COMPUTER HANDBOOKS

Microcomputers

The Amstrad 464, 664 and 6128

Boris Allan

PITMAN

The Amstrad 464, 664 and 6128

Microcomputer Handbooks

The Amstrad 464, 664 and 6128

Boris Allan

Pitman

Computer Handbooks

The complete list of titles in this series and the Pitman Pocket Guide series appears after the Index at the end of this Handbook. The Publishers would welcome suggestions for further additions and improvements to the series.

Consultant Editor: David Hatter

PITMAN PUBLISHING LIMITED
128 Long Acre, London WC2E 9AN

Associated Companies
Pitman Publishing Pty Ltd, Melbourne
Pitman Publishing New Zealand Ltd, Wellington
Copp Clark Pitman, Toronto

First edition 1985

ISBN 0 273 02390 X

Printed in Great Britain at the Bath Press, Avon

Contents

Acknowledgements

I would like to thank Robin Shipp and Sally Tyler for their advice and assistance, and a great debt of thanks goes to David Hatter and Alfred Waller.

The Aims of this Handbook

This Handbook shows how the Amstrad CPC computers can be programmed in a rational manner. My intention is to produce a practical working guide to the CPC computers; not, however, by giving listings of many small examples, but by showing how to construct substantial systems. It is for this reason that detail is concentrated on two main applications: file handling (and statistics), and programming using turtle graphics.

The orientation is towards the promotion of effective programming strategies. Though the study of programming strategies uses Locomotive BASIC (using tape or disk), such strategies have relevance for programming in general, and for CP/M and LOGO in particular. As the cost of a CPC computer with a disk drive is so modest compared with most other systems, any CPC user should aim to be a disk user. Owners of the CPC 6128 will find all the elements of BASIC programming herein, apart from extensions to BASIC with bank switching.

The short accounts of CP/M and LOGO, herein, cannot replace Guides to these topics produced by Amsoft, and only give the outlines of these subjects. The short accounts of languages available on the CPC computers is orientated towards the CPC disk user. If you do not, as yet, have a disk drive you can use these sections to examine the potential of the disk system for the CPC computers.

Effective BASIC

The key to unlock effective programming is an understanding of the ways in which the programming language being used controls the operation of a program—the rationale of the language.

The means available for the control of Locomotive BASIC differ in major and minor respects from those available in other versions of BASIC, and thus it is for this reason that—as with any BASIC—we should start to investigate how such control is exercised in Locomotive BASIC.

There is a problem to be faced here, in that what I wish to say is common knowledge. The reason, however, I wish to re-examine such details is that even old knowledge can become new knowledge when examined from a rational perspective.

Here is an example of the simplest way possible to construct a continuous loop (a sequence of repeated instructions without end):

```
10  number = 1
20  PRINT number : number = number / 2
30  GOTO 20
```

This loop of actions is effective and (as the actions to be repeated are so simple) the result is fairly uncomplicated.

The variable *number* is set to the value *1* in line 10, then (in line 20) the value of the *number* is printed, and the *number* is halved. Finally, the program is directed to return to the actions beginning at line 20.

The outcome of the short program is that the sequence of *PRINT* and halving is repeated until we stop the process by hitting the [ESC] key twice (once to halt execution, twice to stop the restart of execution).

The compressed line 20 could be expanded over several lines, to read

```
10  number = 1
20  PRINT number
30  number = number / 2
40  GOTO 20
```

but either way the output is the same. The first few lines of output are:

```
run
1
0.5
0.25
0.125
0.0625
0.03125
0.015625
```

which shows how the size of the *number* is halved at each occasion.

Further through the sequence we find:

1.88079E-37
9.40395E-38
4.70198E-38
2.35099E-38
1.17549E-38
5.87747E-39
2.93874E-39
0
0

and from there onward there is a swift procession of zeros output on the screen.

This means, of course, that the CPC thinks that *2.93874E-39 / 2 = 0.* Like all other computers, the CPC is a limited machine, and this is one of them. Such a primitive loop mechanism is used because it is probably the first (and irrational) resort of the poor programmer.

Controlling Loops

You may find the use of the [ESC] key to stop the continuous loop crude and displeasing. Rationally, we want to be able to continue to divide by two, while the *number* is not zero: we use a *WHILE* loop.

The loop extends from *WHILE* up to the matching *WEND.*

```
10  number = 1
20  WHILE number > 0
30  PRINT number : number = number / 2
40  WEND
```

The output from this minor program starts as did the *GOTO* loop, and ends

1.17549E-38
5.87747E-39
2.93874E-39
Ready

which shows that once the number reaches zero, then the loop is terminated. The *WHILE WEND* is no more difficult to enter than the absurd *GOTO* loop, and certainly no longer.

A more common form of loop is that which extends from a *FOR* until a corresponding *NEXT*. The most useful situation for such a loop is in the situation where we know exactly how many times we wish to repeat the action: here is a *FOR* loop which can be stopped by use of the [ESC] key:

```
10  number = 1
20  FOR i = 0 TO 300
30  PRINT i, number : number = number / 2
40  NEXT i
```

We can stop the loop using *ESC* to find something of the form:

126 1.17549E-38
127 5.87747E-39
128 2.93874E-39
129 0
130 0

and again we reach the same conclusion, but—as with the use of *GOTO*—we have an uncontrolled irrational exit. At times we might want an uncontrolled exit, but it is better if these situations are at a bare minimum.

At the same time it would be useful if we could test after an action as well as before that action: as well as a test of the form 'WHILE this is true do that', it would be helpful to have a test of the form 'do that UNTIL this is true'.

The UNTIL loop construct can be successfully emulated in Locomotive BASIC by a modified form of the *FOR* loop. First another loop to be stopped by use of [ESC]:

```
10  number = 1
20  FOR i = 1 TO 2 STEP 0
30  PRINT i, number : number = number / 2
40  NEXT i
```

to produce, at the end of a long sequence,

```
1  5.87747E-39
1  2.93874E-39
1  0
1  0
```

which shows that the value of *i* never alters from the initial value *1*. The reason that the value of *1* never changes is that the increase in the value of *i* is set to zero, by *STEP 0*.

Purists might object to the use of a *FOR* loop for this purpose, but it is a technique well worth investigation. (A friend who is not a purist still takes objection ...) The next program shows one form this investigation might take.

We use this curious loop to produce an UNTIL construct:

```
10  number = 1
20  FOR i = 1 TO 2 STEP 0
30  PRINT i, number : number = number / 2
35  IF number = 0 THEN i = 2
40  NEXT i
50  PRINT ''**'';i,number
```

where the addition of line 50 is to indicate the end of the loop, and the key line is number 35.

The output from this program ends with

```
1 5.87747E-39
1 2.93874E-39
** 2        0
```

The explanation of the program is thus that the loop repeats UNTIL *number = 0,* at which point the value of *i* is increased to *2.* As *i* is now *2,* then the loop terminates because the upper limit to the loop (i.e. *TO 2*) has been reached.

Thus we can easily produce an UNTIL construct, which is cleaner than the alternative: where the alternative is a jump out of the loop (I have a great distrust of jumps, because of the confusion they can produce).

A further extension to this technique is to have checks at different points through the loop, so that they can become rather sophisticated—however, a well-designed *WHILE WEND* is usually superior, and rather clearer.

Subroutine Control

Instead of continually dividing by two, we will add one to the value of *number;* the reason for this change will soon become clear. We are going to examine the behaviour of subroutines in the organization of the flow of control in programs.

```
  10  number = 1
  20  GOSUB 1000
 999  END
1000  PRINT number : number = number + 1
1010  GOSUB 1000
1020  RETURN
```

This is a strange arrangement. Line 10 is normal, as is line 20—a call to a subroutine at line 1000—and line 999 indicates the end of the main program before we reach the subroutine.

The subroutine starts with line 1000, a line during which the current value of *number* is output, and its value is increased by one.

Line 1020 signals the end of the subroutine (a *RETURN* to the statement after the original calling statement), but between line 1000 and the end of the subroutine, at line 1020, there is a line (1010) which makes a call to the subroutine at line 1000.

Within the subroutine which starts at line 1000, there is a call to itself (known as a 'recursive' call). So what happens?

The start and end points (with two attempts to print the result of a simple sum) are

1
2
3
4
. . . . . [lots more output]
81
82
83
84
Memory full in 1000
Ready

Without performing any other action, immediately enter (CPC464 only)

? (1 + 1) * (1 + (2))
Memory full
Ready
clear
Ready
? (1 + 1) * (1 + (2))
6
Ready

A good deal has happened here.

First the *number* was steadily incremented by one, until it reached *85,* at which point something happened. We read the strange message 'Memory full in 1000'—and such a small program. We become even more confused when we try to print out the result of the simple sum *(1 + 1) * (1 + (2))*. The result is 'Memory full' for the 464.

Memory cannot be full, however, because the programs are far too small to occupy much more than 1K—much less than the complete 464 memory. What is full is the special set of memory locations called the 'stack'. The stack is the place in which certain types of information are stored temporarily during the execution of the program.

The *RETURN* statement implies that control returns to the statement immediately following the original statement which called the subroutine—BASIC has to remember from whence it came, using the stack to store the information.

After 84 calls to a subroutine, with no *RETURN* from any of the routines, there are 84 pieces of information on the stack to be used when a routine reaches a *RETURN.*

That is a good deal of information, which has to go somewhere, but only so much memory is set aside for the stack, thus memory is full—but only on the stack. On the 464 the evaluation of complicated arithmetical expressions (for example, expressions with parentheses) also uses memory on the subroutine stack, and thus there is no memory available to evaluate the *PRINT* request. The 664 and 6128 use a separate stack for calculations, and thus there is no error message.

The command *CLEAR* clears memory, and so the expression can be evaluated. It is worth experimenting with different types of expressions, to see what can be printed and what cannot, when the stack is full. I can reveal that it is always possible to *PRINT 2*.

The utility of recursive subroutines will become clearer when we examine the HILO example, but the investigation of 'nested' subroutines (subroutines which themselves call subroutines) is important also for more complex applications. If a *RETURN* is encountered, then the return address (the statement number) is 'popped' off the top of the stack, with the next item being the preceding return address.

As a complex application unfolds with many interrelated subroutines, the subroutines may eat up the stack (the appropriate statement numbers are 'pushed' on the stack). When the *RETURN*s are encountered, these statement numbers are popped off again.

If, eventually, there are more pushes than pops, then the stack becomes full. In the increasing of *number*, it was all pushes and no pops.

Although we have used the example of a recursive subroutine to illustrate the importance of the stack (and pushing and popping), the stack is central to any analysis of subroutines. For example, consider the simple program

```
 10  GOSUB 100
 99  END
100  GOSUB 200 : RETURN
200  RETURN
```

which does nothing. First (line 10) a call is made to subroutine 100, and thus the line number 10 is stored on the stack. At line 100 there is a call to a different subroutine (that at 200), and so the line number 100 is pushed on the stack (with an indicator that is the first statement on that line).

When line 200 is encountered, there is a *RETURN* and thus the top value is popped off the stack, which says line 100, first statement. Control returns to that line, and moves on to the next statement, which says *RETURN*. The value is popped off the stack, which says line 100, control moves to line 100, and the program continues to 99 and *END*s.

The HILO Program

Before we move to something slightly more practical, we can learn from a more trivial endeavour. In the study of this trivial endeavour I want to concentrate on the rationale behind the strategies by which we control the sequence of events through our program.

The trivial endeavour to be studied is the construction of a program to implement a number guessing game—one of the classic examples to show how the same problem can be approached from various viewpoints. In reality, however, no endeavour is truly trivial.

The game to be examined requires two players, is very simple (and very old, dating from centuries before computers). The computer chooses a *secret* number, and the human makes a *guess* about that *secret* number.

In the initial *attempts*, probably the *guess* will be wrong and so the human is told whether the *guess* is too big or too small (compared to the *secret*). The *attempts* are totalled, and when correct the number is given. In other words, the computer chooses a *secret* number at random, the number of *attempts* is set to one, and the human makes the first *guess*.

It is at this stage that—*WHILE* the *guess* is incorrect—the comparison is made, and the number of *attempts* is increased by one. We have a classic example of the true utility of a *WHILE* loop.

One other way of coding the program is to use the *GOTO* facility, but this method of (lack of) control is very messy. The unreasonableness and irrationality of *GOTO* is indicated by the short portion of code given below, which is taken from a published program.

There are so many *GOTO*s in this segment partly because there were no *WHILE* control constructs in early versions of BASIC on microcomputers. There are still many BASICs of recent vintage without a *WHILE WEND* or *REPEAT UNTIL* control construct. This might explain the convoluted (and irrational) programming we can find in many places.

```
230  IF A = Y THEN GOTO 300
240  IF A > Y THEN GOTO 270
250  PRINT "YOUR GUESS IS TOO LOW"
       : GOTO 280
270  PRINT "YOUR GUESS IS TOO HIGH"
280  IF B < 6 THEN GOTO 200
```

If anything is to be learnt from this portion of gibberish, it is that a full program, with all these unintelligible *GOTO*s, must be very confusing and confused.

