For the AMSTRAD CPC 464, CPC 664 & CPC 6128

The AMSTRAD Programmer's Guide

Clive Gifford
Tim Hartnell

Pitman

THE AMSTRAD PROGRAMMER'S GUIDE

THE AMSTRAD PROGRAMMER'S GUIDE

Clive Gifford Tim Hartnell

Pitman

PITMAN PUBLISHING LIMITED
128 Long Acre, London WC2E 9AN

A Longman Group Company

First published 1985

British Library Cataloguing in Publication Data
Gifford, Clive
The Amstrad programmer's guide.
1. Amstrad computers — Programming
I. Title II. Hartnell, Tim
001.64'2 QA76.8.A4

ISBN 0–273–02447–7

Library of Congress Cataloging in Publication Data
Gifford Clive
The Amstrad Programmer's Guide.
Includes Index.
1. Amstrad Microcomputer — Programming.
I. Hartnell, Tim. II. Title.
QA76.8.A48G54 1985 005.2'65 85–12339
ISBN 0–273–02447–7

Printed in Great Britain by Butler & Tanner Ltd, Frome and London

CONTENTS

1	How does your computer work?	1
2	First steps in programming	4
3	More programming	11
4	Looping the loop	21
5	Continuing up the learning curve	27
6	READ/DATA and RESTORE	34
7	Random numbers and decisions	41
8	String handling	48
9	Inputs of various kinds	55
10	Commands to aid your programming	61
11	The Amstrad cassette and its uses	64
12	Using discs	70
13	Games! games! games!	79
14	Word processing	96
15	A graphic explanation	100
16	Sounds like fun	128

17 The rudiments of machine code 147

18 System interrupts on your Amstrad 149

19 Windows 154

20 The error commands and ON BREAK 158

21 A pot pourri of commands 161

22 The Amstrad CPC 6128 163

23 How to program more professionally 165

Appendix A Addresses of suppliers 169

Appendix B Memory map 171

Appendix C Error codes and definitions 173

Index 177

PREFACE

Welcome to the Amstrad! So you've bought an Amstrad. You should be very pleased — the machine is well ahead of its time and offers features unavailable on other, more expensive, home computers.

This book has been written to provide everything that the Amstrad's manual lacks and much, much more. And, once you have mastered the BASIC language, in the later stages of the book we have tried to show you a number of ways that you can improve your programs, giving them a more professional appearance, action and structure.

Before you decided that the Amstrad was the micro for you, you must have had some idea of what you wanted to do with it once you got it. Perhaps it was just for playing games (the Amstrad is certainly good for that), or perhaps it was to become a useful addition to your house (to help you to write letters, work out complex problems, manage your finances and help teach the children) or maybe you wanted to learn how to program and cut through the mystique that surrounds computers — to understand what they can and cannot do. All these are good reasons to buy a home computer, and you'll find all are well supported within this volume.

Read this book, have fun with the programs included, and learn to program the Amstrad like a professional!

Clive Gifford, Tim Hartnell
May 1985

ACKNOWLEDGEMENTS

The authors would like to thank all those who helped in the production of this book, especially Catherine, Betty, Bill and Al.

1
HOW DOES YOUR COMPUTER WORK?

In our daily lives we tend to work in a number system based on 10 — the decimal system. We have nine digits to work with (plus zero) and with these we can represent any number we want to. The position of a number indicates its value, with the position being made clear by the presence of trailing zeros. That is, we know 90 is ten times greater than 9, with the zero which follows the nine in 90 making the value of the number clear.

Computers tend to work in a number system based on two — the binary system. This uses only ones and zeros with the sequence and position of digits indicating their value. Here are the numbers one to nine in our normal numbering system (base ten) with their binary (base two) equivalents underneath:

1	2	3	4	5	6	7	8	9
1	10	11	100	101	110	111	1000	1001

As you can see, binary numbers grow long very quickly and they lack the 'instant recognition' Arabic numerals possess. Why on earth should they be used? The answer is simply that your Amstrad is, in essence, a vast collection of switches, and a switch — as you know — can have two states only, 'off' and 'on'. If we use 'off' to represent zero and 'on' to represent one, we can represent any number with a long enough row of switches.

The Amstrad, of course, does not require you to know or understand binary arithmetic. It does the conversions from your base system to its own and back, as needed. Most computers use binary arithmetic. In the early days of programming, programmers had to work with zeros and ones, which meant writing a program was an immensely time-consuming and frustrating task. Any error in running meant that a large number of ones and zeros had to be checked.

In those days, programmers really were a race apart. The intellectual strain must have been tremendous. Fortunately, the computer

does most of the work for us now, in changing numbers from long strings of zeros and ones into numbers we can understand.

But this discussion, while it indicates how the computer manipulates numbers, does not help explain how a computer 'thinks'.

Computer logic, computer decision-making power, is based on the discoveries of an English mathematician from the turn of the century, George Boole. His work is called boolean algebra.

The value of Boole's discoveries lies in the fact that he reduced decision-making to a mathematical process — a process that proved tailor-made for computer use. Your computer stores 'true' and 'false' as numbers.

Let's illustrate this. If you turn your computer on and type in PRINT 1 = 1 (a true statement), you'll see the computer prints up −1, showing it recognizes 'truth' as −1. Now enter a false statement, such as PRINT 3 = 4. This time your computer will return a zero, indicating that a false condition is stored as a zero.

Logical, my dear Watson

Now, as you'll see later, your Amstrad can make decisions which involve 'logical operators'. That is, it can see if statement A *and* statement B are true, if statement A *or* B is true, or if neither statement A *nor* B is true. The computer can even handle long complex chains of logic such as determining if statement A *and* statement B are true *and* statement C *or* statement D is false *or* statement E is false. The heart of the computer's decision-making power, as opposed to its ability to manipulate numbers, lies in this ability to handle boolean algebra and in its ability to translate these findings — expressed mathematically — into an answer we can understand.

This, of course, is not the whole picture. Let's look more closely at your Amstrad. It has a means of accepting information from the outside world (the keyboard and other devices such as a joystick), and a means of translating that information into the ones and zeros it needs.

It has a way of storing that information until it needs it, of comparing new information with previously held facts, and a way of outputting (via the screen or the printer) the information it has developed.

If, for example, you want your Amstrad to add four numbers together, you might first tell it that the letter A represents the number

4, that B equals 5, C equals 9 and D equals 2. It will accept this information, storing it as a number of binary patterns. Then, you might say ADD A and B, then ADD C and D, and print out the results.

There is a fixed binary pattern already in the computer when you buy it, which tells it how to add two numbers when so instructed and how to output the results.

Next, you might get it to determine which of the two results (A + B or C + D) is the larger, and print out this information. The ability to do this is also stored as a built-in binary pattern. Based on the finding of whether A + B or C + D was the larger, the computer can be instructed to do something else . . . and so on.

Read through this greatly simplified explanation until it begins to make some sort of sense. You will then be in a position to start to appreciate how your Amstrad works. In fact, you don't *need* to know how the computer works in order to be able to use it — any more than you have to be able to understand the workings of your television in order to watch a film — but having an introduction to the subject will aid your appreciation of the electronic genie you have at your command.

2
FIRST STEPS IN PROGRAMMING

The BASIC language used by your Amstrad is fairly easy to master. You should be able to program with some ease within about half an hour, even if you have never programmed before. Mastery of programming, of course, will take much longer.

We will follow a simple routine in working through the commands available on your computer. First, a word which the computer recognizes will be explained briefly, and then it will be shown in use. Short programs will be introduced, combining new command words with those already learned; in this way, you can see how different words can be used to build up a program.

From time to time, we will include some major programs; you could, in fact, enter most of these right now. Even if you do not fully understand how the programs work, you can still enjoy running them.

Lining up

All computer programs in BASIC (the computer language you're learning and the one provided for nearly every personal computer in the world) are composed of a series of lines, each of which begins with a number. In general terms, the computer executes a program in numerical order, starting with the lowest line number and proceeding to the highest. From time to time, the computer will redirect action within the program (using such commands as GOTO), but this will be covered later in the book.

From now on, it will be assumed that your Amstrad is turned on and running, and that you're going to enter each program as you come to it. You will find that you gain far more from the book, and learn to program much more quickly, if you enter the programs in this way, trying out the exercises described. This is a self-teaching text, and some application from you is required if it is to do its work effectively.

PRINT, LIST, NEW, RUN

PRINT is one of the most frequently used words in BASIC programming and it means more or less what you would expect; a PRINT statement tells the computer to PRINT something on the screen. The word PRINT and the other commands we'll be studying in this section of the book are words the computer has been pre-programmed to understand.

To show PRINT in action, type in PRINT "TEST" (making sure that you put double quote marks on each side of the word that you want to print), then press the ENTER key. The word TEST should appear under your line PRINT "TEST". Now look underneath both lines on the screen and you should see the word Ready, indicating that the computer has done everything that you told it to do and is now awaiting further instructions.

Type in the following set of instructions (a program) pressing the ENTER key each time you complete a line.

```
10 PRINT "AMERICA"
20 PRINT "THE"
30 PRINT "BEAUTIFUL"
```

Once you've entered the above program into your computer successfully, type in the word RUN and then press the ENTER key. As you can see, the ENTER key is pressed every time you want the computer to act on what you have just entered. The RUN command sets the program in motion and you should see something like the following on your screen:

```
AMERICA
THE
BEAUTIFUL
```

Now there is quite a bit we can learn from this short program. As pointed out a little earlier, program lines are numbered and the program tends to run from the lowest line up to the highest. There is no need to number the program lines in tens, starting at line ten, but you'll find it a good habit to adopt; if you do, you'll find you have room to add new lines between those you have already entered, should the need arise.

The PRINT statement determines how information will be printed on the screen. Change the second line of your program to read like the following line. You change lines simply by typing the whole line in

again with the required changes, and then pressing the ENTER key; this makes the new program line take the place of the old one. (There are other ways to modify program lines which do not involve re-typing the whole line, but these will be discussed later.) Anyway, type in the new line 20 given below:

```
20 PRINT, "THE"
```

This time, when you RUN the program, you'll notice that the output has changed to the following:

```
AMERICA
                THE
BEAUTIFUL
```

The comma before the quote marks in line 20 has moved the PRINT position across the screen.

You can use the comma in this way to format the print output. For example, you can get a row of numbers lined up neatly on one side of the screen.

Now, remove that program from your computer by typing in the word NEW. Typing in NEW, then pressing the ENTER key, instantly clears the contents of the computer's memory.

You can check that the computer is empty by entering the word LIST, then pressing the ENTER key. You'll see the Amstrad's memory is empty, as no program listing appears. We'll be discussing LIST in a moment.

Before we do that, however, enter and RUN the following program:

```
10 PRINT "THIS IS"
20 PRINT "A TEST"
```

The resultant screen display should look something like this:

```
THIS IS
A TEST
```

Now change line 10 to the format given below:

```
10 PRINT "THIS IS ";
```

You'll see that there is a space between the letter s and the closing quote marks, and that there is a semi-colon (;) after the closing quotes. Now run the program again, and you should get a result similar to that shown below:

```
THIS IS A TEST
```

As you can see, the words all run along on the same line. This is because the semi-colon joins the end of one PRINT statement to the following one.

TAB and more on LIST

The TAB (for 'tabulate') command enables the very precise positioning of the start of a line of PRINT and must be followed by a number in parentheses. This dictates how many spaces across the screen will be left empty before the PRINT statement which follows is printed.

```
10 PRINT TAB(2);"2"
20 PRINT TAB(7);"7"
30 PRINT TAB(14);"14"
40 PRINT TAB(32);"32"
```

The control variable, the number within the parentheses, can be a variable rather than a number, as the following program demonstrates:

```
10 FOR J=1 TO 24
20 PRINT TAB(J);J
30 NEXT J
```

The term 'variable' and the use of the FOR instruction in line 10 will be discussed in due course. The output from the preceding program is shown at the top of page 8.

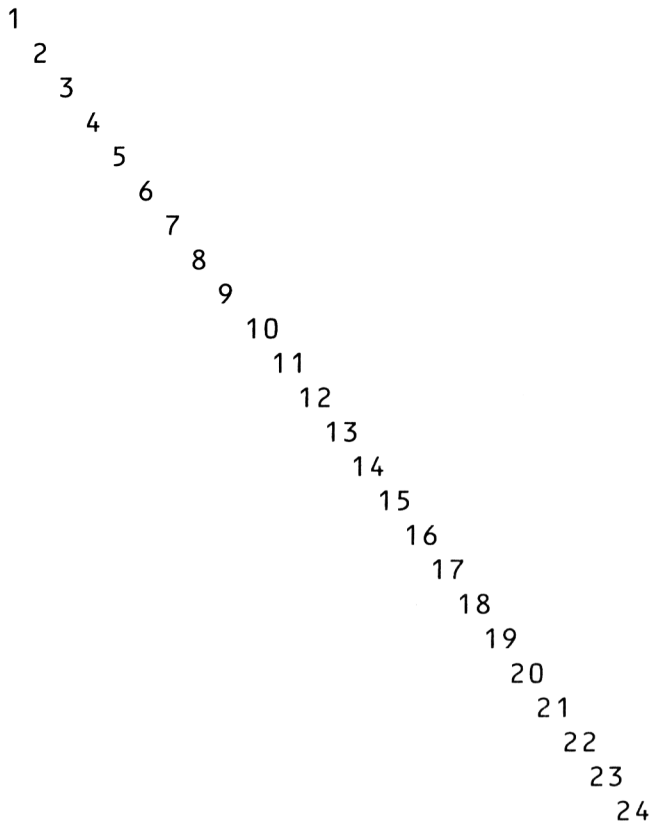
Once this program is in your Amstrad's memory, you can check it by typing in the word LIST and then pressing the ENTER key. Try it . . . and the listing will appear on the screen.

LIST can also be used to get just part of the listing. If you have a program of ten lines, for example, each numbered in tens (so the line numbers would be 10, 20, 30 and so on up to 100), your Amstrad allows you to cause the last four lines of the program to appear on the screen simply by entering:

```
LIST 70 —
```

This lists all the lines following the hyphen; in this example, the computer would list lines 70, 80, 90 and 100. In a similar way, to obtain all the lines up to a particular number, you use the following form:

```
LIST — 40
```



This will list all the lines from the start of the program up to and including the one you have specified (in this case, lines 10, 20, 30 and 40).

If the lines you wish to study are not at the start or the end of the program, you can display them on-screen by entering a command in the following form:

```
LIST 20 — 60
```

This will list lines 20, 30, 40, 50 and 60.

Variables

Nearly all the programs you'll ever write or see written for your computer will use variables. Variables are letters or combinations of

letters and numbers starting with a letter. They are given values during the course of a program execution and are, in effect, those numbers during the run of the program.

To clarify that somewhat mysterious statement, look at the next small program segment:

```
10 A=36
20 B=9
30 C=A+B
40 PRINT A;" + ";B;" = ";C
```

In this program, a value of 36 is assigned to the variable A (and acts as if it were the number 36 right through the program), a value of 9 is assigned to variable B, and the total of A plus B is assigned to variable C (as can be seen in line 40). Variables used in this way are called *numeric variables*. They can change their values during the course of a program, as this example shows:

```
10 A=INT(RND*10)+1
20 B=INT(RND*10)+1
30 PRINT "A IS ";A
40 PRINT "B IS ";B
50 FOR T=1 TO 500:NEXT T
60 PRINT
70 GOTO 10
```

Although a variable can be any letter from A to Z, variable names are not restricted to single letters. Any combination of letters and numbers, so long as it starts with a letter, is an acceptable variable name, so C2PO, R2D and FOXHUNTER are all valid names.

You'll find that the use of explicit variable names can help to make programs easier to understand. For example, in the next program, which works out what percentage a small number is of a large one, the larger number is assigned to a variable called BIGNUMBER, the smaller one to a variable with the name LITTLENUMBER, and the result of the calculating is assigned to a variable called PERCENTAGE.

```
10 BIGNUMBER=761
20 LITTLENUMBER=234
30 PERCENTAGE=LITTLENUMBER*100/BIGNUMBER
40 PRINT "BIG NUMBER=";BIGNUMBER
50 PRINT "LITTLE NUMBER=";LITTLENUMBER
60 PRINT "PERCENTAGE=";PERCENTAGE;"%"
```

As you can see, it is very easy to understand what is happening. You are unlikely to use such long names in your own programs, but there is no reason why you cannot use abbreviated versions of names like those used above. These will still tell you what the variables represent.

Stringing along

So far we have been looking at numeric variables . . . there are also *string variables*. In computer jargon, a string is anything normally enclosed within quote marks in a PRINT statement. It does not necessarily have to be in a PRINT statement, but if it is, and it is a string, it will be within quote marks.

Therefore, the following are strings:

```
"THE SUN IS HOT"
"THE ANSWER TO THE PROBLEM IS"
"I HEARD THE NEWS TODAY, OH BOY"
```

The only difference between the variable assigned to a string, as opposed to that assigned to a number, is that the string variable name ends with a dollar sign. The following are valid string names:

```
A$  FACE$  D123456$  G6H4$
```

Here's a program showing string variables in use:

```
10 N$="NOSE"
20 D$="MY DOG"
30 B$=" NO"
40 C$="HAS"
50 PRINT
60 PRINT D$
70 PRINT C$;B$
80 PRINT N$
```

Note that while the string itself must normally be enclosed within quote marks, the variable name does not have quote marks around it, even when used in a PRINT statement.

You'll notice that numeric variables are widely used within the programs in the first part of this book. String variables are not so common. Strings and string manipulation are discussed in detail a little later on in this book, but you know enough now to recognize them when you come across them.

3
MORE PROGRAMMING

We're now going to add a few more BASIC commands to your programming repertoire. By the end of this chapter, you'll be conversant with all of the following commands: CLS, REM, CONT, GOTO, ON . . .GOTO, GOSUB, ON . . .GOSUB, PAPER, PEN and LOCATE.

Starting off with the simplest command, CLS, this is used to clear the screen. Add a line with CLS at the beginning of one of the programs in the previous chapter and you will be able to see it in operation for yourself.

By now, you must be wondering when the colours (of which the Amstrad boasts 27!) are going to make their entrance. Chapter 15 — A graphic explanation — contains all the details but, in the meantime, here are a few pointers for you to be going on with.

The Amstrad has three graphics modes and, when switched on, the machine defaults to Mode 1 (40 columns by 25 rows of characters with a maximum of four colours). To change the colour of the background, use the PAPER command followed by a number between zero and three. Zero is the defaulted background colour (dark blue) when the machine switches on. Once you've altered the background colour, clear the screen by entering a CLS command.

The PEN command works in a similar way to PAPER, choosing one of the four available colours and making that the colour of the foreground. You do not need to use the CLS command with PEN.

Pens and paper

Here is an example of PEN and PAPER in action; after you have typed in the program, RUN it and see the effect of all the possible combinations of the four colours available on the Amstrad (red, dark blue, light blue and yellow). (You can change these colours using the INK command, which is explained fully in Chapter 15.)

```
10 '  PENS AND PAPER
20 '
30 FOR BACK=0 TO 3
40 FOR FORE=0 TO 3
50 PEN FORE:PAPER BACK
60 CLS
70 PRINT "THIS IS PEN NUMBER ";FORE
80 PRINT
90 PRINT "ON PAPER NUMBER ";BACK
100 FOR B=1 TO 1500:NEXT
110 NEXT:NEXT
120 PAPER 0:PEN 1
```

Looking back at the program above, you can see that the first two lines start with an apostrophe ('), an abbreviated form of the REM command.

The word REM stands for remark. This is a word or phrase which is used within a program to tell those looking at the program what the following section is meant to be doing. REM statements are ignored by the computer, so you can use them to store any information which you feel could help you interpret what sections of the program are doing.

You'll find that REM statements are particularly useful when you return to a program after a long break. You may feel that you will never forget what particular sections of a program do, and this belief will persist until you first try to fathom out the workings of a program which you have not looked at for several days. From that point on, however, it's likely you'll start using REM statements!

Several examples of REM statements are:

```
80 REM END OF GAME, PLAYER TOLD SCORE

100 REM*****TITLE PAGE*****
```

Although you may feel REM statements are a bit of a waste of time with short programs, you'll find they come into their own with long ones. Certainly it is worth getting into the habit of using REM statements.

Horsey Horsey

We'll now be looking at a number of commands, using the same program to show each one in action. The program is called *Horsey*

Horsey, a title which will make much more sense once you've run the program. When you do, you'll see the numbers one to four race each other across the screen, like four tiny horses.

At the beginning of the program, which is intended for two people (although a version in which you play against the computer can easily be created), each player has $30. The two players each bet $5 on their choice of horse to win the race, selecting one of the numbers one to four to win. If the selected horse does win, $10 is paid out to the player who backed it.

