An Introduction to Programming the Amstrad CPC464 and 664

R. A. & J. W. PENFOLD

AN INTRODUCTION TO PROGRAMMING THE AMSTRAD CPC464 and 664

ALSO BY THE SAME AUTHORS

No.BP129	An Introduction to Programming the ORIC-1
No.BP133	An Introduction to Programming the DRAGON 32
No.BP139	An Introduction to Programming the BBC Model B Micro
No.BP142	An Introduction to Programming the ACORN ELECTRON
No.BP143	An Introduction to Programming the ATARI 600/800 XL
No.BP147	An Introduction to 6502 Machine Code
No.BP150	An Introduction to Programming the Sinclair QL
No.BP152	An Introduction to Z80 Machine Code
No.BP154	An Introduction to MSX BASIC
No.BP156	An Introduction to QL Machine Code
No.BP158	An Introduction to Programming the Commodore 16/Plus 4
No.BP165	More Advanced Programming with the Amstrad CPC464 and 664
No.BP166	More Advanced MSX Programming
No.BP167	More Advanced Programming with the Sinclair QL
No.BP169	How to De-bug Your Programs
No.BP170	An Introduction to Computer Peripherals

AN INTRODUCTION TO PROGRAMMING THE AMSTRAD CPC464 and 664

by
R.A. & J.W. PENFOLD

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

PLEASE NOTE

Although every care has been taken with the production of this book to ensure that any projects, designs, modifications and/or programs etc. contained herein, operate in a correct and safe manner and also that any components specified are normally available in Great Britain, the Publishers do not accept responsibility in any way for the failure, including fault in design, of any project, design, modification or program to work correctly or to cause damage to any other equipment that it may be connected to or used in conjunction with, or in respect of any other damage or injury that may be so caused, nor do the Publishers accept responsibility in any way for the failure to obtain specified components.

Notice is also given that if equipment that is still under warranty is modified in any way or used or connected with home-built equipment then that warranty may be void.

All the programs in this book have been written and tested by the authors using a model of the Amstrad CPC464 that was available at the time of writing in Great Britain. Details of the graphics modes may vary with versions of the machine for other countries.

First Published – October 1984
Revised and Reprinted – August 1985

British Library Cataloguing in Publication Data
Penfold, R.A.
An introduction to programming the Amstrad CPC 464 (BP153)
1. Amstrad CPC464 (Computer) – Programming
I. Title II. Penfold J.W.
001.64'2 QA76.8.A4

ISBN 0 85934 128 3

Printed and bound in Great Britain by Cox & Wyman Ltd, Reading

PREFACE

The Amstrad CPC464 is in many ways an unusual home computer, and the most obvious aspect of this type is the inclusion of an integral cassette recorder and a monochrome or colour monitor with the machine. However, the package as a whole represents outstanding value, and there are obvious advantages in having the built-in cassette recorder, and a monitor rather than the more usual course of using a television set to provide the display. The fact that the package costs relatively little does not mean that this computer lacks anything in terms of performance. With its good graphics capability, 80 column text mode, and outstanding sound generator it actually ranks as one of the best home computers currently available.

The excellent hardware is well supported by the built-in computer language, Locomotive BASIC. This has a large range of useful commands available, including instructions that support the excellent graphics and sound capabilities of the machine. Although a complex BASIC of this type can be a little intimidating to the beginner, it makes programming much easier once the various instructions and functions have been mastered. This is not as difficult as one might imagine provided things are taken one step at a time, as (hopefully) this book demonstrates.

In the summer of 1985 Amstrad introduced a second machine, the CPC664. The 664 has much in common with the 464 and indeed all BASIC programs for the 464, including those in this book, are completely compatible with the new 664. The main differences are that the 664 has a built-in disc drive in place of the cassette recorder of the 464, plus a few additional BASIC commands, most of these being concerned with graphics. Chapter 11 covers the special features of the 664.

R.A. & J.W. Penfold

CONTENTS

Page

Chapter 1

VARIABLES & ARRAYS

Consider a typical computer game. A feature of all good games is a running score on the screen. From time to time, you will zap an alien or whatever, and the score will increase. The computer has to be able to remember the current score, update it as necessary, then remember the new score.

Within the BASIC computer language, storage of information which has to be altered periodically is done through the use of variables. These are distinct from constants, numbers such as 1, 100, 11.273, which may be included in program lines and which obviously cannot change in value.

So that variables can be referred to within a program, they are given names. These variable names must not be the same as a BASIC keyword – the words that the computer recognises as specific instructions. In Locomotive BASIC, variable names can contain upper and lower case letters and numerals, but must start with a letter. All the letters and numbers used in a name are used to distinguish one variable from another. Many BASICs only use the first two characters, which can cause difficulties. However, Locomotive BASIC regards upper and lower case letters as the same in a variable name. If a variable name ends with a dollar sign ($), it refers to a string variable. These are handled differently to the numeric variables discussed in this chapter, and therefore have a chapter of their own.

Though many programmers use algebra-style single-letter variable names, it is better whenever possible to use longer names that give a clear indication of what the variable is being used to store. For example, if the variable contains the score, call it "score". However, the use of X and Y as variable names for horizontal and vertical co-ordinates in screen plotting is standard practice. Long variable names do, however, take up more memory, but with the Amstrad there is little risk of running out of space!

BASIC accepts keywords typed in upper or lower case

letters. However, keywords appear in upper case in subsequent program listings on the screen. Variable names always remain in the form in which they are typed in. It is therefore a good idea to type all variable names in lower case, so they stand out from the keywords.

All mathematical and logical operations can be performed on variables, and the results of mathematical operations may alter the contents of a variable, or create a new variable. Thus score=score+10 alters the contents of the variable score, price=cost+tax could create a new variable price, and store in this the value of the contents of cost plus the contents of tax. It could also be that the variable price already exists, in which case the contents would be updated.

One important point to remember is that when variables are used on the right-hand side of an assignment statement like the above, they are not altered in value, unless they also appear on the left-hand side, and they are not destroyed. Therefore, in the example above, cost and tax continue unaltered and could be used in (any number of) further calculations if required.

From our first example, it is clear that the way in which variables are used in computing and the way in which they are used in algebra are very different. A line like X=X+10 is a nonsense in algebra, but lines like it are very common in computing. It should always be remembered that a variable is a labelled storage space, the contents of which can be altered as required. Thus the program line score=score+10 really means look up the contents of the memory location labelled score, add 10 to it, and store the result in the memory location labelled score.

Locomotive BASIC also has integer variables which can be used to contain whole numbers only. These are identified by having a per-cent sign as the last character of the variable name. These variables are limited to containing numbers between −32768 and +32767, but have the advantage that they take up less memory space, and arithmetic operations on them can be performed faster.

If a non-integer value is assigned to an integer variable, it is converted to an integer automatically. This is done by

simple truncation to the nearest whole number less-than the non-integer value. Thus 3.3 and 3.897 would both be truncated to 3, and −3.53 would be truncated to −4 (this last one is the cause of much programmer's insomnia!).

LOOPS

One of the main uses of computers is to perform repetitive tasks. When something has to be done a fixed number of times, a variable is the obvious way of counting the number of times it has been done. In fact, so useful is this that there are special statements in BASIC to do it, FOR, STEP, and NEXT, and WHILE . . . WEND.

FOR . . . NEXT loops are the easiest to start with. These loops are used when an action has to be repeated a known number of times, or within defined limits. The special control variable, which, in Locomotive BASIC must be a floating-point variable, is given a starting value and a terminating value in the FOR statement. For example, the statement FOR entries = 0 TO 20 sets the control variable entries to 0, and sets the value at which the loop will terminate to 20. The program statements which are to be repeated are placed after the FOR statement, and after them is placed the line NEXT entries. When this line is executed, program execution loops back to the statement after the FOR statement, and the value of entries is increased (usually) by 1.

After the control variable has been increased, it is checked to see if it is greater than the terminating value. If it is, the loop terminates. It is important to remember this if the control variable is used for any purpose after the loop has ended. The control variable is always greater than the terminating value.

Normally, the control variable is incremented by 1 each time the statement is executed, but this can be changed by the optional STEP statement. For example FOR columns= 0 TO 39 STEP 5 will increase the control variable by 5 each time, and FOR dots=0 TO 2*PI STEP 0.03 will increase the value by 0.03. This second line also shows that the starting

and finishing values of the control variables do not need to be constants. In fact, the starting value, terminating value, and step size can all be constants, variables, or valid expressions.

It is also possible to have a negative step size. In this case, it is essential that the starting value of the control variable is less than the terminating value, if the loop is to execute more than once.

FOR . . . NEXT loops are always executed at least once, even if the starting control variable value is greater than the terminating value (positive step size assumed). This is because the control variable value is not checked until the NEXT statement is executed.

It is perfectly permissible to use the control variable in calculations or for other purposes within the loop, and this is the main reason for having a variable step size. Using the control variable as a co-ordinate in a LOCATE statement, a character can be printed in successive positions down (or across) the screen, and this is useful for decorative features in screen displays. Using the control variable, it is simple to write a program to print out, for example, the square roots of all the whole numbers between 1 and 10.

You should be careful, however, not to alter the value of the control variable within the loop. This does not cause an error, the computer can cope with it, but it is very bad programming practice and should be avoided. At worst, you may prevent the control variable from ever reaching the terminating value, in which case the loop will continue indefinitely. Also, you should not jump out of a loop using a GOTO. This can thoroughly confuse the computer.

WHILE . . . WEND loops repeatedly execute the statements between the WHILE statement and the WEND statement as long as the condition set in the WHILE statement is true (exactly what true means will be covered in a later chapter). If the condition in the WHILE statement is not true when the statement is executed, the program execution continues at the line after the WEND statement, so a WHILE . . . WEND loop is not necessarily executed at all. This is in contradistinction both to FOR . . . NEXT loops and the REPEAT . . . UNTIL loops found in some other BASICs. It is sometimes helpful

to look on a WHILE . . . WEND loop as a loop with conditional entry, rather than conditional exit.

ARRAYS

So far, we have met simple variables and control variables, but there is a third variety. These are arrays or subscripted variables. These are groups of variables, all with the same name, but distinguished from one another by number, or subscript.

Because arrays can take up a lot of memory, the computer must be instructed to reserve space for them. This is done with the DIM (for DIMension) statement. Thus, DIM subtotals(11) will create an array with 12 elements, numbered 0 to 11. (Most BASICs count array elements from 0, but a few count from 1.) In Locomotive BASIC, it is not always necessary to dimension (or declare, as it is sometimes called) an array. If you don't, the computer will assume (or default to) a size of 10 elements.

To refer to a single element in the array you use the name and number, for example PRINT subtotals(7). The number in the brackets does not have to be a constant. It can also be a valid expression, or a variable. In fact, it is often the control variable of a FOR . . . NEXT loop. Loops and arrays go well together, and are the basis of many powerful programming techniques, including sorts.

Listing 1 is a program which uses arrays and loops together to solve a problem of the sort often found on the Brain Teaser pages of computer magazines.

LISTING 1

```
20 DIM myshare(100)

30 FOR bags=1 TO 100

40 myshare(bags)=bags-(bags*bags
/100)
```

```
50 NEXT bags
60 FOR bags=1 TO 100
70 IF myshare(bags)>bestshare TH
EN bestshare=myshare(bags):highe
st=bags
80 NEXT bags
90 PRINT highest,bestshare
```

The problem is: You are a poor peasant farmer. The local wicked baron takes as his tax a percentage of your crop equal to the number of sacks of corn you grow. That is, one sack, one per cent, ten sacks, ten per cent, and so on. How many sacks should you grow to have the maximum left for your family?

There may be a mathematical formula to work this out, but the easiest way with a computer is to work out the answer for each number of sacks, store the answers in an array, and then search the array for the highest result. Obviously, the highest number of sacks we need consider is 100. More than that would give a negative result.

Line 20 dimensions the array. Lines 30–50 are the first loop, line 40 calculating and subtracting the tax and storing the result in the array. Lines 60–80 are the second loop. Line 70 compares a new variable, bestshare, with the contents of each element of the array. If the array element is higher, bestshare is set equal to it, and highest is set equal to the control variable.

When the loop has completed, bestshare is equal to the highest element, and highest contains the position of that element in the array. Line 90 prints out the solution.

Note that in this program, the first loop completely

finishes before the second commences. It is also possible to have two loops, one inside the other, a process generally called nesting. Listing 2 is a graphic illustration of this. It prints a solid rectangle of stars on the screen.

LISTING 2

```
20 CLS

30 FOR y=2 TO 20:REM top to bottom

40 FOR x=4 TO 36:REM side to side

50 LOCATE x,y

60 PRINT"*"

70 NEXT x

80 NEXT y
```

Line 20 clears the screen. The first loop is between lines 30 and 80. The second is between lines 40 and 70. Note that the second loop is completely contained within the first. This is the rule for loops. Either one must be completely within another, or they must be completely separate, as in Listing 1. If they overlap, an error will occur.

Line 50 positions the cursor, using the control variables as co-ordinates, and line 60 prints the stars.

When you run the program, you will see that each horizontal line is completed before the next is started. The horizontal position is controlled by the X loop.

Having run the program, list it, then swap around lines 30 and 40. If you now try to run the program, you will find it stops with an error message, as the loops are not correctly nested. Now swap around lines 70 and 80. The program should run correctly again, but this time each vertical line

should be completed before the next is started.

It is possible to nest loops more than two deep, if necessary. It is also possible to nest WHILE loops within FOR loops, and vice versa. The same rule of one completely within another, or completely separate, still applies.

The arrays we have met so far have had one dimension, like a list of numbers down a page. In Locomotive BASIC it is also possible to have two-dimensional arrays, like columns of numbers down a page and also across it.

Listing 3 demonstrates the use of a two-dimensional array in conjunction with nested loops. Line 30 dimensions the array, with 6 elements (numbered 0 to 5) in each dimension. Note that this is a total of 6*6=36 elements, not 12. The first loop is between lines 40 and 80, the second between lines 50 and 70. Line 60 stores the result of multiplying the control variables together in the array. The rest of the program is just to print out the results to show it has worked. Compare this part with Listing 2.

LISTING 3

```
20 CLS

30 DIM answers(5,5)

40 FOR factor1=0 TO 5

50 FOR factor2=0 TO 5

60 answers(factor1,factor2)=factor1
*factor2

70 NEXT factor2

80 NEXT factor1
```

```
90 FOR lines=0 TO 5

100 FOR columns=0 TO 5

110 LOCATE 5*columns+2,3*lines+1

120 PRINT answers(columns,lines)

130 NEXT columns

140 NEXT lines
```

More than two dimensions are also possible. You could think of three dimensions as the columns on a page, the lines on a page, and the pages. Listing 4 illustrates this. It is an extension of Listing 3. There is an extra dimension in the array in line 30, and an extra loop, and an extra multiplication in line 70. We can only print two dimensions on the screen, so in the second part of the program, an INPUT statement is used to choose which page to show (line 110). Line 120 is a trap which stops the program if an out-of-range number is entered. Line 200 keeps the program looping round until such an input is made.

LISTING 4

```
20 CLS

30 DIM answers(5,5,25)

40 FOR factor1=0 TO 5

50 FOR factor2=0 TO 5

60 FOR factor3=0 TO 25

70 answers(factor1,factor2,factor3)
```

```
=factor1*factor2*factor3
80 NEXT factor3
90 NEXT factor2
100 NEXT factor1
110 INPUT "Which page (0 to 25)";Pa
ge%
120 IF page%<0 OR page%>25 THEN STO
P
130 CLS
140 FOR lines=0 TO 5
150 FOR columns=0 TO 5
160 LOCATE 5*columns+2,3*lines+1
170 PRINT answers(columns,lines,pag
e%)
180 NEXT columns
190 NEXT lines
200 GOTO 110
```

Multi-dimensional arrays and nested loops are very useful and important parts of programming, and it is worthwhile to take the time to make sure you understand them thoroughly.

Chapter 2

STRING VARIABLES

In the first chapter, we discussed the way the computer stores numbers. The computer also needs to store text. In computing, stored text is commonly referred to as strings. In other words, it is thought of as a line of characters strung together.

As with numeric variables, we can assign a string of characters to a variable, and print the variable. We can also assign the contents of one variable to another (and if we do this, as with numeric variables, the original remains in existence).

String variables are indicated by having a dollar sign as the last character in the variable name, for example name$, search$. To assign a string of characters the characters must be enclosed between inverted commas or apostrophes, thus:- LET name$="John" or alternatively LET name$='Robert'. Strings of characters between inverted commas like this are usually called string literals and are the string equivalent of numeric constants (and so are occasionally called string constants).

As well as simple assignment, there are a range of manipulations that can be performed on strings. The first of these is concatenation, which simple means joining together end to end. Most BASICs use the + sign for this, but it is important not to confuse this with mathematical addition. For example, LET a$="Hello":b$="Sailor":c$=a$ + b$:PRINT c$ will print HelloSailor. Note that no space is inserted between the words.

As well as printing out or manipulating whole strings, it is possible to pick out particular characters in a string. There are three string functions to do this. LEFT$ is used to slice off characters from the beginning (left-hand end) of the string. RIGHT$ slices off characters from the end of the string (right-hand side). MID$ can be used to slice characters out of the middle of a string, but can go right to either end if required. All these functions can be used to slice out single characters,

or a number of consecutive characters. The number of characters, and the string to be sliced, are given in brackets after the variable name. In the case of MID$ we must also give the position of the first character we wish to slice. Thus, if we enter a$="chopper", PRINT LEFT$(a$,4) will print chop, PRINT RIGHT$(a$,3) will print per, and PRINT MID$(a$,2,3) will print hop. In the case of MID$, the starting point is given first, followed by the number of characters we wish to slice. As well as being printed out directly like this, string slices can also be assigned to or concatenated with other strings. Either way, the original string continues intact.

Strings in Locomotive BASIC are not of fixed length. They alter in length to accommodate the number of characters currently assigned to them. The computer must keep a record of how long a string is so that it knows how many characters to print (or whatever). This length is returned by the function LEN( ). There is a maximum length for a string, of 255 characters.

Listing 5 uses most of the ideas we have covered so far. This program has its origin in a letter from a reader to a computer magazine. He wanted a program to create personalities and names for the characters in his adventure game. Personalities are difficult, but this program will give a good yield of acceptable names!

LISTING 5

```
20 REM * Name Creation *
30 source$="CREATINGNAMESFORADVENTU
REGAMESISFUNWITHANAMSTRADCOMPUTER"
40 WHILE 1
50 name$=""
60 FOR namelength=1 TO RND(1)*4+5
```

```
70 name$=name$+MID$(source$,RND(1)*
LEN(source$)+1,1)
80 NEXT namelength
90 PRINT name$
100 WEND
```

Creating words at random is difficult. Just picking out letters of the alphabet at random and assembling them into words is not very successful, as all letters have an equal chance of selection this way, whereas in real language some letters are much more frequent than others. This program overcomes this to some extent by using a phrase of English as a source, source$ in line 30.

WHILE 1 in line 40 and WEND in line 100 are a way of creating an infinite loop. Line 50 sets the variable name$ to an empty string.

Line 60 is the start of a FOR . . . NEXT loop. The function RND is used to give the names a variable length between 5 and 9 characters.

Line 70 uses MID$ to slice out characters from source$ at random, and add them to name$. The function LEN is also used. This makes it easy to use different source strings (by editing line 30) without having to count the characters.

Line 90 prints out the names. The Amstrad's Locomotive BASIC is very fast, and the names scroll up the screen too fast to read. You can stop the program by pressing ESCAPE once, and restart by pressing any other key.

Locomotive BASIC also has string arrays. These must be dimensioned just like numeric arrays. As with numeric arrays, it is possible to have more than one dimension. The length of each string element is variable, like ordinary strings, but subject to the same maximum of 255 characters.