The following program written with the *WHILE* construct is much neater, and is much easier to understand. The program is in a sense 'self documenting', for the structure of the program is such that the intention is clear without the need for a great number of *REM*s (or the short form ').

```
 10  ' hilo in BASIC : Version 1
 20  attempts = 1
 30  secret = 100 * RND(1) MOD 100
 40  INPUT guess
 50  WHILE guess <> secret
 60  attempts = attempts + 1
 70  IF guess < secret THEN PRINT "Too
     small" ELSE PRINT "Too big"
 80  INPUT guess
 90  WEND
100  PRINT "Success in "; attempts
110  END
```

The reason why this program is given as *Version 1* is that there is another neat way to code the program: we can use recursive subroutines.

If we return for a moment to the example in which *number* was halved, you will remember that we did not code that problem in a recursive form. The reason why this was so is to show how deeply we could nest the subroutines.

The halving can be coded recursively, according to a very simple idea. We start with the *number* equal to one, and then we call the halving subroutine. In the halving subroutine we print out the value of the *number,* and divide the *number* by two; we then call the halving subroutine . . .

This sounds as if it is something which could be treated rationally with great ease by a recursive methodology.

```
 10  number = 1
 20  GOSUB 100
 99  END
100  PRINT number : number = number / 2
110  GOSUB 100
120  RETURN
```

and when we execute the program we find

```
1
0.5
0.25
0.125
. . . . . [lots more output]
4.13590E-25
2.06795E-25
1.03398E-25
Memory full in 100
Ready
```

As subroutines can only be nested to a depth of about 84 calls, and it takes about 128 divisions to make one become zero, we cannot expect to reach zero by such a recursive method. Of course, if the size of the stack was increased, we would reach the desired limit, but we need to keep our sense of proportion because extra memory for the stack has to come from some other facility.

Another way of thinking about the HILO game follows the recursive style of the previous halving program:

to *begin* the computer chooses its *secret* number,
the number of *attempts* is set to one, and then the *game* is played;
the game consists of the human making a *guess,*
and finishing if the *guess* equals the *secret;*
otherwise, the relative size of the *guess* is given,
and the *game* is played again.

Here, therefore, is a HILO program which uses recursive subroutines:

```
  10  ' hilo in BASIC : Version 2
  20  GOSUB 1000 : ' begin subroutine
  30  GOSUB 1100 : ' game subroutine
  40  PRINT "Correct in "; attempts
 999  END
1000  ' begin
1010  attempts = 1
1020  secret = 100 * RND(1) MOD 100
1030  RETURN
```

```
1100  ' game
1110  INPUT guess
1120  IF guess = secret THEN RETURN
1130  IF guess < secret THEN PRINT "Too
        small" ELSE PRINT "Too big"
1140  attempts = attempts + 1
1150  GOSUB 1100
1160  RETURN
```

and it is worth comparing the program with that using *WHILE*. If anything, the *WHILE* loop is neater, but the recursive program has its good points. The use of subroutines, for example, makes the alteration of the action that much easier.

Alteration is easier because each component of action is localized to a subroutine, thus one modifies only that routine. However, with Locomotive BASIC it is impossible to 'hide' the workings of the subroutine from the rest of the program.

If by chance, say, one unknowingly modifies a variable, which is used in another routine or in the main program, there can be complications in that the workings of other subroutines may be affected unknowingly.

It is for reasons such as this that it is always a good idea to make the names of your variables as explicit as possible, to save possible contamination. For example, if you only use one or two letters for your variable names, and use the same name for different purposes, trouble is what you deserve.

Here is an actual sequence using the recursive version, though there is no apparent difference between the versions, as far as the user is concerned. Notice the strategy used by the intelligent guesser (me), and try to construct a BASIC program to emulate the action of such an intelligent guesser.

The optimum strategy on the part of the guesser is to continually halve the uncertainty, by a technique known as the 'binary chop'. At each point there is region of uncertainty, and so one chooses the centre of that region. Thus one always starts with the value 50 and then, depending on whether the *guess* is too big or too small, one chooses 75 or 25.

```
run
? 50
Too small
? 75
Too big
? 63
Too big
? 56
Too big
? 53
Correct in 5
Ready
```

If the human inputs are compared to the computer responses, and to the listings of the above programs, the flow of control is clear to see.

The key subroutine in the recursive program is (of course) the recursive subroutine which starts at line 1100.

The *guess* is *INPUT* and—if the *guess* is correct—a *RETURN* is made from the subroutine. The *RETURN* is to line 40 at the first call of the subroutine at 1100.

If the *guess* and *secret* are different, the relative size is *PRINT*ed, the number of *attempts* is increased by one, and call is made to the subroutine at line *RETURN* and control moves to line 1160 of the calling subroutine. Line 1160 contains a *RETURN* and, item by item, information about successive subroutines is popped from the stack until the bottom is reached.

The 'end' reached at the bottom of the stack is the statement after the *GOSUB* in line 30: thus control moves to line 40, and the information about the number of *attempts* is *PRINT*ed.

The recursive version of HILO in Locomotive BASIC is not as impressively clear as it is in Dr LOGO, and those with disks can try this set of procedures (to execute the program/game one simply types *hilo*):

```
? to hilo
>pr [HILO in LOGO—recursive version]
>(local "attempts "secret "guess)
>begin game
>end
```

```
? to begin
>make "attempts 0
>make "secret random 100
>end

? to game
>make "attempts :attempts + 1
>make "guess first rl
>if :guess = :secret [(pr [Success in]
    :attempts) stop]
>if :guess < :secret [pr [Too small]]
    [pr [Too big]]
>game
>end
```

Studying the LOGO procedures may help you to understand the BASIC program.

The CPC computers have a wide range of languages available under both the cassette and disk operating systems, although with CP/M or AMSDOS the range is much wider than that for the cassette filing system.

The first program is written in Amsoft/Abersoft FORTH, running under CP/M. The names of the variables and other words are kept to those we have used already (as much as possible). The FORTH program uses a *WHILE* loop construct (in the routine known as *HILO*).

```
( HILO IN FORTH )
0 VARIABLE GUESS
0 VARIABLE ATTEMPTS
0 VARIABLE SECRET
: BEGINPROC CR RANDOM SECRET !
  0 ATTEMPTS ! ;
: NUMIN QUERY INTERPRET GUESS ! CR ;
: CORRECT? SECRET @ GUESS @ = ;
: RESULT ." SUCCESS IN " ATTEMPTS @ .
  CR ;
: COMPARE SECRET @ GUESS @ > IF
  ." TOO SMALL " CR
  ELSE ." TOO BIG " CR ENDIF ;
: INCHECK 1 ATTEMPTS +!
  NUMIN CORRECT? NOT ;
: HILO BEGINPROC
  BEGIN INCHECK WHILE COMPARE
  REPEAT RESULT ;
```

This is probably the most difficult program to understand (though C has its moments).

Pascal is a very popular programming language, and the version used here is Amsoft/Hisoft Pascal (again running under CP/M, though there is a cassette version). Control is exercised by recursion.

```
 10  PROGRAM hilo ; (* hilo in Pascal *)
 20  VAR secret, guess, attempts : integer ;
 30  PROCEDURE beginproc ;
 40  BEGIN
 50   attempts := 0 ;
 60   secret := random(0) MOD 100
 70  END ;
 80  PROCEDURE game ;
 90  BEGIN
100   attempts := attempts + 1 ;
110   read(guess);
120   IF guess = secret
130    THEN writeln('Success in ', attempts)
140    ELSE BEGIN
150     IF guess < secret
160      THEN writeln('Too small')
170      ELSE writeln('Too big') ;
180     game
190    END
200 END ;
210 BEGIN
220  beginproc ; game
230 END.
```

The final program is written in Small-C-80 for CP/M. This version of the language conforms to that generally available under CP/M, and is not a special Amstrad version. Control is exercised by a *while* loop.

```
/* hilo in C */
# include ''stdio.h''
# include ''rnd.h''

main()

{

 int attempts, secret, guess ;

 attempts = 1 ;
 secret = random(100) ;
 scanf(''%d'', &guess) ;
 while (guess != secret)
 {
  if (guess < secret)
  printf(''\nToo small\n'') ;
  else printf(''\nToo big\n'') ;
  ++attempts ;
  scanf(''%d'', &guess) ;
 }
 printf(''Success in %d\''n'', attempts) ;
}
```

Sending Data to a File

Now to try to understand how control constructs can help us simplify the storing of information in a file (tape or disk). The aim is to construct a consistent set of rational routines to manipulate files on tape or disk—important because they form the basis of much advanced programming.

Obviously any extended use of files will benefit from the use of disks with AMSDOS and CP/M, but the tape system for CPC computers is surprisingly flexible, and remarkably consistent with the use of disk files.

We will start with programs to save information on files, information we can then read back for use at later occasions. The concepts are not difficult to grasp, and are simple to implement.

If you are writing information to a data file quite often you do not know when starting to input exactly how many different data items are to be stored. Often it is too great a problem to find (in advance) how many items have to be stored.

The implication of this is that a standard *FOR* loop is not suitable, because this requires that the number of items is known beforehand.

After some consideration, we realize we want to add information to the file *WHILE* we have not reached the end of input. The outline of the data input and saving program is thus:

```
100  ' Example program - file input
110  GOSUB 51000
       : ' Establish name of data file
120  GOSUB 51100 : ' Input routine
130  WHILE in$ <> "end" AND in$ <> "END"
       : ' Check for end of data
140  GOSUB 51200 : ' Add data to file
150  GOSUB 51100 : ' Input routine
160  WEND
170  CLOSEOUT
999  END
```

This technique of program design is my standard way of approaching the construction of any system of any real complexity—in that I design the program before I know what is in the subroutines.

With some languages, and some versions of languages, this technique is difficult to employ to its full power because when we use subroutines extensively we fill the stack far too quickly (the stack is too small). In the case of some languages we are not able to make recursive calls to subroutines without special trickery (for example, FORTH or FORTRAN IV).

I employ this technique because it works, particularly when a more complex system has to be defined. Others have found that this technique is the most effective and rational approach to programming, and has been given many names.

Perhaps the most apt description of the technique is to call it the 'top down' method, because we start at the top with the problem, go to the program, and then down to the individual constituents of the program (which possibly have their own subconstituents).

We proceed from the problem to the individual lines of code by a process of what is sometimes termed 'stepwise refinement'. We proceed in steps, where at each stage we refine the code, but we delay for as long as possible the actual coding of the program.

Using such techniques we do not enter the program into the computer until late in the day: there is no point, the above program (for file input) is only a set of subroutines. Given the ideas of stepwise refinement we can introduce a set of dummy lines such as

```
51000  RETURN
51100  RETURN
51200  RETURN
```

which have no action, but enable us to test the structure of the program.

We can (using a rational approach) convince ourselves that the program works, before we complicate matters by introducing real subroutines. If errors appear at a later stage, we know the errors are due to the recently introduced subroutine. In rational programming, coding the entire program is left as late as possible.

Though some versions of BASIC make the rational approach more difficult to employ (because of restricted stack space), in Locomotive BASIC we need not worry greatly about filling the stack. We can nest subroutines to a depth of 84 calls, which is enough for most purposes.

Although complicated arithmetic can use some stack memory, and thus reduce the possible number of calls, effectively there is no real problem with the BASIC on the CPC computers. As far as the nesting of subroutines is concerned, we can program without worries.

The input routine

The flexibility which comes from recursive subroutines is particularly apparent in the subroutine at line 51100—the *input routine*—which is the first subroutine we will examine.

```
51100  ' input routine
51110  INPUT "Data"; in$
51120  PRINT in$; " ..OK"; : INPUT r$
51130  IF LEFT$(r$,1) = "y" OR
       LEFT$(r$,1) = "Y" THEN RETURN
51140  PRINT "Revised "; : GOSUB 51100
51150  RETURN
```

After the heading (line 51100), the information to be read in (or *INPUT*) is requested, and is *INPUT* as a string variable, *in*$.

To see why this is a good idea, examine the following interaction.

```
INPUT a
? 1
Ready
PRINT a
 1
Ready
INPUT a
? 1,2
Redo from start
? 1
Ready
INPUT a$
? 1,2
Redo from start
? "1,2
Ready
PRINT a$
1,2
Ready
```

What has happened is that we have used the 'instant' form of *INPUT* to read in a value for the variable *a*, and—if there is only one number—all is well.