The game continues, race after race, until one of the players has no money left. At this point, the solvent player is declared the winner.

```
10 REM  HORSEY, HORSEY
20 X(1)=30:X(2)=30
30 BORDER 9:INK 0,9:INK 1,0
40 INK 2,26
50 PAPER 0
60 CLS
70 FOR T=1 TO 2
80 X(T)=X(T)-5: IF X(T)<1 THEN 620
90 PRINT:PRINT
100 PEN 2
110 PRINT "PLAYER";T;":"
120 PRINT "YOU'VE GOT $";X(T)
130 PRINT:PRINT
140 PRINT "MILL ROOF          = 1"
150 PRINT "OUTSIDE CHANCE = 2"
160 PRINT "RED WINE           = 3"
170 PRINT "BROWN BEAUTY     = 4"
180 PRINT:PRINT
190 PRINT "ENTER THE NUMBER OF THE HORSE YOU
THINK WILL WIN (1-4)"
200 A$=INKEY$
210 IF A$<"1" OR A$>"4" THEN 200
220 A=VAL(A$)
230 Z(T)=A
240 IF T=2 AND Z(1)=Z(2) THEN PRINT "NUMBER 1 HAS
";Z(1):FOR D=1 TO 1000:NEXT:GOTO 90
250 NEXT
260 CLS
270 PEN 3
280 PRINT "HORSEY, HORSEY"
```

```
290 PEN 2
300 PRINT:PRINT "//////////////////////////
///////"
310 LOCATE 1,10
320 PRINT "//////////////////////////////////
/"
330 PEN 3
340 FOR T=1 TO 6:LOCATE 34,T+3:PRINT ";":NEXT
350 LOCATE 1,5
360 PEN 1
370 FOR T=1 TO 4:A(T)=T/32:NEXT
380 FOR B=1 TO 4
390 A(B)=A(B)+INT(RND*2)
400 PRINT TAB(A(B));" ";B;
410 IF B<4 THEN PRINT
420 IF A(B)>32 THEN PRINT:GOTO 470
430 NEXT B
440 LOCATE 1,5
450 GOTO 380
460 SOUND 1,180,100:SOUND 1,120,100
470 CLS
480 PEN 3
490 PRINT "THE WINNER IS ";B
500 PRINT:PRINT
510 IF Z(1)=B THEN PRINT "AND PLAYER 1 GOT IT
RIGHT":X(1)=X(1)+10
520 IF Z(2)=B THEN PRINT "AND PLAYER 2 GOT IT
RIGHT":X(2)=X(2)+10
530 PEN 1
540 FOR T=1 TO 900:NEXT
550 PRINT "STAND BY . . ."
560 FOR T=1 TO 900:NEXT
570 PRINT ". . . FOR A NEW . . ."
580 FOR T=1 TO 900:NEXT
590 PRINT ". . . . . RACE!!!!"
600 FOR T=1 TO 2000:NEXT
610 GOTO 60
620 PAPER 1:PEN 3:BORDER 2
630 PRINT:PRINT "WELL PLAYER ";T
640 PRINT "IS BROKE SO THE"
650 PRINT "WINNER IS PLAYER ";
```

```
660 IF T=1 THEN PRINT 2 ELSE PRINT 1
670 END
```

As this is the first major program you've entered, you may have noticed a number of unfamiliar command words. Some will be explained in the following few pages, others will be explained more fully later in the book. Once you've had some fun with the program, why not sift through the program lines and see if you can work out what some of those unfamiliar words do. Then, return to the text and we'll see if your deductions are correct.

Some more commands

CONT

The first command word we'll look at is CONT, which stands for continue. This is used when for some reason you stop the program and then wish to continue execution. To illustrate this command in action, let's run the program again and, once the race is under way, press the ESC key twice (this is equal to BREAK). An error message will appear (something like BREAK IN 430) and the program stops. Now type CONT and the program will carry on (albeit with the error message still printed on the screen) from where it left off.

LOCATE

The LOCATE command is used a number of times within the *Horsey Horsey* program, particularly in the part of the listing that draws the horse-racing course.

The LOCATE command positions the PRINT cursor. If you have used another computer before, then you may recognize LOCATE as equivalent to PRINT AT or PRINT @. The word LOCATE is followed by two numbers, the horizontal and vertical co-ordinates of the new cursor position.

In Mode 1, the mode we are in at present, the first number (the horizontal figure) can be between one and 40, and the second (the vertical figure) can be between one and 25. The program below shows LOCATE in action:

```
10 CLS
20 PEN 1
30 INK 1,2,24
40 LOCATE 10,10
50 PRINT "X"
60 PEN 2
70 LOCATE 8,23
80 PRINT "X MARKS POSITION 10,10"
```

GOTO

It is pretty obvious what GOTO means and does. You'll recall that earlier it was stated that programs in BASIC tend to be executed from the smallest line number to the highest. GOTO is one of the commands which allow you to break this orderly execution sequence from lowest to highest. When the computer comes across the word GOTO, it goes to the line number indicated.

Harking back to *Horsey Horsey*, part of that program had to be repeated again and again (the routine that drew the horses). The GOTO command in line 450 instructs the program flow to jump back to line 380, the beginning of the part of the program that draws the horses. This type of GOTO is called *unconditional*, because it performs the jump without a condition being fulfilled; soon we will discuss how the computer makes decisions with the IF/THEN statement and you will then be introduced to the *conditional* GOTO.

Jump ahead to the chapter on saving and loading programs (Chapter 11 for cassette, Chapter 12 for disc) and read the first couple of pages to learn how to save programs. Now save the program *Horsey Horsey* on to a blank cassette or disc as we will be returning to it later on in the book.

ON . . . GOTO

This is a variation on the GOTO statement, and is used when you want to go to a series of destinations, depending upon the value held by a particular variable.

Here is a simple program which shows ON . . . GOTO in action:

```
10 REM  ON GOTO DEMO
20 A=INT(RND*4)+1
```

```
30 FOR B=1 TO 500:NEXT
40 ON A GOTO 50,70,90,110
50 PRINT, "ONE"
60 GOTO 20
70 PRINT, "TWO"
80 GOTO 20
90 PRINT, "THREE"
100 GOTO 20
110 PRINT, "FOUR"
120 GOTO 20
```

Line 20 generates the numbers one, two, three or four at random, assigning them to the variable A. Line 40, the important one for this discussion, sends action to line 50 if A equals one (50 is the first number after GOTO), to line 70 if A equals two (70 is the second destination), to line 90 if A equals three and to line 110 if A equals four. Line numbers 50, 70, 90 and 110 spell out the number in full, effectively changing a randomly generated digit into a word.

GOSUB and subroutines

GOSUB, like GOTO, re-directs the program but, instead of just jumping to a line, the GOSUB command orders a jump to a 'subroutine'. A subroutine is a block of code lying within the main program; a well-structured program will be largely made up of subroutines, each one being called by a GOSUB command. At the end of a subroutine, there must be a RETURN command which takes the program flow back to the command following the GOSUB instruction. The program below makes this clearer:

```
10 GOSUB 50:' PRINT TITLE
20 GOSUB 90:' ASK FOR NUMBER
30 GOSUB 120:' PRINT RESULT
40 GOTO 10
50 PRINT:PRINT:PRINT
60 PRINT "NUMBER ADDITION"
70 PRINT:PRINT
80 RETURN
90 INPUT "ENTER THE TWO NUMBERS TO BE ADDED";A
100 INPUT B
110 RETURN
```

```
120 TOTAL=A+B
130 PRINT "THE ANSWER IS ";TOTAL
140 RETURN
```

The main program is nothing more than three GOSUB calls and a GOTO command, the latter directing the computer back to the beginning of the GOSUBs. The three subroutines, however, do all the work: the first, starting at line 50, prints the title, the second asks for the two numbers to be added together, and the final subroutine prints the answer. Note that the REM statements at the end of each GOSUB are optional (as the Amstrad will ignore their content) but they keep the program user informed of what the program does.

The GOSUB command can be used with the ON command in much the same way as ON . . . GOTO except, obviously, the computer jumps to a subroutine instead of a line.

An effect that can be created easily using the GOSUB/RETURN process is simple poetry. All you need to do is write a number of lines, each in a separate subroutine, create a random number between one and the number of poetry lines you've written, and have an ON . . .GOSUB command directing the computer to whichever line has been chosen by the random number.

Inspired by a recent holiday in Austria, you can see the results of my attempts at computer-generated poetry. Not spectacular, I'm afraid . . . but that's no reason why you can't alter the contents of the speech quotes and write some *good* poetry!

Here's an extract from the many poems the Amstrad was happy to compose at random:

```
                    SALZBURG TOWN

TOWERING FORTRESS COPPER DOMES
COBBLED STREETS
MEMORIES IN STONE ECHOED

TOWERING FORTRESS NARROW WALKWAYS
TIMELESS AMONG MOZART'S MUSIC

TOWERING FORTRESS ECHOED
AMONG MEMORIES IN STONE. . . .
DREAMING
MOZART'S MUSIC BAROQUE SPIRES
DREAMING
TIMELESS
```

```
MOUNTAIN AIR . . . LINGERS SOFTLY

TIMELESS
```

And here's the program that allowed the Amstrad to perform such a literary feat:

```
10 REM    POETRY USING ON GOSUB
20 WHILE INKEY$="":WEND
30 CLS
40 PRINT "          SALZBURG TOWN"
50 PRINT
60 RANDOMIZE TIME
70 P=INT(RND*20)+1
80 ON P GOSUB 110,130,150,170,190,210,230,250,
270,290,310,330,350,370,390,410,430,450,
470,490,510
90 FOR T=1 TO 200:NEXT
100 GOTO 70
110 PRINT
120 RETURN
130 PRINT
140 RETURN
150 PRINT
160 RETURN
170 PRINT
180 RETURN
190 PRINT ". . . ."
200 RETURN
210 PRINT "COPPER DOMES ";
220 RETURN
230 PRINT "MOZART'S MUSIC ";
240 RETURN
250 PRINT "LINGERS SOFTLY "
260 RETURN
270 PRINT "MOUNTAIN AIR ";
280 RETURN
290 PRINT "COBBLED STREETS "
300 RETURN
310 PRINT "TIMELESS ";
320 RETURN
330 PRINT "BAROQUE SPIRES "
340 RETURN
```

```
350 PRINT "DREAMING ";
360 RETURN
370 PRINT "ECHOED ";
380 RETURN
390 PRINT "NARROW WALKWAYS ";
400 RETURN
410 PRINT "SHADOWED OVER ";
420 RETURN
430 PRINT "TOWERING FORTRESS ";
440 RETURN
450 PRINT "AMONG ";
460 RETURN
470 PRINT "WANDERING SLOWLY ";
480 RETURN
490 PRINT "MEMORIES IN STONE ";
500 RETURN
510 PRINT "CHURCHES ";
520 RETURN
```

4
LOOPING THE LOOP

We will now begin to look at some commands that always appear together: FOR and NEXT, and WHILE and WEND.

Essentially, FOR and NEXT control a loop which is executed the number of times specified in the FOR statement. The following program fragment should help make this clear:

```
10 REM FOR/NEXT
20 FOR A=1 TO 10
30 PRINT "HERE IS NUMBER.";"OF THE LOOP"
40 NEXT A
```

When you run it, you'll get this result:

```
HERE IS NUMBER. 1 OF THE LOOP
HERE IS NUMBER. 2 OF THE LOOP
HERE IS NUMBER. 3 OF THE LOOP
HERE IS NUMBER. 4 OF THE LOOP
HERE IS NUMBER. 5 OF THE LOOP
HERE IS NUMBER. 6 OF THE LOOP
HERE IS NUMBER. 7 OF THE LOOP
HERE IS NUMBER. 8 OF THE LOOP
HERE IS NUMBER. 9 OF THE LOOP
HERE IS NUMBER. 10 OF THE LOOP
```

The two statements HERE IS NUMBER. and OF THE LOOP are printed ten times with the value of A printed between them. Once you've run it a couple of times, change line 40 to read as follows:

```
40 NEXT
```

All you have done is remove the control variable A; your Amstrad will still understand this — even if you have a number of FOR/NEXT loops, it will still know which NEXT to pair up to which FOR, providing that you have included the correct number of each in the program. For this reason you may prefer to leave the control variable on the end of the NEXT as a reminder.

You can nest loops (that is, have one or more inside the other) if needed. Here is an example of one loop nested inside another — the B loop is nested within the A loop. This routine prints out the multiplication tables from one times one to 12 times 12:

```
10 REM   NESTED LOOPS
20 FOR A=1 TO 12
30 FOR B=1 TO 12
40 PRINT A;" TIMES ";B;" = ";A*B
50 NEXT B
60 NEXT A
```

It is important to ensure that the last NEXT statement that occurs in a program has the same control variable (the letter A, B or whatever) as the first FOR in the program. As you can see in the program above, the first FOR statement refers to the control variable A, as does the last NEXT. The next control variable mentioned in the program is B, and the second last NEXT also refers to B. It is vital that you arrange your FOR and NEXT controls in this way . . . or you'll find the computer will get a little hot under the collar. You can demonstrate this by changing lines 50 and 60 to the following:

```
50 NEXT A
60 NEXT B
```

This will produce an error message from your machine. One way to get around this is to remove the control variables from the NEXTs and let the computer sort them out for itself. In this case, the final two lines of the program would read:

```
50 NEXT
60 NEXT
```

Another way of doing it is to delete line 60 (which you do by typing in 60, with nothing following, then pressing the ENTER key) and change line 50 to:

```
50 NEXT B,A
```

As you'd expect, the program works exactly the same in all cases, except when you get the control variables in the wrong order. You'll find the Amstrad works fractionally quicker when it does not have to check the correctness of the control variables. Therefore, the use of NEXT, without a control variable, is probably the simplest way to use FOR/NEXT loops. However, you'd be best advised — at the begin-

ning of your acquisition of programming skills — to stick to the version given in the first program, using NEXT B and NEXT A. This will help you keep track of what is going on within the program.

FOR/NEXT . . . STEP

There is another aspect of FOR/NEXT loops which must be discussed, and that is the use of STEP. Clear the computer's memory by typing in NEW, then pressing the ENTER key, and enter the following program:

```
10 REM  FOR/NEXT . . . STEP
20 FOR A=1 TO 20
30 PRINT A
40 NEXT A
```

When you RUN this, you'll see the numbers one to 20 printed on the screen. Now, change line 20 so it looks like this:

```
20 FOR A=1 TO 20 STEP 2
```

When you run this, instead of the computer printing out 1, 2, 3, 4, 5, 6 . . . and so on down to 20, you'll see 1, 3, 5, 7, 9 . . . down to 19. The computer has started counting from the first number quoted in the control statement (FROM A = 1 TO . . .) to the closest it can get to the final number, counting up in twos. Change line 20 as shown below and RUN the program again:

```
20 FOR A=1 TO 20 STEP 3
```

This time you should see 1, 4, 7, 10, 13, 16 and 19.

So from this we can learn that FOR/NEXT loops will step up in ones, unless another STEP size is specified. The STEP of one is the default case, which occurs when no other STEP size is specified.

The computer can also count downwards. There is no reason why a FOR/NEXT loop must always go upwards. Try the following:

```
20 FOR A=20 TO 1 STEP-1
```

The STEP size does not have to be a whole number either, as you'll discover if you change line 20 to:

```
20 FOR A=20 TO 1 STEP-0.7
```

It is time now to return to *Horsey Horsey* (see Chapter 3 — More programming) to look at the FOR/NEXT loops in it. Either load the program back into your computer or refer to the listing given in the previous chapter.

The first loop, a 'T' loop, starts in line 70 and ends in line 250.

The first action within the loop is to subtract five dollars from each of the players' stakes, then — still within the loop — the players' bets for the winning horse are taken. Line 240 rejects the second player's chosen horse if it is the same as the first player's.

One of the next FOR/NEXT loops is held in line 370. Notice that this also uses a 'T' as control variable. It is perfectly acceptable to use the same letter for different loops within a program, so long as the loops are not contained within each other. Because there is a slight bias towards horses with lower numbers (as they are checked in numerical order to see if they have crossed the finish line, and if horses one and four have both crossed this line, horse one will be awarded the race because it is checked first), the loop in line 370 gives the horses with higher numbers a slight edge at the start of the race.

The race proper is held within the loop, with control variable B, from line 380 to 430. This runs over and over again, adding a little (line 390) to each horse's total until one of them (see line 420) has a total which is greater than .32, when action is transferred to line 470 where the winner is proclaimed. Lines 510 and 520 check to see if either player has picked the winning horse and, if so, $10 is added to the lucky punter's stake. Line 530 uses another FOR/NEXT loop for a brief delay, and so do the following sections, gradually printing up "STAND BY . . . FOR A NEW . . . RACE!!!!". Another delay loop, line 600, holds things still for a moment and then action goes to line 60 to start the new race.

WHILE and WEND

WHILE and WEND are similar commands to FOR and NEXT in that they control a loop . . . but they do it in a slightly different way. At the beginning of a WHILE/WEND loop lies the WHILE command, and following the command is a condition. At the end of the loop is the WEND command. While the condition after the WHILE command is true, the program continues to loop; however, once the condition is

false, the program continues past the WEND command at the bottom of the loop.

Here is an example of WHILE/WEND in use:

```
10 REM WHILE/WEND DEMO
20 TOTAL=0
30 WHILE TOTAL<>20
40 TOTAL=TOTAL+1
50 PRINT TOTAL
60 WEND
```

Note that the computer continues to add one to the variable T, until T is equal to 20 and the loop is ended.

WHILE/WEND can be very useful — one simple but effective use is to let the computer pause in the execution of the program until a key is pressed; the instruction for that is WHILE INKEY$="":WEND. Therefore, while no key is pressed (the INKEY$ command is still equal to a null string) the computer loops endlessly.

Code maker

Here's a slightly more complex use of WHILE and WEND:

```
10 ' CODE MAKER
20 '
30  INPUT  "ENTER  YOUR  SINGLE  CHARACTER
CODE";CODE$
40 CLS
50 PRINT " Amstrad 64K Microcomputer (v1)"
60 PRINT
70 PRINT " "+CHR$(164)+"1984 Amstrad Consumer
Electronics plc and Locomotive Software Ltd."
80 PRINT:PRINT " BASIC 1.0"
90 PRINT:PRINT "Ready"
100 PRINT CHR$(143)
110 WHILE A$<>CODE$
120 A$=INKEY$
130 WEND
140 SOUND 1,300,40
150 PRINT "YOU ARE NOW IN THE CPC 464 SYSTEM."
160 END
```

In this program, you type a single character code including the control characters. The Amstrad sets up the screen to look exactly like that when it is first powered up. Until the user presses the right code key, he or she cannot use the computer — unless they press the BREAK key. (In Chapter 22 — How to program more professionally — we'll discuss the POKEs to disable the BREAK key completely.)

The WHILE condition occurs when the user's input does not equal the code of the character it's been pre-programmed to accept; when it does, the computer finishes the loop.

Note that in the last line, line 160, use has been made of the END command. This command, as you might have expected, terminates a program and is similar to the STOP command, except that STOP creates a BREAK in the program from which one can continue using the CONT command. Replace END with STOP in line 160, run the program and see the difference between the two program termination commands.

A final point. Tinkering around with the Amstrad has thrown up CALL &BB18, which simulates the command WHILE INKEY$="":WEND. No key is registered, but if you want to have a 'press any key' pause line, there is no neater or quicker way than to use this CALL statement. A word of warning, though — please be careful not to use CALL BB18 as this seems to cause a system reset; in other words, the computer looks as though it has just been switched on.

5
CONTINUING UP THE LEARNING CURVE

Let's look back over the ground we've covered so far. As you can see, there is a great deal we have already done. By now, you should have a fair working knowledge of the following words, and may well be able to manipulate them with a degree of facility:

PRINT, LIST, NEW, RUN, TAB, CLS, REM, GOSUB, CONT, GOTO, FOR/NEXT, ON. . .GOTO, ON. . . GOSUB, WHILE, WEND, FOR/NEXT. . .STEP, PEN, PAPER, INK, END and STOP.

As well as these words, you've learned about such things as program lines, numeric and string variables and how to use a semi-colon in PRINT statements.

We've come a considerable distance already and, in many ways, the worst is now over! The 'learning curve' for BASIC is a gentle, upward slope, with the steepest part representing your first few hours of learning. Once you have those first few hours behind you — which you should have at this point — you'll discover that the additional things you learn interlock neatly with the material you've already mastered. This makes the learning process a satisfying, and not too demanding, one.

Multi-statement lines

We've used these in some of our programs already. A multi-statement line is like the following:

```
10 FOR T=1 TO 20:PRINT T:PRINT T/2:NEXT T
```

As you'll see if you run this, it is a complete program in one line and takes the place of the following four lines:

```
10 FOR T=1 TO 20
20 PRINT T
30 PRINT T/2
40 NEXT T
```

Running this four-line program will demonstrate that it does exactly the same as the multi-statement line.

In general, you'd be well advised to stay away from multi-statement lines, as they make errors harder to spot and can make tracing through a program very difficult. There are also traps. For example, in the next two lines, the second statement (following the colon, which is almost always used to separate statements within a single program line) will never be executed:

```
10 GOTO 50:A=25
10 REM END OF PROGRAM:PRINT "GOODBYE HUMAN"
```

The use of single or multi-statement lines is, in large measure, a question of taste.

There are times when multi-statement lines are desirable, and even essential. For example, they are desirable when spacing out PRINT statements:

```
10 PRINT "EAGLES"
20 PRINT:PRINT
30 PRINT "FLYING"
40 PRINT:PRINT
50 GOTO 10
```

You can also use them when you use a 'dummy' FOR/NEXT loop to introduce a delay:

```
10 PRINT "THE END IS NIGH"
20 FOR T=1 TO 1500: NEXT T
30 END
```

A few functions

By now, you've realized that the programs we've looked at so far include many BASIC words that have not been explained. This is because, after a certain point, it becomes increasingly difficult to avoid assuming certain words are known, especially if worthwhile programs are to be introduced. Some of these commands are mathematically based and will be familiar to anyone who has set foot in a Maths lesson. Others may not be quite so familiar, but *all* are worth a few minutes of your time.

Mathematical functions

ATN
This function returns the arctangent and is used in the following form: A = ATN(n). It returns an answer in radians and works within the range minus PI/two to PI/two.

COS
This instruction returns the cosine and is used as follows: C = COS(n). It returns the answer in radians.

DEG
This function can be used totally on its own, as in 20 DEG; this sets any calculation involving SIN and COS (such as the drawing of a circle) into degree mode and out of RAD or radians mode.

EXP
This function is connected with the LOG instruction as it transforms a logarithm (created using LOG) back to its original value. For example:

```
PRINT LOG(10) 2.30258509
PRINT EXP(2.30258509) 9.99999996
```

LOG
This function returns the natural logarithm of a number, and is used in the form A = LOG(n).

LOG10
The LOG10 function returns the base 10 logarithm of a number.

PI
There are very few people who haven't heard of PI, a figure of magnitude 3.14159265 (that's as far as your Amstrad goes). It is extensively used for various calculations involving circles and curved lines.

RAD
The RAD function changes calculations back into the radians mode. Note that the default mode between degrees and radians is radians.

SIN

The SIN function provides the sine of an angle, and is used in the following form: S = SIN(n). This also returns the answer in radians.

SQR

Pretty straightforward this one: SQR calculates the square root of a figure.

TAN

The tangent of a number is returned by this function, and is used as follows: T = TAN(n). Again, the function returns an answer in radians.

Figurative functions

The Amstrad is blessed with a number of functions that act on a figure, cutting the number of digits or rounding it to a specific value.

ABS

ABS takes the absolute value of a number; that is, it refers to the number but it doesn't bother with the sign. This can be useful when assessing the difference between two variables.