We said in Chapter 1 that the use of loops and arrays together is the basis of sorts, and Listing 6 is an example of

this. This program falls into four parts; the initiation, lines 20 to 40; a routine to enter strings from the keyboard, lines 50 to 130; the sort proper, lines 150 to 260; and a routine to print out the sorted strings, lines 280 to end.

LISTING 6

```
20 REM * Sort Demonstration *

30 entries=-1

40 DIM store$(20)

50 PRINT "Please enter strings."

60 entry$="*"

70 WHILE entry$<>""

80 INPUT entry$

90 IF entry$="" THEN 130

100 LET entries=entries+1

110 store$(entries)=entry$

120 IF entries=20 THEN PRINT "Array
 now full"

130 WEND

140 PRINT "Sorting..."

150 FOR strings=0 TO entries-1
```

```
160 compare$=store$(strings)
170 position=strings
180 FOR rest=strings+1 TO entries
190 IF compare$>store$(rest) THEN c
ompare$=store$(rest):position=rest
200 NEXT rest
210 IF position=strings THEN 260
220 FOR moves=position-1 TO strings
 STEP-1
230 store$(moves+1)=store$(moves)
240 NEXT moves
250 store$(strings)=compare$
260 NEXT strings
270 PRINT "...finished."
280 FOR strings=0 TO entries
290 PRINT store$(strings)
300 NEXT strings
```

Line 40 dimensions the array (one dimension) to take 21 strings. The variable entries is used to count the strings, and this is set initially at −1.

A WHILE . . . WEND loop is used to enter the strings, the actual input being at line 80. Line 90, in conjunction with the WHILE condition, allows the input routine to be ended without entering all 21 possible strings by pressing enter without typing anything. After a string is entered, entries is incremented by 1, and the string stored in the array at this position. This is why entries was started at −1, so the first entry goes into the zero element. Line 120 gives a warning if the array is filled.

The sort method is as follows. Using a FOR loop (line 180), the contents of the first element in the array (selected by the control variable strings in line 160) is copied into the variable compare$, and the variable position is set to zero (line 170). The variable compare$ is then compared to all the (used) elements in the array in turn. If the contents of an element comes before the current contents of compare$ in alphabetical order, this element is copied into compare$ and "position" is set to the position of this element (line 190). At the end of the first pass, the first string in order is thus in compare$, and its position in the array "position". If it is already in the top position in the array, nothing needs to be done, and line 210 checks for this.

If it is not at the top, the next part of the program puts it there. Lines 220 to 240 move all the elements in the array, from one above the element to be moved to the top, to the top element in the array, down by one position. The contents of compare$ are then stored in the zero element of the array (line 250).

On the second pass of the FOR strings loop, the second element is compared with all those below it, and if necessary, the second in order is moved to second position. The process repeats until the last-but-one element is compared with the last.

The strings are then in order and are printed out by lines 280 to 300.

This sorting method is considerably faster than the more

common bubble sort. It also has the advantage that the order of the elements is not disturbed unnecessarily, so it is suitable for multi-field sorts. This means, for instance, if you have data consisting of items and dates in two fields, if you sort the data into alphabetical order and then into date order, items of the same date will remain in alphabetical order.

You may wonder how a computer stores letters in memory. The answer is that each character has a code number, and it is these numbers which are stored. There are two functions which will give the code number of any character, and any character from its code number. The first of these is called ASC( ). This comes from American Standard Code for Information Interchange (ASCII), as the internal code used by the Amstrad and most other computers is based on this. Thus, PRINT ASC("A") will return 65, the code number for a capital A. This function can be used with string variables as well as string literals. If the string contains more than one character, the code returned will be that of the first character in the string. It is, of course, possible to slice out characters from the middle of a string, for example PRINT ASC(RIGHT$ (a$,3)). If the string slice specifies more than one character, again the code returned will be that of the first character of the slice.

Listing 7 is a program that will print out the ASCII code of any key pressed. Some keys, such as CTRL and SHIFT,

LISTING 7

```
20 REM ASCII codes
30 WHILE 1
40 a$=INKEY$: IF a$="" THEN 40
50 PRINT ASC(a$)
60 WEND
```

do not generate codes themselves, but modify the codes returned by other keys pressed simultaneously.

The function CHR$( ) returns, as a string, the single character whose ASCII code is given in the brackets. Thus PRINT CHR$(65) will print a letter A on the screen. This function is useful for printing the graphics characters which are not directly obtainable from the keyboard, and for printing special control characters on the screen, such as CHR$(10), which is a linefeed, or CHR$(12), which is equivalent to CLS.

Listing 8 will print out the character for any code number you enter. The highest valid code number is 255. Codes below 32 are the control codes, and some of these have surprising effects.

LISTING 8

```
20 REM Characters

30 WHILE 1

40 INPUT "Code number";code

50 PRINT CHR$(code)

60 WEND
```

Two other string functions which are almost unique to the Amstrad are UPPER$ and LOWER$. These have the effect of converting all the alphabetical characters in a string into upper and lower case respectively. This can be used in string sort and string search routines to make them non-case-dependent. Examples of the use of these will be found in later chapters.

Chapter 3

DECISIONS

On the face of it the IF . . . THEN statement is very straightforward to use. It takes the form IF condition THEN statement. If the condition is true, the statement is executed, if not, program execution moves on to the next line. However, IF . . . THEN has one or two tricks up its sleeve which can be useful in programming, but which are also pitfalls for the unwary.

It is quite legal to put further statements on the same line as an IF . . . THEN. However, it must be remembered that if the condition in the IF . . . THEN is false, program execution moves on to the next line. This means that all the statements on the line will not be executed, not just the one immediately following THEN. This can be very useful when you want several things to happen when, and only when, the condition is true.

ELSE is an optional extension to the IF . . . THEN statement. When ELSE is used, if the condition after IF is true, all the statements between THEN and ELSE are executed. If the condition after IF is not true, then all the statements after ELSE on the line are executed. Frankly, ELSE can cause more problems than it solves, and I rarely find use for it.

Putting more than one statement on a line saves some memory space compared to giving each statement a line of its own. If a program uses up all memory when run, it sometimes helps to go through it, making two (or more) lines into one. If you ever have to do this, remember not to put extra statements after an IF . . . THEN!

The IF . . . THEN statement acts according to whether a condition is true or false. It is interesting to discover exactly how this is determined. Try entering the following in direct mode:-

```
A=6
PRINT A=6
```

```
PRINT 6=6
PRINT A=5
PRINT 6=5
PRINT A=B
```

On the face of it, you would expect all these lines to produce error messages. In fact, the first two should produce −1, and the last three should produce 0. This is because the computer evaluates these as expressions, to see if they are true or false, and true and false are given numerical values. −1 is used to represent true, and 0 to represent false.

Obviously, A does equal 6, as we have just stored 6 in it. Equally obviously, 6 equals 6, A doesn't equal 5, and 6 does not equal 5. In the last example, A doesn't equal B, because 6 is stored in A, and the computer has not been told to assign a value to B, so it is regarded as holding 0.

These values for true and false hold equally true for comparisons of strings, and also for comparisons of greater than, less than, not equal to, etc.

One advantage of this is that it is possible to test a single variable in an IF . . . THEN statement. Try the following:-

```
IF A THEN PRINT "TRUE"
IF B THEN PRINT "TRUE"
IF NOT A THEN PRINT "FALSE"
```

The first will print TRUE because, in fact, any non-zero value can be regarded as true. The second example will not print because B contains 0, which is false.

The third example is more complicated. NOT reverses the result of a test, so FALSE will be printed if A is false. In this case it is printed. This is because when NOT is used, only −1 counts as true. Any other value is regarded as false. Some other computers, such as Atari and Sinclair models, use +1 rather than −1 to represent true, and with these, NOT still accepts any non-zero value as true, so false would not be printed with these. This is a point to beware of if trying to translate programs written for these computers to run on the Amstrad.

This single variable testing is not just an academic trick. It saves a little memory space, and, more important in many cases, it saves time.

AND & OR

In Locomotive BASIC, AND and OR can be used as logical operators, enabling two or more conditions to be tested in one statement.

AND is used much as it is in the English language. If all the conditions specified after IF, and joined by AND, are true, then the statements after THEN will be executed. If any of the conditions are false, then the statements after THEN will not be executed.

With OR, if any of the conditions after IF, and joined by OR, are true, then the statements after THEN will be executed. If all the conditions joined by OR are true, the statements after THEN will be executed. This is slightly different to the way in which "or" is used in English.

Using AND and OR together is not always straightforward. Consider this example. We have a program with three variables; A, B, C. We want certain things to happen if A=6 and B=3, or if C=7 and B=3. We cannot write the program line thus:-

```
IF A=6 AND B=3 OR C=7 AND B=3 THEN . . .
```

The computer will not associate the two conditions on the right of the OR together, and the two conditions on the left of OR together, as we do in English. If C=7, the statements after THEN will always be executed, and they will always be executed if A=6 and B=3, which is not quite what is required.

One way round this is to use brackets, thus:-

```
IF (A=6 AND B=3) OR (C=7 AND B=3) THEN . . .
```

The conditions within each set of brackets will be evaluated first, to either true or false, and then these results will be

evaluated, and if either is true, the statements after THEN will be executed. This particular conditional could also be written as:-

```
IF B=3 AND(A=6 OR C=7) THEN . . .
```

This would be the preferred form, as it is shorter, and would also execute faster.

If you use NOT in a complex conditional, remember it only affects the one expression immediately following it. It does not operate globally on the entire expression. Thus

```
IF NOT A=6 AND B=3 THEN . . .
```

will execute the statements after THEN if B=3 and A=any value other than 6.

ON

ON is an alternative form of decision making. It is used when it is required to make one choice from a list of possibilities. This is done by branching, as ON can only be followed by one of two statements, GOSUB or GOTO.

The ON statement takes the form ON variable GOSUB (or GOTO) linenumber,linenumber,linenumber . . .

The value of the variable determines which linenumber the program branches to. If it is 1, the program branches to the first in the list, if it is 2, the second, and so on. If the value is 0, or if it is higher than the number of linenumbers in the list, the program will go on to the next statement.

ON . . . GOSUB is most commonly used in conjunction with menus in programs such as data bases. A list of options is printed on the screen, like this:-

1. Enter Data.
2. Load file.
3. Edit file.
4. Search file.

5. Sort file.
6. Save file.
Please enter choice.

The user presses an appropriate number key, the value of which is placed in a variable. This is then used in the ON . . . GOSUB statement to call the subroutine to perform the required function. The ON . . . GOSUB statement would normally be followed by a GOTO statement to send the program back to the start of the menu, so that the menu reappears when the operation is completed, and also if a number key higher than the number of options is pressed.

Though this is the most common use, it is by no means the only one. ON is an under-used facility. Whenever a choice has to be made from several options, the use of ON should always be considered, as it can be considerably faster than a long list of IF . . . THENs to make the choice.

The keyword ON is also used for other purposes in Locomotive BASIC, for example ON ERROR, ON BREAK. These are not really associated with the use of ON in decision making, and are dealt with elsewhere.

Listing 9 is a guess-the-number game which uses a large number of conditional statements. A random number is generated, and you get a clue as to the size. You then have to enter guesses, and the computer tells you how you are doing. No line-by-line description of this program is included, as it is quite straightforward, and REMs are included where necessary.

Try to follow the flow of the program on the listing as you are using it. This is a good way to learn about how programs work.

LISTING 9

```
20 REM * Guess The Number *

30 MODE 1:tries=1:oldiff=5000

40 CLS
```

```
50 LOCATE 5,5:PRINT "I am thinking
of a number..."
60 LOCATE 5,7:PRINT "...it is betwe
en 1 and 5000."
70 REM this loop is to produce a tr
uly random number
80 LOCATE 5,22:PRINT "PRESS A KEY T
O PLAY"
90 WHILE INKEY$=""
100 number=INT(RND(1)*5000)
110 WEND
120 LOCATE 5,22:PRINT SPACE$(19)
130 LOCATE 5,10:PRINT "Here is a cl
ue..."
140 LOCATE 5,15:PRINT "It could be.
.."
150 LOCATE 5,17
160 IF number<100 THEN PRINT "someo
```

```
ne's age."
170 IF number>99 AND number<400 THE
N PRINT "a cricket score."
180 IF number>399 AND number<1000 T
HEN PRINT "the pages in a book."
190 IF number>999 AND number<3000 T
HEN PRINT "the size of a car engine
."
200 IF number>2999 THEN PRINT "some
one's telephone number."
210 LOCATE 5,19:PRINT "Try no. ";tr
ies
220 LOCATE 18,19:INPUT "Your guess
";guess
230 guess=INT(guess)
240 IF guess<1 OR guess>5000 THEN L
OCATE 5,19:PRINT "SILLY!!!":GOTO 22
0
```

```
250 diff=ABS(number-guess)
260 REM skip to end if guess correc
t
270 IF diff=0 THEN 410
280 CLS
290 REM determine size of error and
 print clue
300 LOCATE 5,5
310 IF number<guess THEN PRINT "Too
 big";
320 IF number>guess THEN PRINT "Too
 small";
330 IF diff<10 THEN PRINT " but you
 are very close.";
340 IF diff>9 AND diff<50 THEN PRIN
T " but you are quite close.";
350 IF diff>49 AND diff<200 THEN PR
INT "."
```

```
360 IF diff>199 THEN PRINT " by a l
ong chalk!"
370 IF diff>oldiff THEN LOCATE 5,10
:PRINT "You are further out than la
st time!"
380 tries=tries+1:oldiff=diff
390 LOCATE 5,15:PRINT "Last guess "
;guess
400 GOTO 210
410 CLS
420 x=5
430 FOR y=5 TO 20
440 LOCATE x,y:PRINT "CORRECT!!!"
450 x=x+1
460 NEXT y
470 LOCATE 5,22:PRINT "Play Again (
y/n)?"
480 ans$=INKEY$:IF ans$="" THEN 480
```

```
490 IF LOWER$(ans$)="y" THEN RUN
500 END
```

Chapter 4

INPUT, PRINT & DATA

Almost every practical program will need some means of communicating with the user whilst it is running. The INPUT statement is provided to allow program execution to be temporarily halted so that required data may be typed in.

When the computer stops for input, it signals that it is waiting by printing a question mark on the screen. As the input can be of any length, the computer needs some way of knowing when the user has finished typing in the data. Data entry is assumed to have finished when the RETURN key is pressed, and program execution recommences.

INPUT can be used to obtain both string and numeric data. When entered, the data is placed in a variable, and the type of variable specified in the INPUT statement determines whether the input is to be regarded as string or numeric. "Regarded" because, of course, a string can quite legally contain numbers. On the other hand, if non-numeric characters are entered when a numeric input is required, an error occurs, and the message "? Redo from start" is printed.

It is possible to enter data into several variables from a single INPUT statement, by listing the variables after the INPUT, separated by commas, thus:-

```
100 INPUT NUM1,NUM2,NUM3
250 INPUT NAME$,ADDR$
```

When typing in the various items they must be separated by commas, RETURN being pressed when entry is complete. This applies both to string and numeric input. This means that commas cannot be included in strings entered in response to an INPUT statement. To solve this problem, there is a LINE INPUT statement. This will take any characters typed, including commas and inverted commas, but can only be used for a single variable at a time.

It is perfectly legal to mix string and numeric inputs in a

single statement. Whether such mixing is advisable is another matter. Even the use of multiple INPUTs of one type can cause problems, and should be used with discretion.

The problem with the INPUT statement is that the simple question mark does not give any indication of what sort of input is required. To ease this problem, Locomotive BASIC has provision to include a line of text in the INPUT statement, which is printed on the screen along with the question mark, as an input prompt. When this is used, using a semi-colon between the prompt and the name of the variable will cause the question mark to be printed. Using a commas will cause it to be suppressed.

User-Friendly is a term that is much in use. It means making a program easy and pleasant to use. Input prompts are a vital part of this. They should clearly (and politely) indicate what the user is intended to type in. NAME? appearing on the screen would certainly indicate what is required, but is not very polite. "Please enter name" is better if the question mark is suppressed, otherwise "What is your name, please?" would be the best prompt to use.

One should avoid being obsequeous. If I came across an input prompt "Would you mind entering your name, please?" I personally could not resist entering NO!

It is very difficult to phrase input prompts to deal with multiple input statements, except at the very basic "enter three numbers" level. On the whole, when writing programs which will be used by others (and especially by non-programmers) it is better to have a separate INPUT statement for each item, with a clear and simple prompt for each.

PRINT

As the general-purpose statement to display all text and numeric output, and some graphics on the screen, PRINT is necessarily a very flexible and versatile statement. PRINT is followed by a list of items, which may be string or numeric, variables or constants, or any permutation thereof, and may even contain calculations, string manipulations and logical

comparisons.

Numeric array elements and substrings may be directly included in PRINT statements.

Various punctuation marks are used in the print list, and these control aspects of how the items are to appear on the screen.

The BASIC computer language dates back to the mid-60's, at which time the usual interface to a computer was via a printer and keyboard, rather than a VDU. This explains why the word PRINT is used, and some aspects of how the PRINT statement works.

Whenever a PRINT statement is completed, the computer normally moves on to a new line. If each item to be printed is given a separate statement (not necessarily in a separate program line) each will be printed on a separate line.

If more than one item is included in a PRINT statement, the punctuation mark used to separate them determines how they will be printed. If they are separated by commas, the computer will separate them by a number of spaces. If they are separated by semi-colons, they will be printed right up against each other without any spaces.

The other important punctuation marks in PRINT statements are inverted commas. These are used to enclose string constants. The inverted commas are not printed. This means that it is unfortunately not possible to include inverted commas within a string constant to be printed.

Numeric constants in a PRINT statement do not need to be thus enclosed. The statement PRINT 18 will do exactly that. PRINT HELLO on the other hand will print 0, as HELLO will be interpreted as a variable name. PRINT"HELLO" will print HELLO.

This is a typical PRINT statement:-

```
1000 PRINT "You have scored ";SCORE;" points
from ";TRIES;" attempts."
```

The first item in the print list is a string constant. Note the extra space at the end. The semi-colon means that the numeric variable SCORE will be printed immediately after

it, directly followed by another string constant, another numeric variable, then another string constant. When this statement is executed, something like this will appear on the screen:-

You have scored 18 points from 7 attempts.

This ability to join several items together so that they appear as one on the screen is very valuable in producing readily intelligible output from programs.

The effect of the comma in PRINT statements is to produce a tabular display. The screen is regarded as being divided into fields 13 columns wide (by default). When a comma (not enclosed between inverted commas in a string literal) is encountered in a PRINT list, the print position moves to the start of the next field, moving on to a new line if necessary.

DATA, READ & RESTORE

As well as variable data that may be typed in while a program is running, most programs require constant data. Such data can be stored within a program using the DATA statement, and placed in variables as required using the READ statement.

The DATA statement is followed by the list of constant data, with the items separated by commas. The list can be spread over several program lines if necessary (each beginning with DATA) and these do not necessarily have to be grouped together in the program. The computer, however, always treats all the DATA statements in a program as if they were one long list.

DATA statements may contain both string and numeric constants. String constants do not need to be enclosed in inverted commas. This is, however, necessary if the string is to include a comma.

The READ statement is used to place items from the data list into variables. The READ statement starts with the first item in the list and works through it sequentially. The statement READ A$ will place the first item in the data list into

the string variable A$.

The statement READ A will take the next item in the data list and place it in the numeric variable A, provided it consists of numbers (or other valid numeric characters, such as the decimal point). The READ statement can place any characters into a string variable, but only valid numeric characters into a numeric variable.

Items in DATA statements can be read directly into array elements.

If you try to READ more items of data than there are in the list, an error will result.

To allow data to be used more than once in the running of a program, the RESTORE statement is provided. This resets the data pointer which keeps track of the current position in the data list back to the beginning.

In Locomotive BASIC, the RESTORE statement has an extra feature. It can be used to set the data pointer to a particular program line, so that the READ statement will start from that point. This allows you to jump about in the data list, and is a very valuable and useful feature.