If we try then to read one value from a list of two (that is, *1,2*), we cannot: we are asked to *Redo from start*. If, however, we treat all values as words and input characters into a string, we have a solution.

The way in which we input to the string has to be adjusted to take into account that we want a 'picture' of the input saved in the string.

For example, if we *INPUT* into a string but elements of the string contain commas—recognized by BASIC as delimiters in *INPUT* lists—then we still have to *Redo from start*. We solve that small problem by prefixing the input data by a quote mark, so that commas are treated as part of the text string. This facility is illustrated by the final use of *INPUT a$*, and its corresponding *PRINT a$*.

What happens in the *input routine* is that the data is entered and stored in *in$*, the data is output for checking, and the user is asked to verify its correctness. If the first character of the response (*LEFT$(r$,1)*) is *y* or *Y* then all is well and a *RETURN* is made from the subroutine.

If some other character is first in the input string, then *Revised* is printed and the subroutine is called again. Note that *GOTO 51100* is not used because (though a *GOTO* may be acceptable in this case) it is far neater to have another call to the subroutine.

When control is *RETURN*ed the next line is *51150 RETURN*, and thus everything is strictly delimited, because a record of the places to *RETURN* are stored on the stack.

I have found that—in real programming—a conceptual neatness is very important, a neatness provided, for example, by a clean move of control. There is a certain pleasure to be derived from forcing unwieldy languages to perform sophisticated operations, but practical programming is a different matter.

A clean move of control can be accomplished by a *GOSUB* but only rarely by a *GOTO*; treat uncontrolled jumps with care. It is worth remembering that, though a few odd *GOTO* statements may appear to do no harm, if you get into a *GOTO* frame of mind then the design of a program is made more complicated, and frequently debugging is a pain.

Notice one important benefit of using subroutines: there are two calls to the subroutine, one before the *WHILE* (line 120) and one within the *WHILE WEND* loop (line 150). It would be very silly to have the same lines repeated at both points—at the very least it is more typing—and in addition this routine can be incorporated into other programs.

It is for this reason that the line numbers for the subroutine are so high, in that they will conflict only rarely with line numbers in ordinary programs.

Giving the file a name

The remaining subroutines are concerned with the adding of data to a file, and the subroutines will work without modification with either tape or disk. The equivalence between the two systems is due mainly to the method of implementing file handling on the CPC computers. Begin with the subroutine at line 51000:

```
51000  ' file name
51010  INPUT "File name"; file$
51020  OPENOUT file$
51030  RETURN
```

a routine which asks for a file name. It is true that this routine could be implemented simply as a line within the program, but to turn the request into a subroutine enables a self-contained system to be designed.

Depending on what you want, and on the system being used, the file name can take different forms. On a disk system the name has to be no longer than eight characters, plus an optional 'extension'. For example, a file name might be *TEST.DAT* where the extension is *DAT,* which is distinguished from the main name by the full stop—though usually the name is entered in lower case.

We can test the subroutine by

```
gosub 51000
File name? test.dat
Ready
closeout
Ready
```

and it is a good idea to use the extension notation with both tape and disk systems, because the nature of the file is then clarified. The name of the file is stored in *file*$, and in line 51020 we open an output file with the name stored in *file*$, by use of *OPENOUT file*$.

It is because a file is opened in the subroutine that we had to *closeout* after calling the subroutine at line 51000. With the tape system there is an extra facility (unnecessary with the disk system), due to the need to cope with messages from the tape system.

If, using the tape system, the file name is prefixed with a *!* then these messages are omitted. As long as the tape recorder is set to REC and PLAY there should be no problems. To get ahead of the discussion slightly, if the *!* is not used as a prefix, part of the way through the input while using the tape system you will find:

1 ..OK? y
Press REC and PLAY then any key:
Saving TEST.DAT block 1

and then a wait while the data is being stored on the tape, after the end of which there is another request for *Data?.* If the file name is prefixed with the *!,* then there is still a wait but there is no output on the screen to complicate matters.

Those with disk systems might like to try to use the tape system, which is activated by |*tape*—or |*tape.in* and |*tape.out* if you wish to distinguish between input and output.

The end of input is signalled by entering *end* or *END,* which is the check made in the *WHILE* statement at line 130. The end of the loop (*WEND*) is followed by *CLOSEOUT,* and this closes the file. Within the loop there is the input routine, and a routine to add the input data to the end of the file named by the *OPENOUT* command.

Adding data to a file

This is the subroutine at line 51200, which is defined as

```
51200  ' add data
51210  PRINT #9, in$
51220  RETURN
```

where the *#9* which immediately follows the *PRINT* indicates the 'stream' to which the data is to be sent.

Streams *#0* to *#7* are the eight possible text windows one can use with the CPC (of which, more anon), stream *#8* is the printer stream for a standard parallel printer (Centronics) interface, and stream *#9* is that for input and output to the tape system or disk system (depending on the configuration of your CPC). Line 51210, therefore, directs the output to the file identified by the *OPENOUT* command.

It is now time to see how all these elements tie together to produce the file creation program, but first of all we will *SAVE* our handiwork so far.

```
save ''!input''
Ready
cat
```

This saves a file with the main file name *INPUT* on either the tape or disk system. The ! character is stripped off in the case of both systems, with the result that for the tape system there are no messages. The form of the result for cat will differ according to the system.

In the case of the disk system, you will find there is a file (amongst others)

INPUT .BAS 1K

because the disk system (AMSDOS) adds a file extension *.BAS* to indicate that the file is that of a BASIC program. In the case of the tape system you will have to rewind, and also will be asked to

Press PLAY then any key:
INPUT block 1 $ Ok

plus any other files you have on that tape. The meaning of the tape *CAT* output is that the program extends up to 1 block on the tape, and it is ($) a standard BASIC program. The $ symbol is almost an equivalent of the AMSDOS extension *.BAS.*

Now to use the program.

The file creation program

Here is a sequence of operations which uses the file creation program to produce a small file of information to be known as *TEST.DAT.* The program in memory is that we saved as *!input* thus, depending on the system, it will be *INPUT* or *INPUT.BAS.*

```
run
File name? !test.dat
Data? 1
1 ..OK? y
Data? 2
2 ..OK? y
Data? 3
3 ..OK? y
Data? 45
45 ..OK? n
Revised Data?4
4 ..OK? y
Data? end
end ..OK? y
Ready
```

and, if a *CAT* is entered (see above for a previous use), the result for a tape system is that (among other files) there is the information

*TEST.DAT block 1 * Ok*

where the * indicates that *TEST.DAT* is an ASCII file.

When a file is classed as ASCII, this means that the information stored on that file is stored in such a form that each character is represented individually on the file.

The file contains a 'picture' of the information you have entered, and the only modification to it concerns the way in which the individual characters are stored. As the CPC performs such modifications automatically on sending and receiving information from a file, you need not worry about the modification process.

ASCII stands for the American Standard Code for Information Interchange, and is the most common way in which information is coded for use by computers. ASCII codes, for example, are used in the transmission of information between computers, as well as between computers and devices such as printers. Full lists of these code values are given in the Amsoft documentation (the codes are the basis for the modification of characters for storage by the CPC).

When the floating point number 1 is stored in memory in a computer, it is done so in a special binary form which will take up several bytes of memory.

In fact, the way in which the number 1 is stored in memory bears little obvious resemblance to its observable (or 'picture') form as one digit.

When numbers are stored in their floating point form, the memory occupied by the number 1 is the same as that occupied by the number 1111. If, however, the number is stored as the character '1', then the mode in which it is stored is exactly related to its observable form (the characters '1111' occupy four times as much room as the character '1').

Corresponding to the character '1' there is an internal code (the ASCII value) of 49, which is stored in a byte in memory (computer memory or disk memory). As a byte can contain 256 different values (that is, 0 to 255), there are a possible 256 codes. These internal codes are the means by which the CPC modifies characters for storage.

When the ASCII codes were first developed, the standard format was based around a 7-bit punched paper tape (which reveals its antiquity). The codes did not encompass 8-bit bytes so common on microcomputers, so that there were only 128 possible values produced by these 7 bits (that is, 0 to 127). The ASCII standard only defines the meanings for values 0..127.

The extra 128 values allowed by an 8-bit byte are used by most manufacturers for optional extras. The meanings of these values vary greatly from computer to computer (or printer to printer), because they are not standardized (that is, the values from 128 to 255).

There is provision within Locomotive BASIC, as with most BASICs, of a method to produce the ASCII values corresponding to individual characters, for example:

```
? asc(''1'')
49
Ready
```

and an ASCII file is one in which information is stored according to the exact ASCII equivalents of the characters which have been entered. The file is a picture image of the input, if you use a different set of spectacles to look at the disk file (and the CPC has such a pair of spectacles).

Those with access to the disk system can pursue this examination somewhat further, but first they have to exit BASIC and enter CP/M, by use of the instruction |*CP/M*. Once in CP/M you should check (by use of *dir test.dat*) that the *TEST.DAT* file is present, and then enter

```
A>type test.dat
1
2
3
4

A>
```

By use of the appropriate spectacles the picture image has been reproduced by the CPC. To show that not all files are picture images, try

```
A>type input.bas
```

at which strange things are likely to happen.

If *INPUT.BAS* is a picture image, the picture was painted by Kandinsky. Retrieve the situation by entering

```
A>amsdos
```

or—if the system is dead—[CTRL][SHIFT][ESC], and we can move to writing a program to read in data from a file.

Accepting Data from a File

Producing routines for reading information from a file are not as easily mechanized as writing information to a file. The organization of the transmission of information to a disk file is simpler than the organization of the recovery of that information.

For example, if I am inputting two values at a time, with the intention of sending both values to a file, such as the values *1,2,* then all that is needed for the input is *"1,2*—a picture image.

To read that information from the file, however, we need to specify two variables, and thus the reading of such information varies with the number of values per line of input.

The only standard subroutine is, in fact, that to open a file for reading. Here is one such subroutine:

```
52000  ' file name
52010  INPUT "File name"; file$
52020  OPENIN file$
52030  RETURN
```

Note that the *IN* and the *OUT* of the *OPENxxx* commands are with respect to the user's program. That is, you *OPEN* the file for your *OUT*put and your *IN*put. This is worth remembering—you are the important element, not the file.

The correspondence of this subroutine with the *file name* routine for the sending of information to a file should be noted.

If we only wish to print a copy of the contents of a file, there is a general routine we can use. This is such a useful program that you will find many applications in which such a utility can be valuable. The necessary components of such a system are:

```
  100  ' Example program—reading file
  110  GOSUB 52000 : ' file name
  120  WHILE NOT EOF
  130  GOSUB 52100 : ' read line
  140  WEND
  150  CLOSEIN
  999  END
52100  ' read line
52110  LINE INPUT #9, a$
52120  PRINT a$
52130  RETURN
```

which also uses the subroutine, given above, for setting up the appropriate file name (that is, the subroutine at line 52000). The program is easily converted into a subroutine.

Every file (tape or disk) has a marker to indicate the end of the file, because usually the end of the file is not the same as the space taken up by the file on disk. For example, on tape the minimum size of a file is 2K, and on disk the minimum size is 1K, yet the *TEST.DAT* file only occupies about 12 bytes.

You might suspect, however, as there are four numbers (each of one character) the information would occupy 4 bytes only. There is a little bit more to storing information: after each number there must be a carriage return (1 byte) and a line feed (1 byte), making 3 bytes for each number.

When you press [ENTER] at the end of a line, you instruct the system to perform two actions in one.

The first action is to move to a new line, and the second is to move to the beginning of that new line. The ASCII value for the action produced by carriage return (CR) is *13,* and that for the new line (LF) action is *10.*

The difference between the two actions can be illustrated by

```
print ''1''; ''2''
12
Ready
print chr$(10); ''1''; chr$(10); ''2''

1
 2
Ready
print chr$(13); ''1''; chr$(13); ''2''
2
Ready
```

(1) the first *PRINT* line simply outputs the two characters *1* and *2* one after the other;
(2) the second *PRINT* line moves to a new line before each character (with no return to the beginning of the line), thus the two characters are on different lines, but the *2* is not at the beginning of the line; and
(3) the third *PRINT* line has two CRs, and so the *2* overwrites the *1.*

The difference between the 12 bytes and the full space occupied by the file is effectively 'empty' space.

To establish where the real file ends and the empty space begins, the system inserts an end of file marker ([CTRL]Z—ASCII value 26) at the end of the information in the file, as the file is closed.

The system then knows that the rest of the file from the EOF (end of file) marker to the end of the physical file is to be ignored. It is this *EOF* marker which is tested in the *WHILE* statement at line 120.

We do not need to know how big the file is, because we continue *WHILE* the end of file marker has not been reached (*NOT EOF*). This is the normal way of reading information from a file in many systems.