CINT

The CINT function converts real numbers into integers, but rounds to the nearest figure. Thus, CINT(2.7) would be 2. Note that .5 of a figure (as in 3.5) would be rounded up to a whole number.

CREAL

This instruction creates a real number from a numeric value.

FIX

This function only keeps the number to the left of the decimal point, so FIX(74.9327) would return 74. In effect, FIX almost performs the same function as INT.

INT

Probably the most used of all these functions, INT makes an integer (a whole number without any decimal places) out of a 'real' number — that is, a number which has a decimal point. Therefore, INT(203.6) would return 203; note that INT always rounds down.

MAX

If you enclose a list of numeric expressions in brackets, the MAX function picks out the largest. Therefore, if a statement read: PRINT MAX (23,47,6,9,8,107,56,88), the result would be 107.

MIN

The MIN function performs the exact opposite to MAX in that it finds the smallest number from a list. (Note that in both functions, variables can be used instead of numbers.) For example, MIN (A,B,C,107,D), where A=90, B=109, C=34 and D=77, would result in C, the smallest, being chosen.

ROUND

The ROUND function rounds a number in the same way as CINT, but you can specify the number of decimal places you would like the figure to be rounded to. Therefore, ROUND (2.38769,3) would produce the result 2.388 (due to the 3 after the comma in the bracket).

SGN

This function determines the sign of a variable. If a variable is greater than zero, it returns a 1, and if the variable is equal to zero, then it returns 0; if the variable is less than zero, then a −1 is returned.

UNT

This function converts an unsigned 16-bit integer, such as &BB18, to an 8-bit integer in the numeric range between −32768 and 32767. Its main use is the conversion of address numbers from 16-bit to 8-bit integers. So, a command such as CALL &BB18 would also work as CALL −17640, as this is the figure received when PRINT UNT(&BB18) is entered.

Here is a little demonstration program showing most of these functions in use:

```
10 ' FUNCTIONS DEMONSTRATION
20 A=-10
30 B=7.894325
40 C=3498.5
50 D=29.3786
60 CLS
70 PRINT TAB(15);"FUNCTIONS."
```

```
80 PRINT
90 PRINT "THE LARGEST IS ";MAX(A,B,C,D)
100 PRINT
110 PRINT "THE SMALLEST IS ";MIN(A,B,C,D)
120 PRINT
130 PRINT "THE ABSOLUTES ARE":PRINT
ABS(A);"";ABS(B);" ";ABS(C);" ";ABS(D)
140 PRINT
150 PRINT "THE SIGN VALUES ARE ";SGN(A);
SGN(B);SGN(C);SGN(D)
160 PRINT
170 PRINT "CINT OF A,B,C,D":PRINT CINT(A);
CINT(B);CINT(C);CINT(D)
180 PRINT
190 PRINT "INT OF A,B,C,D":PRINT INT(A);INT
(B);INT(C);INT(D)
200 PRINT
210 PRINT "FIX OF A,B,C,D":PRINT FIX(A);FIX
(B);FIX (C);FIX(D)
220 PRINT
230 PRINT "ROUND TO TWO DECIMAL PLACES":PRINT
ROUND (A,2);ROUND (B,2);ROUND(C,2);ROUND
(D,2)
240 END
```

Defining your own functions

The DEF FN command allows you to define your own functions. This can be very useful for a complicated calculation used a number of times within a program. The syntax for the DEF FN command is:

DEF FN name (formal parameters) = a numeric expression

An example of defining a function can be seen from the following program, which shows you what you'll have to pay on a loan:

```
10 ' DEFINING FUNCTIONS
20 '
30 DEF FN TOTAL(PRINCIPAL)=((PRINCIPAL*I)/
100)+PRINCIPAL
40 INPUT "INTEREST";I
50 INPUT "PRINCIPAL";PRINCIPAL
60 PRINT "TOTAL TO BE PAID";FNTOTAL(PRINCIPAL)
```

The three commands DEF INT, DEF REAL and DEF STR$ are all forms of function definition. The first of these, DEF INT, allows you to define certain variables as integers. At present, all your numeric variables, A, ZZT, SCORE and so on, are real numbers — in other words, they can have decimal places. Many of the variables you use only need to be integer variables and, for the computer, calculating and manipulating integer variables is a faster and more efficient process.

Another way of creating integer variables is to follow the name of the variable with a percentage sign. However, this has to be maintained throughout the program. It really is much simpler to define variables as integer variables at the start of the program.

The syntax for this command is DEFINT followed by the names of the variables separated by commas. If you wish for a whole set of letters to be used as integer variable names then, instead of typing each one in, you can use a command such as DEF INT A – X. All the variables between and including those letters are made into integer variables.

DEF REAL works in the opposite way to DEF INT, making the variables listed after the DEF REAL into real numbers. And lastly, DEF STR$ converts numeric variables to string variables.

There are a couple of points to watch out for before we leave the process of defining functions. Firstly, although within this text the two halves of the DEFining function words have been shown separated with a space, this is purely to make it easier for you to read — do *not* use a space in your own programs. Secondly, none of these commands can be entered in direct mode — all have to be used within a program.

6
READ/DATA AND RESTORE

This is an extremely useful trio of words, usually found together within a program. Basically, the READ command is used to access information stored in a DATA statement, and RESTORE is used to indicate where — in a series of DATA statements — we wish to start accessing the information.

This program should make it clear:

```
10 REM READ/DATA DEMO
20 FOR A=1 TO 10
30 READ B
40 PRINT B
50 NEXT A
60 RESTORE
70 FOR C=1 TO 800:NEXT C
80 PRINT
90 FOR A=1 TO 10
100 READ B
110 PRINT B
120 NEXT A
130 DATA 12,76,85,333,9,23,43,65,88,28 93
```

When you run this, you'll see the numbers in the DATA statement, line 130, printed out one by one as the computer goes through the first A loop, then again as it goes through the second one.

When you run a program, it automatically RESTOREs to the first DATA statement in the program. When the Amstrad comes across a RESTORE statement which is not followed by a number, it RESTOREs to the first DATA statement in the program.

Here is a program to show RESTORE in action directly:

```
10 FOR A=1 TO 10
20 READ B
30 PRINT B
40 IF A=5 THEN RESTORE 70
```

```
50 FOR C=1 TO 50:NEXT C,A
60 DATA 1,2,3,4,5,6,7,8
70 DATA 11,12,13,14,15,16,17,18
```

When you run this, you'll see it print out the numbers 1, 2, 3, 4, 5, 11, 12, 13, 14 and 15. The RESTORE moved the data pointer to the start of line 70, and the computer started reading data from this point. (Note that it doesn't matter how many DATA items you have, so long as there are at least as many as the number to be read.)

There is one thing about READ/DATA which you may find a trifle surprising. It does not matter where in the program the DATA is scattered — the program will find the DATA items and READ them. Check through the following program:

```
10 DATA 1,2
20 FOR A=1 TO 11
30 DATA 3,4,5
40 READ B
50 DATA 6,7
60 PRINT B
70 DATA 8,9,10
80 NEXT A
90 DATA 11
```

So far in this discussion we've been reading numeric items of data, and matching up variables with numbers within a DATA statement. Now, we move on to strings, as the next program shows:

```
10 FOR A=1 TO 5
20 READ H$
30 PRINT H$
40 NEXT
50 DATA "HI","THERE","YOU",
 "CLEVER","MACHINE"
```

Quotation marks have been placed around the characters, but these are not strictly necessary; you only really need them if you want to include punctuation within a string data statement, such as "HELLO, THERE,".

Numeric and string information can be mixed within a program, so long as you ensure that the computer comes to a number when it expects and needs one, and to a string when it needs one. The next program, for example, manages to provide the right kind of information at the right time:

```
10 FOR A=1 TO 5
20 READ H$,B
30 PRINT H$,B
40 NEXT
50 DATA THIS,1,IS,2,A,3,SHORT,4,DEMO,5
THIS        1
IS          2
A           3
SHORT       4
DEMO        5
```

To end our discussion of READ/DATA/RESTORE, here is a more enjoyable use of data statements. In this simple version of *Hangman*, you must guess the mystery word by entering a letter you think might be contained in it. You can get up to seven letters wrong before you lose. If you know the solution, type the whole word in and the Amstrad will tell you if you are correct.

Before you type the program in, there are a couple of points worth mentioning. First off, if you want to construct a different vocabulary, all you have to do is alter the content of the DATA statements. Also, note how the program has been heavily structured — we'll get back to this later on. For now, have a good time with this program and try not to get hanged!

```
10 ' SIMPLE HANGMAN
20 '
30 GOSUB 370
40 GOSUB 310
50 GOSUB 230
60 IF AT>6 THEN 100
70 IF G$=A$ THEN 160
80 AT=AT+1
90 GOTO 40
100 ' LOSE
110 SOUND 1,2000,200
120 CLS
130 LOCATE 12,12:PRINT "YOU LOSE"
140 PRINT:PRINT " THE WORD WAS ";A$
150 END
160 ' WIN
170 CLS
180 FOR T=200 TO 80 STEP -20
```

```
190 SOUND 1,T
200 NEXT
210 LOCATE 10,12:PRINT "WELL DONE, YOU DID IT!"
220 STOP
230 ' PLAYER'S INPUT
240 LOCATE 1,20
250 INPUT "YOUR LETTER TO GUESS (TYPE WHOLE WORD
INIF YOU THINK YOU KNOW)";G$
260 IF G$="" THEN 250
270 FOR T=1 TO LEN(A$)
280 IF MID$(A$,T,1)=LEFT$(G$,1) THEN
 MID$(B$,T,1)=MID$(A$,T,1)
290 NEXT
300 RETURN
310 ' PRINT DISPLAY
320 CLS
330 PRINT:PRINT " SIMPLE HANGMAN"
340 PRINT:PRINT
350 PRINT B$
360 RETURN
370 ' CHOOSE WORD
380 AT=0
390 RESTORE
400 FOR T=1 TO INT(RND*15)+1
410 READ A$
420 NEXT
430 FOR T=1 TO LEN (A$)
440 B$=B$+"-"
450 NEXT
460 RETURN
470 DATA "ALPHABET","PREFABRICATED",
"RIDICULOUS"
480 DATA "TYPEWRIER","INDIVIDUAL",
"CONSOLIDATE"
490 DATA "METAPHYSICAL","ENTERPRISE",
"RELEVANT"
500 DATA "SITAR","OBOE","SEQUEL",
"TRAPEZIUM","CRYPT"
510 DATA "TYRANT","DEMOCRACY","FRANCHISE"
520 DATA "DETERMINED","PUBLISHER"
```

DIM/ERASE

The DIM statement is used when you wish to set up an array to hold a list of numbers, which can then be accessed by referring to a particular element of the array.

Here's an example which should make it clear. For an array to hold numbers, we use a statement in the form DIM A(20) where we want the A array to hold 21 elements. (The number of elements in an array is nearly always one more than the number specified in parentheses in the DIM statement.) Each element of an array is filled with a zero when you first DIMension the array.

The next program DIMensions an array in line 10 and then, using READ statements, will fill each element of the array with a number. The final section of the program will print out each element, in a way that should make it clear what is happening. Notice that the first element of the array is A(0), the second is A(1) . . . and so on, up to A(20).

```
10 DIM A(20)
20 FOR B=0 TO 20
30 A(B)=INT(RND*10)+1
40 NEXT B
50 FOR B=0 TO 20
60 PRINT "A(";B;") IS ";A(B)
70 NEXT B
```

An array is very useful if you wish to choose from a number of possibilities at random . . . as this program shows:

```
10 DIM A(5)
20 FOR B=0 TO 5
30 READ A(B)
40 NEXT B
50 PRINT A(INT(RND*6))
60 FOR C=1 TO 300:NEXT
70 GOTO 50
80 DATA 999,2345,1111,2,9,42
```

The arrays we have looked at so far have been one-dimensional. A multi-dimensional array is set by following the letter name of the array with more than one subscript. Type in and run the following program, and check its output:

```
10 DIM A(4,4)
20 FOR B=0 TO 4
30 FOR C=0 TO 4
40 A(B,C)=INT(RND*9)
50 NEXT C,B
60 FOR B=0 TO 4
70 FOR C=0 TO 4
80 PRINT B;C;"〉"A(B,C)
90 NEXT C,B
```

```
0 0 〉 1
0 1 〉 7
0 2 〉 1
0 3 〉 3
0 4 〉 5
1 0 〉 1
1 1 〉 0
1 2 〉 0
1 3 〉 5
1 4 〉 7
2 0 〉 0
2 1 〉 0
2 2 〉 2
2 3 〉 5
2 4 〉 6
3 0 〉 7
3 1 〉 7
3 2 〉 8
3 3 〉 0
3 4 〉 6
4 0 〉 1
4 1 〉 4
4 2 〉 8
4 3 〉 1
4 4 〉 6
```

Dimensioning the array with the DIM command placed at the beginning of these programs shows clearly the dimensioning process. However, if the size of the array is less than 10 and is a single dimension, then there is no need for it to be DIMmed in the first place; the Amstrad sets up memory for all arrays of under 11 elements.

The Amstrad also provides for string arrays, with a dollar sign in the dimension statement and in subsequent references to the array. The presence of this dollar sign indicates that it is a string, and not a numeric, array. In the next program, a string array is set up in line 10, filled with the loop from lines 20 to 40, and elements are selected randomly by line 50 from the data statements in lines 80 and 90. Type it and see:

```
10 DIM A$(5)
20 FOR B=0 TO 5
30 READ A$(B)
40 NEXT B
50 PRINT A$(INT(RND*6))
60 FOR C=1 TO 250:NEXT C
70 GOTO 50
80 DATA "THESE","ARE","WORDS"
90 DATA "TO","BE","MIXED"
```

Finally, what happens if you have a large array using up a lot of memory and then, halfway through the program, you find you do not need it any more? Well, with Amstrad BASIC, you have a command called ERASE which simply wipes out the array. To erase an array, simply type the name of the array after the ERASE command. Thus, if you want to get rid of both A$(250) and LT(20,40), you would simply enter (as a program line) ERASE A$,LT.

7
RANDOM NUMBERS AND DECISIONS

We've already come across random numbers in this book (take another look at *Horsey Horsey* in Chapter 3). Random numbers, as their name suggests, are generated randomly by the computer. This can be a little misleading as the computer, in fact, chooses numbers from a random numbers table embedded in its memory. To prevent it from using the same area of the random numbers table all the time, we use the RANDOMIZE command followed by some variable. To get the maximum variability, it's worthwhile to make use of the timer, which will randomize the table according to how long the computer has been switched on — a most random variable. Thus, in any programs that use random numbers consistently throughout, the statement RANDOMIZE TIME should be placed at the beginning. (To save memory space, one can add this command directly. This has been done with most of the programs in this book — so if you do not see a RANDOMIZE TIME statement at the beginning of the program, then type it in directly yourself. This will only apply to the games, graphics and sound programs in this book.)

Let's get on to the random number generator itself. The command to create random numbers is known as RND. The command on its own will produce random numbers between zero and one. Type in the two-line program below and let it run for a couple of seconds. You should get similar results to those shown below the program.

```
10 PRINT RND;
20 GOTO 10
```

```
0.498153843 0.42731833 0.668241901
0.343849098 0.564465723
0.326045161 0.893792276
0.362325066 0.439642927
0.122611851 0.672479611 0.7077513
0.940835332 0.977650296
0.112478944 7.18285E-02 0.11415476
```

```
0.433296448 0.35347853 0.53099628
0.699198294 0.37414308 0.728590847
0.935379719 0.439854549
0.473757726 0.593759335
0.875342079
```

These numbers are not going to be much use on their own — what is most needed for random numbers are integers. Fractions can be used — a line such as IF RND ⟩.5 THEN PRINT "YOU ARE DEAD!" gives a player in a game a fifty–fifty chance of survival; and fractions of numbers generated at random are often used in computer decision-making, particularly in games and simulation — but whole numbers between set limits are easier to work with. (By the way, don't worry too much about the IF/THEN command used in the programming example; IF/THEN is the main computer decision-making device and we will be dealing with it later on in this chapter.)

How, then, do we obtain integer random numbers between two specified limits? If we wanted to simulate the score of a football match we would set the limits for each team from zero to about seven (for a realistic answer). There are two things that we now have to do: firstly, we have to set the random number generator to produce numbers between zero and seven (this is done by multiplying the RND generator by seven); and secondly, the resultant figures must be whole numbers — this is done by using the INT function. However, INT rounds fractions down so that the chances of obtaining a seven as a random number is almost non-existent. Therefore, we have to bump up the number that RND is multiplied by from seven to eight. The final format for the RND command is

```
PRINT INT(RND*8);INT(RND*8)
```

Now, say, we had to modify the random number generator to make sure that both teams scored at least one goal. We would have to guarantee that the random number will be between one and seven. All we have to do is add one to the random number and take one off the number that is being multiplied by RND. The following statement is the correct one:

```
PRINT INT(RND*7)+1;INT(RND*7)+1
```

An easy-to-remember rule is that the number added on to the RND statement is that from which the range of random numbers start from. The number that is multiplied with the RND function is the size of the range.

Dice rolls

Random numbers can be used to emulate randomly occurring events in 'real life', such as the result of tossing a coin a number of times. Dice throws produce, if the dice are unbiased, random numbers between one and six, and it is very easy to get the computer to emulate the throw of a single die. Run the following program to illustrate this:

```
10 A=INT(RND*6)+1
20 CLS
30 ON A GOSUB 60,80,100,120,140,160
40 WHILE INKEY$="":WEND
50 GOTO 10
60 PRINT:PRINT "@"
70 RETURN
80 PRINT "@":PRINT:PRINT " @"
90 RETURN
100 PRINT "@":PRINT " @":PRINT " @"
110 RETURN
120 PRINT "@ @":PRINT:PRINT "@ @"
130 RETURN
140 PRINT "@ @":PRINT " @":PRINT "@ @"
150 RETURN
160 PRINT "@ @":PRINT "@ @":PRINT "@ @"
170 RETURN
```

Many dice games require you to throw two dice at once, and then add the pips. At first thought, you might feel the way to do this with the computer would be to get it to generate random numbers between 1 and 12. However, it does not work that way. Although the chance of a single die falling with any face showing is one in six (around 17 per cent), the distribution of totals produced by two dice is as follows: total 2 (2.77 per cent), total 3 (5.55 per cent), total 4 (8.33 per cent), total 5 (11.11 per cent), total 6 (13.88 per cent), total 7 (16.66 per cent), total 8 (13.88 per cent), total 9 (11.11 per cent), total 10 (8.33 per cent), total 11 (5.55 per cent) and total 12 (2.77 per cent).

The different numbers are due to the number of different ways a particular total can be reached. Seven, for example, can be reached in six different ways (1 + 6, 2 + 5, 4 + 3, 3 + 4, 5 + 2 and 6 + 1), while a total of two or of twelve can only be reached in one way (1 + 1 or 6 + 6).

If a random number generator is working properly, and it is used to demonstrate how two dice might fall, we would expect the distribution of the totals to approach the theoretical distribution given in the preceding paragraph. We'll test it to see. Here is a program to throw two dice, followed by the result of a trial. Note that in this case the dice are thrown 100 times.

```
10 REM THROWING DICE
20 DIM A(12)
30 FOR B=1 TO 100
40 C=INT(RND*6)+1
50 C=C+INT(RND*6)+1
60 A(C)=A(C)+1
70 NEXT
80 FOR B=1 TO 12
90 PRINT B;"-";A(B);"%"
100 NEXT
```

```
1 - 0 %
2 - 4 %
3 - 7 %
4 - 9 %
5 - 4 %
6 - 15 %
7 - 20 %
8 - 13 %
9 - 11 %
10 - 9 %
11 - 6 %
12 - 2 %
```

As you can see, the distribution of totals is fairly close to that predicted. Certainly, the percentage scores show a bulge around the seven total.

The program was modified to throw the dice 1000 times, with the idea that the total should more closely approach that which was predicted. Here is the result of throwing a pair of dice 1000 times:

```
1 - 0 %
2 - 4 %
3 - 6 %
4 - 10 %
5 - 10 %
```

6 – 12 %
7 – 15 %
8 – 15 %
9 – 11 %
10 – 7 %
11 – 6 %
12 – 3 %

As you can see, this is closer (as expected) to the theoretical distribution.

Decisions, decisions

As the IF/THEN statement has been included in a couple of examples already, you may have got the gist of this most useful program command, in which case you may want to skip this part and go on to the IF/THEN . . . ELSE command.

IF/THEN is a conditional decision-maker — IF the condition is fulfilled in the statement THEN do something. For example, consider the program line IF A=5 THEN GOTO 200. This roughly translates to 'if the variable A is equal to five then jump to line 200 of the computer program'; this would be known as a *conditional* GOTO as opposed to the unconditional GOTO discussed in Chapter 3.

IF/THEN is the one command that allows the computer to act as if it is thinking for itself and performing rational actions based on its own decisions. Let us now move on to the commands that can be used with the IF/THEN statement. Firstly, let's deal with the computer making comparisons — using the three commands: AND, OR and NOT.

Making comparisons

The Amstrad can use statements such as AND, OR and NOT to compare two or more things, and can act on the result of that comparison.

AND

If two statements to be checked are joined by an AND statement, then the Amstrad will act only if both statements are true.

```
10 A=100
20 B=200
30 IF A=100 AND B=200 THEN PRINT "YES"
```

OR

Two statements joined by an OR will lead to a 'true' finding if either of them is true.

```
10 A=100
20 B=200
30 IF A=100 OR B=100 THEN PRINT "YES"
```

AND and OR can be used in the same program line, as this next example shows. In this program, the Amstrad will report 'YES' if A and B are greater than 10 or if A equals B. Try it with a number of sample values for A and B, until you are sure you understand what is going on.

```
10 INPUT "A";A
20 INPUT "B";B
30 IF A>10 AND B>10 OR A=B THEN PRINT "YES"
40 GOTO 10
```

NOT

This is used to produce, in effect, an 'opposite' condition to be tested, as you'll see in this brief routine:

```
10 INPUT "A";A
20 INPUT "B";B
30 IF NOT A=B THEN PRINT "NO"
40 GOTO 10
```

The final part of this chapter is concerned with another IF/THEN addition — IF/THEN . . .ELSE. This allows for two actions to be performed arising from one condition. IF a condition is fulfilled THEN perform an action ELSE perform another action.