DATA statements are very useful for such things as holding the data to be used by SOUND statements, to play tunes in games. A FOR . . . NEXT loop can be used, looping around the right number of times for the number of notes, with a READ statement and a SOUND statement within it. RESTORE with a line number can be used before the FOR statement to ensure the data is read from the right place.

METRIC CONVERTER PROGRAM

This program, Listing 10, illustrates many of the points made in this Chapter, and also introduces an important programming concept, the subroutine.

This program will perform imperial/metric and metric/imperial conversions for most common units of length, weight, area and volume. This is understandably a somewhat complex program.

This program has to perform a number of functions:-

1. Obtain user-input of the units to be converted.
2. Check whether the units entered are covered by the program.
3. If they are not, print a message and allow further input.
4. If they are, obtain user-input of the quantity to be converted.
5. Calculate and print the result.

Each of these functions is practically a program in its own right, and this is the idea of subroutines. Each separate operation is written as if it were a separate program. These are then called by the GOSUB statement as required. Each subroutine must include a RETURN statement, and when this is executed, the program returns to the statement immediately after the GOSUB.

It is quite possible for a subroutine to be called from within another subroutine, a process known as nesting (as with loops). There is a limit to the depth of this nesting, however, as the computer has to remember the return address for each subroutine, and the memory space for this is limited.

In addition to the subroutines to perform the functions listed above, there are a couple of others. One prints an introduction to the program on the screen, the other prints a list of the units covered by the program.

Line 20 is the program title. Line 30 makes sure the computer is in the right mode, and also clears the screen. Line 40 dimensions the arrays. Line 50 calls the subroutine to print the introduction on the screen. This subroutine consists of lines 1000 to 1080.

Line 1010 prints the program title. Line 1020 has two naked PRINTs in it. This is the simplest way of obtaining two line spaces. Lines 1030 to 1060 are a FOR/NEXT loop. This READs the lines to be printed from DATA statements, and prints them. The DATA statements are in lines 100 to 150. After line 1080, program execution returns to line 60.

The subroutine in lines 2000 to 2060 READs the units

and conversion factors into the two arrays. This is more convenient than using them direct from the DATA statements, though the program could be written in this way.

The subroutine between lines 3000 and 3090 is the input routine for units. As rather a long input prompt is required, it is in two PRINT statements, rather than in the INPUT. This is in part to prevent words being split at the end of screen lines.

If HELP is entered here, the subroutine from line 4000 is called. This is the one to print the list of units from the array.

Typing QUIT causes the program to stop. Note the use of UPPER$ here, so that either upper or lower case input will work.

If the input in line 3030 is not HELP or QUIT, the subroutine from line 5000 is called to search through the conversion data in the array. Line 5010 sets the variable foundflag to 0. The search is done by a FOR . . . NEXT loop in lines 5020–5050. Lines 5030 and 5040 do the actual checking, and if a match is found, set the variable confactor to the appropriate conversion factor, and set foundflag to −1.

When the program returns to line 3070, if a match was not found, foundflag will be set to 0. In this case, the subroutine from line 8000 is called to print a message, and the program flow is diverted back to line 3010 for further input.

Line 3080 is only executed when a match is found. After the quantity is input at line 6050, the subroutine from line 7000 is called. Note how the actual conversion calculation is performed in a PRINT statement in line 7020. This statement is complex, consisting of a numeric variable, a string constant (a single space), a string variable, a string constant (= sign), the calculation, a string constant (another space), and finally a string variable, all separated by semi-colons so they print as one line.

After the conversion, the program goes back to line 6030, so further conversions can be performed without having to re-enter the units. You should always try to make your pro-

grams work in the way which will be most useful.

If 0 is entered at line 6050, the RETURN statement in line 6060 is executed. This takes the program back to line 80, which clears the screen, then goes back to line 70, so that the units can be changed if required.

The END statement in line 90 is really there only as a marker when reading the listing. It is never executed. You should always put an END statement between the main program and the subroutines, whether it is executed or not. If the computer comes across a RETURN when it has not executed a GOSUB, an error occurs.

The placing of DATA statements in a program is arbitrary. Between the main program and the subroutines is as good a place as any. However, when DATA statements are only used by one subroutine, it is sometimes a good idea to include them within the subroutine.

The use of subroutines is vital to good programming, and it is worth taking the time to understand them properly. Again, try to follow the flow of this program on the listing as you are using it.

LISTING 10

```
20 REM metric converter Program
30 MODE 1
40 DIM factors(9,1),unit$(9,1)
50 GOSUB 1000:REM prints instructio
ns
60 GOSUB 2000:REM fills arrays
70 GOSUB 3000:REM input
80 CLS:GOTO 70
```

```
90 END
100 DATA This Program Performs metr
ic/imPerial
110 DATA and imPerial/metric conver
sions.
120 DATA Type 'HELP' as a units ent
ry to see
130 DATA the list of units covered.
140 DATA You must type the units ex
actly as
150 DATA they aPPear in the list.
160 DATA INCHES,2.54,CENTIMETRES,.3
937
170 DATA FEET,.3048,METRES,3.281
180 DATA MILES,1.609,KILOMETRES,.62
14
190 DATA SQ INCHES,6.452,SQ CENTIME
TRES,.1550
```

```
200 DATA SQ FEET,.0929,SQ METRES,10
.76
210 DATA SQ MILES,2.59,SQ KILOMETRE
S,0.3861
220 DATA CUB INCHES,16.39,CUB CENTI
METRES,.06102
230 DATA CUB FEET,.02832,CUB METRES
,35.31
240 DATA GALLONS,4.546,LITRES,.22
250 DATA OUNCES,28.35,GRAMS,.03527
1000 REM instructions
1010 LOCATE 12,5:PRINT "METRIC CONV
ERTER"
1020 PRINT:PRINT
1030 FOR lines=1 TO 6
1040 READ text$
1050 PRINT text$
1060 NEXT lines
```

```
1070 PRINT
1080 RETURN
2000 REM array fill routine
2010 FOR conversions=0 TO 9
2020 FOR systems=0 TO 1
2030 READ unit$(conversions,systems
),factors(conversions,systems)
2040 NEXT systems
2050 NEXT conversions
2060 RETURN
3000 REM units input routine
3010 PRINT "Please enter units to b
e converted"
3020 PRINT "or HELP to see list, or
 QUIT."
3030 INPUT unit$
3040 IF UPPER$(unit$)="QUIT" THEN C
LS:STOP
```

```
3050 IF UPPER$(unit$)="HELP" THEN G
OSUB 4000:GOTO 3010
3060 GOSUB 5000
3070 IF NOT foundflag THEN GOSUB 80
00:GOTO 3010
3080 IF foundflag THEN GOSUB 6000
3090 RETURN
4000 REM help routine
4010 CLS
4020 LOCATE 12,1:PRINT "UNITS COVER
ED"
4030 FOR lines=0 TO 9
4040 LOCATE 1,lines*2+3:PRINT unit$
(lines,0)
4050 LOCATE 20,lines*2+3:PRINT unit
$(lines,1)
4060 NEXT lines
4070 LOCATE 5,24:PRINT "PRESS ANY K
```

```
EY TO CONTINUE"
4080 ans$=INKEY$:IF ans$="" THEN 40
80
4090 CLS
4100 RETURN
5000 REM search routine
5010 foundflag=0
5020 FOR possibles=0 TO 9
5030 IF UPPER$(unit$)=unit$(possibl
es,0) THEN confactor=factors(possib
les,0):conv$=unit$(possibles,1):fou
ndflag=-1
5040 IF UPPER$(unit$)=unit$(possibl
es,1) THEN confactor=factors(possib
les,1):conv$=unit$(possibles,0):fou
ndflag=-1
5050 NEXT possibles
5060 RETURN
```

```
6000 REM quantity input
6010 PRINT
6020 PRINT "Convert ";UPPER$(unit$)
;" to ";conv$
6025 PRINT
6030 PRINT "Please enter quantity t
o be converted"
6040 PRINT "or 0 to change units."
6050 INPUT qty
6060 IF qty=0 THEN RETURN
6070 GOSUB 7000
6080 GOTO 6030
7000 REM calculate/Print
7010 PRINT
7020 PRINT qty;" ";UPPER$(unit$);"
= ";qty*confactor;" ";conv$
7030 PRINT
7040 RETURN
```

```
8000 REM not found message
8010 PRINT
8020 PRINT "The units you have ente
red are not in"
8030 PRINT "the list.  Please enter
 HELP to see the"
8040 PRINT "full list."
8050 PRINT
8060 RETURN
```

Chapter 5

THE SOUND GENERATOR

The CPC464 has quite an advanced sound generator which can simultaneously produce up to three tones plus noise. This is in fact not exceptional by any means, as most of the better home computers have a sound generator with a similar specification. What makes the CPC464's sound generator better than most is the advanced BASIC which fully supports it, and makes it relatively easy to get the most from the sound generator. The word relatively is an important qualification in this statement, since most users tend to find the sound section of a computer one of the more difficult aspects to master. This often leads to users settling for just a few simple beeps and never really fully utilising the sound capability of the machine. This is especially the case with a machine, such as the CPC464, which has an advanced sound generator with what can at first seem like a bewildering array of facilities.

The secret of success with any sound generator is to run a few simple demonstration programs which show just what each instruction, and part of the instruction, actually does. A number of programs of this type, together with descriptions of their action, are included in this Chapter. It is also a good idea to experiment with the sound generator as this is really the only way to fully master it. Even if you fully understand every aspect of all the instructions associated with the sound section of a computer, it is difficult (probably impossible) to correctly relate a set of parameters to the sound they produce unless you have some practical experience with the machine. Even once some experience has been gained it will often be necessary to modify your original program to fine tune it to give exactly the desired effect.

The CPC464's sound is reproduced through an internal loudspeaker in the computer (not the monitor), and a volume control is provided on the side of the machine. A

respectable volume level is available from the internal loudspeaker, and this should suffice for normal requirements. There is a socket at the rear of the machine (the I/O socket) which enables the audio output to be connected to a hi-fi or other amplifier in order to give greater volume. Connection is via a stereo 3.5mm plug, and the output is a stereo output of sorts (one channel drives the left hand output, another drives the right hand output, and the third is mixed into both to give a centre channel). The output level is quite low, and might not drive some amplifiers properly. Although the output socket matches the type of plug fitted to headphones of the type for use with personal stereo units, unfortunately the output is totally inadequate to drive this type of headphones properly.

SOUND

As one might expect, the main instruction for use with the sound generator is SOUND. This can be used with other instructions, but a wide range of effects can be achieved using only the SOUND instruction, and it is advisable to master SOUND before trying out any of the associated commands.

SOUND is followed by seven parameters, but these are not all required in most cases. The first figure selects the channel or channels to which the instruction is directed. This operates in the following manner:-

1 selects Channel A
2 selects Channel B
4 selects Channel C

Thus, if we want the instruction to generate sound from channel B we would use 2 for the first parameter, or if we wanted the sound from all three channels 7 would be used as the number (1 + 2 + 4 = 7). There are other facilities which this parameter can control, but this description is adequate for the time being.

The next number controls the pitch of the sound. In

common with most computer sound generators, this operates in the opposite way to what one would probably expect, with high figures giving low pitches, and vice versa. This is an inevitable consequence of the way in which computer sound generators operate (unless some extra software is used to alter things, which rarely seems to be the case). The sound generator chip takes the computer's 4MHz clock and divides this by 16 to give a frequency of 125kHz. This is then divided by the number used as the pitch value. Obviously high numbers give a higher division rate, and therefore a low output frequency. Pitch values of 4 or less give an output frequency that is beyond the upper limit of human hearing, and only those with good hearing will be able to hear the output with pitch values of around 5 to 8. At the other end of the range the maximum usable pitch value of 4095 gives an output frequency of only about 30Hz which is a very deep bass frequency. The computer's small internal loudspeaker is unable to reproduce very low audio frequencies properly, and this results in an output that gives a sort of buzzing sound.

The third parameter sets the duration of the sound in one hundredths of a second. This gives a good degree of control over the duration, enabling a variety of sound effects as well as music to be easily generated. The maximum duration figure is 32767, which should be more than adequate in practice since it represents a duration of over 5 minutes.

The volume is controlled by the fourth parameter, and when using just a sound instruction (with no ENV instruction) this figure is from 0, which switches off the relevant sound generator channel or channels, to 7 which gives maximum volume. We will discuss the ENV command later, but when this is used in conjunction with a SOUND instruction the volume range is from 0 to 15.

The other three parameters of a SOUND command can be ignored for the moment, and simply set at 0. Thus the command: SOUND 1,250,100,6,0,0,0 would give from channel A a pitch of 250 for one second at a high volume level of six. A slightly richer sound can be produced by using two or three channels simultaneously, but the differ-

ence between one channel and all three producing the same pitch is not very marked, as this short program demonstrates.

```
10 SOUND 1,200,100,6,0,0,0
20 SOUND 7,200,100,6,0,0,0
```

This simply produces from channel A a one second output at a pitch value of 200, followed by the same thing from all three channels. The most noticeable difference between one channel and three is the higher volume of the latter.

The next listing starts with a pitch value of 1 and then steps this upwards to the maximum allowable value of 4095. In other words it takes the sound generator through its entire pitch range.

```
10 FOR P = 1 TO 4095
20 SOUND 1,P,1,7,0,0,0
30 NEXT P
```

If you run this program you will notice that there is an initial rapid fall in pitch, followed by a pitch reduction that is barely detectable towards the end. This is a result of the system of sound generation used, and a change from a pitch (division rate) of 5 to 6 obviously represents a much greater percentage change than a change from (say) 4000 to 4001. Nevertheless, a useful range of higher pitches is still available.

Simple sound effects can be obtained by sweeping the pitch value. For instance, the next program produces a falling pitch lazer zap sound.

```
10 FOR P = 20 TO 80
20 SOUND 1,P,1,7,0,0,0
30 NEXT P
```

This simply sweeps the pitch from 20 to 80 with each pitch being reproduced for the minimum period of one hundredth of a second so that a suitably short sound overall is obtained. An improved zap can be obtained by having a high initial volume level and then reducing this as the pitch falls. This

varying of the volume is called envelope shaping. A falling pitch – falling volume zap sound is produced by the program given below.

```
10 V = 7
20 FOR P = 20 TO 80
30 SOUND 1,P,1,V,0,0,0
40 V = V – 0.1
50 NEXT P
```

This program is much the same as the original, but line 10 starts the volume at 7 while line 40 decrements it by 0.1 on each loop of the program. One might expect this to crash the computer as V (the volume parameter) will not always be an integer. In practice this seems to be alright, and the sound generator effectively converts V to an integer so that there is no need to include a line in the program to achieve this.

MUSIC

It is quite easy to program music, and the computer is capable of producing music in three part harmony. However, unless you have a fair degree of musical expertise it is probably better to settle for simple single line melodies. Also, if you wish to do anything more than add a short single or multipart piece of music to a program it would be advisable to use a composer program rather than program from BASIC.

The CPC464 covers an impressive range of notes, and it actually has a compass of 8 octaves. However, as explained earlier, the number of available pitches at the high frequency end of the range is relatively limited, and low frequency sounds can not be reproduced properly by the internal loudspeaker. This results in some pitches in the top two octaves not being as accurate as those in the lower octaves, and gives a rather thin and unmusical sound from the lowest octave. It is therefore advisable to mainly restrict oneself to the other five octaves when programming music. The computer can not be programmed directly in musical values, and the table in

appendix VII of the CPC464 manual should be used to find the appropriate pitch values for notes.

The program that follows can be used when programming single line melodies.

```
10 P = 1
20 WHILE P( )0
30 READ P,D,V
40 SOUND 1,P,D,V,0,0,0
50 WEND
60 DATA 239,50,5,213,50,5,190,50,5,179,50,5,159,50,
   5,142,50,5,127,50,5,119,100,7,0,0,0
```

A WEND . . . WHILE loop is used to loop the program until the melody has been completed. The required pitch, duration, and volume figures are read from the DATA statement at line 60, in that order. The parameters must therefore be placed in line 60 in sets of three, with the three parameters in the order given above. The program is terminated by using a pitch value of zero, but note that dummy duration and volume figures must be included or the program will crash at line 30 and give an out of data error message.

The sample values in line 60 give an ascending scale of C major, with duration and volume figures of 50 and 5 used for all notes except the final one, which has figures of 100 and 7 respectively. However, by using suitable values any desired tune could be played. With long list of notes it would be advisable to use several data statements as this makes checking and alterations much easier. It may not seem necessary to include a volume setting facility in the program, but apart from giving increased control over the music, this also enables short breaks to be included between notes. This is essential where two or more notes having the same pitch value follow in succession. Without a very brief pause between these notes they would in fact simply merge into one long note, making nonsense of the music.

RENDEZVOUS

So far we have only considered the first parameter in the sound instruction in its channel selecting mode, but it does in fact control other things as well. One of these is the ability of one channel to rendezvous with another channel or channels. In other words, one or two channels wait until a SOUND instruction for the third channel is received, whereupon the SOUND instructions for the two or three rendezvous channels start simultaneously. The following numbers are used to give the specified rendezvous:-

8 Rendezvous with channel A
16 Rendezvous with channel B
32 Rendezvous with channel C

This short program helps to demonstrate how the rendezvous system operates.

```
10 SOUND 12,100,50,5,0,0,0
20 FOR D=1 TO 1000: NEXT D
30 SOUND 33,200,50,5,0,0,0
```

Line 10 is a SOUND instruction for channel C that must rendezvous with channel A. The first parameter (the channel status parameter as Amstrad call it) has a value of 12, which is 4 to select channel C, plus 8 to give a rendezvous with channel A. This instruction is not carried out until a suitable SOUND instruction for channel A is encountered. Line 20 simply provides a short delay, while line 30 is a SOUND instruction for channel A. Note that a channel status value of 33 rather than 1 has been used. This is 1 to select channel A, and 32 to give the rendezvous with channel C. The rendezvous will only occur if both the channels concerned have suitable channel status figures. If you run this program there should be a short delay (about one second) followed by both SOUND instructions being carried out simultaneously.

The next program demonstrates a practical use for the rendezvous facility.

```
10 SOUND 1,120,50,5,0,0,0
20 SOUND 2,50,100,5,0,0,0
30 SOUND 17,100,50,5,0,0,0
40 SOUND 10,150,50,5,0,0,0
```

The first two lines produce tones from channels A and B, but the durations are different with the channel A tone lasting half a second while the channel B tone lasts one second. The next two instructions are again SOUND instructions for channels A and B, but they have channel status numbers that cause them to rendezvous. In this case it means that channel A will not produce its second note until channel B does so. This effect should be clearly audible if you enter and run the program.

The practical importance of this is that when writing two or three part harmonies it is not necessary for each part of the music to have the same rhythm, and rests can be easily programmed. The rendezvous system also provides an easy way of ensuring that the lines of music remain properly synchronised.

HOLD/RELEASE

Another facility provided by the channel status facility is HOLD/RELEASE. Hold is obtained by adding 64 to the channel status number. This short program shows how this facility operates.

```
10 SOUND 65,100,100,5,0,0,0
20 SOUND 66,150,100,5,0,0,0
30 SOUND 68,250,100,5,0,0,0
40 FOR D = 1 TO 1000:NEXTD
50 RELEASE 7
```

The three SOUND instructions at lines 10 to 30 are for channels A, B and C respectively, but as 64 has been added to each channel status number the hold facility is obtained, and these instructions will not be executed as they are

reached. Line 40 merely provides a delay of about one second before the RELEASE instruction at line 50 is reached. The RELEASE instruction removes the hold on the specified channel or channels, and allows any SOUND commands that have been queuing to be completed. In the RELEASE instruction the following numbers are used to select the desired channel.