Some systems (such as Xenix) do not use an EOF marker, and store the exact length of the file in a special location on the disk. AMSDOS, CP/M and the CPC tape system use the much simpler EOF method to delimit the end of a file.

In the subroutine at line 52100 (the *read line* subroutine) there is a command *LINE INPUT #9, a$* which inputs a line of information from the stream *#9*, that is, the file named in the *OPENIN file$* command.

BASIC recognizes the end of a line by the CR and LF pair, and this is the detection method used by *LINE INPUT*. The result of the *LINE INPUT* is placed in the string variable *a$*. The variable named in the *LINE INPUT* statement has to be a string variable, as we are copying ASCII picture images.

The content of a line of input is stored in *a$* and in line 52120 this content is *PRINT*ed. The result is that a 'picture' of an ASCII file is output, without there being any evaluation of the sense of the file—as far as the program is concerned, the file is merely full of characters.

Here is a specimen encounter using a file previously produced by the *INPUT.BAS* program:

```
run
File name? !test2.dat
1,2,3
4,5,6
7,8,9
10,11,12
Ready
```

and we are now in a position to study the different ways in which BASIC programs can be saved, with the implications there are for the construction of program libraries. We will extend the routines for reading files when we implement the statistics system.

Incidentally, this is one method of examining Dr LOGO procedure files from within BASIC. Dr LOGO procedure files are ASCII files, which the system saves with the automatic extension *.LOG* (disk users only).

Before we proceed much further, it is well to learn how to *SAVE* the file reading program in three ways—so see the next section for elucidation of these delights.

Saving Programs

There are three main ways in which a BASIC program can be saved. We will begin by examining the first two of these, so let us start with our recently developed file reading program (if using the tape system remember to first press REC and PLAY):

```
save "!read"
Ready
save "!rd", a
Ready
cat
```

and—depending on whether you are in the tape or disk environment—the result for the disk system includes

```
RD      .       1K
READ    .BAS    1K
```

and the result for the tape system includes

```
Press PLAY then any key
READ    block 1 $ Ok
RD      block 1 * Ok
```

The $ in the tape system indicates that the file *READ* (or its equivalent *READ.BAS* for the disk) is a standard BASIC program file, whereas the * indicates that *RD* is considered to be an ASCII file. Note that the ASCII version was saved with an extra parameter, *A* for ASCII.

If we have an ASCII file, we know that it is possible to list the content of the file by use of the reading file program; thus if we *RUN* the program using *RD* as the file name, when prompted by the program, we should effectively list the content of the ASCII file *RD.*

So, therefore, assuming the disk system (AMSDOS) is operative,

```
run
File name? rd
100  ' Example program—reading file
110  GOSUB 52000 : ' file name
120  WHILE NOT EOF
.....[lots more listing]

52030  RETURN
52100  ' read line
52110  LINE INPUT #9, a$
52120  PRINT a$
52130  RETURN
Ready
```

Thus we are able to list the program on disk (and if we used *!RD* we could perform the same action for the tape system).

How, then, does an ordinary BASIC program file differ from this ASCII version?

One sure way to find out is to try to *RUN* the program using the file name *READ.BAS* (for AMSDOS) or *READ* (for the tape system) when the program prompts for a name. Assuming AMSDOS is in operation, we can find

```
run
File name? read.bas
File type error in 52020
Ready
```

which means that it is not possible to read in an ordinary BASIC program file which has been saved in the normal manner. Thus when we attempt to open the file *READ.BAS* for reading (by *52020 OPENIN*) we find that the file cannot be opened, because the program file is of the wrong type.

The third method of saving BASIC programs is not relevant to the creation of program libraries, but for completeness I give the method at this stage of the discussion. Save the program once more, but this time append a parameter *P*:

```
save ''!rdp'', p
Ready
new
Ready
```

By the addition of the *p*, we have saved the program in '*p*rotected' mode, a special CPC Locomotive BASIC feature.

To have a protected mode program means that if we *LOAD* the program in the normal manner, we can neither *LIST* nor *RUN* it. The only way in which we can activate the protected program is to enter *RUN ''RDP''*.

Once the protected mode program has finished execution, we have to *RUN ''RDP''* and not simply *RUN* again—this mode is designed to provide some protection for normal BASIC programs.

In the AMSDOS or CP/M directories, there is no automatic way in which a protected program can be distinguished from a non-protected program, whereas in the tape system the file of a protected program is distinguished by a % symbol (compared to the $ of an ordinary program file or the * of an ASCII file).

Saving subroutines

The next stage is to begin to create a small library of file input and output subroutines. The library is deliberately kept small so that we can keep track of what is happening. When we save our subroutines, they will be saved as ASCII files, and this should be kept in mind. Start with

```
new
Ready
load ''!input''
Ready
delete 10-999
Ready
```

and if you *LIST* the remaining lines, you will find that only the three subroutines are left (at lines 51000, 51100, and 51200).

Save these subroutines by

```
save ''!input.rtn'', a
Ready
new
Ready
load ''!read''
Ready
delete 10-999
Ready
save ''!read.rtn'', a
Ready
new
Ready
```

We now have two files *INPUT.RTN* and *READ.RTN,* both of which are ASCII files.

I have appended the extension *.RTN* to the file name to indicate that they are sub*RouTiNe* files. A more obvious extension is *.SUB* but I do not use that one because it has a specific use in CP/M combined with the special CP/M *SUBMIT* transient program.

The question to be answered is 'Why use ASCII files?', and the answer lies partly in the use of the BASIC commands *MERGE* and *CHAIN MERGE.*

Merging subroutines

What we hope to achieve is a situation where we can write an applications program in which that program refers to standard subroutines (that is, those we have already written). Once the program has been written, referring to these subroutines, they are then *MERGE*d with the existing program. At that stage we can save the program together with the subroutines.

Another technique, and less expensive on storage, is to save the program without subroutines and then *MERGE* the subroutines from the appropriate file before the program is executed. Either way, we have to *MERGE*—I treat *CHAIN MERGE* as another form of *MERGE*, just as *CHAIN* is another form of *LOAD*.

Start to examine what happens when we *MERGE* files:

```
new
Ready
merge ''!rd''
Ready
list
```

and as a result you find the reading file program is listed.

Remember that *RD* was saved as an ASCII file: to see what happens when an ordinary BASIC program file is *MERGE*d, we enter the parallel sequence

```
new
Ready
merge ''!read''
Ready
list
```

and the content is as before.

This newly discovered ability to load a program by a different method (*MERGE*) might not seem exactly earth shattering in its consequences, but there is more to come.

If we make a slight modification to the process of *MERGE*ing, by having an existing program in memory, and then we *MERGE,* we will see that *MERGE* is not another name for *LOAD*:

```
new
Ready
10  ' A new line—not in the RD program
list
10  ' A new line—not in the RD program
Ready
merge ''!rd''
Ready
list
```

The result is to produce a listing of the program as before, with the addition of an extra line 10.

Rather than use an ASCII file, we can also try an ordinary BASIC program file, attempting to perform an equivalent sequence of actions:

```
new
Ready
10  ' A new line—not in the READ program
list
10  ' A new line—not in the READ program
Ready
merge "!read"
Ready
list
```

The result is exactly as before, in that the new line (10) is added to the program.

Not all is sweetness and light in the ad hoc *MERGE*ing of files, however, because when we try a protected BASIC program file we get different results. For example,

```
new
Ready
10  ' A new line—not in the RDP program
list
10  ' A new line—not in the RDP program
Ready
merge "!rdp"
File type error
Ready
list
10  ' A new line—not in the RDP program
Ready
```

Things are certainly very different, but we should not be surprised. It is not surprising that a protected file cannot be merged, otherwise the program would lose its protection, but this does mean that it is not possible to have protected subroutine libraries.

If there is no real difference in terms of behaviour between ASCII and normal BASIC program files, which type of file is to be preferred?

Making investigations over a series of programs, there is quite a difference in the memory requirements for the storage of programs on disk, with BASIC program files being shorter than the corresponding ASCII file. This is one important consideration.

The reason the program files take up less room is that each BASIC keyword is stored as one number rather than a sequence of characters as in an ASCII file.

You will have noticed that even when you type keywords in BASIC using lower case, when you list the program they are shown in upper case.

Individual keywords are not stored as strings of characters but as a special number. To store keywords as numbers (or 'tokens') is obviously effective in reducing memory requirements for programs.

However, for many purposes in system development, I prefer the ASCII version to the ordinary BASIC program file version. For example, if I am within CP/M, I can use the CP/M command *TYPE* to list the content of the files, and if I am within BASIC I can use the *READ* program.

It is not possible to examine program (non-ASCII) files in this way unless each program is loaded into memory while in BASIC—a very time-consuming activity.

Protected files are unnecessary in program development, unless it is wished to try to hide the code. This is irrelevant in program system development: why hide the code if you are the only person engaged in the development of the system? When you release the system, then is the time to hide the code.

My preferred form of action is to follow this sequence:

first, produce the subroutines one wishes to use in the applications library;
second, decide if the subroutines are all necessary at once (and thus go in one file), or if they can be split into discrete segments (and are best placed in separate files);
third, save all the subroutines on one file (for safety); and,
fourth, save the subroutines again, but this time as one set or in segments as decided in the second action.

The Correlation Coefficient

One of the main tasks of larger computers tends to be the accumulation of numerical and statistical information, and the presentation of such information in summary forms. To summarize the information, it has first to be read from the data files on which it is stored, and then transformed into usable forms. As an example of this type of procedure, let us examine one particular form of statistical analysis, that of the bivariate correlation of two variables whose values are stored on file.

The calculation of the bivariate correlation between two variables is examined in more detail in *Pocket Guide to Statistical Programming* (Pitman Publishing), and the program below is based on ideas in that book. The program is neater than the one given in the *Pocket Guide* because Locomotive BASIC and the CPC filing system are rather more sophisticated than the system used in the *Guide.*

Here is the program:

```
100  ' Bivariate Correlation
110  GOSUB 1000 : ' initializations
120  OPENIN f$
130  WHILE NOT EOF
140  GOSUB 1100 : ' first pass
150  WEND
160  CLOSEIN
170  GOSUB 1300 : ' means
180  OPENIN f$
190  WHILE NOT EOF
200  GOSUB 1200 : ' second pass
210  WEND
220  CLOSEIN
230  GOSUB 1400 : ' moments
240  GOSUB 1500 : ' output
999  END
```

In this case, it is perfectly feasible (and sensible) to write a program which does not use any subroutines, but is purely a linear sequence of statements. The reason why subroutines are used here is to show the structure of what is happening. The structure is:

(1) initialize the system, so that all variables are set to zero, and ask for the name of the disk file, which is then opened;
(2) read in the information from the disk file until the end of file marker, and at the same time accumulate values for the calculation of means; this is the first pass through the data;
(3) close the file, and calculate the means;

(4) reopen the file, and make a second pass through the data, modifying values to form products and cross products;
(5) calculate the covariance, standard deviations and correlation (the moments); and then
(6) output the results.

I will not give the content of the subroutine to output results because that is a matter of personal taste, but I will give details of all the other subroutines.

The first routine is that used to initialize variables:

```
1000  ' initializations
1010  number = 0 : mean1 = 0 : mean2 = 0
1020  sd1 = 0 : sd2 = 0 : cov12 = 0
1030  INPUT "File name ";f$
1040  RETURN
```

The meanings of the program variable names are the number of observations, the means, standard deviations and covariance for statistical variables *var1* and *var2*. All are set initially to zero. The file name is *f$*.

When the first pass through the data is made, only the mean values are accumulated:

```
1100  ' first pass
1110  INPUT #9, var1, var2
1120  number = number + 1
1130  mean1 = mean1 + var1
1140  mean2 = mean2 + var2
1150  RETURN
```

We accumulate the number of observations, and the sums of the values of the two variables (later to be made equal to the means of the two variables).

The sums are converted to means by the next subroutine:

```
1300  ' means
1310  mean1 = mean1/number
1320  mean2 = mean2/number
1330  RETURN
```

and the reason we calculate the means first is that—in the second pass—we subtract the mean from the value of the corresponding input variable.

By this process we minimize problems due to numerical inaccuracies produced by large numbers. For example, the result of the expression

```
?(999999*999999 + 1000001*1000001)/2
    - 1000000*1000000
  256
Ready
```

is incorrect, for the result should be *1*. The problem is (yet again) due to the finite arithmetic of all computers. If we enter the highly related

```
?(999999*999999 + 1000001*1000001
    + 1000000*1000000)/3 - 1000000*1000000
  256
Ready
```

there does not seem to be any rhyme nor reason in these results (the answer should be *0.666666667*).