The IF/THEN . . .ELSE statement cuts down on decision-making space and, in effect, time. It is a neat and tidy way to control decision-making, providing that you do not get muddled over the conditions. Remember that one can have more than one action after a THEN and an ELSE simply by using multi-statement lines. Therefore, the statement IF A=15 THEN PRINT A:GOTO 120 ELSE PRINT B:END is a valid one.

8
STRING HANDLING

You may remember from our discussion on variables that strings are designated by variable names ending with a dollar sign. Strings are very useful parts of your working vocabulary in BASIC, as they can be manipulated to produce a number of valuable results.

The string functions we'll be looking at in this section include STR$, LEN, VAL, LEFT$, RIGHT$ and MID$. Although an expression containing some of these may appear bewildering, they are easy to use if you tackle them carefully.

STR$

STR$ (which is often said aloud as 'string dollar' or 'string string') changes a number into an equivalent string. That is, it changes 45.6 into "45.6" — a transformation which does not appear immediately to be one which has very much value. However, it is useful in some circumstances because strings are in some ways easier to manipulate than numbers. One use will be illustrated later, but first we must look at two other string functions.

LEN

This function finds the length of a string. So, if you told the computer to print LEN(A$), where A$ was "FULTON", it would return 6. If A$ was "45.6", then LEN(A$) would be 4, as the decimal point counts as one of the characters in the string.

VAL

This acts as the opposite of STR$; that is, STR$(42.6) is "42.6" while VAL("42.6") is 42.6.

We can use two of these to produce a program which lines numbers up to the right (instead of to the left, as is usually the case):

```
10 FOR J=1 TO 5
20 READ A
30 A$=STR$(A)
40 PRINT TAB(10-LEN(A$));A$
50 NEXT
60 DATA 23.45,350.07,1.34,1444.75,23.46
```

Below is the output:

```
  23.45
 350.07
   1.34
1444.75
  23.46
```

Line 20 reads the number from the data statement, and line 30 changes this number (A) into a string (A$). Line 40 prints this, using a TAB setting of 10 minus the length (LEN) of the string.

LEFT$, RIGHT$ and MID$

These are used to extract required portions of strings: LEFT$ returns a designated length from the left of a string; RIGHT$ does the same from the rightmost character; and MID$ selects a string of a designated length from the middle of another string.

Here is a program to show LEFT$, RIGHT$ and MID$ in use:

```
10 A$="DAKOTA"
20 PRINT "LEFT$(A$,3)=";LEFT$(A$,3]
30 PRINT "RIGHT$(A$,3)=";RIGHT$(A$,3)
40 PRINT "MID$(A$,2,3)=";MID$(A$,2,3)

LEFT$(A$,3)=DAK
RIGHT$(A$,3)=OTA
MID$(A$,2,3)=AKO
```

As you can see, when A$ is "DAKOTA", LEFT$(A$,3) prints out the three leftmost characters of the string, DAK. RIGHT$(A$,3) returns the three rightmost characters of the string, OTA. MID$(A$,2,3) gives three characters from the string, starting at the second, AKO.

ASC and CHR$

Next, we'll look at two other string-handling words, ASC and CHR$.

The letters, numbers and symbols manipulated by your computer all have 'character codes'. These are generally known as ASCII codes; ASCII stands for American Standard Code for Information Exchange, and is the most widely used encoding system for English Language alphanumerics (that is, letters, numbers and symbols).

The ASC command is used to get the ASCII code number from a symbol, and CHR$ is used to turn the code number into a symbol. The following examples should make that clear:

```
PRINT ASC("A") returns 65
PRINT ASC("B") returns 66
PRINT CHR$(65) returns A
PRINT CHR$(66) returns B
```

You may have noticed that the Amstrad has 31 characters that cannot be printed as normal. These codes, from one to 31, are known as control codes and can only be shown on-screen by pressing the CTRL key and one of the control keys. (If you are curious as to what these codes do, then look them up at the back of the Amstrad users' manual, supplied with every machine.)

These codes cannot normally be printed up on to the screen within a program. However, if you start a program line with PRINT CHR$(1)+CHR$(number of the character code), then it will appear on-screen. By now, you should have developed enough programming skill to write your own loop to print up all 31 of these characters on the screen.

String specifics

Now let's move on to a number of commands specific only to the way the Amstrad handles strings.

UPPER$, LOWER$

These commands, rarely seen in other BASICs, transform a string between upper and lower case. UPPER$ naturally transforms a lower-case string into one filled with upper-case characters, while LOWER$ performs the exact opposite task. The commands can be

useful in programs handling a lot of text where you want the user's input to be in the correct format. Another handy use for such commands is in a complex wordprocessing package, where words specified by the user are highlighted and checked for spelling errors.

SPACE$

This is not a command you are going to be using every minute of the day! SPACE$ creates a string of spaces of a given length; thus, SPACE$(80) creates an 80-character long string filled with spaces.

STRING$

This is really the command that SPACE$ developed from, which does put a question mark over the relevance of including SPACE$ at all. The command STRING$ creates a string of any given character of the specified length. The format for the command is:

```
STRING$(N,"X")
```

where N is the length of the string and X the character that will make up the string.

And if you're looking for oddities in Locomotive BASIC, try putting a '&' character in front of the number signifying the length of the string. You'll find the string is automatically made two-thirds longer than the number specified. As regards the use of such a finding . . .

INSTR

INSTR searches through a string to find the first occurrence of another string. The format for this very useful command is:

```
INSTR (N,"X","Y")
```

where X is the string which is to be searched, Y is the string that is to be looked for (and if found, its position reported) and N is an optional figure allowing the programmer to choose where to search in the first string.

Below are several examples to improve the clarity:

```
PRINT INSTR( "LOCOMOTIVE SOFTWARE", "SOFT")
12
PRINT INSTR(7, "LOCOMOTIVE SOFTWARE","T")
7
```

CLEAR

The final command that we are going to deal with in this chapter is not just to do with strings, but with all variables. CLEAR simply clears all variables and files.

String sort

To finish this chapter on string handling, here is a string-sorting program that puts strings into alphabetical order. Following the program is a couple of sample runs and, finally, a brief discussion on string slicing.

```
10 ' STRING SORT
20 '
30 B=0
40 INPUT "TITLE OF STRING SORT";T$
50 INPUT "HOW MANY WORDS";T
60 DIM W$(T)
70 G=T
80 FOR A=1 TO T
90 INPUT W$(A)
100 NEXT
110 CLS:PRINT T$
120 PRINT
130 Z=1
140 B=Z+1
150 IF B>G THEN 220
160 IF W$(B)>W$(Z) THEN 180
170 Z=Z+1:GOTO 140
180 Q$=W$(Z)
190 W$(Z)=W$(B)
```

```
200 W$(B)=Q$
210 GOTO 170
220 PRINT W$(G)
230 G=G-1
240 IF G>0 THEN 130

PORSCHE
FORD
CHRYSLER
CITROEN
RENAULT
FIAT
FERRARI
BMW
DATSUN
VOLKSWAGEN
ROLLS ROYCE
LOTUS

CAR MAKERS

BMW
CHRYSLER
CITROEN
DATSUN
FERRARI
FIAT
FORD
LOTUS
PORSCHE
RENAULT
ROLLS ROYCE
VOLKSWAGEN

STABLEFORD
MAY
ADAMS
SILVERBERG
GREENE
HEMINGWAY

FAVOURITE AUTHORS

ADAMS
```

```
GREENE
HEMINGWAY
MAY
SILVERBERG
STABLEFORD
```

String slicing

To give it its proper title, concatenation, this involves chopping strings around to create other strings. The commands used to do this include LEFT$, RIGHT$ and MID$ — commands you should by now be familiar with. To add strings together, all you have to do is place a plus sign between them (for example, C$=A$+B$). String slicing can be very useful in text-handling programs — take a look at a textual adventure game!

9 INPUTS OF VARIOUS KINDS

Any program that is truly interactive must involve the user inputting some kind of information for the computer to act on. This input can be as simple as a joystick controlling an arcade-game spaceship or as detailed as the contents of a report that are to be justified, formatted and then output to a printer. In this chapter, an attempt will be made to deal with all the input commands available on the Amstrad.

INPUT

The INPUT instruction is followed by either a string or numeric variable; after entering the required figure, be it a string or a number, you must press the ENTER key. INPUT is not used for action games simply because it would take too long to input the necessary amount of information in this way. Take a look at the section on the INKEY$ command if you want to find out how information can be input without using the ENTER key.

You can put a message within quotation marks between the two halves of an INPUT statement — for example, INPUT "ENTER THE SIZE OF CIRCLE (1–20)"; A; this means that the program can request user-friendly input without the necessity of an extra PRINT statement.

LINE INPUT

This command is very similar to INPUT except that it allows you to enter commas within an inputted string. LINE INPUT is only used with a string variable.

INKEY$

The INKEY$ command allows for quick, single-character input; this is usually done by making a string variable equal to the INKEY$ statement — for example, A$=INKEY$.

As INKEY$ scans the keyboard so fast, it may be necessary to add a program line to halt the progress of the program until a key has been pressed. Use a program line like the following:

```
10 A$=INKEY$: IF A$="" "THEN 10
```

INKEY

A very close relative to INKEY$, the INKEY command returns the values of the key(s) being pressed. The INKEY function is used in the following format:

```
IF INKEY(A)=N THEN . . .
```

The value A is the number of the key (note that it is not the ASCII value but the key number). The value N is the key's status; N can have any one of five values, each corresponding to a different combination of key presses. The five possible values of N are explained below:

Value of N	*Corresponding key status*
−1	The specified key is not being pressed.
0	The specified key is being pressed, but the CTRL and SHIFT keys are not being pressed.
32	Both the SHIFT key and the specified key are being pressed.
128	Both the CTRL key and the specified key are being pressed.
160	The specified key, as well as the CTRL and SHIFT keys, are being pressed down.

Joystick control

The Amstrad has a joystick socket at the back of its case, and all information input from it is controlled by the BASIC command, JOY. This command is followed by either a one or a zero in brackets, signifying the number of the joystick used. (If you do not have the special attachment allowing you to connect up two joysticks to the Amstrad, then always use the number zero in the brackets.

The value created by JOY can be one of six different numbers — check out the table below:

Joystick value	*Action*
1	Up
2	Down
4	Left
8	Right
16	Fire button 2
32	Fire button 1

Artistic input

Having covered the main commands involved with the input of information into your Amstrad, let's take a look now at some example programs to show these in action. The demonstration programs are all based around an artist program that allows you to draw pictures on-screen. For the keyboard-controlled versions (the first two), the keys are: A for up; Z for down; O for left; P for right; and C for changing pen colour.

The first version of *Artist* makes use of the INPUT command.

```
10 ' ARTIST 1
20 ' USING INPUT
30 '
40 CLS
50 X=20:Y912:P=1
60 LOCATE 1,25
70 PRINT "    "
80 LOCATE 1,25
90 PEN 1
100 PRINT "COLOUR:";P;:INPUT
"ENTER DIRECTION";A$
```

```
110 IF A$<>"A" AND A$<>"Z" AND A$<>"O" AND A$<>"P"
AND A$<>"C" THEN 60
120 IF A$="A" AND Y>1 THEN Y=Y-1
130 IF A$="Z" AND Y<24 THEN Y=Y+1
140 IF A$="O" AND X>1 THEN X=X-1
150 IF A$="P" AND X<40 THEN X=X+1
160 IF A$="C" THEN P=P+1: IF P=4 THEN P=0
170 PEN P
180 LOCATE X,Y
190 PRINT CHR$(143)
200 GOTO 60
```

The second version of *Artist* uses the INKEY$ command.

```
10 ' ARTIST 2
20 ' USING INKEY$
30 '
40 CLS
50 X=20:Y=12:P=1
60 LOCATE 1,25
70 PEN 1
80 PRINT "COLOUR:";P
90 A$=INKEY$
100 IF A$<>"A" AND A$<>"Z" AND A$<>"O" AND A$<>"P"
AND A$<>"C" THEN 90
110 IF A$="A" AND Y>1 THEN Y=Y-1
120 IF A$="Z" AND Y<24 THEN Y=Y+1
130 IF A$="O" AND X>1 THEN X=X-1
140 IF A$="P" AND X<40 THEN X=X+1
150 IF A$="C" THEN P=P+1:IF P=4 THEN P=0
160 PEN P
170 LOCATE X,Y
180 PRINT CHR$(143)
190 GOTO 60
```

The final version of *Artist* allows users to input information via a joystick.

```
10 ' ARTIST 3
20 ' USING JOYSTICK
30 '
40 CLS
50 X=20:Y=12:P=1
```

```
60 LOCATE 1,25
70 PEN 1
80 PRINT "COLOUR:";P
90 A=JOY(0)
100 IF A=1 AND Y>1 THEN Y=Y-1
110 IF A=2 AND Y<24 THEN Y=Y+1
120 IF A=4 AND X>1 THEN X=X-1
130 IF A=8 AND X<40 THEN X=X+1
140 IF A=16 THEN P=P+1:IF P=4 THEN P=0
150 PEN P
160 LOCATE X,Y
170 PRINT CHR$(143)
180 GOTO 60
```

Other key commands

While we are discussing the various ways the Amstrad can be made to read the keyboard, it seems a good time to detail the commands that affect the keys themselves.

SPEED KEY

The SPEED KEY command sets the speed of repeat on the keyboard. It has two numbers following the keyword, the first is the start delay before the repeat starts (measured in 0.02-second units) and the second is the rate of repeat (also measured in 0.02-second units). Be careful when you use this command to always have a part of the program which resets the key values, particularly when you're altering the start delays, as it gets difficult to type on the Amstrad. The default rate for the repeat is SPEED KEY 10,10.

KEY DEF

The KEY DEF command lets you define a key to perform a different task. Below is an example of KEY DEF in use:

```
KEY DEF 47,1,65
```

This defines key number 47 (Z) to print a capital A. The first number after the command signifies the number key that is being re-defined. The second number is either a one or zero, and indicates whether the repeat is to be disabled or not (1 for repeat to be kept and 0 for it to be disabled). The final number indicates the new key definition; in our example, the new definition is 65, the ASCII code of the character A.

KEY

This command allows you to fix a totally new function to a key — you don't just change the character that that key prints, but you alter its whole function. For example, KEY 140,"CLS"+CHR$(13) fixes the small ENTER key on the numeric pad so that when CTRL and ENTER are pressed together, the word CLS appears (the CHR$(13) ensures that the command is entered into the computer and is acted upon). Type the above program line into your Amstrad and try it out so that you can see the principle in action.

The next demonstration program defines two keys, the zero key and the one key on the numeric key pad, to perform several commands each. Run the program, which displays some asterisks on the screen, and then press the zero key — the screen clears and the program is listed. Now press the one key and the program runs and a tone is sounded.

```
10 ' KEY DEMO
20 '
30 KEY 128,"CLS"+CHR$(13)+"LIST"+CHR$(13)
40 KEY 129,"SOUND 1,180,30,15"+CHR$(13)+"FOR
T=1 TO 1000:NEXT"+CHR$(13)+"RUN"+CHR$(13)
50 CLS
60 PRINT "KEY DEMO"
70 FOR T=1 TO 20
80 PRINT "**********************";
90 NEXT
```

There are several limitations to note when you're using the KEY command. Firstly, only the group of 32 expansion characters can be used, which range from 128 to 159. Secondly, the strings holding the newly-defined commands and the character codes — such as CHR$(13) — cannot exceed 120 characters in length.

10
COMMANDS TO AID YOUR PROGRAMMING

Here are listed a number of separate commands that all make programming life a little easier (as if we're not pampered enough already).

RENUM

To start off, let's take a look at perhaps the most useful of all these commands — RENUM. This command allows you to renumber the program lines to whatever starting number you require and in whatever number steps you so desire. The syntax for this command is:

RENUM (new line number), (old line number), (increment)

This may seem rather complex, but in practice it is not. If your program was four lines long starting at line 10 and you typed in the command RENUM 100,20,10, then your program line numbers would read 10,100, 110 and 120; the renumbering process starts at the second line of the program, leaving line 10 as it was, and renumbering the remaining three lines in steps of 10 from line 100.

You can also use the command to create more space within a block of program lines where there is no more line space and you need to insert more program instructions. No matter how well a program is designed, there are bound to be a few places where extra lines are required. Also, when you've finished the listing, you can tidy the program up by renumbering it in steps of 10, starting at line 10. To do this, all you have to do is type RENUM and press the ENTER key; entering RENUM is the equivalent of RENUM 10,1,10.

DELETE

DELETE is a rather self-explanatory command. It allows you to delete blocks of program lines; for example, DELETE 100–400 would get rid

of all program lines whose numbers ranged between 100 and 400. Remember that the command is inclusive, so DELETE 100–400 would delete lines 100 and 400 as well.

Much like the LIST command, DELETE can be used with a single number. For example, DELETE 1000– will delete all lines from and including line 1000 to the end of the program, while DELETE –350 will delete all program lines from the start to (and including) line 350.

FRE

The FRE command has several uses, but it is most relevant as a programming aid when used to tell the user how much free memory is still available. When you turn on your machine, type PRINT FRE(0) and you should receive the answer of 43533; the figure '43533' is measured in bytes — you may be more familiar with the term 'K', which is short for 'Kbyte' or 1024 bytes. From the answer 43533, you can work out that you have over 43K of memory free for programming which, by any home computer's standards, is an awful lot of spare memory to play around with.

So, if at any time you are writing a large program and wonder how much memory it is taking up, you can find the answer simply by typing PRINT FRE(0). If you have a 6128 then you have much more spare memory, although this can only be used indirectly. Read the 6128 chapter for more details.

AUTO

The AUTO command stops you from the monotonous chore of typing in line numbers. If you have a program with regular line numbers, then using AUTO could save you some time. Two numbers follow the AUTO command, the starting line number and the sizes of step. For example, AUTO 345,10 would start the generation of line numbers automatically at line 345 in steps of 10.

TRON and TROFF

TRON and TROFF are what are known as 'trace' commands, and they can be very useful when checking the flow of a program. The

trace outputs the number of the line about to be executed (surrounded by square brackets) and displays it on-screen. Users can then check if the program is jumping at the correct time and to the correct place. TRON (TRace ON) switches the trace on and TROFF (TRace OFF) turns it off.

EDIT and editing

You are bound to make mistakes when writing programs or even just typing lines in from this book. Many lines are very short and it is quick to type them in again — but what happens when you get a monster of a line which will take some time to type in again? Luckily, your Amstrad leaps to the rescue (again), providing you with an EDIT command. Type in the line below and RUN it:

```
10 FOR T=1 TO 100:PRINT " EDIT
DEMONSTRATION:PRINT:NEXT
```

You will get a 'NEXT missing in line 10' error message appearing on the screen. What is in fact happening is that due to a missing quotation mark, the computer is counting everything past the first PRINT statement as part of that PRINT statement and, thus, ignoring the NEXT command.

To remedy this, type EDIT 10 to bring the line down to be edited on-screen. Guide the block cursor along the program line using the cursor keys (the block of keys above the numeric keypad) until you reach the last N in DEMONSTRATION. Now, add a quotation mark between the last N and the colon — and the line has been corrected. Pressing the ENTER key puts the new line into the computer's memory. If you wish to get rid of an excess character, then you can either position the cursor one character to the right and press the DEL key, or you can place the cursor over the erroneous character and press the CLR key to wipe it from the line.

If at any time you get in a mess with a line you are attempting to edit, just press the ESC key and the line will still be there in its original form and you can start the editing process again.

There is another method of editing called 'copy cursor editing', but this is fully explained in your Amstrad manual in the first chapter.

11
THE AMSTRAD CASSETTE AND ITS USES

Your Amstrad CPC 464, with its built-in cassette recorder, is fortunately not beset with the problems that abound on other home computers. In this chapter we will discuss how to load and save programs on to cassette, explain he various cassette file-handling statements and, finally, how to create and manipulate cassette data files. If you have a 664 or 6128 then it is worth reading much of what is here and then move on to Chapter 12.

SPEED WRITE

Saving and loading is not a complex task on the CPC 464. You have a choice of two loading and saving speeds; whatever speed you choose for saving should be used for loading. You select the speed by the SPEED WRITE command — if a zero follows the command, the tape runs at the lower speed of 1000 baud; if a one is placed after the command, the machine loads and saves at 2000 baud. When the CPC 464 is switched on, it automatically defaults to a saving/loading speed of 1000 baud.

SAVE and LOAD

Place your cassette in the recorder (it is best to use recommended computer cassettes with five or six minutes a side) making sure that it has been wound on to the 'proper' tape. Type SAVE "The File Name" (the file name can be up to 14 letters long) and press the ENTER key. The Amstrad then gives you on-screen instructions to press the Record and Play keys on the cassette recorder, followed by any key on the keyboard. You are now saving a program!

Loading is a similar operation. Type LOAD followed by a file name in quotes or, if you just want to load the first program on the cassette, just two quotation marks (LOAD" "). The Amstrad again prints up an instruction message on-screen, this time to press only the Play key on the cassette and then any key on the keyboard. The CPC 464 searches through the tape for the first correctly saved program (or if a file name was specified, until it finds that file) and then loads it into its memory.

An alternative to using the LOAD command is RUN" ". This creates an auto-running program — the program loads and then runs automatically.

CAT

To check the contents of a cassette, you can use the CAT command. This CATalogues the entire cassette's contents and states whether the file is a BASIC program file (indicated by a dollar sign ($)), an ASCII text file (indicated by an asterisk (*)), a binary file (indicated by a &) or a protected BASIC program (indicated by a percentage sign (%)).

MERGE

You can merge two programs together to form a single program; this is particularly useful when, say, you have developed a useful routine that is stored on tape which you want to incorporate in a number of different programs. The format to use the command is simply MERGE "file name". The effects of the MERGE command, however, are a little more involved. Firstly, protected files cannot be merged — all variables, files and functions are discarded — and, finally, a RESTORE is performed setting the data pointer to the start of all the data.

If you wish for the program still in memory to be unaffected by the merge, then you will have to RENUMber it to a range above the line numbers of the incoming program.