1 selects Channel A
2 selects Channel B
4 selects Channel C

If more than one channel is to be RELEASED the sum of the appropriate channel numbers is used. Thus in the example given above all three channels are released by using a figure of 7.

FLUSHING

The final facility provided by the channel status parameter is flushing, and this is obtained by adding 128 to the channel status number. This short program demonstrates the effect of flushing.

```
10 FOR C = 1 TO 60
20 SOUND 129,1,1,0,0,0,0
30 NEXT C
```

Lines 10 and 30 are used to loop the program 60 times, and the duration parameter in the SOUND instruction gives a one second tone on each loop. However, the channel status value of 129 gives a tone on channel A with flushing (1 for channel A plus 128 for flushing = 129). The effect of flushing is to make the sound instruction take effect at once, regardless of any queuing on the channel concerned. If you run this program you will find that the sound instruction only runs its full duration after the last loop, and that the others last only as long as it takes to complete each loop. As the

version of BASIC used in the CPC464 is a fast one the total duration of the sound is not much more than a second, and not the one minute that would be obtained without the use of flushing.

One use of flushing is to switch off one or more channels of the sound generator, as in the following demonstration program.

```
10 SOUND 65,200,100,7,0,0,0
20 SOUND 129,1,1,0,0,0,0
30 RELEASE 1
```

Line 10 generates a one second tone on channel A, but with hold. Line 20 flushes channel A and produces no tone due to the volume parameter of zero. Finally, line 30 RELEASES channel A, but no tone should be produced since the flushing at line 20 over-rides the hold at line 10.

NOISE

A noise signal can be generated and mixed into one or more of the tone channels, but note that there is only one noise generator and only one noise sound at a time can be produced. It is not necessary to have a tone produced together with the noise, and by using a small pitch value of 1 or 2 the tone can be made inaudible so that only the noise will be apparent on the output. The type of noise produced is the usual hissing type noise sound, and the pitch can be controlled. The noise is enabled by using a number of between 1 and 31 as the last parameter in a SOUND instruction (a value of zero is used to suppress the noise). 31 gives minimum pitch – 1 gives maximum pitch. A range of 31 pitches may seem rather restrictive, but the main use of noise is in sound effects, and the available range of noises is adequate for this application.

The most common use of noise is in the generation of explosive sounds. The program that follows gives an explosion sound effect.

```
10 P = 12
20 FOR V = 7 TO 0 STEP -1
30 SOUND 1,1,20,V,0,0,P
40 P = P + 2
50 NEXT V
```

The SOUND instruction at line 30 generates sound on channel A, but the tone is too high pitched to be heard. The duration is one fifth of a second on each loop of the program, and the volume and noise pitch values are controlled by variables V and P respectively. V is stepped from 7 to 0 as the program loops, giving maximum volume, steadily decreasing to zero. This shaping of the volume is essential in order to give a reasonable simulation of an explosive sound. Line 40 increases the noise pitch by 2 on each loop of the program, giving a reduction in pitch and something approaching minimum pitch at the end of the signal.

This program is easily modified to give a gunshot sound simulation, and it is merely necessary to use a higher pitch range and give the sound a shorter duration. A suitably modified version of the program is provided below.

```
10 P = 3
20 FOR V = 7 TO 0 STEP -1
30 SOUND 1,1,3,V,0,0,P
40 P = P + 1
50 NEXT V
```

ENV

For most purposes a SOUND instruction plus perhaps a loop or some other support is all that is needed to give the desired effect. However, where complex shaping of the volume or pitch is required it is easier to use a SOUND instruction with either an ENV (envelope – volume) or an ENT (envelope – tone) instruction. We will consider ENV first.

The first parameter in an ENV instruction is merely an identification number (from 1 to 15) that enables the required

ENVelope to be called up in a SOUND instruction. This is accomplished by using the ENV number as the fifth parameter in the SOUND instruction. The next three numbers in the ENV instruction control the number of steps, the decrease or increase in volume provided by each step, and the duration of each step (in hundredths of a second). The number of steps must be in the range 0 to 127, while the volume step size can be between −128 and +127. However, bear in mind that there are only fifteen volume levels plus off, and the volume step size would normally be in the range −15 to +15. The maximum duration per step is 255. As a simple example, to give a step down in volume from 15 to 0 in fifteen steps of one tenth of a second (ten hundredths) in duration, the instruction:-

```
ENV 1,15,-1,10
```

would be used (assuming the ENVelope is to be identified as ENVelope 1).

In practice up to five sections of envelope shaping can be used, with each section requiring three figures to set the number of steps, step size, and step duration. This can make complex envelopes a little difficult to calculate, and it is often worthwhile drawing out the envelope shape before calculating suitable figures for the ENV instruction. Figure 1 gives an example of this. The aim of this envelope is to give an envelope shape similar to that of a piano and many other instruments. A fast initial rise from zero to full volume is required, followed by a fairly rapid fall in volume to a medium level, after which the sound is sustained at that level for a period, and then gradually decays. This is normally termed ADSR (attack, decay, sustain, release) envelope shaping. In Figure 1 the sound rises to full volume in just five one hundredths of a second. With 15 volume levels this requires five steps of minimum duration and a step size of +3. The first part of the ENV instruction would therefore be:-

```
ENV 1,5,3,1
```

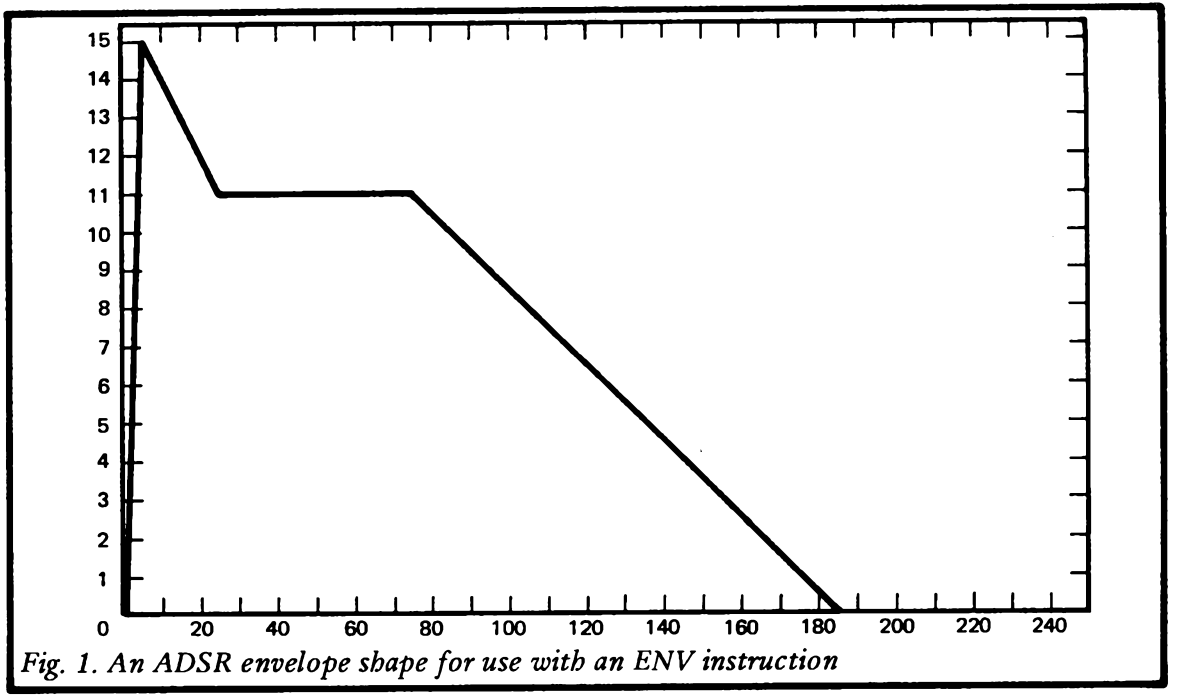

Fig. 1. An ADSR envelope shape for use with an ENV instruction

The next stage lasts 20 one hundredths of a second, and the volume falls four units. This requires four steps of five one hundredths of a second in duration, with the volume falling by one unit per step.

The next section has just one step, there is no drop in volume, and it lasts 50 one hundredths of a second. Finally, the fourth step has eleven single step falls in volume with each step lasting 10 one hundredths of a second. If you work out the entire ENV instruction to give this shape you should end up with the one given below.

```
ENV 1,5,3,1,4,-1,5,1,0,50,11,-1,10
```

This can be used in a simple program to play music, as in the example given below.

```
10 ENV 1,5,3,1,4,-1,5,1,0,50,11,-1,10
20 READ P,D
30 IF P = 0 THEN 70
40 SOUND 1,P,D,0,1,0,0
50 GOTO 20
60 DATA 239,50,213,50,190,50,179,50,159,50,142,50,
   127,50,119,185,0,0
```

This is similar to the music program described earlier, and with the values shown in the DATA statement at line 60 this one also plays a scale of C major. The values in the DATA statement are pairs of note and duration values, in that order. No volume parameter needs to be set from the data statement since the envelope shaping ensures that successive notes of the same pitch stand out from one another and do not merge together. In fact, in this case the volume cannot be altered from the DATA statement since a volume figure of 0 has to be used. Otherwise the required start and finish at zero would not be produced by the ENV instruction, and also the volume parameter would go out of range.

An important point to note is that the duration of each note is set at the SOUND instruction. This is in fact only possible if the duration in the SOUND instruction is less than

that set in the ENV instruction, or if the ENV instruction gives a final volume value of other than zero. Otherwise the sound ends when it reaches zero at the end of the ENV instruction.

ENT

ENT operates in a similar manner to ENV, and as before, the first parameter is simply a number in the range 1 to 15 which identifies the ENT instruction. It is called up from a SOUND instruction by using this number as the sixth parameter in the SOUND instruction. Again, three numbers are used to define the characteristics of each section of the envelope shaping, and up to five sections can be used. The first one sets the number of steps (0 to 239), the second sets the number of pitch units jumped in each step (−128 to +127), and the third controls the duration of each step (0 to 255 in one hundredths of a second).

If we take a simple example, the following program gives a police car siren type sound.

```
10 ENT −1,20,2,1,20,−2,1
20 SOUND 1,50,1000,6,0,1,0
```

The first point to notice is that the envelope number is negative, and this is done to give an automatic repeat of the envelope. Using a positive number gives just one execution of the envelope, and the pitch then stays at its final value until the SOUND instruction is terminated. A positive ENT number is used in the sound instruction.

The values in the ENT instruction give 20 rises in pitch with the pitch being incremented by two in each step. Each step lasts the minimum duration of one hundredth of a second. The second set of three figures simply reverse this process and steps the pitch back to its original figure. The initial pitch value is set by the pitch value in the relevant SOUND instruction. In this case this pitch value is swept upwards by 40, then down by 40, and so on, giving the

required continuous fall and rise in pitch.

If the duration figure used in the SOUND instruction is too short to permit the tone envelope to be completed, the signal is nevertheless terminated after the duration set in the SOUND instruction.

It is perfectly possible to use both ENV and ENT instructions together with a single SOUND instruction. For example, the music program provided earlier can have line 5 added and line 40 modified, as shown below.

```
 5 ENT -1,1,1,4,2,-1,4,1,1,5
40 SOUND 1,P,D,0,1,1,0
```

The ENT instruction simply varies the pitch value by plus and minus 1, giving quite a good vibrato effect.

Chapter 6

GRAPHICS 1 – MODES & COLOURS

In most computers supporting several screen display modes, there is a clear trade-off between the number of colours that can be displayed in a given mode, and the amount of detail that can be shown – more detail – less colours.

The Amstrad is a good example of this. It has three modes, numbered from 0 to 2. Mode 0 has the lowest resolution and the most colours. It can display 20 columns of text (all modes display 25 lines of text), and can use up to 16 colours at once. The graphics resolution is 160 x 200, which means it can plot 160 separate dots across the screen, and 200 up the screen. The large number of colours which can be displayed at once makes this a good mode for games playing, where very fine detail is not generally of great importance.

Mode 1 is the default mode – the one the computer is in when first turned on. It can display 40 columns of text, and has a graphics resolution of 320 x 200. It can display four colours at once. This is a good mode for general-purpose programming, and for scientific and business graphics, where the extra detail can be more important than lots of colour. Four colours are generally quite adequate for these uses.

Mode 2 displays 80 columns of text, and has a graphics resolution of 640 x 200. It can display only two colours at one time. This mode is suitable for graphics when detail is of the utmost importance, and is also ideal for applications such as word-processing and spreadsheets. The problem with this mode is that the text may not be clear on a colour monitor unless the two colours are chosen carefully, and is not generally readable at all on a T.V. used with the optional modulator.

The Amstrad display hardware is capable of producing 27 different colours. You can choose the colours you want to use from these, the number of choices being set by the mode in use. It is rather as if you have 27 bottles of ink from which you can fill your pens. You have 2 pens if you want to write

very small, four for normal writing, and 16 for bold writing. You can fill the pens with any of the inks you like (and can fill two or more pens with the same colour if required). The colour of the paper you use must be chosen from one of the inks in the pens. Of course, if you were drawing on paper, you could go back to your bottles of ink and refill a pen with a different colour. You can do this with the Amstrad computer, but if you do, all the writing already done with that pen will change to the new colour! This can actually be very useful in programming.

This is a useful analogy, because the keywords used to control the colour in Locomotive BASIC are INK, PEN, and PAPER. INK assigns one of the 27 available colours to one of the available pens, PEN selects which pen is to be used for writing (continuing until another PEN command is issued), and PAPER selects which PEN colour is to be used for the background.

In fact, all the pens have default values, so you do not necessarily have to use INK commands. You can use the colours assigned by the computer. Try turning your computer on, and typing "pen 2". When you press ENTER, the "Ready" message will appear in light blue. Type "pen 3" (which will also be in blue) and the "Ready" message will be in red. By default, the mode 1 pens are filled with dark blue (pen 0), yellow (pen 1), light blue (pen 2) and red (pen 3). Pen 0 is used for the background (PAPER), and pen 1 is used for foreground, until changed.

If you now type "paper 1", future text will appear in red on a yellow background. If you set the paper to the current text pen, any text printed will be invisible (i.e. enter paper 3).

To illustrate changing the ink in a pen, reset the computer (SHIFT/CTRL/ESCAPE), then type "ink 0,24". The screen should turn yellow, and of course the yellow writing will be invisible! Next type "ink 1,1". You won't be able to see what you are typing but you should manage. When you press ENTER the writing should all re-appear in dark blue. It may, however, look black. INK 1,2 will give a brighter blue. The first number in the INK command is the number of the pen you wish to refill, the second number is the colour of the

ink (0 to 26).

In fact, you can assign ink colours to all 16 pens no matter which mode you are in, but you can only actually use the number of pens available. That is, pens 0 and 1 in mode 2, pens 0–3 in mode 1, and pens 0–16 (i.e. all of them) in mode 0. The currently selected pen is not altered by a mode change, so if you select pen 3 in mode 1 (red by default) and then change to mode 0, text will still be in red.

If you select a pen number higher than that available in a given mode, the number will be reduced modulo the number available. Thus, if you select pen 4 in mode 1 you will get pen 0, and pen 5 will give you pen 1. This is not altered on a mode change. That is, if you select pen 6 in mode 1, you will get pen 2. If you then change to mode 0, pen 2 will still be selected. However, if you change to mode 2, a further modulo reduction will occur, and pen 0 will be forced. (In this instance, as this will, by default, be the same as the PAPER colour, text writing would be invisible.)

Listing 11 demonstrates the colour capabilities of mode 0. The first loop, lines 50 to 80, puts a line of text in each of the 16 default colours on the screen. The pen 0 line cannot be seen as it is the same as the background. The second part of the program, lines 90 to 140, are two nested FOR . . . NEXT loops, which cause all 16 pens to cycle through all 26 colours. This obviously leaves all pens set to colour 26! The rest of the program resets the four pens used by mode 1 to their default colours, so you can see what you are doing! You may like to try to modify this program so that it resets all 16 pens (HINT: Use a FOR . . . NEXT loop and DATA statements. Alternatively you could simply replace lines 140 to 210 with CALL &BBFF, an operating system subroutine which resets screen colours and mode, but not current pen selections.)

The ink command also allows any pen to be set to produce a flashing effect. This is done by specifying two ink colours, which will appear alternately. An on-off flash can be obtained by setting one of the ink colours to the current background colour.

The rate of flash is controlled by the SPEED INK

command. This is followed by two numbers, which must be integers. The first gives the time the first colour is to appear, the second the time the second colour is to appear, so both speed of flash and relative duration of the colours is controllable. These commands are beautifully easy to use, and experimentation is recommended, but don't get too goggle-eyed!

LISTING 11

```
20 REM colour demonstration

30 MODE 0

40 CLS

50 FOR colour=0 TO 15

60 PEN colour

70 PRINT "This is Pen ";colour

80 NEXT colour

90 FOR quill=0 TO 15

100 FOR colour=0 TO 26

110 INK quill,colour

120 FOR delay=1 TO 200:NEXT delay

130 NEXT colour

140 NEXT quill
```

```
150 MODE 1

160 INK 0,1

170 INK 1,24

180 INK 2,2

190 INK 3,3

200 PEN 1

210 END
```

LINE DRAWING

The Amstrad has a comprehensive set of commands for drawing lines and plotting points on the screen. These use a system of co-ordinates to give the horizontal and vertical positions.

Though the resolution in the three modes differs, the co-ordinate values are the same. This means a given set of DRAW commands will draw the same shape in the same place on the screen in all three modes. This is a considerable benefit. Of course, the amount of detail possible and fineness of line will differ.

In fact, all three modes have the same vertical resolution, 200 lines. The co-ordinate system uses a scale of 400 vertically, so this means that points 0 and 1 in the co-ordinate system will refer to the same point on the screen, as will 2 and 3, 4 and 5, and so on. This means there is no point in using odd numbers when specifying vertical co-ordinates.

The horizontal co-ordinates use 640 points horizontally. This matches the highest resolution mode, mode 2. Mode 1 has 320 lines horizontally, which means that points 0 and 1 refer to the same point, as in the vertical. This means that

mode 1 has equal resolution in the two directions, which can be useful for some graphics display purposes. Mode 2 has only 320 points horizontally, which means that points 0,1,2, and 3 refer to the same point.

The on-screen graphics co-ordinates run from 0 to 639 horizontally, 0 to 399 vertically. The position 0,0, called the origin, is in the bottom left-hand corner (unlike the text origin which is top left) but it can be moved to anywhere on the screen (or, indeed, off it) with the ORIGIN command.

With the Amstrad, it is quite legal to draw off the screen. The off-screen position is correctly recorded, which means if you draw off the screen, and then back on again, the second line will have the correct angle. This makes some perspective-drawing effects easy to achieve.

The computer has a graphics cursor, as well as a text cursor, but the graphics cursor is invisible. When lines are drawn, they are drawn from the current cursor position to the new position specified.

There are two ways of specifying co-ordinates, relative and absolute. Relative means that the values specified are a displacement relative to the current cursor position. Absolute means that the values specified are absolute co-ordinates on the imaginary screen grid, or to put it another way, relative to the graphics origin. Negative co-ordinates are legal in both relative and absolute.

On the whole, it is likely that you will find the absolute co-ordinate system the easiest and most useful, but relative co-ordinates can simplify some jobs.

The three main graphics commands are MOVE, DRAW, and PLOT. These are the absolute forms. The relative forms have an extra R on the end, MOVER, DRAWR, PLOTR. MOVE simply moves the graphics cursor without making a mark on the screen. DRAW draws a line from the graphics cursor position to the new position specified. PLOT is similar to MOVE, but it plots a point at the new cursor position.

Listing 12 illustrates MOVE and DRAW, and relative and absolute co-ordinates. It draws a box on the screen, the size

and position of which are entered in INPUT statements in lines 50 to 70. The actual box-drawing part of the program has been written as a subroutine, which you could adapt for use in your own programs.