To keep things tight, we will minimize the size of numbers by subtracting the means before any more complex manipulation. This is what is achieved by the second pass through the data.

```
1200  ' second pass
1210  INPUT #9, var1, var2
1220  var1 = var1 - mean1
1230  var2 = var2 - mean2
1240  sd1 = sd1 + var1*var1
1250  sd2 = sd2 + var2*var2
1260  cov12 = cov12 + var1*var2
1270  RETURN
```

The values of the variables (*var1* and *var2*) are modified by subtraction of the corresponding mean values. These transformed values are then used in further calculations. The variables *sd1, sd2* and *cov12* are used to create the sums needed for the calculation of the correlation coefficient. This calculation is performed by the subroutine at line 1400.

The reason why the subroutine is given the name *moments* is that the variance (square of the standard deviation) and covariance are known in statistical theory as the *second moment* and the *product moment* respectively.

```
1400  ' moments
1410  sd1 = SQR(sd1/number)
1420  sd2 = SQR(sd2/number)
1430  cov12 = cov12/number
1440  correlation = cov12/(sd1*sd2)
1450  RETURN
```

The input for the data file is via the earlier procedures, where the information is entered as pairs of values (for example, "56, 78). In the case of the cassette filing system, the cassette has to be rewound between the first and second passes through the file.

A small example: the correlation between the two variables whose values are

1, 2
2, 3
3, 4
4, 5
5, 4
6, 3

is 0.458682472.

The Primary Colours

We will pursue our rational approach to programming when we begin to study CPC graphics, and the rationale behind them. At the outset, however, it is worth establishing the nature of computer graphics in general, and in the specific case of the CPC computers. It so happens that it is in the case of graphics that there is the greatest differences between the 464 and 664/6128, but even these are not great.

The first thing to clarify when using CPC graphics is the difference between an *INK* and its colour. The CPC computers have a possible 27 different colours, but not all of these can appear on screen at once. Before discussing how the selection from these 27 colours is made, we will see how they are arranged. The CPC colour system is based (as are all computer colours) on the primary colours of light, which are *R*ed, *G*reen and *B*lue, and not the red, yellow and blue of painting.

The initials of the primary colours of light explain why colour monitors are often known as RGB monitors. The CPC computers can only provide these three colours, though they can be combined in various ways (and this is also the case for nearly all colour computers). What distinguishes the CPC computers are the number of different 'intensities' of these colours which can be used in combination.

The CPC colour scheme is based on an ordering of greyness of the primary colours, with each primary colour being of three possible intensities. The greyness of the colour is given by the weighting of that colour in Table 1.

BLUE is the least intense colour on a monochrome monitor, and on the CPC it can have the values 1 or 2 (plus 0 for no colour); RED has the values 0, 3 and 6; and GREEN (the most intense colour) has the values 0, 9 and 18. (For those interested, colours on the CPC are calculated in base 3 arithmetic.)

Table 1 *CPC colours*

Blue	*Red*	*Green*	*Grey*	*Name*
0	0	0	0	Black
1	0	0	1	Blue
2	0	0	2	Bright Blue
0	3	0	3	Red
1	3	0	4	Magenta
2	3	0	5	Mauve
0	6	0	6	Bright Red
1	6	0	7	Purple
2	6	0	8	Bright Magenta
0	0	9	9	Green
1	0	9	10	Cyan
2	0	9	11	Sky Blue
0	3	9	12	Yellow
1	3	9	13	White
2	3	9	14	Pastel Blue
0	6	9	15	Orange
1	6	9	16	Pink
2	6	9	17	Pastel Magenta
0	0	18	18	Bright Green
1	0	18	19	Sea Green
2	0	18	20	Bright Cyan
0	3	18	21	Lime Green
1	3	18	22	Pastel Green
2	3	18	23	Pastel Cyan
0	6	18	24	Bright Yellow
1	6	18	25	Pastel Yellow
2	6	18	26	Bright White

The CPC has 27 colours, therefore, numbered from 0 to 26, where each number is measured on a greyness scale: the colour numbers are in fact known as the 'Grey Level' in the CPC manuals. The Grey Levels are defined by the increasing brightness of the colours on a monochrome monitor.

I noted, above, that the addition of the primary colours was performed in base 3 arithmetic, and this enables us to experiment with different colour combinations. In a later section, we produce a method by which we can enter the desired intensities of primary colours (0, 1 or 2) and assign these colours to a particular *INK*.

Before I introduce the method, however, we need to examine the notion of *INK*s in Locomotive BASIC.

Colours and resolution

As the aim is ultimately to produce a turtle graphics system in which we can mix text and graphics windows, we have to establish the ways in which text and graphics are related. This relationship is closely associated with the different text and graphics modes available on the CPC computers.

One of the most obvious differences between the modes is that, depending upon the one in which we are operating, there are different numbers of colours we can display on screen at any one time. It is at this point that we learn there is a distinction between a 'logical' colour and an 'actual' colour.

The different logical colours are given numbers (starting with zero), and these are the numbers of the *INK*s. Any logical colour (or *INK*) can be associated with up to the 27 actual colours—though only one colour at a time. The amount of memory available for graphics on the CPC machines is fixed at 16K, and this can either provide more logical colours with less resolution or less colours with more resolution.

The fewest number of logical colours we can use is two. The two-colour mode provides the highest resolution, because if there are only two colours, memory is freed to keep track of more plotting points (and characters) on the screen.

Table 2 shows a list of modes available, together with information concerning resolution and the number of different logical colours it is possible to use in each mode.

Table 2 *CPC modes*

Mode	*Cols*	*Rows*	*Pixels*	*Colours*
0	20	25	160x200	16 (0..15)
1	40	25	320x200	4 (0..3)
2	80	25	640x200	2 (0..1)

Holding back from the plunge into an *INK*y morass, we will first establish how many distinct *INK*s there are for each mode.

This experiment will be undertaken with the aid of the command *PEN*. This is the program:

```
10  a$ = "XHXHXHXH"
20  FOR i = 0 TO 15
30  PEN i : PRINT a$
40  NEXT i
50  PEN 1
60  END
```

and note that—because the loop has strict limits—a *FOR NEXT* loop is used as it is most efficient for this example.

PEN establishes what *INK* (logical colour) is to be used by the *PRINT* command, and also establishes the logical colour used in graphics plotting. Starting with mode 0, *RUN* the program: there are sixteen examples of *XHXHXHXH*, all in different colours—a couple of examples are flashing from one colour to another.

If, however, you count the examples, you find there are fifteen different cases of *XHXHXHXH*, because there is a blank line where the first printing should have been. In mode 0 there are sixteen different *INK*s (0..15), and *PEN 0* refers to the background colour, because *INK 0* is the background logical colour.

The background is always *INK 0*, and normally the text is *PRINT*ed in *INK 1*—which explains why the last line of the program sets the *PEN* back to 1. In mode 0, each of the fifteen *INK*s has a distinct actual colour. What happens, then, when we use mode 1, which has four *INK*s?

RUN the program in mode 1, and you discover that though sixteen examples of *XHXHXHXH* are output, there are only four different colours. There is the background colour, and three different colours for use in printing text (or drawing graphics).

Enter

```
ink 1, 26
Ready
```

and one of the colours changes to white (code number 26). This is the distinction between *INK* and colour (or logical colour and actual colour). There are fewer *INK*s than there are colours, but any *INK* can be any of the 27 colours. (Or, any logical colour can take any actual colour.)

A few more quick changes:

```
ink 0, 0
Ready
border 13
Ready
```

I find this a pleasing arrangement (grey/white border, 13, black background, 0, and white text, 27).

A completely displeasing arrangement is

```
border 11, 15
Ready
```

and the border flashes between colour numbers 11 and 15. Even more displeasure is produced by

```
ink 0, 0, 26
Ready
ink 0, 26, 0
READY
```

and one's eyes begin to ache. At first the background (*INK 0*) is set to flash between the two colours numbered 0 and 26 (black and bright white); the text (*INK 1*) is then set to flash in the opposite manner 26 and 0 (bright white and black).

The text and background are set to flash in opposing senses, so that text is always visible (if a pain on the eyes). If the program is run again in mode 0, the problems are magnified. Return to sanity by

```
border 13
Ready
ink 0, 0
Ready
ink 1, 26
Ready
```

and *RUN* the *INK* program in mode 2. There are only two different logical colours available in mode 2, so that the output is a blank line, *XHXHXHXH,* blank line, *XHXHXHXH,* and so forth.

We can now extend the program to include a base 3 function *FNrgb*

```
  10  GOSUB 1000 : ' initialize
  20  GOSUB 2000 : ' print all inks
  30  GOSUB 3000 : ' change ink colours
 999  END
1000  DEF FNrgb(r,g,b) = b + r*3 + g*9
        : ' initialize
1010  a$ = "XHXHXHXH"
1020  RETURN
2000  FOR i = 0 TO 15 : ' print all inks
2010  PEN i : PRINT a$
2020  NEXT i
2030  PEN 1
2040  RETURN
3000  INPUT "ink"; in : ' change ink colours
3010  WHILE in > -1
3020  GOSUB 4000 : ' request levels
3030  INK in, FNrgb(red,green,blue)
3040  GOSUB 2000
3050  INPUT "ink"; in
3060  WEND
3070  RETURN
4000  INPUT "red"; red : ' request levels
4010  INPUT "green"; green
4020  INPUT "blue"; blue
4030  RETURN
```

This program allows you to alter the balance of red, green and blue, to produce an actual colour which is then assigned to a specified *INK* number. The program is stopped by the input of an *INK* of −1 (line 3010).

It is not possible to set flashing colours using this program, though it is easy to modify the program to do so. Flashing *INK*s are set by use of the command with three parameters: the first is the ink number (as usual) and the next two set the pair of colours between which the *INK* flashes (*see* above).

It is at this point we can see the advantage to a rational approach. The program is easily modified:

```
  10  GOSUB 1000 : ' initialize
  20  GOSUB 2000 : ' print all inks
  30  GOSUB 3000 : ' change ink colours
 999  END
1000  DEF FNrgb(r,g,b) = b + r*3 + g*9
        : ' initialize
1010  a$ = ''XHXHXHXH''
1020  RETURN
2000  FOR i = 0 TO 15 : ' print all inks
2010  PEN i : PRINT a$
2020  NEXT i
2030  PEN 1
2040  RETURN
```

```
3000  INPUT ''ink'; in : ' change ink colours
3010  WHILE in > -1
3020  GOSUB 4000 : ' request levels
3030  colour1 = FNrgb(red, green, blue)
3040  GOSUB 4000 : ' request levels
3050  colour2 = FNrgb(red, green, blue)
3060  INK in, colour1, colour2
3070  GOSUB 2000
3080  INPUT ''ink'; in
3090  WEND
3100  RETURN
4000  INPUT ''red'; red : ' request levels
4010  INPUT ''green'; green
4020  INPUT ''blue'; blue
4030  RETURN
```

It might make sense to turn the portion of code with the two *request levels* calls into a distinct subroutine. The principal interest from the point of view of rational programming is the use of multiple parameters to the function *FNrgb.*

The use of graphics implies a distinction between textual output and graphical output, and this distinction is dependent on an understanding of the use of 'streams' on the CPC.

Input and Output Streams

The rational approach to the study of windows is eased by the means used to control devices for the CPC computers. There are ten streams for input and output on the CPCs, plus a graphics stream. A 'stream' is a conceptual device which is used by the CPC to simplify input and output for a variety of physical devices, in keeping with rational modern practice which stresses the similarities between devices, rather than emphasizing their differences.

Stream #9 has already been encountered in the context of ASCII file input and output and, as you will have noticed, the transmission of data to and from files has few differences from the handling of ordinary input and output. In the case of ASCII files, the only real differences come with the *OPENIN*/*OPENOUT* file opening commands and the *CLOSEIN*/*CLOSEOUT* closing commands.

For example, the *READ.BAS* program can be turned into a file copy program by altering the *PRINT* statement (plus an *OPENOUT* and *CLOSEOUT*). What we produce is a situation where the file is treated as a picture (or ASCII) equivalent to the video display, and so—where we used to *PRINT* on the screen—we *PRINT* exactly the same information on the file space. The CPC operating system takes care of the pictorial conversion.

```
100  ' Example program—copy file
110  GOSUB 210 : ' names
120  WHILE NOT EOF
130  GOSUB 170 : ' read/write
140  WEND
150  CLOSEIN : CLOSEOUT
160  END
170  ' read and write file
180  LINE INPUT #9, a$
190  PRINT #9, a$
200  RETURN
210  ' file names
220  INPUT "File name in"; file$
230  OPENIN file$
240  INPUT "File name out"; file$
250  OPENOUT file$
260  RETURN
```

This file copy program will not only operate when using the disk system, but also (with more manual effort) when using the tape system. The new program is a BASIC program which, in part, emulates the *FILECOPY* program provided with the CPC CP/M system. The specially written *FILECOPY* program is, however, customized to CP/M on the CPC, and it also deals with files of all types, not just ASCII files.

