 GOTO 890
1230 '
1240 '
1250 ' Move
1260 '
1270 d%=INSTR(direction$,command$)
1280 IF d%=0 THEN P$="I don't know what you mean":GOSUB
2680: GOTO 1340
1290 d%=INSTR(dir$(position),command$)
1300 IF d%=0 THEN P$="You can't go in that direction":GOSUB
2680: GOTO 1340
1310 IF (position=4 OR (position=3 AND d%=1)) AND backdoor=
closed THEN P$="The back door is closed":GOSUB 2680: GOTO
1340
1320 IF (position=1 OR (position=2 AND d%=2)) AND frontdoor=
closed THEN P$="The front door is closed":GOSUB 2680: GOTO
1340
1330 position=ASC( MID$( dest$(position),d%,1 ) )-64 :
GOSUB 2140
```

```
1340 command$=nul$ : RETURN
1350 '
1360 ' get
1370 '
1380 o%=0
1390 FOR i%=0 TO items
1400 IF INSTR(UPPER$(object$(i%)),noun$) THEN o%=o%+1 :
GOSUB 1440 : GOTO 1430
1410 NEXT
1420 IF o%=0 THEN P$="I don't know what that is":GOSUB 26
1430 command$=nul$ : RETURN
1440 IF i%)=fixed THEN P$="You can't take the "+object$(i
:GOSUB 2680: GOTO 1480
1450 IF held=heldmax THEN p$="Your hands are full":GOSUB
2680:GOTO 1480
1460 IF objloc(i%)=position THEN P$= "Got it":GOSUB 2680:
held=held+1:objloc(i%)=0 : GOTO 1480
1470 IF objloc(i%)=0 THEN P$="You've already got it":GOSU
2680 ELSE P$="It's not here":GOSUB 2680
1480 RETURN
1490 '
1500 ' put
1510 '
1520 o%=0
1530 FOR i%=0 TO items
1540 ff%=0
1550 IF INSTR(UPPER$(object$(i%)),noun$) THEN o%=o%+1 :
GOSUB 1600
1560 IF o%)0 THEN i%=items
1570 NEXT
1580 IF o%=0 THEN P$="I don't know what that is":GOSUB 26
1590 command$=nul$ : RETURN
1600 IF objloc(i%)=0 THEN P$="You drop the "+object$(i%):
GOSUB 2680:held=held-1:objloc(i%)=position ELSE P$="You
haven't got a " +object$(i%):GOSUB 2680
1610 RETURN
1620 '
1630 ' pull
1640 '
1650 d%=INSTR("HANDLEDOOR",noun$)
1660 IF d%=0 THEN P$="No point":GOSUB 2680: GOTO 1690
1670 IF d%=7 THEN ' goto the open door routine
1680 IF position=2 THEN switch=-(1+switch): P$="The handl
clicks into the "+sw$(ABS(switch))+" position":GOSUB 2680
ELSE P$="I see no handle":GOSUB 2680
1690 command$=nul$ : RETURN
1700 '
1710 ' inventory
1720 '
1730 temp=0
1740 P$="You are carrying ":GOSUB 2680
```