Line 1010 moves the graphics cursor to the absolute position specified in line 50. The next four lines draw the box, using relative co-ordinates. The order of drawing is left side, top, right side, bottom. Note that drawing up and to the right are positive displacements, down and to the right are negative.

LISTING 12

```
20 REM box drawing

30 MODE 1

40 WHILE 1

50 INPUT "Bottom corner position (x
,y)";x,y

60 INPUT "Height";high

70 INPUT "Width";wide

80 CLS

90 GOSUB 1000

100 WEND

1000 REM box drawing subroutine

1010 MOVE x,y
```

```
1020 DRAWR 0,high

1030 DRAWR wide,0

1040 DRAWR 0,-high

1050 DRAWR -wide,0

1060 RETURN
```

Locomotive BASIC does not include any circle drawing command. It is, however, quite easy to draw a circle using the in-built SIN and COS functions. This is illustrated in Listing 13, which draws an interesting pattern of ellipses within a circle, using the default colours in mode 1.

Line 30 sets the colour of the border around the screen area. The border can be set to any of the 27 available colours, or to flash between any two of them, irrespective of current pen settings and the number of colours available in the current mode. The colour number(s) in the BORDER statement therefore come from the list of 27 colours, and not the pen numbers. In this case, however, I have used a current pen colour.

Lines 50 to 80 draw a box around the screen. Note the extra parameter on the end of the DRAW statements. The colour used for graphics line drawing is not controlled by the current text pen setting. It is controlled by this extra parameter, which must be a number of an available pen. If no colour is specified in a DRAW statement, the last colour specified remains in use, defaulting to pen 1 colour.

Lines 100 to 140 draw the circle. In line 100, x and y are the co-ordinates of the centre of the circle, and d sets the diameter (actually, the number is the radius in graphics units, and would be better called r, but I have been using d for this parameter for too long to change now!). Line 110 moves the graphics cursor to the first point on the circle, and the FOR . . . NEXT loop in lines 120 to 140 do the actual drawing. Note that line 130 selects pen 2 colour.

Lines 160 to 250 draw the pattern of ellipses. This is a modification of circle drawing, the circles being stretched in a direction controlled by line 170 (the number by which PI is divided in this line controls the number of ellipses) and by an extent controlled by the constant in line 190 (100 in this case). Lines 200 to 230 do the drawing. Note that the order of SIN and COS in lines 200/210 are reversed compared to line 130. Thus the ellipses are drawn clockwise, and the circle is drawn widdershins. Line 220 moves the cursor to the start point of each ellipse. Line 230 draws in pen 3 colour.

LISTING 13

```
20 MODE 1

30 BORDER 2

40 REM draw box

50 DRAW 0,399,1

60 DRAW 639,399,1

70 DRAW 639,0,1

80 DRAW 0,0,1

90 REM now draw circle

100 x=320:y=200:d=196

110 MOVE x+d,y

120 FOR c=0 TO 2*PI STEP 0.03

130 DRAW x+d*COS(c),y+d*SIN(c),2
```

```
140 NEXT c

150 REM now draw the ellipses

160 d=92

170 FOR e=0 TO PI STEP PI/5

180 FOR p=0 TO 2*PI STEP 0.03

190 f=100*COS(p)

200 x1=x+f*SIN(e)+d*SIN(p+e)

210 y1=y+f*COS(e)+d*COS(p+e)

220 IF p=0 THEN MOVE x1,y1

230 DRAW x1,y1,3

240 NEXT p

250 NEXT e

260 END
```

Listing 14 demonstrates PLOT. It simply plots dots on the screen at random positions in random colours. The results can be quite pretty.

LISTING 14

```
20 MODE 0

30 WHILE 1
```

```
40 PLOT RND(1)*640,RND(1)*400,RND(1
)*15
50 WEND
```

Finally in this section, Listing 15 is the program used to produce the screen display in the cover photo. Again, it is a modification of the circle-drawing formula. In this case, the value of d is decremented (line 140) producing a spiral. The spiral is not drawn, the cursor is simply moved to points on it (line 120). A second spiral, related to the first, is generated by line 130, in conjunction with line 100. Lines are drawn from the points on the first spiral to points on the second spiral. The variable x controls the colour. The available colours are taken in sequence, except that the background colour is suppressed (line 80) and the two flashing colours are omitted (line 90).

LISTING 15

```
20 MODE 0
30 BORDER 2
40 PAPER 5:CLS
50 x=1:d=350
60 WHILE d>34
70 FOR c=0 TO 2*PI STEP 0.1
80 IF x=5 THEN x=6
90 IF x=14 THEN x=1
```

```
100 e=d-50
110 IF e<0 THEN e=0
120 MOVE 320+d*COS(c),200+d*SIN(c)
130 DRAW 320+e*COS(c-0.5),200+e*SIN
(c-0.5),x
140 d=d-1
150 IF e<0 THEN e=0
160 x=x+1
170 NEXT c
180 WEND
190 GOTO 190
```

WINDOWS

Normally the whole screen of the monitor or television set, apart from the border area, is available when PRINTing, LISTing, etc. However, it is possible to divide the screen into sections, and then PRINT or LIST in each section, or "window" as they are usually termed. The CPC464 can have up to eight of these windows. As one would expect, the WINDOW instruction is used when defining a window, and this instruction has five parameters. The first of these is the channel or stream number, and this is a means of identifying the windows so that the appropriate one can be selected in (say) a PRINT instruction. The other four parameters specify the limits of the window using the ordinary screen co-

ordinate system. The first two of these parameters determine the width and lateral position of the window, as they are respectively the left and right most columns occupied by the window. Similarly, the last two figures specify the top and bottom lines that are occupied by the window.

As a simple example,

```
WINDOW#1,10,20,15,22
```

would produce a window having 1 as its channel number, with screen co-ordinate 10–15 at the top left hand corner, and co-ordinate 20–22 at the bottom right hand corner. Of course, the exact area of the screen that this would occupy is dependent on the screen mode used, as the co-ordinate numbering is different for each mode.

DICE

The "Dice" program of Listing 16 helps to show the way in which windows can be used. The idea of the program is to graphically represent a pair of dice on the screen. Operating the spacebar causes the dice to roll a few times and then display the final numbers with the spots in the appropriate patterns. Two windows are used in the program, and these produce the bodies of the dice.

Line 20 sets the required mode, and in this case high resolution is not needed, but a wide range of available colours is an advantage, and mode 0 has therefore been selected. Line 30 sets the paper colour to light blue and line 40 gives a bright green border. The windows are defined at the next two lines, and these are given channel numbers 1 and 2. Figure 2 shows the size and positions of the windows, as well as the positions of the spots on each die. Each window is 7 co-ordinates wide by 14 high, but as each screen co-ordinate (in mode 0 anyway) is about twice as wide as it is high, this gives reasonably square dice. Similarly, each spot is one co-ordinate wide by two high, and reasonably square.

Lines 70 to 140 set the paper and pen colours for each

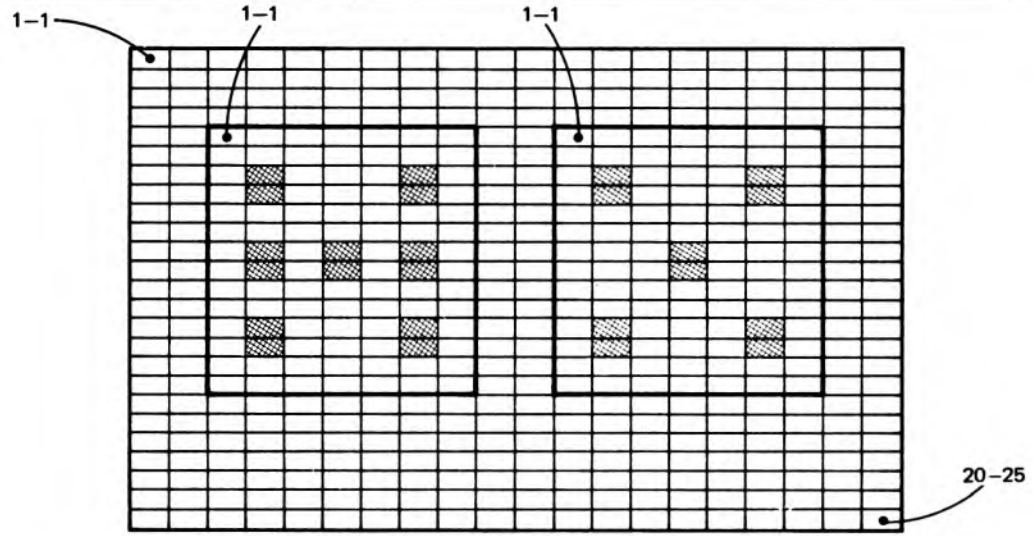

Fig. 2. The screen layout used for the DICE program

window, with the appropriate channel numbers being used at lines 110 to 140. The right hand die is yellow with orange spots, while the right hand one has a bright magenta background with red spots. However, you should have no difficulty in changing these to any colours you prefer. Lines 150 and 160 are used to clear the backgrounds of the windows to the set colours. The next few lines produce two random integers from 1 to 6 and assign these to variables l and r (left die and right die). Line 200 simply provides a clicking sound effect each time the dice roll.

The spots are placed on the dice using a series of IF . . . THEN instructions and a series of subroutines. Four subroutines are used for each die, and the first of these draws the centre spot. For the left hand die this is the subroutine starting at line 400. A new instruction is introduced here, and this is LOCATE. It is used to place the cursor at the desired position within a window. In this case it is used to position the cursor at the point where one of the spots must be placed, and then a PRINT instruction is used to print a suitable character on the screen at this position. The character we are using here is number 143, and if you refer to the character set in the CPC464 manual (Appendix III) you will find that this is simply a solid block of the current pen colour. Two of these blocks are needed to produce each spot, and each one requires separate LOCATE and PRINT instructions.

As already pointed out, the first subroutine draws the centre spot. The next draws the top-right and bottom-left spots, while the third produces the other two corner spots. The fourth routine draws the spots to the left and right of the centre one. By calling up the appropriate subroutines at the IF . . . THEN instructions the correct pattern of spots can be produced. For example, the second and third subroutines give the pattern for a four, and the final three subroutines give the correct pattern for a six. Separate sets of subroutines are used for the two dice.

An important point to note about the LOCATE instruction is that it does not use the normal screen co-ordinates. Instead, each window has its own co-ordinates starting at 1–1 in the top left hand corner. Thus the centre spot for the left hand

die is not as co-ordinates 6–11 and 6–12, but is instead at 4–7 and 4–8. Also, because of this system the co-ordinates used in the subroutines for the left hand die are exactly the same as those for the right hand die. The only difference between the two sets is the channel number used in the LOCATE and PRINT instructions.

Lines 330 to 360 are used to loop the program a few times to give the rolling of the dice. Variable x is set randomly at 0, 1, or 2, on each loop of the program, and the dice roll until it is set at 1. This gives typically about three rolls of the dice, but the number can be anything from one to about a dozen times. The old spots must be blanked before a new set is added, but this is achieved quickly and neatly by simply sending a CLS instruction to each window. Line 370 is used to hold up the program until the spacebar is operated, whereupon the old dice are blanked and the program branches back to line 180 to set the dice rolling again.

LISTING 16

```
20 MODE 0

30 INK 0,2

40 BORDER 18

50 WINDOW#1,3,9,5,18

60 WINDOW#2,12,18,5,18

70 INK 2,25

80 INK 3,15

90 INK 4,8

100 INK 5,6
```

```
110 PAPER#1,2
120 PAPER#2,4
130 PEN#1,3
140 PEN#2,5
150 CLS#1
160 CLS#2
170 RANDOMIZE TIME
180 l = INT (RND*6+1)
190 r = INT (RND*6+1)
200 SOUND 1,1,2,7,0,0,2
210 IF l = 1 THEN GOSUB 400
220 IF l = 2 THEN GOSUB 430
230 IF l = 3 THEN GOSUB 400:GOSUB 43
0
240 IF l = 4 THEN GOSUB 430:GOSUB 48
0
250 IF l = 5 THEN GOSUB 400:GOSUB 43
0:GOSUB 480
```

```
260 IF l = 6 THEN GOSUB 430:GOSUB 49
0:GOSUB 530
270 IF r = 1 THEN GOSUB 580
280 IF r = 2 THEN GOSUB 610
290 IF r = 3 THEN GOSUB 580:GOSUB 61
0
300 IF r = 4 THEN GOSUB 610:GOSUB 66
0
310 IF r = 5 THEN GOSUB 580:GOSUB 61
0:GOSUB 660
320 IF r = 6 THEN GOSUB 610:GOSUB 66
0:GOSUB 710
330 x = INT (RND*3)
340 IF x = 1 THEN 370
350 CLS#1:CLS#2
360 GOTO 180
370 a$ = INKEY$:IF a$ <> CHR$(32) TH
EN GOTO 370
```

```
380 CLS#1:CLS#2
390 GOTO 180
400 LOCATE#1,4,7:PRINT#1, CHR$(143)
410 LOCATE#1,4,8:PRINT#1, CHR$(143)
420 RETURN
430 LOCATE#1,6,3:PRINT#1, CHR$(143)
440 LOCATE#1,6,4:PRINT#1, CHR$(143)
450 LOCATE#1,2,11:PRINT#1, CHR$(143)
460 LOCATE#1,2,12:PRINT#1, CHR$(143)
470 RETURN
480 LOCATE#1,2,3:PRINT#1, CHR$(143)
490 LOCATE#1,2,4:PRINT#1, CHR$(143)
500 LOCATE#1,6,11:PRINT#1, CHR$(143)
510 LOCATE#1,6,12:PRINT#1, CHR$(143)
520 RETURN
530 LOCATE#1,2,7:PRINT#1, CHR$(143)
540 LOCATE#1,2,8:PRINT#1, CHR$(143)
550 LOCATE#1,6,7:PRINT#1, CHR$(143)
```

```
560 LOCATE#1,6,8:PRINT#1, CHR$(143)
570 RETURN
580 LOCATE#2,4,7:PRINT#2, CHR$(143)
590 LOCATE#2,4,8:PRINT#2,CHR$(143)
600 RETURN
610 LOCATE#2,6,3:PRINT#2, CHR$(143)
620 LOCATE#2,6,4:PRINT#2, CHR$(143)
630 LOCATE#2,2,11:PRINT#2, CHR$(143
640 LOCATE#2,2,12:PRINT#2, CHR$(143
650 RETURN
660 LOCATE#2,2,3:PRINT#2, CHR$(143)
670 LOCATE#2,2,4:PRINT#2, CHR$(143)
680 LOCATE#2,6,11:PRINT#2, CHR$(143
690 LOCATE#2,6,12:PRINT#2, CHR$(143
700 RETURN
710 LOCATE#2,2,7:PRINT#2, CHR$(143)
720 LOCATE#2,2,8:PRINT#2, CHR$(143)
730 LOCATE#2,6,7:PRINT#2, CHR$(143)
```

```
740 LOCATE#2,6,8:PRINT#2, CHR$(143)

750 RETURN
```

WINDOW SWAP

WINDOW SWAP is an instruction that can be used to swap over the channel numbers of two windows. For example, to direct data for channel 3 to channel 4 (and vice versa) the instruction:–

WINDOW SWAP 3,4

would be used. This command can be used to direct data that would normally go to the main screen (which is channel 0) to a window. For example, WINDOW SWAP 0,4 would result in data being sent to the channel 4 window instead of the main screen. It should be possible to modify the dice program to use this instruction and one set of subroutines to accommodate both dice. You might find it instructive to try out this idea.

Chapter 7

GRAPHICS 2 – ANIMATION

The principle of animation is simple. Print a character on the screen, then block it out and print and the same character in an adjacent position. Repeat this at appropriate speed and the character will appear to be moving across the screen.

It is also possible to animate an object consisting of more than one character, and this is very easy if only horizontal motion is involved. Listing 17 illustrates this. It moves a bucket, consisting of two characters from the graphics set and two underline characters across the bottom of the screen. By putting a space (CHR$(32)) at each end of this string, it effectively blocks itself out as it moves. Motion of the bucket is controlled from the keyboard using the z key for left movement and the \ key for right movement. These keys are checked using the INKEY function in lines 60 and 70. INKEY returns −1 when the key specified is not pressed, and various other values when it is pressed, depending on whether the shift key, control key, or both, are also down. We can therefore check simply for pressed or not pressed by using IF NOT, as explained in Chapter 3.

Lines 80 and 90 are to prevent the bucket moving off the edges of the screen.

Line 95 needs some explanation. When this program is run, a double image of the bucket is sometimes visible as it moves. CALL &BD19 calls an operating system subroutine which causes the machine to wait for the start of the next frame of the video display. I tried this to see if it would reduce the double effect. You may agree with me that it is not effective in this case.

Once you can move a bucket, you can catch things in it. Let's see how to tackle vertical motion, then the two ideas can be put together into a game.

LISTING 17

```
20 REM bucket movin9

25 CLS

30 bucket$=CHR$(32)+CHR$(205)+"__"+

CHR$(204)+CHR$(32)

40 bx=17:by=24

50 WHILE 1

60 IF NOT INKEY(71) THEN bx=bx-1

70 IF NOT INKEY(22) THEN bx=bx+1

80 IF bx<1 THEN bx=1

90 IF bx>34 THEN bx=34

95 CALL &BD19

100 LOCATE bx,by:PRINT bucket$

110 WEND
```

Listing 18 causes the heart shape from the graphics set (CHR$(228)) to move vertically down the screen.

Line 40 seeds the random number generator. Line 45 sets initial values to two variables which will be used to remember the last position of the falling heart for blocking out purposes. Line 50 sets the horizontal position of the heart at random.

The descent is controlled by a FOR . . . NEXT loop, lines 60 to 100. The control variable is used as the vertical

co-ordinate. Line 70 prints the heart, line 80 blocks it out at the previous position. Line 90 updates the variables odx and ody.

LISTING 18

```
20 REM vertical motion

30 CLS

40 RANDOMIZE TIME

45 odx=1:ody=1

50 LET dx=INT(RND*30+5)

60 FOR dy=1 TO 23

70 LOCATE dx,dy:PRINT CHR$(228)

80 LOCATE odx,ody:PRINT CHR$(32)

90 odx=dx:ody=dy

100 NEXT dy

110 GOTO 50
```

Listing 19 is the Bucket Game which brings these two ideas together. The first part of this program, lines 20 to 70, does things which only need to be done once, such as defining the "bucket" string, and printing the screen labels.

The second part, lines 90 and 100, do things which must be done for each bomb. Line 100 selects which of two characters will be printed, the heart, or a downwards-arrow. The idea of the game is that you must catch the hearts (scoring 10 points)

and avoid the arrows (penalty – points halved if you don't!). Line 110 sets the horizontal position as in Listing 18.

Line 120 calls the subroutine which controls the main play, lines 1000 to 1120. It can be seen that this is constructed from Listings 17 and 18 interwoven together, with a few extras, such as PEN and SOUND commands.

When each bomb reaches the bottom of the screen, line 130 compares the horizontal co-ordinates of bomb and bucket. If a catch has occurred (the bomb must be over the underline characters to count) the routine from line 2000 is called. This routine actually uses the variable c, which otherwise controls the bomb colour, to determine how to change the score (lines 2010 and 2020, which include appropriate sound effects). This routine also prints the score on the screen.

A round consists of 20 bombs. After this, line 150 calls the subroutine from line 3000, which offers the option of another round or exiting the program. Line 3015 is interesting. The INKEY function checks to see if a key is pressed, but does not remove a character from the input buffer. This means that all the keypresses from moving the bucket are stored (up to the limit of the buffer). Line 3015 strips these keypresses, so that the program does not fall through line 3030. Line 3060 holds the program in this routine if one of the two play keys is pressed by mistake, which can easily happen. This routine does not interpret the player's response.

Line 160 determines what action to take. If the y key is pressed, another round follows. If any other key is pressed, line 170 resets the default colours and screen mode, and the program ends.