CHAIN

CHAIN is similar to the MERGE command, except that it replaces the program in memory rather than merges with it. After the CHAIN

command comes the file name and an optional figure; this optional figure allows you to specify the line number at which the program will start once it is loaded into the CPC 464's memory.

CHAIN MERGE combines the two commands so that a program from tape is merged with the program in the computer's memory, and the program automatically starts from the line number specified in the CHAIN MERGE statement. If no line number is mentioned in either of these commands, the program begins from the lowest one available.

Before leaving the subject of saving and loading programs, it is worth noting that if the character ! (an exclamation mark) is placed at the front of the file name, the cassette messages produced by the CPC 464, such as Press PLAY and any key:, are removed.

Data files

The rest of the loading and saving commands are all concerned with data files. To make a data file, you must open a file with a file name, output the data to the cassette and then close the file. To load the file back into the CPC 464, you must open a file with the same name as the one wanted, input the data from tape, check for the end of the file and then close it.

That may sound complicated, but it is really more simple than you would think, particularly when you see the names of the commands involved. There are several commands that can be used, but for the sake of simplicity we will not clutter up this chapter with any more than one way to construct data files.

To save a file

```
OPENOUT "File Name"
PRINT #9, all data to be printed on to cassette
CLOSEOUT
```

To load a file

```
OPENIN "File Name"
```

INPUT #9, all the data to be loaded from cassette
IF EOF=1 THEN CLOSEIN (This last command checks for the *E*nd *O*f *F*ile. If that is reached, then the file is closed.)
CLOSEIN

Database

To demonstrate the data file commands, here is a demonstration program showing how one can save and load data files. It has been kept deliberately simple so that the cassette-handling commands are at their most prominent within the program. Enter up to 50 items of data, then save the file on cassette with a file name. This file can be retrieved at a later date, simply by choosing the load file mode.

It would not be difficult to add to this program — in fact, it has been written in such a way that any additional lines or rearrangement should be an easy task. Why not try adding a string search feature or increasing the number of files handled by the computer? One easy task would enable the data to be dumped to the printer.

One final point about the screen display. When the first item in the file is displayed, you must hold down a key to maintain the flow of data printed. This seemed to be the best way of displaying up to 50 items of data, so that all could be examined in the user's own time.

```
10 ' DEMO DATABASE
20 '
30 DIM A$(50)
40 PRINT:PRINT
50 PRINT "1 TO ENTER DATA, 2 TO SAVE DATA, 3 TO
LOAD DATA, 4 TO PRINT DATA ON-SCREEN"
60 INPUT G:IF G>4 OR G<1 THEN PRINT:GOTO 50
70 ON G GOSUB 120,250,370,490
80 GOTO 40
90 '
100 ' ENTER DATA
110 '
120 CLS
130 INPUT "HOW MANY ENTRIES";N
140 IF N>50 OR N<1 THEN PRINT:GOTO 130
150 PRINT:PRINT
160 FOR T=1 TO N
```

```
170 PRINT "ENTER DATA NO.";T
180 INPUT "       ";A$(T)
190 PRINT:PRINT
200 NEXT
210 RETURN
220 '
230 ' SAVE DATA
240 '
250 CLS
260 INPUT "FILE NAME (UP TO 14 CHARS.)";F$
270 PRINT:PRINT
280 OPENOUT F$
290 FOR T=1 TO 50
300 PRINT #9,A$(T)
310 NEXT
320 CLOSEOUT
330 RETURN
340 '
350 ' LOAD DATA
360 '
370 CLS
380 INPUT "FILE TO BE LOADED ";F$
390 PRINT:PRINT
400 OPENIN F$
410 FOR T=1 TO 50
420 INPUT #9,A$(T)
430 NEXT
440 CLOSEIN
450 RETURN
460 '
470 ' PRINT DATA
480 '
490 CLS
500 FOR T=1 TO 50
510 PRINT A$(T)
520 WHILE INKEY$="":WEND
530 NEXT
540 RETURN
```

Tips with cassettes

Though using cassettes with the CPC 464 is pretty straightforward, there are a number of things you may do which will damage the contents of your tapes. So, be warned, and follow these tips and keep your cassettes in good condition.

(1) NEVER store your cassettes near a device which creates an electro-magnetic field, such as a television. This can wipe the tapes clean or, if less potent, corrupt them.
(2) NEVER store your tapes near an abnormally hot or cold area, such as a radiator or an open window. Extreme temperatures can badly affect tapes.
(3) ALWAYS label your tapes with the program's loading name and, if possible, use the tape counter to give its position on tape.
(4) ALWAYS use the CAT command to catalogue what is actually on the cassette . . . it's a lot better than guesswork!

12
USING DISCS

Connecting your disc drives

If you have an Amstrad CPC 664 or 6128, then you obviously do not have to connect your drive to the computer. Just skip the next three paragraphs, and go straight to the section headed 'The discs'.

Before you attempt to connect the disc drive to your 464, make sure the power to the drive itself, and to the computer, is switched off. In about the middle of the computer, at the back, is an edge connector marked with the words *FLOPPY DISC*. Insert the interface unit firmly into this connector. The other end of the flat grey ribbon is to be connected to the drive unit itself (the plug about 4 inches from the end of the ribbon is for a second disc drive).

Once you have the drive attached, turn on the disc unit, and then the computer. The disc must always be turned on *before* you turn on the computer. In addition, you should make sure you have removed the disc from the drive before you turn it on.

It is best to place the disc unit to the *right* of the monitor, rather than the left, as the left-hand side of the monitor appears to send out electrical interference which could cause the drive unit to work less reliably than might otherwise be the case. It is important as well that the drive, and your discs, are kept away from string magnetic fields, such as those produced by hi-fi loudspeakers.

The discs

You will see that the 3-inch discs which your computer uses have a 'shutter' in the top left-hand corner of each side. This is used — much as the lug you can remove on a cassette — to prevent overwriting information you want to keep on a disc.

Figure 1 shows what the disc looks like.

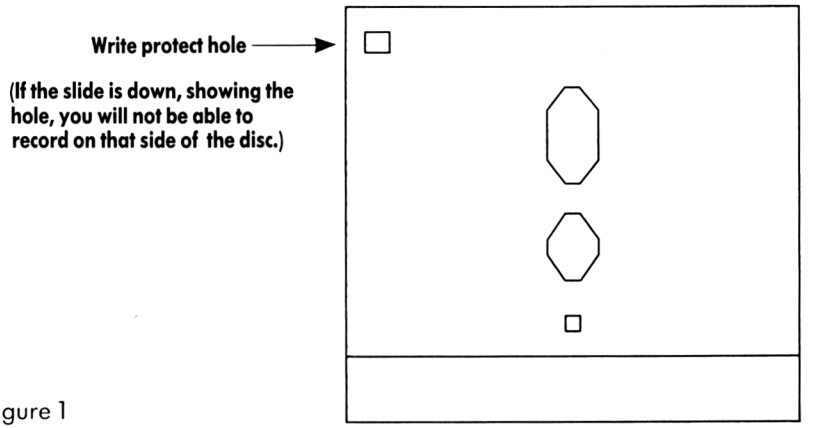


Figure 1

When the drive is turned on, the bright green light near the bottom of it lights up: this is the 'power on' indicator. About halfway up, on the right-hand side of the front panel of the disc unit, is a push button which is used to eject discs from the drive. On the other side of the front of the unit is a light which glows dull red. This red indicator light performs two tasks. It flashes brightly if you have only one drive connected, when information is being recorded on the disc, or when it is being read from the disc. If you have two drives, the red light on the second drive (known as *Drive B*, with the first drive always being called, by convention, *Drive A*) is on all the time, so that you are left in no doubt as to which drive is B.

AMSDOS

AMSDOS is the AMStrad Disc Operating System. It ensures that BASIC files can access the disc in exactly the same way (although very much more quickly, of course) as they did the datacorder.

All the BASIC commands (with the exception of CAT which has a special meaning which we'll see in action shortly) work with the disc drive and the cassette and have exactly the same effect.

This means you can use all of the following, and see them performing with either a disc drive or a datacorder:

LOAD, RUN, SAVE, CHAIN, MERGE, CHAIN MERGE, OPENIN, OPENOUT, CLOSEIN, CLOSEOUT, EOF, INPUT #9, LINE INPUT #9, WRITE #9, LIST #9

To erase a file from the disc, take the following steps. Assume that the name of the file is *turtle*.

Enter the following directly:

f$="turtle":|era,@f$ ⟨ENTER⟩

If you precede this with CAT, and follow it with CAT, you will see that *turtle* has vanished from the disc.

To remove a file on disc, take the following steps. Assume that the original name of the file is *turtle* and that you want to change it to *terrapin*.

Enter the following directly:

one$="turtle":two$="terrapin":|ren,@two$,@one$ ⟨ENTER⟩

Again, CAT will show you that *turtle* has vanished from the catalogue and has been replaced by *terrapin*.

It was mentioned earlier that CAT behaves differently with discs than it does with the datacorder. The datacorder command CAT simply spools through the tape, putting up each file name as it comes to it. With the disc, CAT sorts all files on the disc into alphabetical order (as you can see above) and then prints them out, complete with an indication of the length of the file, and a message regarding the space left on the disc.

The message BAD COMMAND tells you that either you've entered a name which is too long or one which contains incorrect spaces or punctuation. TYPE MISMATCH indicates that you've left off the quote marks (") and SYNTAX ERROR means you have spelt RUN incorrectly.

If you get this message:

```
Drive A: read fail
Retry, Ignore or Cancel?
```

it means that, for some reason (wrong disc; incorrectly or not formatted; or corrupted data) the computer has been unable to read from the disc currently in the drive.

If you have been used to using cassettes, you may be surprised to see the old familiar message Press PLAY then any key: come up on the screen when you tell the computer to load a program. This is because the computer is ignoring the disc drive, either because it has not been turned on, is not connected, or at some stage you have used the command |TAPE ('bar tape') to indicate that you wish to write to, and read from, the datacorder rather than the disc drive. To reverse

|TAPE, simply enter |DISC, and try again. (Note that although SPEED WRITE can be typed in while you have a disc connected, it is not a valid disc command, but only works with the datacorder; the disc system already writes as fast as it can.)

CP/M

The widely-used operating system CP/M (which stands for 'Control Program Microcomputer') is supplied, as you know, with the master disc as part of the disc system. Its name tells you pretty accurately what it does.

CP/M was invented by Gary Kildall, about a decade ago, when he wanted to connect up a disc unit to his (by today's standards) very primitive personal computer. No way existed, so Gary decided to develop a method all by himself. He did it, and then thought that perhaps one or two other microcomputer owners might be interested in the control system he'd developed.

Gary was working for the semiconductor manufacturer, Intel, at the time, and had — in fact — written part of his control system at work. He asked Intel if they wanted to market it. He was told that microcomputers were an unimportant fad, and that Intel had no interest whatsoever in getting involved in toy products. 'You can do what you like with it,' they said.

Kildall did. Starting with modest mail-order advertisements, he sold the system to other microcomputer owners, just as he had expected. But what he had not anticipated was just how much interest there would be in his system. The market he had discovered was so big that Kildall quit his job at Intel, and founded his own company — Digital Research — to sell the product. Today he is a multimillionaire, and software products from Digital Research continue to be among the most successful items on the market.

Kildall's success was based on two things. Firstly, his control system actually worked, and gave microcomputer owners a way to hook discs up to their computers. As well as this, he had inadvertently, simply by being first, created a *standard* disc-control system. Discs created on one computer running under CP/M would work with another computer, so long as it was also furnished with CP/M. There are now more than 3000 CP/M products available, and although they need some fiddling to make them work (and the lack

of memory on the 464 and 664 but not the 6128 is a serious barrier to using a lot of the 'heavier' CP/M software, which assumes 64K is available, rather than the 39K or so you have on the 464 and 664), a significant number of these are now available to you and your Amstrad.

You know you're in CP/M because the following appears:

```
CP/M 2.2 — Amstrad Consumer Electronics plc.
```

The 2.2 will be 3.0 on the 6128. This is followed, on the next line, by A⟩ which indicates that the CP/M system is ready and 'listening to you'. It is the CP/M equivalent of the BASIC READY message.

Once you're in CP/M, your Amstrad will no longer understand BASIC, and any attempt to enter a BASIC command will simply cause the word you entered to be reprinted on the screen, followed by a question mark, which indicates the system does not understand you.

However, although it does not understand BASIC, CP/M does have a vocabulary of its own. This includes DIR (directory), to get a list of the material on your disc, and FORMAT, to prepare a new disc so that you can write to it. (You return the system so that it can understand BASIC by entering AMSDOS, then pressing the ENTER key.)

Copying programs from disc to disc

Single-disc drive

You have evoked CP/M with |CPM ⟨ENTER⟩. A⟩ is now on the screen. Type in DISCCOPY; the disc drive will whirr a few times, before this message appears:

```
Please insert source disc into drive A then
press any key
```

Remove the CP/M disc and replace it with the one you want to copy. Then follow the prompts. Part of the disc will be stored into RAM; then you'll be asked to insert the 'destination disc' so this part of the original disc can be recorded on it. You'll then be asked to insert the source disc again, so the next section can be copied into RAM.

Two-disc drives

With two drives, the process is simpler and easier than it is with one. Instead of DISCCOPY, you use the command COPYDISC (!). You simply have the source disc in one drive, and the destination disc in the other. Full on-screen prompts will guide you as to what to do.

Checking copied discs

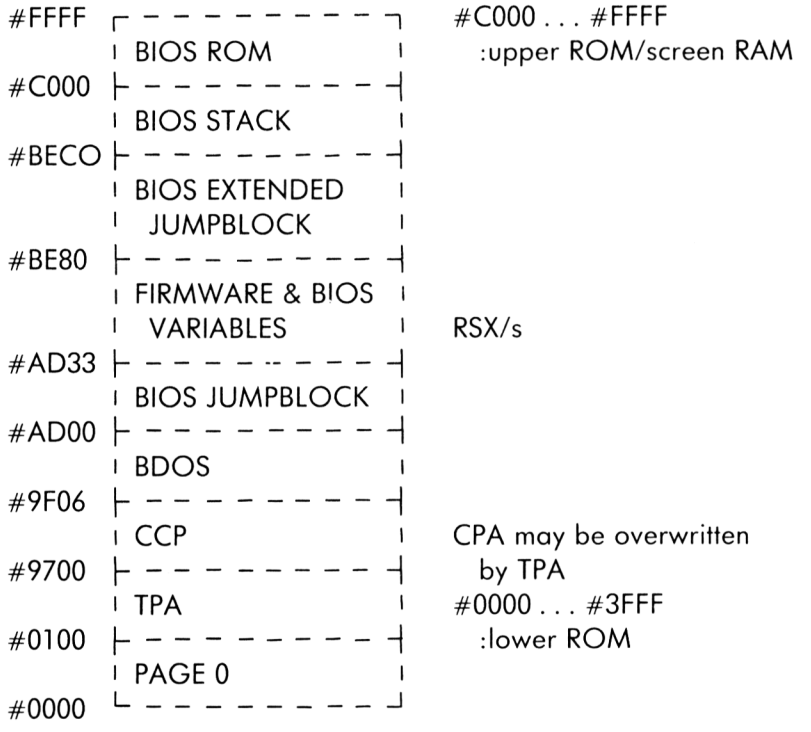
If you want to check the copy you've made against the source disc, you can use the DISCCHK facility (for a single-drive system) or CHKDISC (if you have two drives). The screen prompts will tell you what to do.

CP/M utilities

The disc-copying and checking facilities are just a few of the utilities provided on your system disc as part of CP/M. Here's a summary of them:

AMSDOS.COM	returns to AMSDOS and BASIC (the opposite of |cpm)
ASM.COM	8080 assembler
BOOTGEN.COM	copies boot and configuration sectors between discs
CLOAD.COM	copies cassette file to disc
CSAVE.COM	copies file from disc to cassette
CHKDISC.COM	compares two discs (needs two drives)
COPYDISC.COM	copies one disc to another (two drives needed)
DDT.COM	Dynamic Debugging Tool, 8080 debugger
DISCCHK.COM	compares two discs (using single drive)
DISCCOPY.COM	copies one disc to another (using single drive)
DUMP.COM	displays file in hex on screen
ED.COM	text editor
FILECOPY.COM	copies files from one disc to another (using a single drive)
FORMAT.COM	formats disc
LOAD.COM	reads file in Intel HEX format; produces .COM file

MOVECPM.COM	constructs CP/M system of given size (used to make room for RSXs)
PIP.COM	Peripheral Interchange Program (copies files on disc; between other peripherals)
SETUP.COM	changes parameters in configuration sector
STAT.COM	status report on files, discs, users and IOBYTE (can also be used to modify IOBYTE)
SUBMIT.COM	takes CP/M commands from file (instead of from keyboard)
SYSGEN.COM	writes CP/M system on to system tracks
XSUB.COM	used with SUBMIT.COM for buffered program input



CP/M store map (positioned as high as possible)

Logo

As well as CP/M, your system disc contains Digital Research's Logo. To evoke the Logo from BASIC, turn the system disc (or your security

security copy of it) over to side two (so that the words 'Dr LOGO' are visible) and enter the command |CPM. The drive does its whirring bit again, and a message will appear on the screen telling you that Logo has been loaded.

Logo was developed by a team of mathematicians under the leadership of Seymour Papert (author of the most important Logo text, *Mindstorms*) as an easy entry for children into computer programming; adults as well have found it a delight to use. The instant, on-screen feedback, and the elegance of many of the patterns traced by the turtle as it ambles around the screen make the language a delight to use.

Logo primitives

The basic building blocks of Logo are known as *primitives*, and you can do a great deal with just a few of these.

Try the following, once you have Logo running, pressing the ⟨ENTER⟩ key after each one:

```
forward 100
back 100
right 90
forward 100
right 90
forward 130
left 80
back 100
```

Four of the most important words are the two which move the turtle across the screen (*forward* and *back*) and those which control its turning (*right* and *left*). The movement primitives must be followed by a number which tells the turtle how many 'steps' to take across the screen. Those controlling turns are followed by a number which is the number of degrees the turtle will turn from the direction in which it is now facing.

You can use the primitive *cs* to clear the screen. When you've done that, enter the next line, and follow it with ⟨ENTER⟩:

```
repeat 4[fd 110 rt 90]
```

You will discover that this draws a square on your screen. The *repeat* primitive, as is pretty obvious, causes the turtle to carry out the commands within the square brackets the number of times indicated

by the number which precedes the opening bracket.

A full list of Logo primitives is given in your manual. However, to show you just how flexible Logo can be, here are five very interesting designs you can create on your screen, just using the commands we have discussed so far:

Pinwheel

```
REPEAT 45[FD 130 RT 92]
```

Double Fan

```
REPEAT 20[FD 120 RT 182 FD 240 RT 182 FD 120]
```

Shadow Star

```
REPEAT 30[FD 120 RT 190 FD 240 RT 190 FD 120]
```

The Inner Circle

```
REPEAT 100[FD 100 RT 93 FD 10 RT 97 FD 101 RT 180]
```

Bird of Paradise

```
REPEAT 70[FD 130 RT 181 FD 260 RT 182 FD 128]
```

If you want to expand your knowledge of Logo, there are a number of worthwhile books around. As most Logo dialects are pretty similar to each other, you'll find nearly all programs you come across will work with minimal modification on your system. The best books I have come across on the subject are:

IBM PC and PCjr Programming Primer by Don Martin, Marijane Paulsen and Stephen Prata (Howard W. Sames & Co., Inc., Indianapolis, Indiana, 1984).
Learning Logo on the Apple II by Anne McDougall, Tony Adams and Pauline Adams (Prentice-Hall of Australia, Pty Ltd, 1982).
Logo Programming by Anne Moller (Century Communications Ltd, London, 1984).
Logo, a Language for Learning by Anne Sparrow (Pan Books, London and Sydney, 1984).
Apple Logo by David D Thornburg (Addison-Wesley Publishing Company, London, 1983).

An easy-to-read introduction to Logo graphics, the majority of which you can run without modification in Dr. Logo, is in the 64-page booklet *Spectrum+ Logo* by Tim Hartnell (Interface Publications Ltd, 1985).

13
GAMES! GAMES! GAMES!

Here is a selection of exciting games for you to key in and enjoy. The listings here cover the whole spectrum of computer games, including those involving adventure, arcade, strategy and logic.

Writing games is an excellent way of improving your programming. Even the mere act of typing in those in this chapter will teach you something about program design and may give you a few tips on how to present your own games. The introduction to each game gives a couple of hints on how to play — and win!

Red Dawn

As you skim the gently flowing waves of the Martian Sea, things look calm. You are the captain of one of the Federation's scout ships and must search the area on the lookout for the Facewatchers, a band of mutants that threaten to attack at any time. Your craft, humble by the Federation's standards, only has 50 plasma bolts . . . so use them fairly sparingly! Anyway, there doesn't seem to be a need for them right now . . . But wait, what's that on the horizon? It is . . . Yes, it's a Facewatcher.

You grip the control stick (your joystick) tightly; this is going to be some battle. . . . Good luck.

We couldn't have a collection of games without a good old 'Zap 'em up' game, could we? This one is quite tricky to play as the Facewatcher bounces round the screen. A score of over a thousand is something to be quite proud of!

```
10 ' R E D    D A W N
20 '
30 CLS
40 BORDER 6
```

```
50 X=12
60 PEN 3
70 FOR T=1 TO 40:LOCATE T,25:PRINT
CHR$(244);:NEXT
80 LOCATE 1,1
90 PRINT "||||||||| R E D   D A W N     ||||||||"
100 C=1:D=1
110 B=2:A=INT(RND*30)+5
120 PEN 1
130 ' MAIN LOOP
140 LOCATE A,B:PRINT " ";
150 A=A+C:B=B+D
160 LOCATE A,B:PRINT CHR$(225);
170 IF A>39 THEN C=-1
180 IF A<2 THEN C=1
190 IF B<3 THEN D=1
200 IF B>21 THEN D=-1
210 Z=JOY(0):IF Z=16 THEN MF=MF+1:K=22
220 M=X
230 LOCATE X,23:PRINT " ";
240 IF Z=4 AND X>1 THEN X=X-1
250 IF Z=8 AND X<40 THEN X=X+1
260 LOCATE X,23:PRINT CHR$(229);
270 IF K<2 THEN 320
280 LOCATE M,K:PEN 2:PRINT ":";:PEN 1:K=K-1
290 IF K=B AND X=A THEN SOUND
1,80,10:SC=SC+137:K=0:LOCATE A,B:PRINT
"X";:FOR T=1 TO 20:NEXT:LOCATE A,B:PRINT
" ";:GOTO 100
300 SOUND 1,2000,2
310 LOCATE M,K+1:PRINT " ";
320 IF MF>50 THEN 350
330 GOTO 130
340 '
350 ' END OF GAME
360 '
370 CLS
380 LOCATE 12,12
390 PRINT "You Scored:";SC
400 SOUND 1,180,100
410 END
```

In the City

Time to don the pinstripe suit, the bowler hat and the umbrella. Tuck today's copy of the *F.T.* under your arm and make your way up to the city — for today you are dealing in stocks and shares.