Another important stream is #8, the stream devoted to the line printer. There is a program in there, struggling to emerge: a program to list the contents of an ASCII file onto the line printer.

All that is needed is a small modification or two of the preceding *copy file* program for the new program to work (a program which will function for both tape and disk systems):

```
100  ' Example program—printing file
110  GOSUB 210 : ' names
120  WHILE NOT EOF
130  GOSUB 170 : ' print line
140  WEND
150  CLOSEIN : CLOSEOUT
160  END
170  ' read and print file
180  LINE INPUT #9, a$
190  PRINT #8, a$
200  RETURN
210  ' file names
220  INPUT "File name in"; file$
230  OPENIN file$
240  RETURN
```

The reasons I have given these two file programs are partly because they are useful and partly because they illustrate the power of the notion of streams.

I noted that there were ten streams, and we have distinguished two specific ones (that for the printer, and that for files); the other eight are unspecific in that they treat the same type of device. The other eight streams are devoted to pseudodevices known as 'text windows'.

Windows and Co-ordinates

A 'text window' is a specifically delimited portion of the video display which is given a special label to identify that window.

The special label is the stream number for the window. Under normal circumstances, the whole video display is one window and its stream number is #0. To issue the command *PRINT,* without further specification, is equivalent to *PRINT #0.* A text window has almost an independent existence, because even though it is a segment of the video display it is able to be treated as an object.

A closer examination of the form and format of windows is given later, but (as a beginning) here is a program which enables the user to investigate the effects of windows—in a simple manner:

```
  10  GOSUB 1000 : ' initializations
  20  GOSUB 2000 : ' set window
  30  WHILE 1
  40  GOSUB 3000 : ' draw window
  50  INPUT a$
  60  WEND
 999  END
1000  DEF FNrn(x) = 1 + RND*x : ' initializations
1010  MODE 0 : INK 1, 0 : INK 0, 26
1020  RETURN
2000  INPUT ''window'; i : ' set window
2010  PAPER #i, 3
2020  RETURN
```

```
3000  WINDOW #i, FNrn(20), FNrn(20),
          FNrn(25), FNrn(25) : ' draw window
3010  INK 3, FNrn(26) : CLS #i
3020  RETURN
```

One window number worth trying is 0 (the main stream; that is, the default stream).

With *WINDOW 0* there are many strange consequences. What happens with *WINDOW 0* is that the prompt for input (the ?) can be seen moving around the screen, as new positions for the text window are defined.

Using windows other than 0, you will be able to see how one window can overlap another: for example, at times the prompt (?) (window 0) will overlap one of the windows drawn by the program. One window overwrites the other.

Employing other window numbers can produce strange effects, such as 'windows' 8 or 9:

```
run
window? 9
Improper argument in 2010
Ready
```

and the result is the same for *WINDOW 8*. There are no *WINDOW*s numbered *8* and *9,* though there are streams so numbered, because the Locomotive BASIC interpreter for the CPC considers that trying to set a *PAPER* for streams 8 or 9 is silly—thus such an action is not allowed. If you have asked for *WINDOW 0,* and then stop input by [ESC], you will discover that—quite probably—the BASIC output is restricted to a small window on screen.

We use BASIC's mode 0 because it provides more *INK*s than the other modes, though the program will still work, of course, in the other modes. It is possible to change the mode after the event but, however, if (after using *WINDOW 0*) you change mode, by entering *MODE 2*, it may not be possible to see the text. This lack of clarity can be remedied by entering *PAPER 0*, though you have to type 'in the dark' to do so.

Obviously, therefore, one has to be very careful about redefining *WINDOW*s, *INK*s and *MODE*s at the same time—it is very easy to lose track on the interactions. If one programs in a regular, systematic and rational manner, the possibility of such interactions coming without any warning is lessened.

Limits to the extents of the windows are given by the use of co-ordinate systems which are specific to each mode. We meet the other (special) window, the graphics window, in the next section, and it is at that point we will find that the co-ordinates for the graphics system are independent of the mode in use. The co-ordinates used for text windows are those given earlier for the rows and columns of the different modes, with the extra proviso that the origin for text co-ordinates is always top left, starting with column 1, row 1.

The CPC manuals have TEXT and WINDOW planners for the different modes.

The graphics window

The origin for graphics is at the bottom left of the graphics window (0,0). In normal circumstances the graphics window ranges over the entire screen, with the origin being at the bottom left of the complete screen. It is always the case that the value of the co-ordinate to the bottom right is (639,0) and that for the top left of the screen is (0,399). The top right corner is thus (639,399). These co-ordinate values are independent of the BASIC mode.

Without any attempt to redefine the graphics window, enter and then *RUN* the following program:

```
10  CLS : CLG
20  FOR i = 1 TO 25
30  PRINT ''OO''
40  NEXT i
50  DRAWR 200,200
60  INPUT a$
70  DRAWR 200,0
80  END
```

The outcome of the program up to line 60 is: a series of *OO*s at the left of the screen; a line drawn at 45 degrees to the vertical, starting at the bottom left corner of the screen; and an input prompt (?) bottom left waiting for you to hit the carriage return. The graphics line (which had started bottom left) has moved bodily up one line. The line still appears to run through a *O* (the same *O*) but all the letters (the text) have moved up that one line, to accommodate the input prompt.

The graphics drawing command

DRAWR 200,200

means draw, relative to the initial starting position, a line 200 units rightward and 200 units upward. The other drawing command

DRAWR 200,0

means draw a line 200 units rightward and 0 units upward (a line, that is, directly to the right). These movements are exactly true for all the CPC machines, though there is also an optional third parameter which gives the *INK*. The CPC 664 and 6128 have an extra (fourth) parameter which sets the form of drawing mode.

When the [ENTER] key is hit, a line is drawn to the right from where the first line finished. Where the first line actually finished, however, is not where the first line now appears to end. The drawing of the line starts from the absolute position at which the previous line finished. Though the line on screen has moved upward one character row, the new graphics line is drawn from the position before the movement.

The result is that the horizontal line appears to start one character row beneath the diagonal line's endpoint. The reason for this strange behaviour is that the graphics co-ordinate system is independent of the text windows though, with scrolling of the text window, graphics lines may be scrolled along with the text.

Rationally, therefore, to stop scrolling of the graphical output we have to use a text window which does not scroll.

If a slight modification is made to the program, in that line 60 is removed, but all else is as before:

```
10  CLS : CLG
20  FOR i = 1 TO 25
30  PRINT ''OO''
40  NEXT i
50  DRAWR 200,200
70  DRAWR 200,0
80  END
```

the outcome is that the two graphics lines join, but both are scrolled upwards. A final modification:

```
10  CLS : CLG
20  FOR i = 1 TO 25
30  PRINT ''OO''
40  NEXT i
50  DRAWR 200,200
70  DRAWR 200,0
80  WHILE 1 : WEND
90  END
```

where the modification is an endless loop at line 80, which has to be broken by a [ESC][ESC]. When the program is *RUN* there is no scrolling, and the graphics design is in its correct position.

In line 10 there are two commands *CLS* and *CLG,* one to clear the text screen and the other to clear the graphics screen. To clarify the difference between *CLS* and *CLG,* stop the program by [ESC][ESC], and then enter in instant mode

```
cls : drawr 200,200
```

This produces a cleared screen and a diagonal line: the diagonal line starts in the middle of the screen, where the last line finished.

Now try the result of

```
clg : drawr 200,200
```

and the screen clears, with a line drawn from the bottom left. The reason for this seeming ability to clear the screen, either by use of *CLS* or by use of *CLG,* is that both graphical and textual output use the same 'screen'.

On the CPC (and many other computers with 64K or less) the pixels used in graphics are exactly the same as elements of text characters. There is no information available which will enable the system to distinguish between an element of a text character and a graphics pixel. There is only one portion of memory given over to storing information about the screen, and not two memory banks to store information.

The overlaying of the textual and graphical output occurs in the CPC's memory, and does not happen on the screen. For some purposes we need to distinguish between (say) text and graphics, so information is separately maintained about the co-ordinates of, and cursor position in, the distinct text windows as well as the graphics window.

A rational approach to programming means we have to try to keep our graphical activities as distinct as possible from our textual activities, but in clearing the display screen, therefore, we modify certain types of information, but not other forms:

CLS clears the text screen, and the clearing of graphical output is incidental, because the graphics co-ordinates are unaffected.
CLG clears the graphics screen, and the clearing of the textual output is incidental, because the position of the text cursor is not altered.

These differences become important when we are developing the turtle graphics system, because in that system we want to maintain a clear distinction between text and graphics. The isolation of text from graphics (and vice versa) is the next step, so we now have to develop the turtle graphics screen.

The turtle graphics screen

Typing in the dark may be good for the soul, but it is a bad way to assure accuracy of input. We must have some means by which such information can be monitored by the user. Given that the CPC does not keep information about graphics and text in different portions of memory, we need some form of split screen display to promote ease of use.

A split screen display is defined by having a text window area which is distinct from the graphics window area. By 'distinct' we mean that textual input and output does not interact or interfere in any way with the graphical display. Conversely, the graphical output (that is, the drawing of lines) should not affect the textual display.

In terms we have recently been using: the textual display should be restricted to a window, which is less than the full video display; and the graphical display should be restricted to a window whose extent is less than the full video display. The two windows should not overlap, but should meet exactly at one edge.

It happens also that, to give colour to the graphical display, we need to define a second text window of exactly the same extent as the graphics window. The superposition of text window and graphics window also means that (if desired) we can *PRINT* information on the graphics screen, by use of the superposed text window.

It is normal, in cases such as this, to reserve the area occupied by the top twenty lines of text for graphical output, and to leave the lower lines for the input and output of text. The CPC has twenty-five lines of text for all modes, but (as noted above) each mode has a different number of characters per line. The best mode to use is mode 1 because it has 'high' resolution graphics with a fair number of colours.

We will, of course, use the default window stream 0 for normal textual display, so that when we enter information at the keyboard it appears in the text window. The extent of the text window is defined by:

```
WINDOW #0, 1, 40, 21, 25
```

That is, *WINDOW #0* is to extend from character 1 on the left to 40 on the right, and row 21 at the top to 25 at the bottom. (For modes 0 and 2 the only alteration will be to the number of characters on the right.)

On the 464, to give colour to the graphics screen we need also to define a text window of exactly the same extent as the graphics window. (The 664 and 6128 have a new command *GRAPHICS PAPER* to accomplish this task.) If this other text window is to complement the first text window, then it has to extend from row 1 to row 20. Therefore:

```
WINDOW #1, 1, 40, 1, 20
```

That is, *WINDOW #1* is to extend from character 1 on the left to 40 on the right, and row 1 at the top to 20 at the bottom. The two windows meet at lines 20 and 21, and both extend over the full width of the screen.

The normal extent of the graphics window is the complete video display, unless the window's dimensions are altered by the user's program. The dimensions of the graphics window are defined by graphical co-ordinates (0..639, 0..399) which are independent of mode. Each line of text thus has a 'depth' which extends over 400/25 = 16 graphical units. This is another way of saying that the height of a text character is 16 graphical units.

To be coterminous with the upper text window, therefore, the graphics window has to start $16 \times 5 = 80$ units up from the bottom of the screen (the window extends the full width of the screen, that is, from 0 to 639).

A text window has a fixed starting point, to which the cursor always returns when the window associated with a stream *#i* is cleared (by *CLS #i*). This starting point is always the top left of the window: in normal circumstances, when *CLS* is used, the cursor goes to the top left of the complete screen. It is possible, by use of *LOCATE,* to define another position, but only after the window is cleared.

There is no fixed starting point for a graphics window, but there is a customary starting point—which is the lower left corner of the graphics window. There is no fixed graphics starting point, because it can be altered to any location within the window by use of the *ORIGIN* command.

ORIGIN not only fixes the origin (0,0) for the graphical co-ordinates, but also fixes the extent of the graphics window. The first two parameters of the command fix the position of the new graphical origin (in terms of the standard graphical co-ordinates), and the next four parameters give the extent of the graphics window (again, in terms of the standard graphical co-ordinates). The order of co-ordinates is *left, right, top, bottom,* which is consistent with the order for text windows.

To define a graphics window which is exactly coterminous with the upper text stream, we need a graphics window from co-ordinates 0 to 639 (horizontally), and co-ordinates 399 (top) to 80 (bottom). Thus to position the origin in the centre of the graphics window, the co-ordinates need to be 319,239.

The graphics origin is set by

```
ORIGIN 319, 239, 0, 639, 399, 80
```

and the background colour is that of the top text window. The lines are drawn in the same colour as the text of the lower text window.