The routine in lines 4000 to 4050 is a simple routine to print the highest score since the program was run on the screen.

The last routine, from line 5000, was an afterthought. This causes the program to wait until a key is pressed before starting each round, so the player can get ready. This routine is called by line 85.

This game has been written using the default colours in

mode 1. You may find the arrows a little difficult to see (they are bright yellow). If so, it is easy to put a line somewhere near the beginning of the program (line 45, say) redefining the colour for PEN 1.

Animation using the text co-ordinates will always be a little jerky, but can be quite fast and exciting, and is useful for simple games of this sort.

LISTING 19

```
20 REM Bucket Game

30 MODE 1

40 BORDER 18:PAPER 2:CLS

50 bucket$=CHR$(32)+CHR$(205)+"__"+
CHR$(204)+CHR$(32)

55 bx=17:by=24

60 LOCATE 1,1:PEN 0:PRINT "SCORE"

70 LOCATE 34,1:PRINT "HISCORE"

80 RANDOMIZE TIME

85 GOSUB 5000:REM 'Press any key' r
outine

90 FOR bombs=1 TO 20: REM round ini
tiation
```

```
100 IF RND>0.5 THEN bomb$=CHR$(228)
:c=3:ELSE bomb$=CHR$(241):c=1
110 dx=INT(RND*25)+7
120 GOSUB 1000:REM play routine
130 IF (dx=bx+2) OR (dx=bx+3) THEN
GOSUB 2000:REM scoring
135 LOCATE odx,ody:PRINT CHR$(32)
140 NEXT bombs
150 GOSUB 3000:REM another round?
155 GOSUB 4000:REM hiscore
160 IF LOWER$(ans$)="y" THEN 85
170 CALL &BBFF:PAPER 0:PEN 1:CLS
180 END
1000 REM play loop routine
1010 odx=7:ody=1
1020 FOR dy=1 TO 23
1030 LOCATE odx,ody:PRINT CHR$(32)
1040 LOCATE dx,dy:PEN c:PRINT bomb$
```

```
1050 odx=dx:ody=dy
1055 SOUND 1,dy*50,5
1060 IF NOT INKEY(71) THEN bx=bx-1
1070 IF NOT INKEY(22) THEN bx=bx+1
1080 IF bx<1 THEN bx=1
1090 IF bx>34 THEN bx=34
1100 LOCATE bx,by:PEN 0:PRINT bucke
t$
1110 NEXT dy
1120 RETURN
2000 REM scoring
2010 IF c=3 THEN pts=pts+10:SOUND 1
,50,50
2020 IF c=1 THEN pts=CINT(pts/2):SO
UND 1,1000,100
2030 LOCATE 1,2:PEN 0: PRINT pts
2040 RETURN
3000 LOCATE 15,10
```

```
3010 PEN 3
3015 WHILE INKEY$<>"":WEND
3020 PRINT "Another Go? (y/n)"
3030 ans$=INKEY$:IF ans$="" THEN 30
30
3040 LOCATE 15,10
3050 PRINT SPACE$(18)
3060 IF LOWER$(ans$)="z" OR ans$= "
\" THEN 3000
3070 RETURN
4000 IF pts>hiscore THEN hiscore=pt
s
4010 LOCATE 34,2
4020 PEN 0
4030 PRINT hiscore
4040 pts=0
4050 RETURN
5000 REM start routine
```

```
5010 LOCATE 10,10

5020 PEN 3

5030 WHILE INKEY$<>"":WEND

5040 PRINT "Press any key to start"

5050 a$=INKEY$:IF a$="" THEN 5050

5060 LOCATE 10,10

5070 PRINT SPACE$(22)

5080 RETURN
```

TEXT AT GRAPHICS CURSOR

Where smoother animation is required, it is possible to place text characters at the position of the text cursor. Characters can then be moved on the screen by a pixel at a time, giving much smoother movement.

The command to do this is TAG. Normal printing at the text cursor is resumed with TAGOFF, or automatically if the BASIC returns to command mode.

When TAG is in force, the colour in which characters are printed is not controlled by PEN commands. Instead, they will be printed in the current graphics foreground colour, as set in any previous DRAW or PLOT commands (or pen 1 colour by default). When TAG is in force, the background to colours is always the current graphics paper colour, pen 0 colour by default.

The Amstrad has a number of graphics colour modes, which allow graphics drawing and TAG printing to react to what is already on the screen. The default is colour mode 0, which forces the current foreground and background colours

– that is, ignores what is already there. In mode 1, the colour which appears is the colour plotted XOR the colour already there. Mode 2 uses AND and mode 3 uses OR.

For animation, XOR is the most useful, as plotting the same thing twice in the same place under XOR restores what was originally there, so a character can be moved across a multi-coloured background without changing the background. Also, provided the graphics background colour is left as pen 0 colour, the character's paper colour is effectively transparent, as (number XOR 0)=number.

By using XOR, it is also possible in a very simple way to make the moving character pass in front of some objects on the screen, and behind others. Listing 20 illustrates this. This draws a desert scene with yellow sand, blue sky, and a couple of red pyramids. A small black tank moves across the desert, behind the first pyramid, and in front of the second.

To do this, we must reset a number of pen colours. When doing this type of effect, it is much easier if a pen number of 8 or higher is used for the character, and pen numbers less than 8 for the background. In this case pen 8 is used for the tank.

Pen 3 is used for the desert. This pen is set to yellow by the INK command in line 30. 3 XOR 8 is 11, and as we want the tank to be black, pen 11 is set to black in line 50.

We have to use different pens for the two pyramids. Pen 5 is used for the first, pen 1 for the second. Both these pens are set to red in lines 60 and 70. 1 XOR 8 is 9. As we want the tank to pass in front of the second pyramid, pen 9 is set to black (line 40). 5 XOR 8 is 13. As the tank is to pass behind the first pyramid, pen 13 is set to red (line 80).

Note that the colour for pen 8 does not have to be reset, as it never actually appears!

Lines 80 and 90 clear the screen to yellow. Lines 100 to 120 produce the sky by setting a window and clearing it to blue. Lines 140 and 150 draw the pyramids by calling the subroutine from line 1000. This uses a FOR . . . NEXT loop to fill the triangular shapes by drawing lines close together.

There is no BASIC statement to set the graphics colour mode. Instead it is done by sending control codes to the vdu driver. This is done in line 180. CHR$(23) is the code to set

the colour mode, and this is directly followed by the mode required. You must use either a semicolon, or the + (concatenation) sign between the two characters. Any other print separator (such as a comma) **will not work**. Line 180 resets the mode at the end of the program.

The tank is moved across the screen by the subroutine from line 2000. The principle is similar to the way the bombs were erased in the last program, with variables to remember the old position, but in this case, only the x co-ordinate changes, so we have x and ox, but no oy, and the erasure is by reprinting the character rather than overprinting a space. Line 2050 actually does the overprinting, line 2070 the printing. Line 2020 is to make the first erasure by line 2050 at a point off the screen. The calls to &BD19 should make a tangible improvement to the animation in this program. Line 2085 is simply short delay to control the speed of movement.

At the end of the program, line 190 waits for a keypress, then line 200 resets the mode to 1 and colours to defaults.

The tank, CHR$(254), is a redefined character (line 130). Figure 3 shows how the numbers in the SYMBOL statement are arrived at (the first, 254, being the code for the character to be redefined). Of course, it is not necessary to work out the numbers in decimal, as the Amstrad will accept binary numbers, indicated by preceding them with &X.

LISTING 20

```
20 MODE 0

30 INK 3,12

40 INK 9,0

50 INK 11,0

60 INK 1,3

70 INK 5,3
```

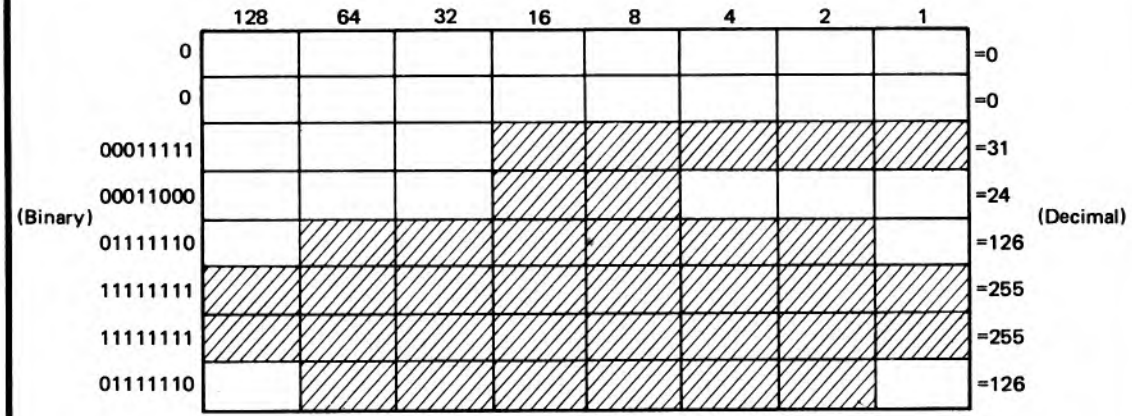

Fig. 3. The tank character (Mode 0 aspect ratio)

```
80 INK 13,3
90 PAPER 3:CLS
100 WINDOW #1,1,20,1,10
110 PAPER #1,6
120 CLS#1
130 SYMBOL 254,0,0,31,24,126,255,25
5,126
140 c=5:x=250:y=150:GOSUB 1000
150 c=1:x=450:y=200:GOSUB 1000
160 PRINT CHR$(23);CHR$(1);
170 GOSUB 2000
180 PRINT CHR$(23);CHR$(0);
190 WHILE INKEY$="":WEND
200 MODE 1:CALL &BBFF: PAPER 0
210 END
1000 REM draw Pyramids
1010 w=200
1020 FOR v=1 TO 100
```

```
1030 MOVE x-w/2,y+v
1040 DRAWR w,0,c
1050 w=w-2
1060 NEXT v
1070 RETURN
2000 TAG
2010 PLOTR 0,0,8
2020 y=212:ox=-64: REM tank start P
oint
2030 FOR x=0 TO 640 STEP 4
2040 MOVE ox,y
2045 CALL &BD19
2050 PRINT CHR$(254);
2060 MOVE x,y
2065 CALL &BD19
2070 PRINT CHR$(254);
2080 ox=x
2085 FOR d=1 TO 100:NEXT
```

```
2090 NEXT x

2100 TAGOFF

2110 RETURN
```

JOYSTICKS & TEST

When movement on the screen has to be controlled in 4 directions, joysticks are a much better bet than using the keyboard Listing 21 is a computer version of the game where you have to move a ring along a bent wire without the ring and wire touching. Things are somewhat the other way round, however, in that you have to draw a yellow line between two parallel red lines, without touching them. This program also illustrates the TEST function, which returns the ink value of a specified point on the screen.

The pattern taken by the parallel lines is set a random. As this game is most fun played competitively, and as different patterns can vary markedly in difficulty, the program has been written so that a pattern can be repeated any number of times.

The subroutine from line 1000 sets the co-ordinates for the parallel lines using RND (line 1020) and stores them in an array. The subroutine from line 2000 then draws the screen, outlining it in red (lines 2020–2050) as well as drawing the parallel lines (2060–2160). Line 2170 sets the start point for the yellow line.

The WHILE . . . WEND loop in lines 80–110 repeatedly calls the two routines starting at lines 3000 and 4000, until the yellow line reaches the right-hand side of the screen.

The routine from line 3000 does most of the real work. Lines 3020 and 3030 read the joystick, and amend the drawing co-ordinates px and py as appropriate. Diagonal movement is not permitted. Note that although most switch-type joysticks have the same type of plug, there is some difference of opinion between manufacturers as to which

way round joysticks should work. If you find your joysticks work in the wrong sense for this program, you can either change over the + and − signs in lines 3020 and 3030, or hold the joystick upside down!

Line 3040 checks the point which is about to be plotted to see if it is red (ink 3). If it is, px and py are reset to their previous values (see also line 3010), a one-second time penalty is incurred, and a buzz is produced. Otherwise the new point is plotted (line 3050).

The routine from line 4000 prints the elapsed time on the screen, using the function TIME. The variable t is set to the value of the internal clock in line 70. Line 4020 calculates the elapsed time in seconds by subtracting t from the current TIME value, and dividing by 300 (as the internal clock increments every 1/300 of a second). You can thus see how the one-second penalty for a contact in line 3040 is added, by subtracting 300 from the value of t.

Line 4020 also uses a PRINT USING specifier. This is a string of characters which controls how output is to be formatted on the screen. In this particular case, the time is printed with up to 3 digits ahead of the decimal point, and 2 after it. There are many PRINT specifiers, both for strings and for numbers, and the best way to learn how to use them is to experiment.

After completion of the yellow line, the routine from line 5000 asks if you want to repeat the screen. If you answer y, the program goes to line 60 to re-draw the screen.

If you answer n, you are offered the choice of a new screen. If you answer y, the program goes to line 50. Otherwise it terminates and the computer is reset to mode 1 and default colours.

An interesting feature of the Amstrad is that the joysticks actually put characters into the keyboard buffer when moved. These stored characters would interfere with lines 5030 and 6030, so these lines have been written so that they will only respond to the y and n keys.

LISTING 21

```
20 REM * BUZZLINES *
30 MODE 0
40 DIM y%(8)
50 GOSUB 1000:REM set Pattern
60 GOSUB 2000:REM draw screen
70 t=TIME
80 WHILE P%%<630
90 GOSUB 3000:REM move
100 GOSUB 4000:REM display time
110 WEND
120 GOSUB 5000:REM 'try again?'
130 IF LOWER$(ans$)="y" THEN GOTO 6
0
140 GOSUB 6000:REM 'new screen?'
150 IF LOWER$(ans$)="y" THEN GOTO 5
0
160 MODE 1:CALL &BBFF:PAPER 0:PEN 1
```

```
170 END
1000 REM screen co-ords
1010 FOR Points=0 TO 8
1020 y%(Points)=RND*200+100
1030 NEXT Points
1040 RETURN
2000 REM draw screen
2005 MOVE 0,0
2010 PAPER 8:BORDER 3:CLS
2020 DRAWR 0,399,3
2030 DRAWR 639,0
2040 DRAWR 0,-399
2050 DRAWR -639,0
2060 MOVE 0,y%(0)+20
2065 Points=0
2070 FOR x%=79 TO 639 STEP 80
2080 DRAW x%,y%(Points)+20
2090 Points=Points+1
```

```
2100 NEXT x%
2110 Points=0
2120 MOVE 0,Y%(0)-20
2130 FOR x%=79 TO 639 STEP 80
2140 DRAW x%,y%(Points)-20
2150 Points=Points+1
2160 NEXT x%
2170 Px%=10:Py%=y%(0)
2180 RETURN
3000 REM movement routine
3010 a%=Px%:d%=Py%
3020 Px%=Px%+4*(JOY(0)=8)-4*(JOY(0)
=4)
3030 Py%=Py%+2*(JOY(0)=1)-2*(JOY(0)
=2)
3040 IF TEST(Px%,Py%)=3 THEN Px%=a%
:Py%=d%:t=t-300:SOUND 1,1000,20
3050 PLOT Px%,Py%,1
```

```
3060 RETURN
4000 REM Print elapsed time
4010 LOCATE 2,2
4020 PRINT USING "###.##";(TIME-t)/
300
4030 RETURN
5000 REM 'try again' routine
5010 LOCATE 2,4
5020 PRINT "Try Again? (y/n)"
5030 ans$=LOWER$(INKEY$):IF ans$<>"
y" AND ans$<>"n" THEN 5030
5040 LOCATE 2,4
5050 PRINT SPACE$(16)
5060 RETURN
6000 REM 'new screen' routine
6010 LOCATE 2,4
6020 PRINT "New Screen? (y/n)"
6030 ans$=LOWER$(INKEY$):IF ans$<>"
```

```
y" AND ans$<>"n" THEN 6030
6040 LOCATE 2,4
6050 PRINT SPACE$(16)
6060 RETURN
```

Chapter 8

BINARY & HEX

A great deal of programming can be undertaken without having an understanding of the way in which a computer operates, or even having to understand a few basic principles, especially when using a high level language such as the Amstrad CPC464's Locomotive BASIC. However, even when using a sophisticated high level language there are still some aspects of programming which can only be fully understood if some of the more important fundamentals of computer operation are understood. The binary numbering system is perhaps the best example of this. Computers do not operate directly in our ordinary numbering system, but do everything in binary of one form or another.

The numbering system we normally use is commonly called the decimal system and is, of course, based on the number 10. There are ten single digit numbers from 0 to 9. This system of numbering is not very convenient for an electronic circuit in that it is difficult to devise a practical system where an output has ten different voltage levels so that any single digit decimal number can be represented. It is much easier to use simple flip/flops which have just two output levels, and can only represent 0 or 1. However, this bars such circuits from operating directly in the decimal numbering system. Instead, the binary system of numbering has to be utilised.

This system is based on the number 2 rather than 10, and the highest single digit number is 1 rather than 9. If we take a decimal number such as 238, the 8 represents eight units (10 to the power of 0), the 3 represents three tens (10 to the power of 1), and the 2 represents two hundreds (10 to the power of 2 or 10 squared). Things are similar with a binary number such as 1101. Working from right to left again, the 1 represents the number of units (2 to the power of 0, the 0 represents the number of twos (2 to the power of 1), the next 1 represents the number of fours (2 to the power of 2), and the final 1 represents the number of eights (2 to the power

of 3). 1101 in binary is therefore equivalent to 13 in decimal (1 + 0 + 4 + 8 = 13).

The table given below shows the number represented by each digit of a 16 bit number when it is set high. Of course, a digit always represents zero when it is set low.

Bit	0	1	2	3	4	5	6	7	8
	1	2	4	8	16	32	64	128	256

Bit	9	10	11	12	13	14	15
	512	1024	2048	4096	8192	16384	32768

A binary digit is normally termed a bit, and a group of 8 bits are called a byte.

Using 16 bits any integer from 0 to 65535 can be represented in binary fashion, or using 8 bits any integer from 0 to 255 can be represented, and this exposes the main weakness of the binary numbering system. Numbers of modest magnitude are many binary digits in length, but despite this drawback the ease with which electronic circuits can handle binary numbers makes this system the only practical one at the present time.

Addition of two binary numbers is a straightforward business which is really more simple than decimal addition. A simple example is shown below:–

First number	11110000
Second number	01010101
Answer	101000101

As with decimal addition, start with the units column and gradually work towards the final column on the left. In this case there is 1 and 0 in the units column, giving a total of 1 in the units column of the answer. In the next column two 0s give 0 in the answer, and the next two columns are equally straightforward. In the fifth one there are two 1s to be added,

giving a total of 2. Of course, in binary the figure 2 does not exist, and this should really be thought of as 10 (one 2 and no units), and it is treated in the same way as ten in decimal addition. The 0 is placed in the answer and the 1 is carried forward to the next column of figures. The sixth column again gives a total of 10, and again the 0 is placed in the answer and the 1 is carried forward. In the seventh column this gives a total of 3 in decimal, but in this binary calculation it must be thought of as the binary number 11 (one 2 and one unit). Therefore, 1 is placed in the answer and 1 is carried forward. In the eighth column this gives an answer of 10, and as there are no further columns to be added, both digits are placed in the answer.

A detailed explanation of how a computer handles complex calculations in binary would be out of place here, and provided you understand how numbers can be represented by an electronic circuit using this direct binary system you should be able to use and understand the functions of the CPC464 that require a knowledge of binary.

BIN$

A useful feature of the CPC464 which is not often found in home computers is its BIN$ function. In the basic form of this function the specified (decimal) number is converted into its binary equivalent. For instance:-

```
PRINT BIN$(8)
```

would return a value of 1000.