You have £1000 to dabble with and must buy shares in the companies quoted on the exchange. The Amstrad gives you all the prompts that you need. Be careful never to buy more shares than you can afford; the computer won't let you cheat and will stop your illegal enterprise.

```
10 ' IN THE CITY
20 '
30 CLS
40 PRINT
50 PRINT TAB(10);" IN THE CITY."
60 WHILE INKEY$="":WEND
70 SOUND 1,180
80 X=4:I=10
90 B=1000
100 DIM A(3,4)
110 FOR A=1 TO X
120 A(1,A)=A+INT(RND*250)
130 A(3,A)=0
140 NEXT
150 CLS
160 PRINT
170 PRINT "NO. SHARES PRICE ORIGINAL P/L"
180 PRINT "============================
=========="
190 FOR A=1 TO X
200 PRINT
210 Z=A(1,A)-A(3,A)
220 PRINT A;TAB(X+2);A(2,A);TAB(16);A(1,A);
TAB(27);A(3,A);TAB(34);Z
230 NEXT
240 PRINT:PRINT
250 PRINT "BANK ";B;" INTEREST RATE:";I
260 PRINT
270 INPUT "WHICH COMPANY (1-4)";A
```

```
280 IF A>4 THEN PRINT "NO SUCH COMPANY ON THIS
EXCHANGE":PRINT:GOTO 270
290 PRINT:PRINT
300 INPUT "BUYING OR SELLING ";A$
310 PRINT:PRINT
320 INPUT "HOW MANY SHARES";C
330 IF LEFT$(A$,1)="S" THEN C=-C
340 IF C>0 THEN A(3,A)=A(1,A)
350 B=B-C*A(1,A)
360 IF B<=0 THEN 500
370 A(2,A)=A(2,A)+C
380 IF A(2,A)<0 THEN 520
390 FOR A=1 TO X
400 A(1,A)=A(1,A)+INT(RND*(A(1,A)/4))-INT
(RND*(A(1,A)/4))
410 IF A(1,A)<1 THEN A(1,A)=1
420 NEXT
430 I=INT(ABS(I+RND*8-X)+1)
440 B=INT(B+B*I/100)
450 IF RND>0.05 THEN 150
460 PRINT "COLLAPSE"
470 P=A(2,1)+A(2,2)+A(2,3)-1000
480 PRINT "YOUR VALUE:";P
490 END
500 PRINT "BANKRUPT"
510 END
520 PRINT "FRAUD"
530 END
```

Battlefield Munchman

Munchman's maze, where he normally resides, has become the centre of a battle between the Galaxians and the Platform Climbers. His maze is now a mess and there are two less ghosts than usual; the other two ghosts, as well as the missing power pills, are victims of the ensuing conflict. Still, it's work as usual for the greedy pill gobbler.

You control your Munchman with keys A for up, Z for down, the comma key for left and the full-stop key for right. Note how the ghosts seem to home in on your position. This game is not the fastest

Munchman game around, but there has to be a compromise between the ghost's intelligence and the overall speed to get a competent implementation of the game on-screen.

```
10 ' BATTLEFIELD MUNCHMAN
20 '
30 DEFINT A-Z
40 S=0:LV=3
50 GOSUB 790
60 GOSUB 660
70 LOCATE A,B+1:PEN 1:PRINT A$
80 '
90 ' MAIN LOOP
100 '
110 GOSUB 170
120 GOSUB 170
130 GOSUB 360
140 IF S)6000 THEN GOSUB 820:GOSUB 660:GOTO 70
150 GOTO 90
160 '
170 ' ACCEPT PLAYER'S INPUT AND MOVE
180 '
190 IF (X(1)=A AND Y(1)=B) OR (X(2)=A AND
Y(2)=B) THEN 570
200 LOCATE 7,1:PRINT S;
210 LOCATE A,B+1:PRINT " ";
220 D$=INKEY$:IF D$=" " THEN 270
230 IF D$="," THEN A$="e":C=-1:D=0
240 IF D$="." THEN A$="c":C=1:D=0
250 IF D$="A" THEN A$="b":D=-1:C=0
260 IF D$="Z" THEN A$="d":D=1:C=0
270 IF (C=-1 AND P(A-1,B)=1) OR (C=1 AND
P(A+1,B)=1) OR (D=-1 AND P(A,B-1)=1) OR (D=1
AND P(A,B+1)=1) THEN 290
280 A=A+C:B=B+D
290 LOCATE A,B+1:PRINT " ";
300 LOCATE A,B+1
310 PEN 1
320 PRINT A$
330 IF P(A,B)=2 THEN P(A,B)=0:LOCATE
A,B+1:S=S+10:SOUND 1,300,1
340 RETURN
```

```
350 '
360 ' MOVE GHOSTS AND CHECK FOR LOSE
370 '
380 FOR T=1 TO 2
390 LOCATE X(T),Y(T)+1
400 PEN 2
410 IF P(X(T),Y(T))=2 THEN PRINT ".";ELSE PRINT
" ";
420 PEN 1
430 IF X(T)=A AND Y(T)=B THEN 570
440 IF X(T)>A AND P(X(T)-1,Y(T))<>1 THEN X(T)
=X(T)-1
450 IF X(T)<A AND P(X(T)+1,Y(T))<>1 THEN X(T)
=X(T)+1
460 IF Y(T)>B+1 AND P(X(T),Y(T)-1)<>1THEN
Y(T)=Y(T)-1
470 IF Y(T)<B+1 AND P(X(T),Y(T)+1)<>1THEN Y(T)
=Y(T)+1
480 IF X(2)=X(1) AND Y(2)=Y(1) THEN K=INT
(RND*2)-INT(RND*2):L=INT(RND*2)-INT(RND*2):
IF P(X(1)+K,Y(1)+L)<>P THEN X(1)=X(1)+K:Y(1)
=Y(1)+L
490 LOCATE X(T),Y(T)+1
500 PEN 3
510 PRINT "f";
520 PEN 1
530 IF RND>0.94 THEN LOCATE X(2),Y(2)+1:PRINT "
";:X(2)=INT(RND*30)+5:Y(2)=INT(RND*20)+3:IF
P(X(2),Y(2))=1 OR A=X(2) OR B=Y(2) THEN 530
540 NEXT
550 RETURN
560 '
570 ' EATEN
580 '
590 SOUND 1,800,40
600 LV=LV-1
610 IF LV=0 THEN CLS:LOCATE 12,12:PRINT "YOUR
SCORE:";S:END
620 GOSUB 1430
630 GOSUB 660
640 GOTO 70
```

```
650 '
660 ' DRAW SCREEN
670 '
680 MODE 1
690 PEN 1
700 CLS:PRINT "SCORE:";S;" MUNCHMAN C.1984"
710 PEN 2
720 FOR Y=1 TO 24:FOR X=1 TO 40
730 LOCATE X,Y+1
740 IF P(X,Y)=1 THEN PRINT CHR$(143);
750 IF P(X,Y)=2 THEN PRINT CHR$(46);
760 NEXT:NEXT
770 RETURN
780 '
790 ' INITIALISATION
800 '
810 DIM P(40,24)
820 FOR Y=1 TO 24:FOR X=1 TO 40
830 READ P(X,Y)
840 NEXT:NEXT
850 DATA 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1
860 DATA 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1
870 DATA 1,2,2,2,2,2,1,2,2,2,2,1,2,2,2,2,2,
2,2,1
880 DATA 1,1,2,2,2,2,1,1,1,2,2,1,2,2,2,2,2,
2,2,1
890 DATA 1,2,2,2,1,2,2,2,1,2,2,2,2,2,2,2,2,
1,2,1
900 DATA 2,2,2,2,2,2,1,1,2,2,2,2,2,2,2,2,2,
2,2,1
910 DATA 1,2,2,2,2,2,2,1,1,1,2,2,2,2,2,1,2,
2,2,2
920 DATA 2,2,1,1,1,1,1,1,2,2,1,1,1,1,1,1,1,
1,2,1
930 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
940 DATA 2,2,1,1,1,1,1,2,2,2,2,1,2,2,1,2,2,
1,2,1
950 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
```

```
2,2,2
960 DATA 2,2,2,2,1,1,1,1,1,2,2,2,2,1,2,2,1,
2,2,1
970 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
980 DATA 1,2,2,2,1,1,1,1,1,2,2,2,2,2,2,2,1,
1,2,1
990 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1000 DATA 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1010 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1020 DATA 2,2,2,2,1,1,1,2,2,2,2,2,2,2,2,2,2,
2,2,1
1030 DATA 1,2,2,2,2,1,1,2,2,1,2,2,2,2,2,2,2,
1,2,2
1040 DATA 2,2,2,2,2,2,1,1,1,2,2,2,1,2,1,2,2,
2,2,1
1050 DATA 1,2,2,2,2,2,2,1,1,2,2,2,2,1,2,2,2,
2,2,2
1060 DATA 2,2,2,2,2,1,1,1,2,2,2,2,1,1,1,2,2,
2,2,1
1070 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1080 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1090 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1100 DATA 2,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1110 DATA 1,2,2,2,2,2,2,1,1,1,2,2,1,1,2,2,2,
2,2,1
1120 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1130 DATA 1,2,1,1,2,2,1,1,2,1,1,1,1,1,2,1,1,
2,1,1
1140 DATA 1,1,1,2,2,2,2,2,2,1,2,1,2,2,2,2,2,
2,2,1
1150 DATA 1,2,2,2,1,1,1,2,2,2,2,2,2,2,2,2,2,
2,2,2
```

```
1160 DATA 1,2,2,2,1,2,2,1,1,2,2,2,1,2,2,1,2,
2,2,1
1170 DATA 1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1180 DATA 2,2,2,2,1,2,1,2,2,1,2,2,2,1,1,2,2,
2,2,1
1190 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1200 DATA 1,1,1,1,1,2,2,2,1,2,2,2,2,2,2,2,2,
2,2,1
1210 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2
1220 DATA 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1230 DATA 1,2,2,2,2,2,2,1,2,2,2,2,2,2,1,1,1,2,1,
1,1,2
1240 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1250 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,
2,2,1
1260 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1270 DATA 1,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,1,
2,2,1
1280 DATA 1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1
1290 DATA 1,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,
2,1,1
1300 DATA 1,2,2,2,2,2,2,1,1,1,2,2,2,2,2,2,2,1,
1,2,1
1310 DATA 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1
1320 DATA 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1
1330 S=0
1340 BORDER 0:INK 0,0:INK 2,2:INK 1,24
1350 A=18:B=14:C=0:D=0:E=A:F=B
1360 SYMBOL AFTER 96
1370 SYMBOL 97,60,126,255,255,255,255,126,60
1380 SYMBOL 98,0,66,195,231,255,255,126,60
1390 SYMBOL 99,60,126,248,240,240,248,126,60
```

```
1400 SYMBOL 100,60,126,255,255,231,195,66,0
1410 SYMBOL 101,60,126,31,15,15,31,126,60
1420 SYMBOL 102,28,62,42,107,127,127,109,73
1430 A=18:B=14:C=0:D=0:E=A:F=B
1440 A$="e"
1450 RESTORE 1490
1460 FOR T=1 TO 2
1470 READ X(T),Y(T)
1480 NEXT
1490 DATA 4,3,35,22
1500 RETURN
```

Letter Slide

Take yourself back to your early childhood . . . do you remember those squares with little interlocking tiles, each with a letter on it. There was always one space among the tiles, and you had to shuffle the pieces around the square, trying to create words or to get them back into alphabetical order. Well, this program re-creates those tile puzzles of yesteryear.

In this computerized version, you must get the tiles back into alphabetical order with the space occupying the bottom right-hand corner. You must enter both the co-ordinates for the piece you want to move and where you want to move it to; the co-ordinates are shown on the right-hand side of the screen. Illegal moves are rejected immediately.

This may take you some time to figure out — give your brain a stretch instead of your fingers for a change!

```
10 ' LETTER SLIDE
20 '
30 GOSUB 390
40 GOSUB 190
50 GOSUB 190
60 SOUND 1,180:SOUND 1,240:SOUND 1,120
70 PRINT:PRINT:PRINT
80 PEN 2
90 PRINT "WHICH PIECE DO YOU WANT TO MOVE ?"
100 INPUT X
```

```
110 IF A(X)=32 THEN 100
120 INPUT "TO WHERE";Y
130 IF A(Y)<>32 THEN 120
140 A(Y)=A(X)
150 A(X)=32
160 GO=GO+1
170 GOTO 50
180 '
190 ' PRINT OUT
200 '
210 PEN 2
220 CLS
230 PRINT TAB(14);"LETTER SLIDE"
240 PRINT TAB(13);"=============="
250 PRINT:PRINT
260 PRINT "MOVE NUMBER:";GO
270 PRINT:PRINT
280 L$=CHR$(143)+CHR$(143)+CHR$(143)
+CHR$(143)+CHR$(143)+CHR$(143)+CHR$(143)
+CHR$(143)+CHR$(143)+CHR$(143)+CHR$(143)
290 PEN 3
300 PRINT TAB (8);L$
310 PRINT TAB(8);CHR$(143);:PEN 1:PRINT "
";CHR$(A(1));" ";CHR$(A(2));" ";CHR$(A(3));"
";CHR$(A(4));" ";:PEN 3:PRINT CHR$(143);"1 2 3
4"
320 PRINT TAB(8);CHR$(143);" ";:PEN 1:PRINT
CHR$(A(5));" ";CHR$(A(6));" ";CHR$(A(7));" "
;CHR$(A(8));" ";:PEN 3:PRINT CHR$(143);"5 6 7
8"
330 PRINT TAB(8);CHR$(143);" ";:PEN 1:PRINT
CHR$(A(9));" ";CHR$(A(10));" ";CHR$(A(11));"
";CHR$(A(12));" ";:PEN 3:PRINT CHR$(143);"9
10 11 12"
340 PRINT TAB(8);CHR$(143);" ";:PEN 1:PRINT
CHR$(A(13));" ";CHR$(A(14));" ";CHR$(A(15));
" ";CHR$(A(16));" ";:PEN 3:PRINT
CHR$(143);"13 14 15 16"
350 PEN 3:PRINT TAB(8);L$
360 RETURN
370 PEN 2
```

```
380 '
390 ' INITIALISE
400 '
410 DIM A(16)
420 FOR B=1 TO 16
430 READ M
440 A(B)=M+64
450 NEXT
460 GO=1
470 RETURN
480 DATA 9,14,5,2,11,6,1,4,12
490 DATA 7,-32,10,13,8,3,15
```

Another Brick in the Wall

I'm sure the title didn't fool you! Yes, this is a version of the famous *Breakout* game, souped up and ready to run on the Amstrad.

Move your bat left and right with the 7 and 9 keys on the numeric key pad. (You will be surprised how comfortable it is to use those two keys — a fact discovered after many, many hours of extensive play-testing!)

```
10 ' ANOTHER BRICK IN THE WALL
20 '
30 H=0
40 INK 0,0:INK 1,22:INK 3,3
50 INK 2,2:INK 3,6
60 BORDER 0
70 SPEED KEY 1,2
80 GOSUB 660
90 SPEED KEY 1,2
100 FOR Z=1 TO 5
110 A$=A$+A$
120 NEXT Z
130 PEN 3:CLS
140 LOCATE 3,1
150 PRINT A$
160 LOCATE 3,2
170 PRINT A$
180 LOCATE 3,3
```

```
190 PRINT A$
200 G$=A$:LOCATE 3,22:PRINT G$
210 PEN 1
220 LOCATE 8,25
230 PEN 3
240 FOR X=1 TO 22
250 LOCATE 2,X
260 PRINT "d";
270 LOCATE 35,X
280 PRINT "d";
290 NEXT X
300 IF LV=0 THEN 570
310 FOR T=1 TO 1500:NEXT:SOUND 1,100
320 P$=INKEY$
330 IF P$="9" AND A<30 THEN A=A+2
340 IF P$="7" AND A>2 THEN A=A-2
350 IF A=2 THEN LOCATE A,21:PEN 3:PRINT
"d";:PEN 2:PRINT" abc ":GOTO 380
360 IF A=30 THEN LOCATE A,21:PRINT "abc";:PEN
3:PRINT"d":PEN 2:GOTO 380
370 LOCATE A,21:PEN 2:PRINT " abc "
380 LOCATE D,C:PRINT " "
390 IF C>21 THEN LV=LV-1:SOUND
1,400:C=6:D=INT(RND*25)+10:LOCATE
2,22:PEN 3:PRINT G$+"d":PEN 1:GOTO 300
400 IF D>33 OR D<4 THEN G=-G:SOUND 1,140,15
410 C=C+F
420 D=D+G
430 LOCATE D,C:PEN 1:PRINT B$:PEN 2
440 IF C<4 THEN GOSUB 490
450 IF C<>21 THEN GOTO 470
460 IF D=A+2 OR D=A+3 OR D=A+4 THEN F=-F:SOUND
1,100,15
680 F=1:G=1:C=6
690 D=10+INT(RND*25):A$="d"
700 B$=CHR$(231)
710 REM***DEFINE GRAPHICS***
720 SYMBOL AFTER 95
730 SYMBOL 97,1,31,127,255,255,255,255,248
740 SYMBOL 98,255,255,255,255,255,255,0,0
750 SYMBOL 99,128,248,254,255,255,255,255,31
```

```
760 SYMBOL 100,254,254,254,0,239,239,239,0
770 REM*** a b c = SHIELD***
780 REM***   d   = BRICKS***
790 RETURN
```

Knightsbridge

Here's a variant of *Chess/Checkers* that's sure to get you in a twist!

This game is played on a seven-by-seven square board. The Amstrad tells you which piece you must move; each piece is indicated by entering a two-digit number, the first being the co-ordinate of the side of the board, and the second being the co-ordinate at the top. You must then decide where to move to. All the pieces move like Knights in *Chess* (hence the title).

You capture an enemy piece by landing on top of it. The first player to capture five of the enemy's seven pieces is the winner. The Amstrad is a worthy opponent, both fast and skilful — you'll need all the luck you can muster!

```
10 ' KNIGHTSBRIDGE
20 '
30 GOSUB 840
40 GOSUB 550
50 IF HU=5 OR CO=5 THEN 750
60 GOSUB 340
70 GOSUB 550
80 IF HU=5 OR CO=5 THEN 750
90 GOSUB 120
100 GOTO 40
110 '
120 ' PLAYER MOVES
130 '
140 Q=0
150 M=INT(RND*66)+11
160 Q=Q+1
170 IF Q=500 THEN 750
180 IF H(M)<>72 THEN 150
190 PRINT "YOU MUST MOVE THE PIECE ON";M
200 INPUT N
210 IF N=99 THEN Q=500:GOTO 750
```

```
220 '
230 ' CHECK IF MOVE LEGAL
240 '
250 P=0
260 CT=1
270 IF M+Z(CT)=N THEN P=1
280 IF CT<8 THEN CT=CT+1:GOTO 270
290 IF P=0 THEN PRINT "ILLEGAL MOVE":GOTO 200
300 IF H(N)=67 THEN HU=HU+1:PRINT
"WELL PLAYED!":FOR R=1 TO 800:NEXT
310 H(M)=46:H(N)=72
320 RETURN
330 '
340 ' COMPUTER MOVES
350 '
360 Q1=0
370 Q1=Q1+1
380 K=INT(RND*66)+11
390 IF Q1=500 THEN 750
400 IF H(K)<>67 THEN 370
410 PRINT "I HAVE TO MOVE THE PIECE ON";K
420 W=1
430 FOR T=1 TO 1800:NEXT
440 IF K+Z(W)<11 OR K+Z(W)>77 THEN 460
450 IF H(K+Z(W))=72 THEN PRINT "GOTCHA
!!!":CO=CO+1:FOR P=1 TO 300:NEXT:GOTO 510
460 IF W<8 THEN W=W+1:GOTO 440
470 W=1
480 IF (K+Z(W)<11 OR K+Z(W)>77)AND W<8 THEN
W=W+1:GOTO 480
490 IF H(K+Z(W))<>46 AND W<8 THEN W=W+1:GOTO 490
500 IF W=8 AND H(K+Z(W))<>46 THEN Q1=500:GOTO
750
510 X=K:Y=K+Z(W)
520 H(X)=46:H(Y)=67
530 RETURN
540 '
550 ' PRINT BOARD
560 '
570 CLS:PRINT:PRINT:PRINT
580 PRINT TAB(8); "MY SCORE IS";CO
```

```
590 PRINT TAB(7);"AND YOURS IS";HU
600 PRINT
610 PRINT TAB(8);"1 2 3 4 5 6 7"
620 PRINT TAB(8);"---------------"
630 FOR M=70 TO 10 STEP -10
640 PRINT TAB(5);M/10;
650 FOR N=1 TO 7
660 PRINT CHR$(H(M+N));" ";
670 NEXT
680 PRINT M/10:PRINT
690 NEXT
700 PRINT TAB(8);"---------------"
710 PRINT TAB(8);"1 2 3 4 5 6 7"
720 PRINT
730 RETURN
740 '
750 ' END OF GAME
760 '
770 GOSUB 550
780 IF HU=5 THEN PRINT "YOU HAVE BEATEN ME HUMAN"
790 IF CO=5 THEN PRINT "THIS VICTORY IS JUST MY
FIRST STEP TO WORLD DOMINATION!"
800 IF Q=500 THEN PRINT "I ACCEPT YOUR WISH TO
CONCEDE"
810 IF Q1=500 THEN PRINT "I CONCEDE TO A MASTER"
820 END
830 '
840 ' INITIALISE
850 '
860 CLS:PRINT "PLEASE STAND BY . . ."
870 DIM H(99),Z(8)
880 X=0:Q1=0:Q=0
890 RANDOMIZE TIME
900 HU=0:CO=0
910 FOR A=1 TO 99
920 IF A>77 OR A=70 OR A=60 OR A=68 OR A=69 OR
A=50 OR A=59 OR A=40 OR A=48 OR A=49 THEN 970
930 IF A=30 OR A=38 OR A=39 OR A=20 OR A=28 OR
A=29 OR A=18 OR A=19 OR A<11 THEN 970
940 H(A)=46
950 IF A>70 AND A<78 THEN H(A)=67
```

```
960 IF A>10 AND A<18 THEN H(A)=72
970 NEXT
980 FOR A=1 TO 8:READ Z(A):NEXT
990 DATA -8,-21,-12,-19,19,12,21,8
1000 RETURN
```

14
WORD PROCESSING

Letter Writer

One of the most useful pieces of software available for both business and home users is a wordprocessor. *Amsword*, the Amstrad's version of Tasman's *Tasword 2*, is widely regarded as a superb wordprocessor.