```
1750 o%=0
1760 FOR i%=0 TO items
1770 IF objloc(i%)=temp THEN P$="a "+object$(i%)+" ":
GOSUB 2680: o%=o%+1
1780 NEXT
1790 IF o%=0 THEN P$="nothing ":GOSUB 2680
1800 command$=nul$ : RETURN
1810 '
1820 ' close
1830 '
1840 d%=INSTR("DOOR",noun$)
1850 IF d%<>1 THEN P$="I can't close that":GOSUB 2680:
GOTO 1910
1860 IF position>4 THEN P$="What door?":GOSUB 2680: GOTO
1910
1870 IF (position=1 OR position=2) AND frontdoor=closed
THEN P$= "The front door is already closed":GOSUB 2680:
GOTO 1910
1880 IF (position=3 OR position=4) AND backdoor=closed
THEN P$="The back door is already closed":GOSUB 2680:
GOTO 1910
1890 IF (position=1 OR position=2) THEN P$="You have closed
the front door":GOSUB 2680: frontdoor=closed
1900 IF (position=3 OR position=4) THEN P$="You have closed
the back door":GOSUB 2680: backdoor=closed
1910 command$=nul$ : RETURN
1920 '
1930 ' open
1940 '
1950 d%=INSTR("DOOR",noun$)
1960 IF d%<>1 THEN P$="I can't open that":GOSUB 2680: GOTO
2030
1970 IF position>4 THEN P$="What door?":GOSUB 2680: GOTO
2030
1980 IF (position=1 OR position=2) AND frontdoor=open THEN
P$="The front door is already open":GOSUB 2680: GOTO 2030
1990 IF (position=3 OR position=4) AND backdoor=open THEN
P$="The back door is already open":GOSUB 2680: GOTO 2030
2000 IF (position=1 AND frontdoor=closed) OR (position=4
AND backdoor=closed) THEN P$="Hard luck. You have locked
yourself out":GOSUB 2680:GOTO 2580
2010 IF position=2 THEN P$="You have opened the front door":
GOSUB 2680: frontdoor=open
2020 IF position=3 THEN P$="You have opened the back door"
:GOSUB 2680: backdoor=open
2030 command$=nul$:RETURN
2040 '
2050 ' quit
2060 '
2070 P$="Do you really want to stop (Y/N) ":GOSUB 2680
2080 INPUT "", command$
```

```
2090 IF UPPER$(command$)="Y" THEN END
2100 command$=nul$ : RETURN
2110 '
2120 ' look
2130 '
2140 p$="You are in the "+loc$(position):GOSUB 2680
2150 p$=". Exits lead ":GOSUB 2680
2160 FOR i%=1 TO LEN(dir$(position))
2170 temp$=MID$(dir$(position),i%,1)
2180 p$=ex$(INSTR(direction$,temp$)-1)+", ":GOSUB 2680
2190 NEXT
2200 p$="and you can see ":GOSUB 2680
2210 temp=position
2220 GOSUB 1750
2230 command$=nul$ : RETURN
2240 '
2250 ' plug in
2260 '
2270 IF position=5 THEN p$="Your family are watching TV an
stop you.":GOSUB 2680:GOTO 2340
2280 IF objloc(1)()10 THEN p$="The monitor isn't here":
GOSUB 2680:GOTO 2340
2290 IF objloc(3)()10 THEN p$="The computer isn't here":
GOSUB 2680:GOTO 2340
2300 IF position()10 THEN p$="There's no socket in here.":
GOSUB 2680: GOTO 2340
2310 IF INSTR("CTM640MONITORAMSTRADCPC464COMPUTER",noun$)
=0 THEN p$="I can't plug that in.":GOSUB 2680: GOTO 2340
2320 plugged=onn
2330 IF switch=off THEN p$="Nothing happens.":GOSUB 2680
ELSE p$="Your computer springs into life.":GOSUB 2680
2340 command$=nul$ : RETURN
2350 '
2360 ' help
2370 '
2380 RESTORE 3190
2390 p$="The commands are :" : GOSUB 2680
2400 FOR h1=0 TO 25
2410 READ p$:p$=" "+p$:GOSUB 2680
2420 NEXT
2430 command$=nul$ : RETURN
2440 '
2450 ' "finished?"
2460 '
2470 fr=10 'finish room
2480 IF position()fr THEN RETURN
2490 ready=(objloc(1)=fr) AND (objloc(3)=fr) AND (objloc(4
=fr) AND (objloc(5)=fr) AND (objloc(6)=fr)
2500 ready=ready AND switch AND plugged
2510 catout=((objloc(0)=4) OR (objloc(0)=1)) AND backdoor
=closed AND frontdoor=closed
```