There is a slightly more complex version of this command where the first number is the one to be converted to binary, and the second number specifies the number of digits. For example:-

```
PRINT BIN$(8,12)
```

would print on the screen

000000001000

In other words it gives a conversion of 8 (in decimal) to the equivalent binary number, giving an answer with 12 digits and the leading zeros not suppressed. Note that specifying fewer digits than the converted number requires will not result in it being shortened. Thus the command:-

```
PRINT BIN$(8,2)
```

would print on the screen

1000

and not

00

The CPC464 will accept binary numbers, but these must be preceded by &x to indicate to the computer that the number is in binary and not decimal. The computer can therefore give binary to decimal conversion. For instance, the command:-

```
PRINT &x1000
```

would print on the screen 8.

The largest number that can be accommodate is 65535, or 1111111111111111 (i.e. 16 digits) in binary. Negative numbers can be accommodated, but the way in which these are handled in the binary system goes beyond the scope of this book ["An Introduction To Z80 Machine Code" (BP152) by the same publisher and authors as this book gives more detail on this topic, and covers machine code programming on the CPC464].

HEXADECIMAL

While on the subject of numbering systems it would perhaps be worthwhile dealing with another system which you will inevitably come across quite frequently, and this is the hexadecimal system. A problem with binary numbers is that they tend to have many digits with each digit being either 0 or 1, which makes them rather difficult to deal with in many circumstances. On the other hand, binary numbers give a graphic representation of the state of each bit in the registers of the microprocessor, and this is something that is often important. Decimal numbers are easier to use in that they are much shorter and are in a familiar form. However, converting a decimal number into an equivalent binary one is not a very quick and easy process, especially where large numbers are concerned, and this is inconvenient when it is necessary to visualise things on a bit by bit basis.

The hexadecimal system gives the best of both worlds in that it requires just a few digits to represent fairly large numbers, and is in fact slightly better than the decimal system in this respect. On the other hand, it is easy to convert hexadecimal to binary, and it is easy to use when operating at bit level. The hexadecimal system is based on the number 16, and there are sixteen single digit numbers. Obviously the numbers we normally use in the decimal system are inadequate for hexadecimal as there are six too few of them, but this problem is overcome by augmenting them with the first six letters of the alphabet. It is from this that the system derives its name. The table following helps to explain the way in which the hexadecimal system operates.

Decimal	*Hexadecimal*	*Binary*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	00010000
17	11	00010001
163	A3	10100011

What makes hexadecimal so convenient is the way in which multidigit numbers can be so easily converted into binary form. The reason for this is that each hexadecimal digit represents four binary bits. Take the hexadecimal A3 in the above table for instance. The digit A represents 1010 in binary, and the digit 3 converts to 0011. A3 therefore represents 10100011 in binary. You may find that you can memorise the four bit binary number represented by each of the sixteen hexadecimal digits, but a little mental arithmetic is all that is needed to make the conversion if you can not.

The digits in a hexadecimal number represent, working

from right to left, the number of units, 16s, 256s, and 4096s. You are unlikely to use hexadecimal numbers of more than four digits in length when using the CPC464, and often you will deal with hexadecimal numbers only two digits long.

The CPC464 can provide decimal to hexadecimal conversion using the HEX$ function. As a simple demonstration try typing:-

```
PRINT HEX$(16)
```

into the computer. This should return a value of 10 (one 16 and no units).

The CPC464 will accept hexadecimal numbers, but they must be preceded by either & or &H to indicate to the computer that they are in hexadecimal. This facility can be used to give hexadecimal to decimal conversion. For example, typing:-

```
PRINT &FF
```

into the computer should return a value of 255.

LOGIC OPERATIONS

The BASIC words AND and OR can be used almost as if they are plain English words, as we have seen in a previous chapter. Exclusive or (XOR) has also been briefly covered in an earlier chapter when it was used as a bitwise logic operator. In fact all three of these logic operators can be used in the bitwise mode, and as this is an important topic we will consider bitwise operation in some detail here.

In this mode two (usually) 8 bit binary numbers are compared bit by bit to produce an 8 bit binary answer. If you enter numbers into the computer in decimal form when performing bitwise operations this can produce what at first sight seem to be rather nonsensical answers. Try typing the following command into the CPC464:-

```
PRINT 15 AND 245
```

This should return a value of 5, and not 260 as one might expect. If we convert the three numbers into binary this gives the following result:-

15	00001111
245	11110101
5	00000101

This may not seem any less confusing, but what actually happens here is that the computer compares the two numbers bit by bit, and a 1 is placed in the answer only if there is a 1 in that bit of both numbers. There is a 1 in the least significant bit of both numbers, and a 1 appears in the least significant bit of the answer. The situation is different for the most significant bit as a 1 only appears here in one of the numbers, and a 0 therefore appears in this bit of the answer.

This may seem to be of only academic interest, but there are practical applications for this type of logic operation. The AND function is particularly useful for use as a mask when only certain bits of a number are of interest, and the others must be effectively eliminated in some way. The way in which this is done is to use a masking number which has a 1 in each bit that is of interest, but a 0 in any bits which must be masked. To read (say) the two least significant bits the masking number would be 3, since these bits (when set to 1) represent 1 and 2 respectively, giving a total of 3. The following example shows how this masking operates:-

85	01010101
3	00000011
1 (answer)	00000001

All that is happening here is that a 0 in a bit of the masking number ensures that this bit cannot be at 1 in both of the

ANDed numbers, and that particular bit therefore has to be 0 in the answer. If a bit of the masking number is set at 1, that bit of the answer will be set at 0 or 1, depending on which of these occurs in the other number, and this bit is not masked.

The OR function operates in a similar way, but a 1 appears in a bit of the answer if there is a 1 in that bit of the first number or the second (or both). Thus, if 85 and 3 are ORed the following answer is obtained:-

85	01010101
3	00000011
87 (answer)	01010111

The XOR function again operates in a similar fashion, but a 1 appears in the answer only if there is a 1 in that bit of one or other of the numbers, but not if a 1 appears in that bit of both numbers. EORing 85 and 3 gives this result:-

85	01010101
3	00000011
86 (answer)	01010110

Chapter 9

INTERFACING

Many of the interfacing problems that occur with other home computers are avoided by the system approach of the CPC464, which has a built-in cassette recorder and is supplied with either a monochrome or a colour monitor (together with the necessary connecting leads). Provided Amstrad accessories such as joysticks are used with the machine, these will plug straight into the appropriate socket without any problems. Of course, the use of a built-in cassette recorder does not totally avoid the problems associated with this medium of program storage, and it is still necessary to use a good quality ferric tape and to keep the tape heads clean in order to obtain reliable results. The use of the standard (1000 baud) rate rather than the fast (2000 baud) rate also helps to give optimum reliability.

PRINTER

An attractive feature of the CPC464 is its built-in parallel (Centronics type) printer interface which enables it to be used with a wide range of printers from inexpensive dot matrix types to high quality daisy wheel machines suitable for word processing. If possible I would always recommend the use of a ready-made printer lead, even if you are familiar with electronics and soldering, as this can save a great deal of time and frustration. However, if a suitable lead proves to be difficult to get or you prefer to make your own, it is simply a matter of using a piece of 23 way ribbon cable about 1 metre long (and definitely no more than two metres in length) to connect a suitable edge connector to a Centronics 36 way plug. Each pin of the edge connector connects to the pin of the same number on the Centronics plug. The diagram in Appendix V of the CPC464 manual gives the pin numbering for the printer port, and Centronics plugs are conveniently

marked with pin numbers. Note that several of the pins do not need to be connected (only those listed in Appendix V of the manual). Also, note that there is no Acknowledge Handshake line implemented on the printer port. The way in which handshaking operates is that a negative pulse is produced on the Strobe line when a fresh character is available on the eight data lines (D0 to D7), to indicate to the printer that new data is available. The printer then indicates on the Acknowledge or Busy handshake line when it has finished processing the data and is ready to receive another character. Only the Acknowledge line or the Busy line need to be implemented, and not both. The lack of an Acknowledge input is therefore not important provided the user is aware of its absence and uses the Busy line instead.

A slight problem that could arise when making up a printer lead is that of obtaining a suitable connector for the printer port. A 2 by 17 way 0.1 inch pitch female edge connector is required, but a connector having the right number of ways might not be obtainable. A simple solution is to buy a larger type and carefully cut it down to size using a hacksaw.

One final point is that although eight data outputs are provided, D7 is in fact just connected to ground. This is quite acceptable for ASCII codes which only use numbers up to 127, and do not require D7. However, some printers can use control codes which require numbers in the range 128 to 255, and these can not be accommodated by the printer port.

PRINTING

Programs can be listed on a printer using the normal LIST command, but with a channel number of 8 rather than the usual default channel (which prints on the screen). Thus the command:-

```
LIST#8
```

would list the entire program to the printer. If (say) only lines 10 to 100 must be listed to the printer, then it is merely

necessary to specify this line number range in the command in the normal way (i.e. LIST 10–100,#8).

You can also PRINT data on the printer using the normal PRINT command with a channel number of 8 (i.e. PRINT#8,"this will be printed on the printer").

A useful command associated with the printer port that should not be overlooked is WIDTH. This sets the maximum number of characters per line (assuming a value less than the maximum number of characters per line of the printer is selected). For instance, the command:-

```
WIDTH 25
```

would limit the maximum line length to 25 characters. This command can be very useful, and it was used to print out the listings in this book to line length that would fit the page size properly.

DISC PORT

The floppy disc port although primarily intended for use with an Amstrad disc drive can act as a general purpose interface for other devices. In particular, it is an ideal port for user add-ons. For someone with suitable experience it should be quite easy to utilise this port, but it is only fair to point out that **it would be unwise for anyone without the necessary experience to dabble with this port.** Doing so could easily result in expensive damage to the computer. The information on interfacing the disc port that is provided below is only intended for those readers who possess the necessary technical experience.

The disc port provides the full data, address, and control buses, together with a 5 volt supply output (which seems to be able to supply at least 100 milliamps). Connection to this port is via a 2 by 25 way 0.1 inch pitch edge connector, and a suitable connector should be readily available from most of the larger electronic component retailers. The standard method of interfacing with a system based on a Z80 micro-

processor is to decode only the lower eight lines (A0 to A7) of the address bus to provide up to 256 input and output addresses. There are separate memory and input/output maps, with MEMRQ going low when a memory device is accessed, and IORQ going low when an input or an output device is accessed. In addition to the address bus, IORQ therefore has to be decoded as well in order to ensure that add-on circuits are not spuriously operated when a memory circuit having the same address is accessed.

In common with some other Z80 based home computers, the CPC464 does not strictly adhere to the standard Z80 method of interfacing. The system adopted in this case is to use the upper eight address lines (A8 to A15) to select the required input/output circuit, with the lower address lines being left free to provide additional addresses if an input/output circuit requires more than one address. The point of this system is that it enables considerably less than full address decoding to be utilized, and helps to simplify the hardware. The only drawback of the system is that by using the wrong address more than one input/output circuit at a time can be activated. This is not likely to cause any damage to the computer though, or even a crash of the system for that matter, but it could result in something like break-up or distortion of the display. Care must therefore be exercised when directly addressing any input/output device, whether it is an internal circuit or an external one. When using machine code only input and output instructions where the B register provides the upper eight bits of the address bus can be used, and block instructions that utilize the B register as a counter can not be used properly.

EXTERNAL CIRCUITS

The area of the input/output map that is free for user add-ons is from &F800 to &FAFF. However, full decoding of the address bus is not needed, and the basic system of address decoding suggested by Amstrad is to have external circuits activated when address line A10 goes low. If more than one

input/output address is required up to 256 addresses are available by decoding all or some of the eight least significant address lines with A10.

Figure 4 shows how a simple 8 bit (latching) output port

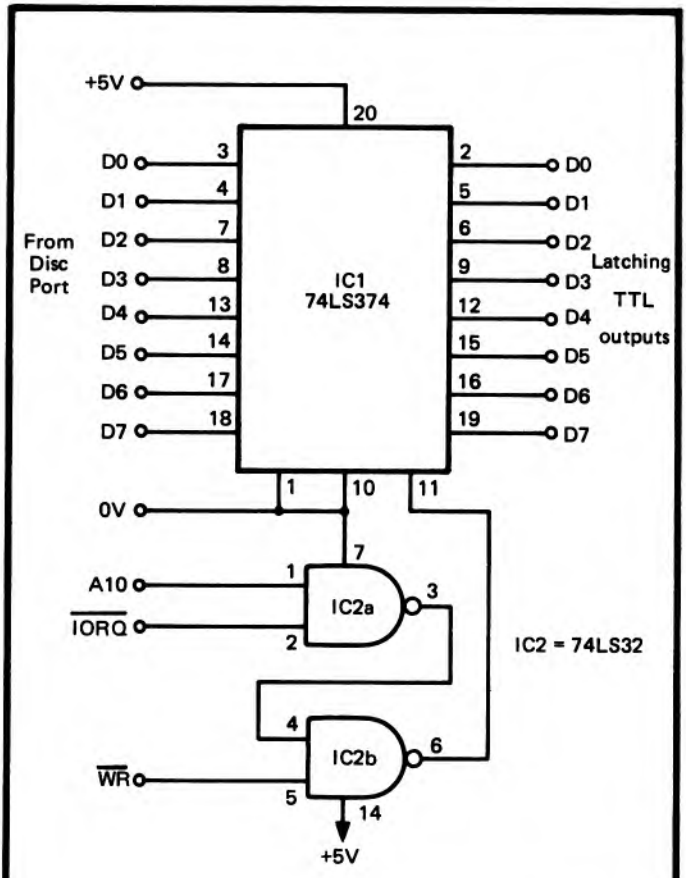

Fig. 4. The circuit diagram of an 8 bit output port

can be added to the disc port. IC1 is an octal D type flip/ flop with three-state outputs. In this application is it used as an 8 bit data latch with the address decoder providing a negative latching pulse to the clock pulse input at pin 11 of IC1 when data is written to the port. The address decoder is very simple, and it just comprises two 2 input OR gates from a 74LS32 device which are wired so that the negative latching pulse is provided when A10, IORQ, and WR (the write line) are all low.

Data can be written to the port from BASIC using the OUT instruction, and the port is at address &F800. In fact the use of only partial address decoding means that the port appears at numerous output addresses, but &F800 is a safe and convenient address to use in practice. As a simple example of writing data to the port:-

```
OUT &F800,255
```

would set all the output lines high.

Figure 5 shows an additional circuit that can be used to provide an 8 bit input port having a latching facility. IC2a of Figure 4 plus one of the previously unused gates of IC2 (IC2c) are used to decode the A10, IORQ, and RD (read) lines, so that the a negative enable pulse is supplied to IC4 when input/output address &F800 is read. The decoding of the WR and RD lines enables the input and output ports to share the same address without the two circuits interfering with one another in any way. IC4 is an octal transparent latch. With the Strobe input left unused it operates as a straightforward 8 bit input port. However, if the Strobe input is normally taken low, a positive pulse can be used to latch data into the port. This can be useful if the port is driven from a signal source such as an analogue to digital converter which is operating in the continuous conversion mode. The circuit can be arranged so that the converter latches the result of each new conversion into the port. Reading the port then takes the latest reading from the converter, and avoids the need for handshake lines.

The port can be read from BASIC using the INP function.

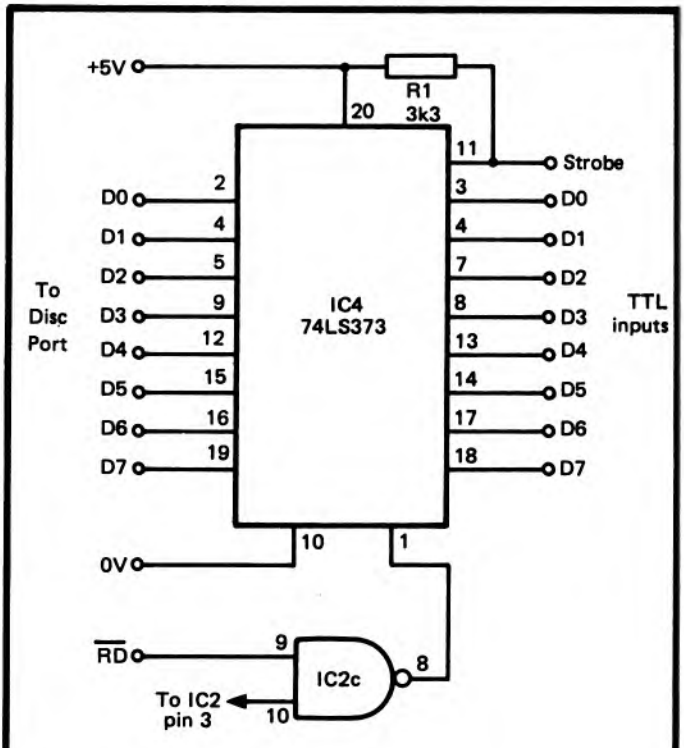

Fig. 5. Additional circuitry to provide an 8 bit input port

For instance:-

```
PRINT INP(&F800)
```

would read the port and print the returned value on the screen.

These circuits illustrate the basic method of interfacing to

the disc port, and the same general scheme of things could be used to interface other circuits to the port, including Z80A peripherals such as the Z80A PIO. As the CPC464 uses the high speed (4MHz) version of the Z80 (the Z80A) any Z80 peripherals used with the computer must also be the high speed A versions.

The disc port includes a Sound terminal which is an audio input. About 50 millivolts RMS is needed to fully drive this. Of course, there is also a stereo audio output available on the rear panel of the computer, and connections are made to this via a 3.5 millimetre stereo jack socket (the type that is used with personal stereo headphones). This output can be used to drive a suitable audio amplifier, but there is not an adequate output level to drive a pair of headphones. The light pen input is another useful facility of the disc port. The registers of the 6845 CRT controller, including the light pen registers, can be read by first writing the register number to output address &8C00 and then reading input/output address &8F00. The light pen registers are registers 16 and 17 incidentally. These do not directly provide horizontal and vertical screen positions, but combine to form a 16 bit number which must then be processed to provide the screen co-ordinates.

PEEK & POKE

We have covered the OUT instruction and the INP function which are used with input/output devices, and mention should be made of POKE and PEEK which are the equivalents for memory circuits. With many computers these are both very important, but the hardware and built-in software of the CPC464 are such that you are unlikely to use them very often, if at all. The CPC464 has 64k of RAM (random access memory) which covers all the Z80A microprocessor's address range of 0 to 65535. You can PEEK or POKE any address in this range, but note that POKEing some addresses could crash the computer (but you can not damage it using PEEK or POKE). Some addresses, in fact half of them, are shared by RAM (where programs and other data are stored) and ROM

(which contains the built-in software – the Locomotive BASIC and the computer's operating system). Only the RAM is accessible using PEEK and POKE.

OUTPUT PORT

When the printer port is not being used as such it is quite possible to use this as an 8 bit latching output. Data for this port is written to address &EF00. Bits 0 to 6 of this port connect to D0 to D6 respectively of the printer port connector. As explained earlier, D7 of this port is connected to ground, but bit 7 is available at the Strobe output. However, this bit is inverted, whereas the other seven bits are not. Obviously this bit could be re-inverted if necessary.

It is possible to read the Busy input, which is bit 6 of input address &F500. It is possible to read just one bit of an input port while masking off all the other bits, and this is achieved using the bitwise AND function which was described in the previous chapter of this book.

There is also a BASIC instruction which can be used to monitor one, or perhaps several bits of an input port, and this is the WAIT instruction. In its most simple form this merely monitors the specified bit or bits of the given address, and loops until the monitored bit (or one of the monitored bits) goes high. For example:-

```
WAIT &F800,3
```

would loop until bit 0 or bit 1 of the input port at address &F800 went high. What this instruction actually does is to bitwise AND the value returned from the specified port with the masking number given in the instruction, and the computer repeats this procedure until an answer of other than zero is obtained.