However, many users only want a letter-writing program allowing them to quickly edit mistakes, automatically print the address and output the formatted letter to the printer.

Letter Writer, the program that follows this introduction, is a simple attempt to provide a letter-writing program that is easy to use. The program parts are carefully noted, so that you can alter them at will to suit your own requirements. It would certainly make the basis for a more complex system with full justification, word count, and a host of other facilities should you want to make your own modifications.

Using *Letter Writer* is straightforward. Type up to the line on the screen (which gives you 66 characters to a line) and then press the ENTER key; the first line of your letter is now registered in the Amstrad's memory. Up to 60 lines can be held by the Amstrad, but you can manipulate that figure if your needs are different. You must note that if a line exceeds 66 characters when entered, it is wiped out and you must type it again. Instructions are provided on-screen indicating the controls you'll need to save a letter on to tape for later retrieval, to start a letter from scratch at any time or to print out a finished letter.

One final point. A fictitious address appears in lines 280–310 — this should be replaced with your own address in the same format.

```
10 ' LETTER WRITER .2
20 '
30 K=0:T=1
40 MODE 2
50 INK 0,0
```

```
60 PRINT:PRINT:PRINT " PRESS 2 FOR GREEN, ANY
KEY FOR WHITE"
70 A$=INKEY$:IF A$="" THEN 70
80 IF A$="2" THEN INK 1,18 ELSE INK 1,26
90 PRINT:PRINT:PRINT " DATE PLEASE":LINE INPUT
D$
100 CLS
110 GOSUB 840
120 DIM W$(62)
130 LOCATE 1,1
140 FOR T=1 TO 60
150 LINE INPUT W$(T):IF LEN(W$(T))>60 THEN
SOUND 1,100,9:LOCATE 1,T:PRINT " |";:GOSUB
840:W$(T)="":GOTO 150
160 IF W$(T)="\" THEN K=T:T=60
170 IF W$(T)="^" THEN CLS:GOSUB 840:GOTO 130
180 IF W$(T)="/" THEN 400
190 IF W$(T)="]" THEN 560
200 IF T=40 THEN GOSUB 840
210 NEXT T
220 PRINT #8,CHR$(27);CHR$(69)
230 PRINT #8,CHR$(27);CHR$(65);CHR$(16)
240 WIDTH 66
250 '
260 ' NOTE THAT TO CHANGE DUMMY ADDRESS, JUST
ENTER OWN ADDRESS INTO THE DATA STATEMENTS
270 '
280 PRINT #8,TAB(45);"18 EDWARD WAY,"
290 PRINT #8,TAB(45);" ASHFORD,"
300 PRINT #8,TAB(45);" MIDDX."
310 PRINT #8,TAB(45);" TW15 3AY."
320 PRINT #8," "
330 PRINT #8,TAB(42);D$
340 PRINT #8," "
350 FOR T=1 TO K-1
360 PRINT #8," ";W$(T)
370 NEXT T
380 END
390 '
400 ' SAVE FILE
410 '
```

```
420 CLS
430 LOCATE 27,8
440 PRINT "ENTER FILE NAME (UP TO 14 LETTERS)"
450 PRINT:PRINT
460 INPUT FL$
470 PRINT:PRINT
480 OPENOUT FL$
490 FOR T=1 TO 60
500 PRINT #9,W$(T)
510 NEXT
520 PRINT #9,D$
530 CLOSEOUT
540 CLS:GOSUB 840:GOTO 130
550 '
560 ' LOAD FILE
570 '
580 CLS
590 LOCATE 30,4
600 PRINT "ENTER FILE NAME TO LOAD"
610 PRINT:PRINT
620 INPUT FG$
630 PRINT:PRINT:PRINT
640 OPENIN FG$
650 FOR T=1 TO 60
660 INPUT #9,W$(T)
670 NEXT
680 INPUT #9,D$
690 IF EOF=1 THEN CLOSEIN
700 CLOSEIN
710 CLS
720 GOSUB 840
730 LOCATE 1,1
740 FOR T=1 TO 60
750 PRINT W$(T)
760 WHILE INKEY$="":WEND
770 NEXT
780 GOSUB 840
790 T=61:INPUT "WHAT NEXT";W$(T)
800 GOTO 160
810 '
820 ' MENU PRINTED
```

```
830 '
840 LOCATE 61,1
850 PRINT "--LETTER WRITER-- ";
860 FOR G=2 TO 24:LOCATE 61,G:PRINT "|";:NEXT G
870 LOCATE 64,8:PRINT "'\' = print"
880 LOCATE 64,12:PRINT "'^' = re-start"
890 LOCATE 64,16:PRINT "'/' = SAVE file"
900 LOCATE 64,20:PRINT "']' = LOAD file"
910 LOCATE 1,T
920 RETURN
```

15
A GRAPHIC EXPLANATION

In recent years, the greatest advances in home computers have undoubtedly been in the field of graphics. Four or five years ago, home computers under £400 (and there weren't that many around) often had no more than block graphic displays — that is, the same resolution as their text display. To gain a small block medium resolution of 60 by 100, micro owners would often have to resort to expensive peripheral packages.

Your Amstrad, on the other hand, has some excellent graphic features already built-in: 27 colours, a maximum resolution of 640 by 200, commands to draw lines, plot complex shapes, test for the colour of a certain position, perform animation and much, much more. And the nice part is that using these features doesn't gobble up its precious memory.

In this chapter, we'll be looking at the range of features that the Amstrad offers, explaining each in turn. The chapter is split into three major sections — the commands dealing with colour and the text with screen, those involved user-definable graphics creation, and those controlling high-resolution graphics. In each section, there are a number of programs — some are purely for demonstration, but you'll find many of great use, especially when incorporated within your own programs.

Colour, modes and text commands

By now, you will have noticed that the Amstrad has a number of modes that relate to the number of characters you can have across the screen, the number of colours that can be present on the screen at any one time, and the level of resolution for the graphics commands.

Mode 1 is used most frequently, and allows 40 characters across the screen when processing text, up to four colours on the screen at any one time and has a graphics resolution of 320 by 200.

Mode 2 is more likely to be used for business purposes as it allows 80 characters per line (making it ideally suitable for CP/M — which is available on disc for the Amstrad). Only two colours — a background and a foreground colour — are permitted, but you're given the opportunity to make use of the highest resolution available on the Amstrad, namely 640 by 200.

Mode 0 only allows 20 characters across the screen and has a maximum resolution of 160 by 200. However, its great advantage is that up to 16 colours can be displayed on-screen at one time.

All three modes are accessed by the MODE command, which is followed by a number zero, one or two.

You may be a little confused over the term 'high resolution'. Later on in this chapter, we'll be looking at the various commands controlling the use of high-resolution graphics, but for the moment, it's worth discussing what 'high resolution', and the term 'pixel', actually mean.

If you look carefully at a character on the screen, you can see that it's made up of a number of tiny dots. Each dot is called a 'pixel', and each pixel has its own allotted place on the screen. The 'high-resolution' dimensions indicate the number of pixels across the screen by the number of pixels up and down the screen.

As each character is made up of a number of pixels, we can say that high-resolution graphics commands allow you to access each individual pixel whereas text commands only allow you to alter whole characters and their position.

Let's now look at the text screen and see how characters are printed on to it. Many of the characters available can be found simply by pressing a key on the keyboard; however, there are a number of characters that cannot be accessed straight from the keyboard and these have to be printed using the CHR$ command. If you print CHR$(n), where n is the code of the character between 32 and 128, you'll obtain a character on the screen. For example, printing CHR$(224) will give you a smiling face on the screen. (A complete list of the characters available is at the back of your Amstrad manual.)

We have already discussed the PRINT statement, and seen how TAB can be used to move it over a specified number of columns and how the semi-colon can be used as a separator. Let's now take a look at the LOCATE command in action.

If you have used other computers, then you will recognize a command that allows you to print at a certain column co-ordinate and a certain line co-ordinate. Many computers' versions of this command

are called PRINT AT or PRINT @, but on the Amstrad it is called LOCATE.

LOCATE is followed by two numbers, essentially the X co-ordinate and the Y co-ordinate. Here's an example of LOCATE in use:

```
10 ' LOCATE, LOCATE
20 '
30 CLS
40 FOR Y=1 TO 25:' Y COORDINATE
50 FOR X=1 TO 39 STEP 3:' X COORDINATE
60 LOCATE X,Y
70 PRINT STR$(X+1);
80 NEXT:NEXT
90 GOTO 90
```

Two commands look after the position of the text cursor. They are VPOS and POS, and here they are:

```
10 ' POS/VPOS CREATE A QUARTER SCREEN
20 '
30 CLS
40 FOR T=1 TO 3000
50 A$=INKEY$:IF A$="" THEN 50
60 IF POS(#0)>20 THEN PRINT:GOTO 90
70 IF VPOS(#0)>12 THEN END
80 PRINT A$;
90 NEXT
```

The above program uses VPOS and POS to create a mini-screen, almost like a window. Whatever key is pressed is displayed on the screen and, when you get more than 20 characters in a horizontal line, a new line is started (as occurs on the proper screen). When 12 whole lines have been filled, the program stops.

You may notice the 0 in brackets after both VPOS and POS. This refers to the stream type (discussed primarily in Chapter 19). Zero is the stream code for the whole screen but POS and VPOS can also be used to check the position of the windows, cassette and printer cursors; POS and VPOS are very useful commands to use in programs like wordprocessors. The bracketed #N, where N is a stream number between zero and nine, must be placed after each VPOS or POS statement.

As you have almost certainly already deduced, VPOS handles the vertical position of the cursor while POS looks after the horizontal

position. You will meet their two close relatives, XPOS and YPOS, in the section concerned with high-resolution graphics.

A rainbow of colours

Your Amstrad is gifted with great colour facilities — some 27 colours in all, which can be used for the foreground, background and border of your screen display. Many users who have had to put up with other machines in the past will welcome less common colours such as grey and brown which are often very useful in games and pictures.

Returning to our discussion of the three screen modes, you will remember that there is a limit to the number of different colours you can have on the screen at one set time. This number varies with the mode, and there is a trade-off between the level of resolution and the number of colours permitted.

The easiest way to see each of these colours is via the BORDER command. This is a very easy-to-use command that defines the colour of the border of the screen; the number following the command indicates the colour. Table 1 shows the available colours and their codes.

The short example program below displays a different colour border every time you press a key on the keyboard:

```
10 ' COLOUR BORDER
20 '
30 FOR T=0 TO 26
40 BORDER T
50 WHILE INKEY$="":WEND
60 SOUND 1,400-(12*T)
70 NEXT
```

A flashing border can be created by placing two numbers after the BORDER command — the display will then flash between the two colours. Type a few BORDER commands followed by two numbers to see the effect created. If you do not like the speed of the flashing, try the SPEED INK command. The format for this command is:

SPEED INK A, B

Both A and B are numbers of frames (there are 50 frames a second, as a rule), and they determine how long the border will stay

Table 1 Amstrad's colours and codes

Colour code	*Colour*
0	Black
1	Dark blue
2	Bright blue
3	Red
4	Magenta
5	Mauve
6	Bright Red
7	Purple
8	Bright magenta
9	Green (shows as khaki/green-brown)
10	Cyan
11	Sky blue
12	Yellow
13	White (shows as grey)
14	Pastel blue
15	Orange
16	Pink
17	Pastel magenta
18	Bright green
19	Sea green
20	Bright cyan
21	Lime green
22	Pastel green
23	Pastel cyan
24	Bright yellow
25	Pastel yellow
26	Bright white

one colour before it changes to the next. Thus, if you want the first colour to be present for half a second and the second colour for a second and a half, then the SPEED INK command would be followed by the numbers 25 and 75. Here's a demonstration program for you to type in that incorporates several SPEED INK and BORDER settings. Experiment for yourself and see what effects you can create!

```
10 ' SPEEDY BORDERS
20 '
30 GOSUB 160
40 BORDER INT(RND*27),1
50 FOR T=1 TO 3500:NEXT
60 GOSUB 160
70 SPEED INK 5,5
80 BORDER 6,4
```

```
90 FOR T=1 TO 3500:NEXT
100 GOSUB 160
110 SPEED INK 80,10
120 BORDER 24,13
130 FOR T=1 TO 5000:NEXT
140 GOSUB 160
150 END
160 ' RESET AND PAUSE
170 BORDER 1
180 SOUND 1,100,20,15
190 SPEED INK 20,20
200 FOR T=1 TO 1000:NEXT
210 RETURN
```

The SPEED INK command can also be used to modify the next command that we will look at — the highly versatile INK command.

From earlier chapters, you have seen that we can modify the colour used for the foreground and the background using the commands PEN and PAPER, respectively. In Mode 1, for instance, we saw how to change to any of the available colours: dark blue, light blue, yellow or red.

The INK command allows us to change the colours that can be used by the PEN and PAPER commands. To make this clearer, consider a palette of four colours from which an artist can choose any one to use on the canvas. Now, if you think of the INK command being like a box of different paints — all you have to do is specify which colour you want to use. Following the INK command are at least two numbers, the first being the PEN/PAPER colour; PEN 0 is the background and PEN 1 is the foreground colour when your Amstrad is switched on. The second number is the new colour that is assigned to that PEN/PAPER number; this number comes from Table 1 given earlier in this chapter. Here's an example:

```
INK 1,5
```

This re-defines PEN 1 with colour 5 (mauve). Let's take a look at another:

```
INK 1,24
```

This time, PEN 1 is re-defined to its default colour 24 (bright yellow).

The best mode to demonstrate colours is Mode 0 that allows up to 16 different ones. Below is a short program that shows each of Mode 0's colours:

```
10 ' MODE 0'S COLOURS
20 '
30 MODE 0
40 FOR T=0 TO 15
50 PEN T
60 PRINT "COLOUR:";T
70 NEXT
80 PEN 12
```

You can see that several of the colours flash on and off. This is achieved by adding a third number to the INK statement; this means the PEN colour now flashes between the second number and the third number (when these last two numbers are translated into colour codes). Therefore, a command such as INK 0, 15, 6 would make PEN 0 (the background) flash between colours 15 (orange) and 6 (bright red).

The flashing rate of these colours can be altered in the same way as the BORDER command's colours; that is, via the SPEED INK command. And now you've seen that flashing colours can be very useful highlighting pieces of text, and you have seen the INK command and related commands, PEN, PAPER and BORDER, in action, why not attempt to add some extra colour to programs that you have already typed in on your Amstrad?

Your Amstrad also allows you to superimpose characters on to one another. At first, this may not seem to be of much practical value. However, you could superimpose an underline character under letters already on-screen, thus underlining an important word or phrase in a screenful of text. The program below shows how this can be done:

```
10 ' TRANSPARENT INKS
20 '
30 CLS
40 MODE 1
50 LOCATE 12,12
60  PRINT "This is an example of Underlining
using"
70 PRINT:PRINT "Transparent Ink generated by
CHR$(22)"
80 LOCATE 12,12
90 PRINT CHR$(22)+CHR$(1);"----------------
----------------------"
```

```
100 PRINT:PRINT "-------------------------
------------"
110 PRINT CHR$(22);CHR$(0)
```

The first part of line 90 and all of line 110 perform the necessary feat. By printing CHR$(22), the computer is in a position to allow you to superimpose characters over each other. However, this effect has still to be switched on by the CHR$(1) function following the CHR$(22). To switch the effect off, replace the CHR$(1) with CHR$(0).

Type PRINT CHR$(22);CHR$(1) directly into the computer and then experiment with the shapes that can be created by placing two or more characters on to the same position. Once you've played around and formed some amazing characters more by accident than judgement, move on to the next section where we'll discuss how to define characters *exactly* as you want them.

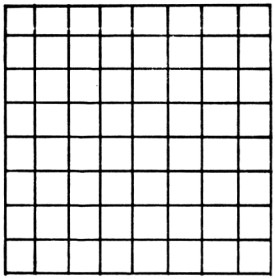
User-definable graphics

The characters that can be printed using the CHR$ command can also be re-defined using the SYMBOL and SYMBOL AFTEr commands. Any shape that you can design within the confines of the eight-by-eight grid can be turned into any characters 32 to 255. You'll be able to define your own shapes for games, foreign languages . . . or even a different style typeface can be created very easily.

The SYMBOL AFTER command first specifies the area of characters that you want to re-define. If there's no SYMBOL AFTER command used, the computer will only allow you to re-define the characters from 240 onwards. Therefore, if you want to re-define any of the upper-case alphabet, you would have to use something like SYMBOL AFTER 64. If you have re-defined a number of characters and want to change them back to their original design without stopping the program, then add the following as a line in your program: SYMBOL AFTER 255 — this wipes out all the re-defined characters.

Let's move on to the command that actually does the re-defining — SYMBOL. It has nine parameters, the first being the character that the command is to re-define, and the other eight being the individual line values of the eight-by-eight grid; the eight-line values can be any figure between zero and 255. Look at the sample grid below — you can see it has numbered rows and numbered columns.

128 64 32 16 8 4 2 1

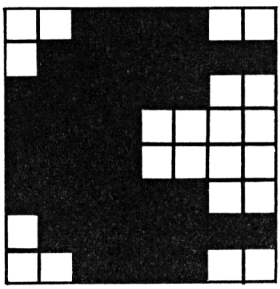


= **X1**
= **X2**
= **X3**
= **X4**
= **X5**
= **X6**
= **X7**
= **X8**

The symbol command would look like: SYMBOL N,X1,X2,X3,X4,X5,X6,X7,X8. But what are the numbers to define the columns? First, let's look at the design of the grid; the grid is made up of 64 squares, which can either be 'on' (black/having a foreground colour) or 'off' (white/having no foreground colour or being the same as the background). If the square in line X1 and column eight was 'on', then the value of that line would increase by eight. Whatever the number at the top of the column, if that column is 'on', then the value of the line that the square is in increases by the column's number. Thus, if all of row X1 was 'on', the value of that line would be 255; consequently, if nothing was 'on', the value would be zero.

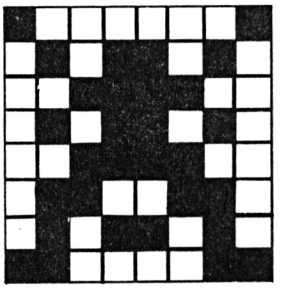
Let's illustrate this point with a proper example. Below are two re-defined characters that you may see in an arcade game, a pac-man shape and a space invader. Beside each shape is a number for each line. Looking along a line of the grid, add up the column values for the 'on' squares and confirm that they add up to the total given.

128 64 32 16 8 4 2 1



= **60**
= **127**
= **252**
= **240**
= **240**
= **252**
= **127**
= **60**

128 64 32 16 8 4 2 1



= 129
= 90
= 60
= 90
= 60
= 102
= 90
= 195

You should now begin to appreciate how user-defined graphics (or UDGs for short) are created.

One thing to learn from creating a large number of UDGs is that what you draw on a piece of graph paper doesn't always look the same when displayed on the computer screen. Remember that, on the Amstrad, though the characters are created on an eight-by-eight grid, the pixels themselves are not quite square. Also, you have the problems encountered when you move into a different graphics mode and your beautiful UDGs have suddenly become short and squat, or as thin as a rake. It is best to sketch your planned design out on paper and then experiment with it on-screen, changing the odd piece here and there until you're happy with the result.