All this definitional work, plus some definition of colours, is encapsulated in one subroutine:

```
40000  ' set windows
40010  MODE 1 : INK 2, 25 : INK 1, 0
         : INK 0, 23 : ' mode and colours
40020  WINDOW #0, 1, 40, 21, 25
         : PAPER #0, 0 : CLS #0 : ' window 0
40030  WINDOW #1, 1, 40, 1, 20
         : PAPER #1, 2 : ' window 1
40040  BORDER 13
40050  ORIGIN 319, 239, 0, 639, 399, 80
         : ' graphics window
40060  GOSUB 41000  : ' clear graphics
40070  RETURN
```

Given the preceding discussion, most parts of this subroutine should be easily understood, apart from the first line (that is, line 40010) and the subroutine (41000) to clear graphics (the call is made at line 40060).

Before examining the subroutine to set windows, here is the definition of the subroutine to clear graphics:

```
41000  ' clear graphics early version
41010  CLG : CLS #1 : ' clears top screen
41020  x = 0 : y = 0 : angle = 0
         : ' initialize TG variables
41030  ps = 1 : dist = 0 : ' set penstate
         and distance to move
41040  RETURN
```

The subroutine involves other turtle graphics variables (x y angle ps dist) but even so, at the moment, consider the subroutine as a means to clear the graphics screen.

Subroutine 40000 associates colour number 25 with *INK* 2, colour 0 with *INK* 1, and colour 23 with *INK* 0. Thus we have selected three different colours and associated them with specific *INK*s: the colours are black (0), pastel cyan (23) and pastel yellow (25).

We will produce a pastel yellow graphics screen, a pastel cyan text screen, with text and graphics written in black. This tasteful arrangement is surrounded by a grey border (13), in what is called 'white' by Amstrad.

With these two subroutines we have the complete turtle graphics screen. There is the upper graphics screen, which is initialized to start at the centre of the graphics screen. The lower text screen starts with the *Ready* signal at the top left of that screen. Both screens are self-contained, and both are surrounded by a pleasant grey border.

A rather more vivid (but still pleasing) display can be produced by altering the colour numbers by

```
40010  MODE 1 : INK 2, 8 : INK 1, 1
          : INK 0, 10 : ' mode and colours
40040  BORDER 0
```

These new assignments produce an arrangement of colours which distinctly focuses one's attention on the display. As such matters are very much a matter of personal taste, try to experiment with different colours by use of instant commands such as

```
ink 2, 12
```

so that it is possible to change each ink individually, without having to rewrite the subroutine.

Drawing on the graphics screen

The first use we will make of the turtle graphics screen is not at all exceptional, because here is the subroutine to be used:

```
1000  ' random squiggle
1010  GOSUB 40000 : ' set graphics
1020  WHILE 1 : ' forever
1030  DRAWR 40*(RND - 0.5), 40*(RND - 0.5)
1040  WEND
1050  RETURN
```

The subroutine (when called by *GOSUB 1000*) sets up the turtle graphics screens, and then draws random lines on the graphics screen (until [ESC][ESC]).

DRAWR is an instruction to draw a line from the present co-ordinates to a new point. The distances to be moved horizontally and vertically are given as the two parameters.

Each parameter is the same: *40*(RND − 0.5)*, so that movements horizontally and vertically are approximately the same. There is a third possible parameter to *DRAWR* which gives an *INK*, and the 664 and 6128 have a fourth parameter which indicates the form of drawing (normal, XOR, AND or OR).

The best way of coming to terms with these parameters is to introduce them into the above subroutine, as *INPUT* values (a new subroutine called from line 1015?)

The result of *RND* is a random number from 0 to 1, and thus, if 0.5 is subtracted, the resulting number ranges from −0.5 to 0.5. As that result is multiplied by 40, then the final outcome ranges from −20 to 20. The lines therefore are drawn in all directions, but the changes in the co-ordinates are never more than twenty units. The effects produced by the 664's and 6128's (fourth) mode parameter can be intriguing.

To follow the sequence of events more closely, we can slow the pace of drawing down by use of a Locomotive BASIC interrupt facility, that known as *EVERY*.

Timers and Interrupts

The CPC has four inbuilt timers (0..3). These can be interrogated by the user by the BASIC interrupt commands, and to control events which have to be performed regularly, at fixed intervals.

There is a hierarchy of precedence amongst the interrupt commands: *REMAIN* takes priority over all other interrupt commands and disables the appropriate timer; the other two (*AFTER* and *EVERY*) are of equal precedence. With the latter two, the last program command to be issued takes precedence (overrides) any previous command.

There is also an order of precedence with the timers. If there is a conflict between interrupt commands, then that issued for timer 3 takes precedence over timer 2, which takes precedence over timer 1, and then timer 0. If a timer is not specified, then it is assumed to be timer 0 (that of lowest priority).

The *AFTER* command takes the form

AFTER delay, time GOSUB ⟨line_number⟩

which arranges for a subroutine (at ⟨*line number*⟩) to be called at some point in the future. One use for such a facility is in a game, where *AFTER* a certain time users are informed that they have taken too long, and thus have failed.

The *EVERY* command repeats the action, as defined in the subroutine specified, at intervals—given by the value of the *delay*.

EVERY delay, timer_number GOSUB ⟨line_number⟩

The timings for the *delay* (for both commands) are given in 1/50ths of a second. To set the speed at which we draw lines, therefore, we make the drawing of a line the subject of a subroutine, and vary the *delay* between calls to that subroutine.

My initial subroutine is

```
1000  ' interrupt demo
1010  GOSUB 40000 : ' graphics screen
1020  INPUT delay : ' measured in 1/50 sec
1030  EVERY delay GOSUB 1100
          : ' timer 0, draw line
1040  WHILE 1 : WEND : ' endless loop
1050  RETURN
```

and the content should be self-explanatory. The only addition necessary is the line drawing subroutine at line 1100:

```
1100  ' draw relative line
1110  DRAWR 40*(RND - 0.5), 40*(RND - 0.5)
1120  RETURN
```

which is exactly the same as the previous line drawing command sequence.

The first time we use the subroutine, the screen clears and we are faced with an input prompt. We input a *delay*—try the value 25—which slows down the drawing so that we can see the progress of the random squiggle program. Hit [ESC][ESC] and when we enter

```
gosub 1000
```

the pattern of events is not repeated. Instead of the screen clearing instantly, there is more drawing, and only then is the screen cleared.

At the point we enter a new *delay*, there may be a drawing on the screen: enter a new *delay* of 1, and the drawing is much faster. Stop, by [ESC][ESC], and rerun the subroutine at line 1000. The speedy drawing (with a *delay* of 1) continues for quite a while—or so it seems—before we clear the screen or have the input prompt. If the new *delay* is set to a large value, the speedy drawing continues for quite a while, and then slows to use the new delay.

These inconsistencies are due to the fact that we have confused the timer by the use of [ESC][ESC] without properly terminating the subroutine. Every time we use [ESC][ESC] and do not issue a *RETURN* within the program, we leave a return address on the stack.

The result, therefore, is that there are confused instructions still hanging around to complicate the control mechanisms in BASIC. We have found another example of the drawbacks to 'non-clean' control.

Though it might appear a good idea to be able to stop programs by use of a break command [ESC][ESC], we have to be wary about cluttering up memory with dross.

The way in which we clear memory is to issue the command *CLEAR* (mentioned above when we were discussing the subroutine stack), so that each time we finish by use of [ESC][ESC] we should enter

? 1
Break in 1020
Ready
clear

The need to reset the stack in this manner is tedious in the extreme, and militates against neat programming. In the case of the more complex procedures we will use later, this is no way for a sensible system to behave. Note that *CLEAR* should not be confused with *CLEAR INPUT* (only on the 664 and 6128), because the latter command only clears the keyboard input buffer.

We have found, yet again, that the desire for the quick and easy way out conflicts with the desire for a rational system that works, and a system which does not create more problems than it solves. It might seem that perhaps I am being too idealistic, but I want my programs to function correctly under most conditions.

Clearing up *BREAK*s

We need to incorporate some mechanism into our programming, which waits so that when we issue an [ESC][ESC] the stack is automatically cleared. The overseeing mechanism is known as the *ON BREAK GOSUB* command. That is, whenever a *BREAK* (for example, [ESC][ESC]) occurs while BASIC is executing, then call the subroutine at the line given by the command.

The best place for this new action to be situated is in the subroutine at line 41000, which clears the graphics screen. Here, therefore, is the final version of the subroutine to clear graphics:

```
41000  ' clear graphics final version
41010  ON BREAK GOSUB 41100
       : ' clear memory
41020  CLG : CLS #1
41030  x = 0 : y = 0 : angle = 0
41040  ps = 1 : dist = 0
41050  RETURN
```

where the main difference to the first version is that line 41010 is new (and all other lines have moved down, to make way). The subroutine at line 41100 is, therefore, the important new feature.

The new subroutine is very short:

```
41100  ' clears stack and variables
41110  CLEAR
41120  END
```

and it leaves everything neat and tidy. Note that the routine is terminated by an *END* rather than a *RETURN,* and this is a deliberate move. (664 and 6128 users might wish to add *CLEAR INPUT* to line 41110.)

If the subroutine at 41100 is ended with *CLEAR* : *RETURN* rather than *CLEAR* : *END,* then to issue a [ESC][ESC] gives the outcome

```
? 6
Unexpected RETURN in 41120
Ready
```

(assuming a break was made after the input of 6).

When we issue the *CLEAR* instruction, the return address for subroutine 41100 is cleared from the stack—BASIC forgets that 41100 is supposed to be a subroutine. Thus when the *RETURN* in line 41120 is encountered, there is no return address to be found: the *RETURN* is thus unexpected.

The interruption comes cleanly if there is an *END*, because everything is left in order. Incidentally, if the subroutine is altered to a special form it is impossible to break into the program. (There is a new command on the 664 and 6128 *ON BREAK CONT* specifically for this purpose.)

If you want to try out this facility, SAVE THE PROGRAM NOW.

Only when the program has been saved should you modify the subroutine, and then issue the command to *GOSUB 1000*

```
41100  ' never ends
41110  RETURN
```

This is one way in which you can make sure a program runs until you (the programmer) want it to end (at some point you introduce an *ON BREAK STOP* command, or a *NEW* to erase the program). To produce a 'turnkey' program which runs as soon as it is loaded, and continues to run until the machine is switched off, start your program with

```
  1 KEY DEF 66,1,0,0,0
  2 ON ERROR GOTO 10
 10 CLEAR : WHILE 1 : ON ERROR GOTO 10
 20 GOSUB 100 : ' main program
 30 WEND
 40 END
 50 RETURN
100 ' main program starts here
110 ' and ends with a RETURN
```

and save the program in protected mode. The main program is that in the subroutine at line 100.

The instruction to *KEY DEF 66,1,0,0,0* changes the operation of key number 66 (the *ESC*), to have a delay of 1, and assigns the ASCII code 0 to that key. Notice that the first parameter is a key number, and the third, fourth and fifth parameters are ASCII values.

The reassignment is performed for the [ESC] not only in the normal mode, but also in the shifted and control modes. The key codes are given in CPC documentation concerning ASCII and keyboard characters.

The *ON ERROR GOTO* command instructs the BASIC interpreter to *GOTO* the specified line, if any error occurs in the program (deliberate user errors can be used to crash programs). By sending control back to line 10, *CLEAR*ing and reasserting the error trap, the program continues for ever—or at least until the whole system is reset. On an error, therefore, the system as a whole is rerun.

The Turtle Graphics Routines

We have a graphics screen, with the *ORIGIN* being dead centre. In our initializations (the subroutine at line 41000) we specify values for several variables—five in all. The variables and their designation are:

x The x co-ordinate of the turtle;
y The y co-ordinate of the turtle;
angle The direction in which the turtle points;
ps The penstate, draw a line or just move; and
dist The distance the turtle is to move.

The theory behind turtle graphics is explained in *Guide to LOGO* (from Amsoft), as well as in *Pocket Guide to LOGO* (from Pitman Publishing).

For those without access to either of these tomes, the essence of turtle graphics is the 'turtle', an imaginary reptile which lives on the graphics screen. Being an object with a head, the turtle points in a certain direction (known as *angle* in my system), and it can be told to move forward a certain amount (*dist*).

Any movement of the turtle is obviously in the direction it is facing. To direct the movement of the turtle, therefore, all you do is specify an *angle*, and a *dist*ance to move. That is not quite all, for you have to say from where the turtle is to start drawing.

You have to specify initial co-ordinates (those known as *x y*), and normally the turtle starts in the middle of the screen (*x = 0* and *y = 0*). You also have to specify the initial direction (*angle = 0*), which conventionally is directly upwards and measured in degrees counterclockwise.

Once the turtle has been given initial co-ordinates and direction, there is no need to specify any more co-ordinates, because all drawing is by relative angles and distances.