```
2520 IF NOT catout AND objloc(3)=fr AND objloc(0)()0 THEN
p$=" The cat keeps jumping on the keyboard":GOSUB 2680:
objloc(0)=fr
2530 ready=ready AND catout
2540 IF NOT ready THEN RETURN
2550 PRINT
2560 p$="Well done. You have completed the adventure in":
GOSUB 2680
2570 p$=STR$(goes)+" goes. Now you can enjoy a quiet
afternoon's computing":GOSUB 2680
2580 PRINT:PRINT
2590 P$="Do you want to play again? (Y/N) ":GOSUB 2680
2600 INPUT "", command$
2610 IF UPPER$(command$)="Y" THEN RUN
2620 END
2630 '
2640 ' Printing routine
2650 ' Enter with P$=text to be printed
2660 ' Return with cursor in following position
2670 ' therefore CRLF needed before )
2680 p$=p$+" "
2690 ln=LEN(p$):xp=POS(#0)
2700 IF INSTR(p$," ")=0 AND (ln+xp))(scrwidth-2) THEN
PRINT:xp=0
2710 FOR pr%=1 TO ln-1
2720 sp=INSTR(pr%,p$," ")
2730 IF sp+xp )(scrwidth-2) THEN PRINT :xp=xp-scrwidth
2740 PRINT MID$(p$,pr%,1);
2750 NEXT
2760 RETURN
2770 '
2780 '
2790 ' Directions
2800 '
2810 DATA North,South,East,West,Down,Up
2820 '
2830 ' Locations No:(rooms)
2840 ' name,directions out,resp. room numbers out (A=1
B=2...)
2850 '
2860 DATA front garden,N,B
2870 DATA hall,NSEU,CAEH
2880 DATA kitchen,NS,DB
2890 DATA back garden,S,C
2900 DATA lounge,W,B
2910 DATA bathroom,E,H
2920 DATA front bedroom,N,H
2930 DATA bedroom landing,NSEWUD,IGJFKB
2940 DATA back bedroom,S,H
2950 DATA box room,W,H
2960 DATA attic,D,H
```

```
2970 '
2980 ' Objects  No:(items+1)
2990 ' name,init pos ( (=rooms)
3000 '
3010 DATA black cat,2
3020 DATA CTM640 monitor,3
3030 DATA TV set,5
3040 DATA Amstrad CPC464 computer,7
3050 DATA work table,11
3060 DATA kitchen chair,3
3070 DATA computer manual,9
3080 '
3090 ' fixed objects from no.7
3100 '
3110 DATA red handle,2
3120 DATA power socket,10
3130 DATA bed of roses,1
3140 DATA rusty dustbin,4
3150 DATA priceless old painting,11
3160 '
3170 ' Help command
3180 '
3190 DATA CLOSE,CONNECT,DROP,END,EXAMINE
3200 DATA FINISH,GET,HELP,INVENTORY,LEAVE,LIFT
3210 DATA LIST,LOOK,OPEN,PICK,PLUG,PRESS,PULL
3220 DATA PUSH,PUT,QUIT,SHUT,STOP,TAKE,TURN,WHERE
```

Putting it into BASIC

If you have already studied and understood the listing of the adventure then you can skip this section, otherwise read on. The program begins at the beginning and prints the title page. It waits for you at line 360. Pressing any key starts the program proper.

Lines 430–460 set some variables whose purpose will become clear later. Lines 490 to 510 read the first six items from the data list (line 2810) in array 'ex$'. Lines 550 to 580 read the next 33 data items (3*11, lines 2860 to 2960) into arrays 'loc$', 'dir$' and 'dest$', respectively. Lines 620 to 650 read the next 24 data items into arrays 'object$' and 'objloc'. From line 670 more variables are set up, including 11 curious string variables. Each of these is formed from a number of words which can all mean the same thing and will be used in the main loop of the program to help the CPC464 to work out what your commands mean.

Let's pause here for a recap on arrays, previously discussed in Chapter 2. Arrays allow you to access 'lists of things' much more

easily. To illustrate this, RUN the adventure and press **ESC** when you get the command prompt.

Now type in the following lines:

```
5000 FOR n=0 TO 11

5010 PRINT n;" ";loc$(n)

5020 NEXT n
```

Do not RUN this or all the variables will be lost. Instead type:

```
CLS:GOTO 5000
```

You will get a list of all the data items, called elements, of array 'loc$'. Try it with the other arrays. All of these are one-dimensional arrays and are the basis of the entire program. Using these arrays the CPC464 can keep track of what is where and in what condition.

Back to the listing now. Messages are printed on the screen by the subroutine that begins at line 2680. What is to be printed is assigned to variable 'p$'. At line 2140 we find that 'p$' is formed from two strings that are joined end to end. The first string is a standard one, 'You are in the', and the second will depend upon your position. Initially your position is in the kitchen, because variable position was set to three at line 710 and element three of array 'loc$' is 'kitchen'.

The subroutines at 2150 and 2200 put 'p$' together in the same way to print out the exits and contents of the location, respectively. Once that is done it is up to you to input your commands.

When you do this, whatever you type in is given the variable name 'command$'. This then has to be analysed and checked before any instruction can be carried out. This 'making sense' of a string is called 'parsing', hence the programmer's comment on line 960. Here the INSTR function demonstrates its worth. If line 1000 is passed over then the following lines search your input to find out if there are any recognisable words within it. Any that are recognised (those contained in those curious variables on lines 720 to 820) are acted upon: otherwise you get a comment back. Each action is performed on its own subroutine and each of these is clearly marked on the listing.

Only when you get all the correct objects in the correct room and some other conditions are met does the game end. By studying the routine beginning at line 2470 you can find out what you need to do in order to end the game. (If you want to find out by playing the game and not cheating, then skip the next three paragraphs!)

Begin at line 2470. The finish room is room 10, i.e. the room that is the tenth element of array 'loc$'. This is the box room. At line 2480 a test is made to see if your current position is room 10, i.e. variable position=10. If it is not, then the game continues.

If it *is* the correct room, we move on to line 2490 which looks complex but isn't really. The value of the variable 'ready' will depend upon the values of the five conditions in the line. If 'ready' is to be true, all five conditions must be true. These five conditions are simply elements of array 'objloc', which are the objects you have to collect and put in the room. If you look through the data list (lines 3010 to 3070) you will see that elements 1, 3, 4, 5 and 6 are the CTM640 monitor, Amstrad CPC464 computer, small work table, kitchen chair and computer manual. The number after each item in this list is the number of the room it is in to start with. It is this number that must equal 10 at the end.

Variable 'ready' does not depend only on the objects being correct. You can see at line 2500 that 'ready' also depends upon variables 'switch' and 'plugged' being true. These are set from lines 1650 and 2270, respectively. And finally, at line 2530 you see that 'ready' also depends on 'catout' which is set at line 2510. In order for 'catout' to be true, element 0 of array 'loc$' (the cat) must either be in location 1 or 4 (front or back garden) and both doors must be shut. If not, and you are not holding the cat (line 2520), then the cat jumps on the keyboard.

To sum up, when you have the monitor, CPC464, manual, chair and table in the box room, turned the big red handle, plugged in, put the cat in the garden and closed the front and back doors, that is the end of the game!

This is the substance of this adventure game. As you move around, pick things up and change the states of things like doors, then the arrays and other variables are altered and updated. If you look back a few pages, you will see that is exactly how our original PDL design was structured.

YOUR HOUSE COULD BE A CASTLE

To further aid your efforts in understanding how this adventure works, we will now make some modifications to the program. Another location will be added, more objects too, and another object to get to the box room. Interested? Then read on...

How about a garage to the east of the front garden? The first thing to do is change the variable 'rooms' on line 430 from 11 to 12. This alters the DIMensioning of the arrays later. Next, add a twelfth DATA statement to the list beginning line 2860. Each line has three pieces of data. First the name of the location - garage. The second part is the direction(s) out of the location. This will be west into the front garden - W. Finally, a code letter for the location(s) adjoining. The locations are coded A to K for elements 1 to 11 of array 'loc$'. The front garden is element 1 so the letter is A. Enter this line:

```
2970 DATA garage,W,A
```

Now we must alter the data concerning the front garden. Either EDIT or retype line 2860 to read:

```
2860 DATA front garden,NE,BL
```

The E is added because you can now go east out of the garden and the L is added because being the twelfth element of the array the garage has code letter L (twelfth letter of alphabet).

If you now RUN the program you will be able to go into and out of the garage, but of course there is nothing there.

So, now to put something in the garage. A car seems like a good idea! First you need to change the variable 'items' on line 430 from 11 to 12, then add a twelfth DATA line to the objects list beginning at line 3010. Note that some objects are movable but others are not. The number of movable objects is determined by the variable 'fixed' on line 440. At present this has value 7 which means that the first 7 items can be moved; the rest cannot. We don't want to be able to move the car, so don't change 'fixed'. And

we must make sure that our new DATA line does not become one of the first seven. Enter the following line:

```
3160 DATA second hand car, 12
```

The 12 is the location number of its position, the garage - when you RUN the program there will be a car there!

Now let's put something in the garage that you can take away - a bicycle. Once again change variable 'items' (line 430), this time to 13. Now change variable 'fixed' from 7 to 8.

Our DATA line must now be one of the first eight, so we enter:

```
3080 DATA brand new bicycle, 12
```

Now RUN. You should be able to pick up the bike, but not the car.

Finally we can alter the finishing conditions so that you need a datacassette in the boxroom before the game will end.

First, increase variables 'items' and 'fixed' by one again, then add another new line:

```
3075 DATA datacassette, 9
```

The 9 is the number of the back bedroom, the ninth element of array 'loc$'. To make the datacassette one of the items you must collect, you must add it to the line that checks if variable 'ready' is true. So, to the end of the line 2490 add:

```
AND (objloc(7)=fr)
```

The datacassette is the seventh element in its array and now must be in the finish room. RUN the program to make sure it works!

By now you should have a good idea of how the program works. So why not add more rooms, a cellar, or a shed, or, better still, why not change your house into a castle with banqueting halls, dungeons, towers, maidens to be rescued, and dragons to be fought. It is quite feasible. Just take your time, think logically, step by step. Even if you are not sure, have a go.

The best way to learn is by experiment. Learning is an adventure too!

TESTING

Having had your wits tested by playing ADVENTUR, it is now time to have your memory tested by running SAT11.

Chapter 12
WHAT NEXT?

ADVANCED AMSTRAD

Long, long ago (back in Part 1, in fact) there was a hint that the numeric keyboard on the right of your CPC464 keyboard could be persuaded to produce effects other than merely making life more easy for the typing-in of numeric values. You probably already use the CTRL-ENTER short-cut for loading and running programs; what you are about to learn is how you can put your own time-saving, single-key functions into the CPC464 to suit the type of programs that you write most often, or a particular one you are developing at a given moment.

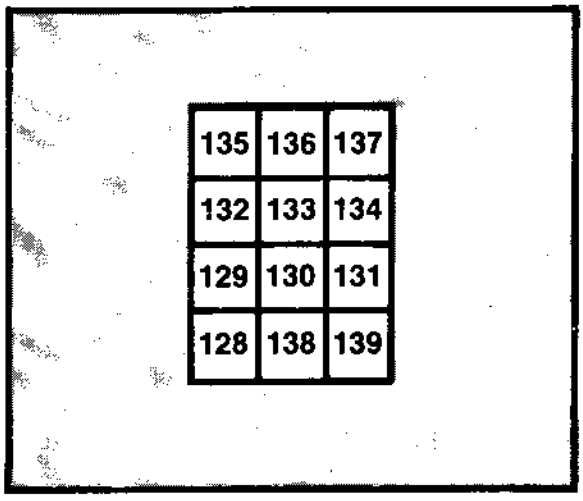
One of the defaults that operate on initial power-on of the CPC464 is the definition of which characters are generated when the keys are pressed on the numeric keypad. You can modify this at any time. First of all press the zero key on the numeric keypad. You get a zero on the screen, don't you? Now try entering the following:

```
KEY 128,"hello"
```

Surprise, surprise, when you press the zero key again you now get 'hello', instead of zero. Our new keyword, KEY, allows you to assign whatever string you wish to any of the 12 keys of the numeric keypad. The command structure is:


```
KEY(key number),(string expression)
```

The argument <key number> is an integer between 128 and 140, which refers to actual key positions as shown in the following diagram.



You will notice that the numbers only go up to 139 and not 140. The thirteenth key is our old friend ENTER which, with CTRL, you know very well gives:

```
run"
```

So the <string expression> can therefore be a command for direct execution during program input. Enter the following:

```
KEY 129,"mode 2"
```

If you now press the '1' key you will see:

```
mode 2
```

come up on the monitor. If you now press the RETURN key, the command will be executed exactly as though you had typed it in directly.

And this isn't all! You can also type in:

```
KEY 129,"mode 2"+chr$(13)
```

Remember this number. It is the ASCII code for CR, or, in plain English, *C*arriage *R*eturn (the carriage referred to is that of the electromechanical typewriters which used to be the only way of talking to computers). As you may have realised, this is exactly what you get when you press the ENTER key. So by pressing key '1' on the numeric keypad the command is executed immediately and you are automatically put into MODE 2. Try it for yourself.

Here is an example of a short program you might like to put on a special 'program development' cassette, along with the setting up of values for your favourite BORDER, PEN, PAPER and

INK:

```
10 key 128,"mode 0"+chr$(13)

20 key 129,"mode 1"+chr$(13)

30 key 130,"mode 2"+chr$(13)

40 key 131,"list"

50 key 132,"list"+chr$(13)

60 key 133,"save""

70 key 134,"renum"

80 key 135,"auto"
```

When you are writing and debugging a long program, having these functions on single keys will save a lot of tedium. You may find that different types of program require different functions. If there are a lot of screen messages, for example, you may like to assign 'print' to a key to save you having to type it in every time.

FAST LOADING

Another one of the CPC464's power-on defaults is the speed at which programs are written onto the datacassette. For maximum reliability, both the datacassettes that accompany this course are written at this speed (1000 baud for the technically minded). It is possible to double the writing speed by entering:

```
SPEED WRITE 1
```

Any subsequent SAVE command will then write the program at this new speed (2000 baud).

When programs written at 2000 baud are loaded by the CPC464, it automatically adjusts its reading rate. In fact, you can mix programs written at different speeds on the same cassette. If you use top-quality cassettes and keep the reading head clean, there is little likelihood of any reading problems.

To change the speed back you enter:

```
SPEED WRITE 0
```

HARD COPY

'Hard copy' is computer jargon for computer output or program listings which are printed on paper. This section is intended for anyone who has bought an Amstrad DMP-1 dot matrix printer so that they can get printed output from their CPC464.

If you plug the printer into the connection port at the rear of your CPC464, turn on the power and issue PRINT or LIST commands, nothing will happen. This is because all input and output data is handled in 'streams', and the printer stream has not been specified. There are 10 of these streams, numbered 0 to 9, but we won't be looking at them in detail until Part 3 of this course. For the moment we will limit ourselves to the streams numbered 0 and 8.

The stream is an optional argument in PRINT and LIST commands, and the default is 0. For example:

```
LIST 2000-
```

will produce an output on the screen of all the program lines from number 2000 onwards, since 0 is a screen stream. If, however, you enter:

```
LIST 2000-,#8
```

all the lines from 2000 onwards will be sent to the attached printer rather than to the screen.

Similarly, you can enter:

```
PRINT #8,"This is an example of printer output"
```

and the string will be sent to the attached printer.

If you turn back to the ESTIM program listing in Chapter 5, the subroutine for printing out the estimate appears to have an unused variable '# str'. If you look at the print statements in this subroutine you will see that this is really the variable 'str' in the stream position of the PRINT command, and, since it was never set to anything, it stays at zero - the screen stream. So set 'str=8' and you will get your printed estimate.

DATABASE

Our last program on Datacassette A is possibly the best of the lot.

If you want to catalogue your record collection, your favourite recipes, or the names and addresses of your friends, this is a program for you. It is like an electronic filing cabinet, except that

you can use lots of different labels at the same time. Once you have built up your database file you can look it up under lots of different headings. Soups, for example, or reggae, or whatever.

There is an example of a database file recorded after DATABASE on the cassette. When the main menu asks what option you want, type R; it will then ask for a file name. Enter

```
Countrys
```

and learn a bit about geography.

TESTING

This final test, SAT12, will go through *all* the topics covered in this part of the course. As we said before, it isn't an exam, it's merely an aid for you to check that there isn't anything you should go back over to make sure that you understand before going on to the next stage.

The next stage from now is Part 3 of this course. From now on you will start having to learn about the strange world of binary arithmetic, hexadecimal numbers and interrupts. Good luck!

LIST OF PROGRAMS

Datacassette A contains the following programs in the same order that they are referred to in this book. Datacassette B contains the Self-Assessment Test (SATs) which the reader should complete at the end of every chapter except Chapter 5.

Chapter 1

SHEP

Chapter 2

CHATEAU
VILLAGE

Chapter 3

SINCOS
PIECHART

Chapter 4

DIGITALC
ALARM

Chapter 5

ESTIM

Chapter 6

FLIGHTPL

Chapter 7

WORDPUZL

Chapter 8

BLITZ

Chapter 9

ZOUNDS
ZAPPOW2

Chapter 10

MINUET

Chapter 11

ADVENTUR

Chapter 12

DATABASE
COUNTRYS

LIST OF KEYWORDS

The following is a list, chapter by chapter, of all the Amstrad BASIC keywords covered in this book. Not all the variations and extensions have been dealt with since, although this is not a beginners book, knowledge of the CPC464's internal procedures is necessary for correct understanding.

The keywords marked with an asterisk (*) were introduced in Part 1, but are given a more thorough treatment here.

Chapter 2

DIM
READ
DATA
AUTO
RENUM
RESTORE
DELETE
LIST*

Chapter 3

PEN*
PAPER*
INK*
PLOT*
DRAW*
ORIGIN*
CLG
TAG
TAGOFF

Chapter 4

FOR-NEXT*
IF-THEN-ELSE*
WHILE-WEND

Chapter 5

ZONE
PRINT*
PRINT USING
SPACE$
STRING$
SPC
TAB
INPUT*

Chapter 6

SIN
COS
TAN
DEG
PI
RAD
ATN
SQR
ABS

Chapter 7

INSTR
ASC
VAL
LEN
LEFT$
RIGHT$
MID$
STR$
TIME*
CHR$
AND
OR
XOR
UPPER$

Chapter 8

SPEED INK
RND*
INT
LOCATE*
TEST
INKEY

Chapter 9

SOUND*
ENV*
ENT*

Chapter 12

KEY
PRINT #8
LIST #8
SPEED WRITE

INDEX