There is another form of this instruction where a third parameter (the inversion) is included. What happens here is that the returned value is bitwise ANDed with the masking number, and then exclusive ORed with the inversion. The

idea of this is that the program can be made to loop until a certain bit or certain bits go low rather than high since the exclusive ORing simply inverts the specified bit or bits. For example:-

```
WAIT &F800,3,3
```

would loop the program until either bit 0 or bit 1 of the input port at &F800 went low, rather than until one of these bits went high.

Chapter 10

INTERRUPTS

An unusual feature of Locomotive BASIC is its ability to generate interrupts, or at least a form of interrupts. Strictly speaking interrupts are generated by items of hardware in the computer which activate special interrupt inputs of the microprocessor. A typical application of interrupts would be to give a timer function. Here a hardware circuit would generate an interrupt at regular intervals, and typically about 100 times a second (300 times a second in the case of the CPC464). A software interrupt routine associated with the timing circuit increments the number stored in a series of memory locations by one each time an interrupt is generated by the timer circuit. Reading the number in these memory locations therefore gives a timer function, which could even be used as a real-time clock with suitable mathematical manipulation of the readings that are obtained. Most home computers use interrupts a great deal, including such things as scanning the keyboard and synchronising sound channels.

The point about interrupts is that they are not normally apparent to the user of the computer. Whether in the command mode or running a program, the interrupts normally continued to operate in the background, and the fact that they are continually interrupting any program that is running is not apparent due to the short time taken to complete each interrupt routine. When an interrupt is generated the microprocessor finishes the current instruction, carries out the interrupt routine, and then continues where it left off. The interrupts generated from BASIC in the CPC464 are analagous to real interrupts, but are different in that they are essentially software rather than hardware generated. However, they still have the ability to run in the background while a program is running. They can only operate while a program is running though, and cannot be used in the command mode.

AFTER

There are two instructions that can be used to generate interrupts: AFTER and EVERY. Taking AFTER first, this generates an interrupt after the specified time has elapsed. The way the system operates in practice is to have a subroutine associated with each AFTER instruction. After the specified time has elapsed the program breaks off from its normal procedure and goes to the subroutine. After this routine has been completed the program picks up where it left off. If required up to four AFTER instructions can be used at once, with each instruction using a different channel number (0, 1, 2, or 3). The default channel is zero. The short program given below helps to show the way in which the AFTER instruction operates.

```
10   AFTER 50,0 GOSUB 100
20   AFTER 100,1 GOSUB 200
30   AFTER 150,2 GOSUB 300
40   AFTER 200,3 GOSUB 400
50   PRINT "program looping"
60   GOTO 50
100  PRINT "Interrupt 0"
110  RETURN
200  PRINT "Interrupt 1"
210  RETURN
300  PRINT "Interrupt 2"
310  RETURN
400  PRINT "Interrupt 3"
410  RETURN
```

The first four lines of the program set up four AFTER interrupts, specifying the time to elapse before they interrupt, the interrupt channel, and subroutine to be called by each interrupt (in that order). The delay before the interrupt is called is in fiftieths of a second incidentally. Lines 50 and 60 simply loop the program indefinitely, printing program looping on the screen. The four subroutines are at lines 100 to 410, and these merely print "Interrupt" followed by the

number of the channel that generated the interrupt. If you run the program you should find that in amongst the numerous "program looping" messages that are printed on the screen, "Interrupt 0", "Interrupt 1", and so on appear after the appropriate time delays (1, 2, 3, and 4 seconds).

EVERY

The EVERY instruction differs from AFTER only in that rather than just once, the specified subroutine is called repeatedly. The delay given in the instruction sets the time before the first execution of the subroutine, and the period between subsequent executions, again in fiftieths of a second. This short listing demonstrates the EVERY instruction.

```
10   X = 0
20   EVERY 50 GOSUB 100
30   WINDOW 1,34,40,1,1
40   PRINT "program looping"
50   GOTO 40
100  X = X + 1
110  PRINT 1,X
120  RETURN
```

Line 10 starts variable X at a value of 0, while line 20 calls the subroutine at line 100 at one second intervals. Note that with both the AFTER and the EVERY instructions the subroutine must end with RETURN, like a normal subroutine called with a GOSUB. The subroutine simply prints a number in the WINDOW (see line 30) at the top right hand corner of the screen. The number starts at one and is incremented by one each time the subroutine is called. In other words it provides a simple seconds counter action. Lines 40 and 50 provide a continuous loop which repeatedly prints "program looping" on the screen.

If you run the program the seconds count should appear in the top right hand corner of the screen, but note that each number will only appear momentarily as no means of pre-

venting the screen from scrolling (and scrolling the numbers off the top of the screen) has been included in the program.

DI (disable interrupts) and EI (enable interrupts) are two instructions associated with the timed interrupts. As their names imply, they are respectively used to disable interrupts to prevent an unwanted disruption of a routine, and then to re-enable interrupts after the routine has been completed. REMAIN is a function associated with timed interrupts, and it prints the remaining count from the specified channel (e.g. REMAIN(2) would return the number of fiftieths of a second remaining before the channel 2 timer would time out and generate an EVERY or AFTER timed interrupt). An important point to realise about the REMAIN function is that it sets the specified timer channel to zero, and prevents the relevant subroutine from being called. As the EVERY and AFTER instructions use the same four timers it is not possible to simultaneously use EVERY and AFTER instructions that use the same channel number. The maximum delay number that can be used is 32767, and this represents a delay of nearly 11 minutes, which should be adequate for most practical applications.

Chapter 11

THE AMSTRAD CPC664

The Amstrad CPC664 is a new model introduced in the summer of 1985. It has much in common with the 464 model, and indeed all BASIC programs for the 464 (including those in this book) are completely compatible with the new model.

The main change in the 664 model is the inclusion of a single disc drive in place of the cassette recorder built into the 464. This disc drive uses the same 3 inch microdiscs as the add-on drive for the 464, and software on disc for the 464 can be used on the 664, and vice versa. This size of disc has not gained the popularity of the 3½ inch size, but is freely available.

As the 664 does not have a built-in cassette recorder, it is fitted with a cassette interface so that cassette programs for the 464 can be loaded. There is also an interface for a second disc drive.

The advantage of the disc drive is that it allows programs and data to be loaded and saved much more quickly than they can be on cassette. A disc can also store more than can be stored on a cassette. The disc filing system also allows much more versatile and flexible data file handling. Applications such as word processing and extensive data base use are not really feasible without a disc system.

The keyboard on the new model has also been tidied up, with a new and more tasteful colour scheme. The numeric keypad is now configured as function keys, though it can still be used for numeric entry. The cursor key cluster has been made larger, with styling rather like a MSX machine. The extra size of the disc drive has made the new model slightly wider than the old. Like the 464, the monitor supplies all power (5 volts for the computer and 12 volts for the disc drive) so only one mains plug is necessary.

As well as the addition of the disc drive, the version of Locomotive BASIC supplied with the 664 has been enhanced with the addition of some new commands. Most

of these are concerned with the graphics.

A command has been added to fill enclosed areas with colour. This is an improvement on the method of using windows to put blocks of colour on the screen, as the areas can be any shape. This command takes the form 'FILL n', where n specifies the colour to be used for the fill. The area is filled from the cursor position up and sideways, then down and sideways, until a pixel of non-background colour is found. This is essentially similar to the FILL and PAINT commands found in several other BASICs.

A new command is provided for plotting dotted or dashed lines. This command is 'MASK', and the pattern of the line is derived from the bit pattern in the binary form of the mask., For example, either 85 or 17Ø would give dotted lines, these numbers being Ø1Ø1Ø1Ø1 and 1Ø1Ø1Ø1Ø in binary. 15 is ØØØØ1111 in binary, and this would give a dashed line.

GRAPHICS PEN and GRAPHICS PAPER commands have been added for simpler colour control in graphics work, and you can also change colour by means of an added parameter to the MOVE command.

For improved smoothness in animation, there is now a FRAME command which causes the machine to wait for the frame flyback on the video display before updating the graphics display. This reduces flicker as characters move about the screen, and can also be used to improve scrolling and colour-change effects.

A further command allows a character to be read from the screen. This can be useful both in games and in text handling applications for screen editors.

The 'transparent' background colour, previously only available in text printing, can now also be used in graphics. This makes it simpler to place characters on the screen using TAG for animation. The EOR method described in Chapter 7 is now unnecessary in some instances, though it remains the most elegant way of animating a character over multi-coloured backgrounds, and remains the only way of moving a character in front of some colours and behind others, as in listing 20.

The line-drawing commands now include an extra parameter. This allows the last point in the line to be left unplotted, and is for use when drawing boxes using the EOR plotting option. Normally, this results in points at the corners being 'unplotted' because they are plotted twice. The new parameter avoids this.

Unfortunately, it will not be possible to upgrade the existing 464 model to use the new commands. Though all BASIC programs written on the 464 can be loaded and run on the 664, programs written on the 664 using the new commands cannot be run on the 464.

Some extra error handling commands have been added to cover disc operations, but this is beyond the scope of this book.

A disc containing CP/M and DR LOGO is included with the computer. It is hoped that some CP/M business software suitable for running on the Amstrad CPC664 will become available in the near future.

Notes

Notes

ALSO OF INTEREST

BP152: AN INTRODUCTION TO Z80 MACHINE CODE

R.A. & J.W. Penfold

Home computers are equipped with built-in software that enables them to be easily programmed to do quite complex tasks. The price that is paid for this programming ease is a relatively slow running speed, far lower than the speed at which the computer is really capable of running. Machine code programming entails direct programming of the microprocessor without using a built-in high level computer language such as BASIC. This gives a vast increase in running speed, but is something that can only really be undertaken by someone who has a reasonable understanding of the microprocessor and some of the other hardware in the computer.

Machine code programming is not as difficult as one might think, and once a few simple concepts have been grasped it is actually quite straightforward (although admittedly never as quick and easy as using a high level language). This book takes the reader through the basics of microprocessors and machine code programming, and no previous knowledge of these is assumed. The microprocessor dealt with here is the Z80 which is not one of the most simple types, but is generally acknowledged as one of the most powerful 8 bit devices, and is by no means excessively difficult for beginners. The Z80, or in most cases now the faster version the Z80A, are used in many home computers, including several of the most popular machines such as the Sinclair ZX81 and ZX Spectrum, plus the Memotech MTX500 and MTX512 machines, and the Amstrad CPC464. A few simple demonstration programs that can be run on these computers are included in this book.

112 pages *1984*
0 85934 127 5 **£1.95**

Please note following is a list of other titles that are available in our range of Radio, Electronics and Computer Books.

These should be available from all good Booksellers, Radio Component Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the publisher enclosing payment to cover the cost of the book plus adequate postage.

If you would like a complete catalogue of our entire range of Radio, Electronics and Computer Books then please send a Stamped Addressed Envelope to:

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

160	Coil Design and Construction Manual	£1.95
202	Handbook of Integrated Circuits (IC's) Equivalents & Substitutes	£1.95
205	First Book of Hi-Fi Loudspeaker Enclosures	£0.95
208	Practical Stereo and Quadrophony Handbook	£0.75
214	Audio Enthusiasts Handbook	£0.85
219	Solid State Novelty Projects	£0.85
220	Build Your Own Solid State Hi-Fi and Audio Accessories	£0.85
221	28 Tested Transistor Projects	£1.25
222	Solid State Short Wave Receivers for Beginners	£1.95
223	50 Projects Using IC CA3130	£1.25
224	50 CMOS IC Projects	£1.35
225	A Practical Introduction to Digital IC's	£1.75
226	How to Build Advanced Short Wave Receivers	£1.95
227	Beginners Guide to Building Electronic Projects	£1.95
228	Essential Theory for the Electronics Hobbyist	£1.95
RCC	Resistor Colour Code Disc	£0.20
BP1	First Book of Transistor Equivalents and Substitutes	£1.50
BP2	Handbook of Radio, TV, Ind & Transmitting Tube & Valve Equivalents	£0.60
BP6	Engineers and Machinists Reference Tables	£0.75
BP7	Radio and Electronic Colour Codes and Data Chart	£0.40
BP14	Second Book of Transistor Equivalents & Substitutes	£1.75
BP24	52 Projects Using IC741	£1.75
BP27	Chart of Radio, Electronic, Semi-conductor and Logic Symbols	£0.50
BP28	Resistor Selection Handbook	£0.60
BP29	Major Solid State Audio Hi-Fi Construction Projects	£0.85
BP32	How to Build Your Own Metal and Treasure Locators	£1.95
BP33	Electronic Calculator Users Handbook	£1.50
BP34	Practical Repair and Renovation of Colour TVs	£1.95
BP36	50 Circuits Using Germanium, Silicon and Zener Diodes	£1.50
BP37	50 Projects Using Relays, SCR's and TRIACS	£1.95
BP39	50 (FET) Field Effect Transistor Projects	£1.75
BP42	50 Simple L.E.D. Circuits	£1.50
BP43	How to Make Walkie-Talkies	£1.95
BP44	IC 555 Projects	£1.95
BP45	Projects in Opto-Electronics	£1.95
BP47	Mobile Discotheque Handbook	£1.35
BP48	Electronic Projects for Beginners	£1.95
BP49	Popular Electronic Projects	£1.95
BP50	IC LM3900 Projects	£1.35
BP51	Electronic Music and Creative Tape Recording	£1.95
BP52	Long Distance Television Reception (TV-DX) for the Enthusiast	£1.95
BP53	Practical Electronics Calculations and Formulae	£2.95
BP54	Your Electronic Calculator and Your Money	£1.35
BP55	Radio Stations Guide	£1.75
BP56	Electronic Security Devices	£1.95
BP57	How to Build Your Own Solid State Oscilloscope	£1.95
BP58	50 Circuits Using 7400 Series IC's	£1.75
BP59	Second Book of CMOS IC Projects	£1.95
BP60	Practical Construction of Pre-amps, Tone Controls, Filters & Attenuators	£1.95
BP61	Beginners Guide To Digital Techniques	£0.95
BP62	Elements of Electronics – Book 1	£2.95
BP63	Elements of Electronics – Book 2	£2.25
BP64	Elements of Electronics – Book 3	£2.25
BP65	Single IC Projects	£1.50
BP66	Beginners Guide to Microprocessors and Computing	£1.95
BP67	Counter, Driver and Numeral Display Projects	£1.75
BP68	Choosing and Using Your Hi-Fi	£1.65
BP69	Electronic Games	£1.75
BP70	Transistor Radio Fault-Finding Chart	£0.50
BP71	Electronic Household Projects	£1.75
BP72	A Microprocessor Primer	£1.75
BP73	Remote Control Projects	£1.95
BP74	Electronic Music Projects	£1.75
BP75	Electronic Test Equipment Construction	£1.75
BP76	Power Supply Projects	£1.95
BP77	Elements of Electronics – Book 4	£2.95

BP78	Practical Computer Experiments	£1.75
BP79	Radio Control for Beginners	£1.75
BP80	Popular Electronic Circuits – Book 1	£1.95
BP81	Electronic Synthesiser Projects	£1.75
BP82	Electronic Projects Using Solar Cells	£1.95
BP83	VMOS Projects	£1.95
BP84	Digital IC Projects	£1.95
BP85	International Transistor Equivalents Guide	£2.95
BP86	An Introduction to BASIC Programming Techniques	£1.95
BP87	Simple L.E.D. Circuits – Book 2	£1.35
BP88	How to Use Op-Amps	£2.25
BP89	Elements of Electronics – Book 5	£2.95
BP90	Audio Projects	£1.95
BP91	An Introduction to Radio DXing	£1.95
BP92	Easy Electronics – Crystal Set Construction	£1.75
BP93	Electronic Timer Projects	£1.95
BP94	Electronic Projects for Cars and Boats	£1.95
BP95	Model Railway Projects	£1.95
BP96	C.B. Projects	£1.95
BP97	IC Projects for Beginners	£1.95
EP98	Popular Electronic Circuits – Book 2	£2.25
BP99	Mini-Matrix Board Projects	£1.95
BP100	An Introduction to Video	£1.95
BP101	How to Identify Unmarked IC's	£0.65
BP102	The 6809 Companion	£1.95
BP103	Multi-Circuit Board Projects	£1.95
BP104	Electronic Science Projects	£2.25
BP105	Aerial Projects	£1.95
BP106	Modern Op-Amp Projects	£1.95
BP107	30 Solderless Breadboard Projects – Book	£2.25
BP108	International Diode Equivalents Guide	£2.25
BP109	The Art of Programming the 1K ZX81	£1.95
BP110	How to Get Your Electronic Projects Working	£1.95
BP111	Elements of Electronics – Book 6	£3.50
BP112	A Z-80 Workshop Manual	£2.75
BP113	30 Solderless Breadboard Projects – Book 2	£2.25
BP114	The Art of Programming the 16K ZX81	£2.50
BP115	The Pre-Computer Book	£1.95
BP116	Electronic Toys Games & Puzzles	£2.25
BP117	Practical Electronic Building Blocks – Book 1	£1.95
BP118	Practical Electronic Building Blocks – Book 2	£1.95
BP119	The Art of Programming the ZX Spectrum	£2.50
BP120	Audio Amplifier Fault-Finding Chart	£0.65
BP121	How to Design and Make your Own P.C.B.s	£1.95
BP122	Audio Amplifier Construction	£2.25
BP123	A Practical Introduction to Microprocessors	£2.25
BP124	Easy Add-on Projects for Spectrum ZX81 & Ace	£2.75
BP125	25 Simple Amateur Band Aerials	£1.95
BP126	BASIC & PASCAL in Parallel	£1.50
BP127	How to Design Electronic Projects	£2.25
BP128	20 Programs for the ZX Spectrum & 16K ZX81	£1.95
BP129	An Introduction to Programming the ORIC-1	£1.95
BP130	Micro Interfacing Circuits – Book 1	£2.25
BP131	Micro Interfacing Circuits – Book 2	£2.25
BP132	25 Simple Shortwave Broadcast Band Aerials	£1.95
BP133	An Introduction to Programming the Dragon 32	£1.95
BP134	Easy Add-on Projects for Commodore 64 & Vic-20	£2.50
BP135	The Secrets of the Commodore 64	£2.50
BP136	25 Simple Indoor and Window Aerials	£1.95
BP137	BASIC & FORTRAN in Parallel	£1.95
BP138	BASIC & FORTH in Parallel	£1.95
BP139	An Introduction to Programming the BBC Model B Micro	£2.50
BP140	Digital IC Equivalents and Pin Connections	£3.95
BP141	Linear IC Equivalents and Pin Connections	£3.95
BP142	An Introduction to Programming the Acorn Electron	£1.95
BP143	An Introduction to Programming the Atari 600XL and 800XL	£2.50
BP144	Further Practical Electronics Calculations and Formulae	£3.75
BP145	25 Simple Tropical and M.W. Band Aerials	£1.95

BERNARD BABANI BP153

An Introduction to Programming the Amstrad CPC464 and 664

■ The excellent hardware of the Amstrad CPC464 or 664 running with Locomotive BASIC go to make up an extremely potent and versatile machine and this book has been written to help the reader expand the potential of this powerful combination, with the minimum of difficulty.

■ The authors adopt a step-by-step approach starting with the fundamentals and then moving on to more advanced topics, with many example programs being included to illustrate and clarify points.

■ In a book of this size it is impossible to fully cover every aspect of machines as complex as the Amstrad CPC464 or 664, but the authors have tried, as far as possible, to complement the information supplied by the manufacturer rather than just duplicate it.

■ The text is divided into the following chapters: 1, Variables & Arrays; 2, String Variables; 3, Decisions; 4, INPUT, PRINT & DATA; 5, The Sound Generator; 6, Graphics 1 – Modes & Colours; 7, Graphics 2 – Animation; 8, Binary & Hex; 9, Interfacing; 10, Interrupts; 11, The Amstrad CPC664.

GB £ NET +002.50

ISBN 0-85934-128-3

£2.50

00250

9 780859 341288

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/