With no claims for originality, I will start with a square, and give the procedure in Dr LOGO for the CPC:

```
?to square :side_length
>repeat 4 [fd :side_length rt 90]
>end
```

where the square has a side of *:side_length*. To draw a side of a square you move forward (*fd*) an amount *side_length*, and at the end of the side turn right (*rt*) through *90* degrees. You are left pointing in the direction in which the next side is to be drawn. We *repeat 4* times the drawing of a side, and thus produce a square. The equivalent subroutine in BASIC is

```
500  INPUT side_length : ' draw square
510  dist = side_length
520  FOR i = 1 TO 4
530  GOSUB 42000
540  angle = angle + 90
550  NEXT i
560  RETURN
```

and the drawing routine is that at line 42000. Before any drawing takes place, the turtle graphics system is activated by the call to *GOSUB 40000,* and so (for reference) here are the appropriate routines:

```
40000  ' set windows
40010  MODE 1 : INK 2, 25 : INK 1, 0
         : INK 0, 23
40020  WINDOW #0, 1, 40, 21, 25
         : PAPER #0, 0 : CLS #0
40030  WINDOW #1, 1, 40, 1, 20
         : PAPER #1, 2
40040  BORDER 13
40050  ORIGIN 319, 239, 0, 639, 399, 80
40060  GOSUB 41000
40070  RETURN

41000  ' clear graphics
41010  ON BREAK GOSUB 41100
41020  CLG : CLS #1
41030  x = 0 : y = 0 : angle = 0
41040  ps = 1 : dist = 0
41050  RETURN

41100  CLEAR : END : ' clear up

42000  ' draw line
42010  angle = angle - 360 * INT(angle/360)
42020  x = x + dist * SIN(angle*PI/180)
42030  y = y + dist * COS(angle*PI/180)
42040  IF ps THEN DRAW x,y
       ELSE MOVE x,y
42050  RETURN
```

```
42500  ' draw to point x,y
42510  IF ps THEN DRAW x,y ELSE MOVE x,y
42520  RETURN

43000  ' change penstate
43010  CLS
43020  ps = 1—ps
43030  RETURN
```

To show how subroutines build on subroutines (and why I was so concerned to see how deeply they could be stacked) consider subroutines 4000 and 5000. Subroutine 5000 draws a square, goes forward a distance equal to the side of the square, increases the angle by 20 degrees, and increases the side of the square by 0.5. Another square is drawn, and the process is repeated *WHILE* the side of the square is less than 600 units.

```
5000  ' inc squares
5010  dist = 1
5020  WHILE dist < 600
5030  GOSUB 4000 : ' square
5040  GOSUB 42000 : ' line
5050  angle = angle + 20
5060  dist = dist + 0.5
5070  WEND
5080  RETURN
```

The subroutine is a slightly modified version of the earlier subroutine to draw a square.

```
4000  ' square
4010  FOR i = 1 TO 4
4020  GOSUB 42000 : ' line
4030  angle = angle + 90
4040  NEXT i
4050  RETURN
```

The best way to experiment with the routines is to experiment . . . One rather nice way of performing such experimentation is to use the function keys.

Defining function keys

My turtle graphics system has a short initialization program, which goes:

```
10  GOSUB 20000 : ' set function keys
20  GOSUB 40000 : ' initialize
99  END
```

and the content of the subroutine at 20000 is:

```
20000  k$ = CHR$(13)
       : ' carriage return
20010  KEY 139, "GOSUB 41000"+k$
       : ' [ENTER] clear graphics
20020  KEY 138, "GOSUB 47000"+k$
       : ' [.] set x,y
20030  KEY 128, "GOSUB 42000"+k$
       : ' [0] draw
20040  KEY 135, "GOSUB 44000"+k$
       : ' [7] left turn
```

```
20050  KEY 137, "GOSUB 45000"+k$
       : ' [9] right turn
20060  KEY 136, "GOSUB 46000"+k$
       : ' [8] set distance
20070  KEY 133, "GOSUB 43000"+k$
       : ' [5] change pen state
20080  KEY 131, "MODE 2"+k$
       : ' [3] listing mode
20090  KEY 129, "RUN"+k$
       : ' [1] run program
20100  KEY 130, "GOSUB 42500"+k$
       : ' [2] drawto x,y
20110  RETURN
```

These statements assign functions to the keys, where the function is that given by the string (for example, *"GOSUB 43000"+k$*), and the key is identified by a special number (in this case, *133*).

Each key number corresponds to a function key (this time it is *[5]*), so when that key is pressed that function is activated: thus to press function key *[5]* lifts the turtle's pen (if down) or puts it down (if up).

I was unable to add many more functions because the memory available for function key assignment became too little, but this selection is adequate. Note that *k$ = CHR$(13)*, that is, *k$* is the carriage return operation.

Certain of the above functions are obvious, and subroutines have been provided for others, but there are certain new ones (and associated subroutines), the idea of which is to allow on-screen doodling with turtle graphics. The appropriate subroutines are:

```
44000  ' left turn
44010  CLS
44020  INPUT ''Left''; turn
44030  angle = angle - turn
44040  RETURN

45000  ' right turn
45010  CLS
45020  INPUT ''Right''; turn
45030  angle = angle + turn
45040  RETURN

46000  ' distance forward
46010  CLS
46020  INPUT ''Distance''; dist
46030  RETURN

47000  ' set x,y co-ordinates
47010  CLS
47020  PRINT ''x is ''; x, ''y is ''; y
47030  INPUT ''x''; x
47040  INPUT ''y''; y
47050  RETURN
```

A simple example of how the turtle graphics routines can be used is the subroutine to draw a spiral, in which the turtle turns through a certain angle (*a1*), moves forward a certain distance, turns through the same angle, moves forward through a slightly larger distance (an extra *i1*). The routine asks for the values of *a1* and *i1*:

```
2000  ' outspiral
2010  INPUT a1, i1
2020  GOSUB 40000 : ' initialize
2030  WHILE 1
2040  angle = angle + a1
2050  dist = dist + i1
2060  GOSUB 42000 : ' draw line
2070  WEND
2080  RETURN
```

The snowflake

A rather more complex example is the drawing of a snowflake—which makes extensive use of recursive subroutines. The main 'driver' subroutine is

```
6000  ' snowflake
6010  INPUT "Side"; dist
6020  INPUT "Order"; order
6030  FOR i = 1 TO 3
6040  ang(order) = 120
6050  GOSUB 6100 : ' drawside
6060  NEXT i
6070  RETURN
```

in which the main side of the snowflake (the basic equilateral triangle) is entered as *dist,* the complexity of the snowflake (the order) is entered as *order,* and from 6030 to 6060 the basic equilateral triangle is drawn (with twiddly bits due to the subroutine *drawside*).

The content of the *drawside* subroutine is as follows:

```
6100  ' drawside
6110  IF order = 0 THEN GOSUB 42000
      ELSE GOSUB 6200 : ' draw else drawtri
6120  angle = angle + ang(order)
6130  RETURN
```

and in this routine there is either a line drawn (if the order is zero) or there is a call to the subroutine known as *drawtri.* The format of *drawtri* is:

```
6200  ' drawtri
6210  order = order—1 : dist = dist/3
6220  ang(order) = −60 : GOSUB 6100
      : ' drawside
6230  ang(order) = 120 : GOSUB 6100
      : ''drawside
6240  ang(order) = −60 : GOSUB 6100
      : ''drawside
6250  ang(order) = 0 : GOSUB 6100
      : ''drawside
6260  order = order + 1 : dist = dist*3
6270  RETURN
```

which breaks up the subroutine into a further four calls to *drawside* (which has already called *drawtri*). The best way to see how this system works is to call the *snowflake* routine and try the effects of different sides and orders.

There is a detailed explanation of how the snowflake program works in *Introducing Pascal* (Collins), and the next example is based on the program given therein. The program was written by Robin Shipp in Hisoft Pascal for the 464 (using a tape system). Robin has implemented many special turtle graphics commands which I will not list here, so I will give the bones of his procedures:

```
1250  PROCEDURE drawside(order : integer;
      angle, side : real);
1260  FORWARD;
1270
1280  PROCEDURE drawtri(order : integer;
      side : real);
1290  FORWARD;
1300
1310  PROCEDURE snowflake(order : integer;
      side : real);
1320  BEGIN
1330    drawside(order, 120, side);
1340    drawside(order, 120, side);
1350    drawside(order, 120, side)
1360  END;
1370
```

```
1380  PROCEDURE drawside;
1390  BEGIN
1400   IF order > 0
1410    THEN drawtri(order, side)
1420    ELSE fwd(side);
1430   turn(angle)
1440  END;
1450
1460  PROCEDURE drawtri;
1470  BEGIN
1480   drawside(order-1, -60, side/3);
1490   drawside(order-1, 120, side/3);
1500   drawside(order-1, -60, side/3);
1510   drawside(order-1, 0, side/3)
1520  END;
```

I have not given the main program or the routines to move *fwd* and *turn*, but I think you can see how the outlines of the Pascal program follow the same rational pattern. Incidentally, the term *FORWARD* indicates that the procedure is called by a second procedure, which itself calls the first procedure.

It would be easier in fact to produce a Dr LOGO set of procedures, but I hope to have shown how BASIC, when approached from a rational viewpoint, can produce easy to understand programs.

A final FILL for the 664 and 6128

This is a short section for the 664 and 6128.

Keep the snowflake, drawn using BASIC, on screen and move the turtle (that is, the graphics cursor) to some point inside the snowflake. The positioning can be most easily achieved in an interactive environment by using the function keys to lift the turtle's pen, and draw to specified co-ordinates. Issue the command

```
FILL 1
```

and the snowflake is filled with colour, the same colour as the normal pen colour.

The *FILL* command fills a shape with colour, where the boundaries are set by the lines already drawn and by the edges of the graphics window (you cannot leave graphics to *FILL* our lower text window). The shape to be filled can be of arbitrary complexity, but if it is very complex and you have a very large program, then there might not be enough memory available.

The *FILL* uses its own stack to store information about where it has been, if at any time it needs to backtrack—*FILL* is effectively a recursive procedure. As we have found with our recursive subroutines, at times the stack runs out, though with *FILL* you start with a possible 6000 recursive calls, until memory is used by BASIC programs, and so the number of recursive calls is reduced.

In conjunction with the other new commands (particularly *MASK*), the potential for graphical applications is enhanced on the 664 and 6128.

Index

Computer Handbooks

Languages

Assembly Language for the 8086 and 8088 Robert Erskine
C Language Friedman Wagner-Dobler

Business Applications

dBASE III Peter Gosling
SuperCalc and SuperCalc 2 Peter Gosling
VisiCalc Peter Gosling

Microcomputers

The Amstrad 464, 664 and 6128 Boris Allan
The Apricot Peter Gosling
The Sinclair QL Guy Langdon and David Heckingbottom

Operating Systems

Introduction to Operating Systems Lawrence Blackburn and Marcus Taylor

Word Processing

Wordwise and Wordwise+ Wendy Chuter

Pocket Guides

Programming

Programming John Shelley
Statistical Programming Boris Allan
BASIC Roger Hunt
COBOL Ray Welland
FORTH Steven Vickers
FORTRAN Philip Ridler
FORTRAN 77 Clive Page
LOGO Boris Allan
Pascal David Watt

Assembly Languages

Assembly Language for the 6502 Bob Bright
Assembly Language for the 8085 Noel Morris
Assembly Language for the MC 68000 Series Robert Erskine
Assembly Language for the Z80 Julian Ullmann

Microcomputers

Acorn Electron Neil Cryer and Pat Cryer
Commodore 64 Boris Allan
Programming for the Apple John Gray
Programming for the BBC Micro Neil Cryer and Pat Cryer
Sinclair Spectrum Steven Vickers
The IBM PC Peter Gosling

Operating Systems

CP/M Lawrence Blackburn and Marcus Taylor
MS-DOS Val King and Dick Waller
PC-DOS Val King and Dick Waller
UNIX Lawrence Blackburn and Marcus Taylor

Word Processors

Introduction to Word Processing Maddie Labinger
IBM Displaywriter Jacquelyne A. Morison
Philips P5020 Peter Flewitt
Wang System 5 Maddie Labinger
WordStar Maddie Labinger

COMPUTER HANDBOOKS

Microcomputers

Computer Handbooks provide handy, compact references to microcomputers, languages, operating systems, business applications and word processing. They are an essential companion for anyone using computers at home or in the office.

Also available

The popular and bestselling series of *Pitman Pocket Guides.*

A complete list of all titles in these two series is given inside.

ISBN 0-273-02390-X

UK £3.95
AUS $7.95
CAN $5.95
US $5.95

The Amstrad 464, 664 and 6128 Allan PITMAN

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/