Remember that you can create a shape from a number of UDGs printed together. Below is a program that draws a rather nice looking horse and rider from 11 UDGs all grouped together. Why not, as a programming exercise, add these horses into the *Horsey Horsey* gambling game given earlier in this book?

```
10 ' HORSE
20 '
30 SYMBOL AFTER 64
40 SYMBOL 65,0,0,0,0,0,0,0,3
50 SYMBOL 66,0,0,0,0,0,1,113,207
60 SYMBOL 67,0,28,18,42,87,132,15,238
70 SYMBOL 68,0,0,16,248,60,124,199,131
80 SYMBOL 69,5,13,13,9,9,9,9,0
90 SYMBOL 70,9,16,0,144,157,178,192,192
100 SYMBOL 71,221,209,33,1,194,242,11,10
110 SYMBOL 72,128,0,0,0,0,0,0,128
```

```
120 SYMBOL 73,3,2,2,2,2,1,0,0
130 SYMBOL 74,192,192,96,32,16,32,0,0
140 SYMBOL 75,13,28,25,19,18,8,0,0
150 CLS
160 LOCATE 18,12
170 PRINT "ABCD"
180 PRINT TAB(18);"EFGH"
190 PRINT TAB(18);"IJK"
```

To make things a whole lot simpler, here is a program that will help you design your own characters, a *UDG designer*. To use the designer, you must enter in the value of each line (a value from zero to 255) and the computer will convert your number into a large-scale version of the resultant character. Once the UDG is finished, the computer prints a row of them on the screen and waits for you to press the ESC key to break the program. Your UDGs can now be designed with ease . . . so type it in and start designing!

```
10 REM*****UDG DESIGNER*****
20 DIM G(8),D(8)
30 PRINT "WHICH MODE, 1 OR 0":INPUT L:IF L<>1 AND
L<>0 THEN PRINT:GOTO 30
40 CLS
50 MODE L
60 PRINT TAB(4); "UDG DESIGNER"
70 PRINT:PRINT:PRINT
80 PRINT " WHICH CHARACTER DO "
90 PRINT "YOU WISH TO REDEFINE"
100 INPUT N:IF N>255 OR N<32 THEN GOTO 100
110 SYMBOL AFTER N-1
120 PRINT:PRINT"OK."
130 CLS
140 PRINT TAB(4);"UDG DESIGNER"
150 PRINT:PRINT:PRINT
160 PEN 3
170 PRINT CHR$(143);CHR$(143);CHR$(143);
CHR$(143);CHR$(143);CHR$(143);CHR$(143);
CHR$(143);CHR$(143);CHR$(143)
180 FOR T=1 TO 8
190 PRINT CHR$(143);" ";CHR$(143)
200 NEXT
210 PRINT CHR$(143);CHR$(143);CHR$(143);
```

```
CHR$(143);CHR$(143);CHR$(143);CHR$(143)
CHR$(143);CHR$(143);CHR$(143);
220 PEN 1
230 FOR T=1 TO 8
240 LOCATE 5,20:PRINT " ":LOCATE 5,20
250 INPUT G(T):IF G(T)>255 OR G(T)<0 THEN 250
260 D(T)=G(T)
270 LOCATE 2,5+T
280 IF D(T)>127 THEN PRINT CHR$(143);:D(T)=
D(T)-128 ELSE PRINT " ";
290 IF D(T)>63 THEN PRINT CHR$(143);:D(T)=
D(T)-64 ELSE PRINT " ";
300 IF D(T)>31 THEN PRINT CHR$(143);:D(T)=
D(T)-32 ELSE PRINT " ";
310 IF D(T)>15 THEN PRINT CHR$(143);:D(T)=
D(T)-16 ELSE PRINT " ";
320 IF D(T)>7 THEN PRINT CHR$(143);:D(T)=
D(T)-8 ELSE PRINT " ";
330 IF D(T)>3 THEN PRINT CHR$(143);:D(T)=
D(T)-4 ELSE PRINT " ";
340 IF D(T)>1 THEN PRINT CHR$(143);:D(T)=
D(T)-2 ELSE PRINT " ";
350 IF D(T)>0 THEN PRINT CHR$(143);:D(T)=
D(T)-1 ELSE PRINT " ";
360 NEXT
370 LOCATE 4,24:PRINT "PRESS A KEY"
380 WHILE INKEY$="":WEND
390 SOUND 1,180
400 SYMBOL N,G(1),G(2),G(3),G(4),
G(5),G(6),G(7),G(8)
410 CLS
420 PRINT TAB(4);"UDG NUMBER:";N
430 PRINT
440 FOR T=1 TO 8:PRINT "LINE NO:";T"=";G(T):
PRINT:NEXT
450 FOR T=1 TO 13:PRINT CHR$(N);" ";:NEXT
460 ON BREAK GOSUB 480
470 GOTO 460
480 REM***RETURN TO MODE 1***
490 MODE 1
500 END
```

Tired of your typeface?

Are you, as the title suggests, getting a little bored with the Amstrad's alphabet? Do you fancy livening it up? Type in the following program and re-define both the upper- and lower-case alphabet!

The new-style typeface is a little difficult to describe — it's easier for you to type in the first couple of SYMBOL commands and then run them to get a glimpse of what the new set will look like. If you like the look of what you see on-screen, type in the rest of the program . . .

```
10 REM ' RE-DEFINED ALPHABET
20 SYMBOL AFTER 64
30 SYMBOL 65,56,100,226,226,254,226,226
40 SYMBOL 66,252,226,226,226,252,226,252
50 SYMBOL 67,124,226,224,224,224,226,124
60 SYMBOL 68,252,226,226,226,226,226,252
70 SYMBOL 69,254,224,224,224,248,224,254
80 SYMBOL 70,254,224,224,224,248,224,224
90 SYMBOL 71,124,226,224,224,238,226,124
100 SYMBOL 72,226,226,226,226,254,226,226
110 SYMBOL 73,56,56,56,56,56,56,56
120 SYMBOL 74,14,14,14,14,142,142,124
130 SYMBOL 75,226,228,232,240,232,228,226
140 SYMBOL 76,224,224,224,224,224,224,254
150 SYMBOL 77,130,198,234,242,226,226,226
160 SYMBOL 78,130,194,226,242,234,230,226
170 SYMBOL 79,124,226,226,226,226,226,124
180 SYMBOL 80,252,226,226,226,252,224,224
190 SYMBOL 81,124,226,226,226,234,228,122
200 SYMBOL 82,252,226,226,226,252,232,228
210 SYMBOL 83,124,226,224,124,14,142,124
220 SYMBOL 84,254,56,56,56,56,56,56
230 SYMBOL 85,226,226,226,226,226,226,124
240 SYMBOL 86,226,226,226,226,100,40,16
250 SYMBOL 87,226,226,226,242,234,198,130
260 SYMBOL 88,134,78,60,56,120,228,194
270 SYMBOL 89,134,78,60,56,56,56,56
280 SYMBOL 90,254,14,28,56,112,224,254
290 SYMBOL 97,0,0,124,14,126,142,126
300 SYMBOL 98,0,224,224,252,226,226,252
310 SYMBOL 99,0,0,124,226,224,226,124
320 SYMBOL 100,0,14,14,126,142,142,126
```

```
330 SYMBOL 101,0,0,124,226,252,224,124
340 SYMBOL 102,0,60,114,112,120,112,112
350 SYMBOL 103,0,0,124,142,142,126,14,124
360 SYMBOL 104,0,224,224,252,226,226,226
370 SYMBOL 105,0,56,0,56,56,56,56
380 SYMBOL 106,0,28,0,28,28,28,156,120
390 SYMBOL 107,0,224,226,228,232,244,226
400 SYMBOL 108,0,112,112,112,112,112,60
410 SYMBOL 109,0,0,244,234,234,234,234
420 SYMBOL 110,0,0,252,226,226,226,226
430 SYMBOL 111,0,0,124,142,142,142,124
440 SYMBOL 112,0,0,252,226,226,252,224,224
450 SYMBOL 113,0,0,126,142,142,126,14,14
460 SYMBOL 114,0,0,252,224,224,224,224
470 SYMBOL 115,0,0,124,224,124,14,124
480 SYMBOL 116,0,112,248,112,112,112,56,0
490 SYMBOL 117,0,0,226,226,226,226,124
500 SYMBOL 118,0,0,226,226,100,56,16
510 SYMBOL 119,0,0,234,234,234,234,116
520 SYMBOL 120,0,0,142,92,56,116,226
530 SYMBOL 121,0,0,142,142,142,126,14,124
540 SYMBOL 122,0,0,254,28,56,112,254
```

Finally, here is a small companion program to the last one, in which the numbers are re-defined to match the re-styled letters of the alphabet.

```
10 ' RE-DEFINED DIGITS
20 SYMBOL AFTER 47
30 SYMBOL 48,56,100,226,226,226,100,56
40 SYMBOL 49,12,28,60,92,28,28,28
50 SYMBOL 50,124,142,28,56,112,224,254
60 SYMBOL 51,124,142,14,28,14,142,124
70 SYMBOL 52,14,30,46,78,254,14,14
80 SYMBOL 53,254,224,224,252,14,142,124
90 SYMBOL 54,124,226,224,252,226,226,124
100 SYMBOL 55,254,14,28,56,112,224,224
110 SYMBOL 56,124,142,142,124,142,142,124
120 SYMBOL 57,124,142,142,126,14,142,124
```

The high resolution commands

As mentioned already, the Amstrad has three modes which provide 160 by 200, 320 by 200 and 640 by 200 high-resolution graphics, respectively. We know how to set up each of these modes and how to choose the required colours, so let's move on to using the actual commands we use to create pictures screen.

The PLOT commands seem a good place to start. PLOT must be followed by two numbers — an X co-ordinate and a Y co-ordinate — and is used to plot a pixel on the screen. The X co-ordinate is the value of the pixel's position *across* the screen, while the Y co-ordinate is the pixel's position *vertically*. As you would expect, the lowest value of X is the left-hand side of the screen starting at pixel one and moving across and up to 640.

The Y co-ordinate acts a little differently to how you might expect; its lowest number is at the bottom of the screen (value one) and rises until it reaches the top (400). You should be able to see the apparent anomaly in the above: if the Amstrad's vertical resolution is rated at a maximum of 200, how can the vertical co-ordinates range from one to 400? Well, the simple answer is go and ask the people who wrote the BASIC Locomotive software! But, from a programming point of view, there *is* a resolution of 200, and two numbers equal the same pixel. For instance, type in PLOT 1, 2 and then PLOT 1, 3. You should see no difference whatsoever on-screen as both numbers apply to the same pixel. However, please note that this only occurs with the vertical co-ordinates.

Following the two co-ordinates after the PLOT command is an optional parameter determining the colour of the pixel. To show this in action, type in the short routine below:

```
10 ' COLOUR PIXEL DEMO
20 '
30 MODE 0
40 INK 0,0
50 FOR T=1 TO 15
60 PLOT 320,200,T:PLOT 324,200,T
70 FOR D=1 TO 1000:NEXT
80 NEXT T
```

The PLOT command can be used in a variety of situations. It moves the graphics cursor's position and can be used for relocating the

cursor all round the screen. PLOT can also be used for graphs and creating fairly complex shapes. The latter function is illustrated in the next graphics routine; it draws a three-dimensional cone in the bottom-left quadrant of the screen. The plotting calculations that the Amstrad has to carry out to create this kind of picture are very involved indeed, so you will have to wait a minute or so for the final result. Just consider yourselves lucky — when a similar program was run on the ORIC 1, it took over 10 minutes to complete!

```
10 ' 3D CONE
20 '
30 DEFINT A-Z
40 CLS
50 MODE 1
60 FOR A=-100 TO 100
70 J=0:K=1:T=10
80 V=T*INT(SQR(10000-A*A)/T)
90 FOR B=V TO -V STEP-T
100            C=INT(80+30*SIN((SQR(A*A+B*B))/
12)-0.7*B)
110 IF C<J THEN 150
120 J=C
130 PLOT A110,C-15,1
140 K=0
150 NEXT:NEXT
160 SOUND 1,100,300,15
```

You can also use the PLOT command as a basis for a short routine to draw circles. There is no circle command on the Amstrad, but the following routine is the simple way to create circles and ellipses:

```
10 ' CIRCLE-DRAWING ROUTINE
20 '
30 CLS
40 FOR C=1 TO 360
50 DEG
60 PLOT 320,200,0
70 PLOT 320+100**COS(C),200+100*SIN(C),1
80 NEXT
```

The all-important line in the above program is line 70, where the plotting position is calculated and the plot executed. Let's have a closer look. The first figure, 320, indicates the horizontal position of

the centre of the circle; for the above routine, the middle of the screen has been chosen. The next number, 100, is the size of the circle's radius. The second part of the line shows the vertical position of the centre of the circle together with the radius figure again.

Bearing this information in mind, it is very simple to draw circles to your own specification. Change the 320 and the 200 in line 70, and match the figures in line 60 to your new circle centre co-ordinates. Now change the size of the circle by altering the two 100s in line 70. You should also be able to draw ovals, simply by making the two radius figures in line 70 different magnitudes; try altering just the first figure to 70 and see what happens.

If the first figure is smaller than the second, you should get an oval shape stretching *up* the screen; if the second figure is the smallest, then the oval will look more like a squashed circle stretching *along* the screen.

The PLOTR command is a very close relative of PLOT. The difference is that PLOTR plots a point from a position *relative* to the graphics cursor, whereas PLOT uses the *absolute* cursor position. Let's work our way through an example.

The cursor position is (320,200). Thus, PLOT 10, 10 would draw a pixel in the bottom left-hand corner of the screen, moving the cursor position to (10, 10). PLOTR, on the other hand, would plot a point 10 pixels above and 10 pixels to the right of the cursor position. The cursor position would now be at (330,210).

PLOTR can be used within the circle-drawing routine. In fact, it shortens it slightly, as you can see from the routine below:

```
10 ' CIRCLE-DRAWING ROUTINE USING PLOTR
20 '
30 CLS
40 FOR C=1 TO 360
50 DEG
60 PLOT 320,200,0
70 PLOTR 100*COS(C),100*SIN(C),1
80 NEXT
```

Drawing, moving and testing

Moving on from PLOT, three other high-resolution commands are DRAW, MOVE and TEST. All deal with the absolute cursor position and all have a version that deals with the relative cursor position. If you want to activate the relative versions of the commands, you just add an 'R' to the end of the names of the commands.

The DRAW command allows you to construct a line from the cursor position to any specified point around the screen. DRAW is a most useful command and a number of interesting effects stem from its use. We will be looking at a couple of these in a moment, but first let's see the syntax of the command:

DRAW X, Y, C.

where X is the X co-ordinate of the new cursor position, Y is the Y co-ordinate of the new cursor and C is the optional parameter for the colour of the line. The line is drawn between the old graphics cursor position and the new graphics cursor position. If you opt to use DRAWR, as with PLOTR, the X and Y values are added to the old cursor position if you want to work out the new cursor position. Check out the next demonstration program:

```
10 ' THE CAVE OF LIGHT
20 '
30 Z=100:F=0
40 CLS
50 GOSUB 120
60 WHILE INKEY$=""
70 F=F+1:IF F=20 THEN F=0:GOSUB 120
80 PLOT 320,200
90 DRAW INT(RND*640)+1,INT(RND*200)+1,
INT(RND*3)+1
100 WEND
110 END
120 FOR T=1 TO 180 STEP 2
130 DEG
140 PLOT 320,200
150 PLOTR Z*COS(T),Z*SIN(T),1
160 NEXT
170 Z=Z-INT(Z/15)
180 IF Z<25 THEN 200
```

```
190 RETURN
200 SOUND 1,350,100
210 GOTO 210
```

You'll need a joystick for the next program, although it can be easily converted to work from the keyboard via INKEY$. All you have to do is move the cursor around on-screen and hit the fire button when you want to draw a line.

```
10 ' DRAW A SHAPE
20 '
30 CLS
40 X=320:Y=200
50 N=JOY(0)
60 IF N=1 AND Y<400 THEN Y=Y+1
70 IF N=2 AND Y>1 THEN Y=Y-1
80 IF N=4 AND X>1 THEN X=X-1
90 IF N=8 AND X<640 THEN X=X+1
100 IF N=16 THEN DRAW X,Y
110 GOTO 50
```

The MOVE command simply places the graphics cursor in a new position without lighting a pixel. The MOVER command is the relative version of MOVE.

TEST checks for the colour of a given point on the screen. The syntax of TEST and TESTR are slightly different to MOVE and DRAW and, of course, neither of the colour-testing commands alter the cursor position. For example, PRINT TEST (X, Y) would give the colour of pixel at co-ordinates (X,Y).

Here is a program showing the TEST command in action:

```
10 ' STONEWALLING
20 '
30 A=10
40 B=200
50 CLS
60 FOR T=580 TO 640
70 PLOT T,1,3
80 DRAW T,400,3
90 NEXT
100 PLOT 1,1,1
110 TAG
```

```
120 WHILE TEST (A+14,B<>3
130 PLOT A,B
140 PRINT ">";
150 A=A+2
160 WEND
170 TAGOFF
180 LOCATE 15,12
190 PRINT "CCRRAASSHH!!"
```

TAG and TAGOFF, XPOS and YPOS

As you probably noticed in the last program, a command called TAG came into operation. What does TAG actually do? Well, the TAG command links a given stream to a graphics cursor, so that characters can be printed at the graphics cursor position. One immediate use can be seen in the previous program — it allows characters to be moved round the screen with much more flexibility and then, using the TEST command, we can check if a character is close to coming into contact with a different colour; TAG is invaluable when writing games programs.

The TAG command should be followed by a stream number; if none is specified, then the computer assumes that stream zero (the normal text screen) is to be TAGged with the graphics cursor. TAGOFF can also be seen in the last program. This does the exact opposite to TAG, switching off any link between the text and the graphics cursor.

Another feature of the Amstrad's graphics is the facility to check up on the position of the graphics cursor. As the cursor used for this task is invisible, it is necessary to have two commands that will always tell you its whereabouts on the screen.

XPOS provides the horizontal value of the invisible cursor, while YPOS deals with the vertical value. These can be used to create a 'wrap-around' effect on the graphics screen. On the text screen, any line of characters that tries to continue past the right-hand edge of the screen are transferred to the line immediately below. Using TAG, in conjunction with XPOS and YPOS, we can create the same effect on the graphics screen.

Take a look at the following demonstration if you're still a bit puzzled:

```
10 ' TEXT ON THE GRAPHICS SCREEN
20 '
30 A=10:B=200
40 CLS
50 TAG
60 RESTORE 180
70 A=10
80 FOR T=1 TO 54
90 IF XPOS>620 THEN A=10:B=B-40
100 MOVE A,B
110 IF YPOS<41 THEN B=360:GOTO 100
120 READ H$
130 PRINT H$;
140 A=A+40
150 NEXT
160 WHILE INKEY$="":WEND
170 GOTO 60
180 DATA T,H,I,S," ",P,R,O,G,R,A,M," "," ","
","  ",S,P,A,C,E,S," ",T,E,X,T," ",O,N," ","
",T,H,E,"  ",G,R,A,P,H,I,C,S,"  ","  ","  ","
",S,C,R,E,E,N
```

Here are a couple more graphics routines for your enjoyment. Type them in and try them out.

```
10 ' LEONARDO
20 '
30 MODE 0
40 CLS
50 A=INT(RND*640)+1
60 B=INT(RND*400)+1
70 C=INT(RND*15)+1
80 IF RND<0.4 THEN PLOT A,B,C:GOTO 110
90 IF RND<0.8 AND RND>0.39 THEN DRAW A,B,C:GOTO
110
100 IF RND>0.79 THEN GOSUB 140:GOTO 110
110 SOUND 1,INT(RND*600)+60,5
120 GOTO 50
130 '
140 ' DRAW A CURVE
150 '
160 FOR T=1 TO INT(RND*315)+45 STEP 2
```

```
170 DEG
180 PLOT A,B,C
190 PLOTR 100*COS(T),90*SIN(T),C
200 NEXT
210 RETURN
```

That last program came up with some pretty random displays — try the following for rather more ordered shapes. Just press a key when you want the next display.

```
10 ' 3 TO 12 SIDED SHAPES
20 '
30 FOR T=3 TO 12
40 MODE 1
50 ORIGIN 320,200
60 R=100
70 GOSUB 150
80 WHILE INKEY$="":WEND
90 ORIGIN 0,0
100 NEXT
110 END
120 '
130 ' DRAW SHAPE
140 '
150 DEG
160 MOVER 0,R
170 FOR N=0 TO 360 STEP INT(360/T)
180 DRAW R*SIN(N),R*COS(N)
190 NEXT
200 LOCATE 10,24
210 PRINT "A";T;"SIDED SHAPE"
220 RETURN
```

The graphics window

Like the other windows available on the Amstrad, this one must be defined in terms of its size. If the graphics window is used, all graphics are restricted to within that window and cannot appear anywhere else on the screen.

ORIGIN is the command that sets up the graphics window and is followed by six numbers, as shown below:

ORIGIN X co-ordinate value, Y co-ordinate value, left, right, top, bottom

The final four values determine the size and position of the graphics window and, once you have put these in, the computer may slightly alter them so that the window is a measurement of whole pixels only; this makes it easier to write whatever routines you want within the graphics window. A useful application for this feature is that of statistics, where numerical data can be shown on one portion of the screen while another portion has been allocated to display a graphical representation of that data.

The CLG command is just like CLS, but is used to clear the graphics window instead.

Here is a short program using ORIGIN and CLG to create a window within which a graph is displayed:

```
10 ' THE GRAPHICS WINDOW
20 '
30 INK 0,0
40 CLS
50 PEN 3
60 LOCATE 1,1
70 FOR T=1 TO 25:PRINT "TEXT:TEXT:
TEXT:TEXT:TEXT:TEXT:TEXT:TEXT:";
80 NEXT
90 ORIGIN 0,0,10,630,200,10
100 CLG
110 FOR T=30 TO 600
120 R=104+90*SIN(2*PI*T/600)
130 PLOT T,R,2
140 NEXT
150 GOTO 150
```

Animation

There have already been a number of examples of moving graphics in this book, from the simple cantering effect in *Horsey Horsey* to the more complicated movement in some of the action games. Also, we have already seen how movement can be achieved via the LOCATE command, and how TAG can be used to give humble text characters

movement normally associated with that found on the graphics screen.

What we haven't seen yet are any examples of real-time animation on the Amstrad. Animation is a tricky and arduous area of home computer graphics — good results can be achieved but, more often than not, it can take many hours of work before you see any success. Due to the need for extremely fast frame movement, you normally find that machine code is the only medium where you can be assured of success. However, there are certain types of computerized animation that your Amstrad can excel at.

Remember that horse we created out of a number of UDGs earlier on? Well, it makes a triumphant return in the next program which produces a cantering colt. The technique used with this program is the simplest form of animation available on computers — the Amstrad draws a shape; pauses for a short while; moves the shape's position and re-draws it, blanking out the old one. The main addition to the program is the movement of both the horse's legs and the jockey as well as the whole figure's movement across the screen.

```
10 ' CANTERING HORSE
20 '
30 DEFINT T
40 ON BREAK GOSUB 430
50 SYMBOL AFTER 64
60 SYMBOL 65,0,0,0,0,0,0,0,3
70 SYMBOL 66,0,0,0,0,0,1,113,207
80 SYMBOL 67,0,28,18,42,87,132,15,238
90 SYMBOL 68,0,0,16,248,60,124,199,131
100 SYMBOL 69,5,13,13,9,9,9,9,0
110 SYMBOL 70,9,16,0,144,157,178,192,192
120 SYMBOL 71,221,209,33,1,194,242,11,10
130 SYMBOL 72,128,0,0,0,0,0,0,128
140 SYMBOL 73,3,2,2,2,2,1,0,0
150 SYMBOL 74,192,192,96,32,16,32,0,0
160 SYMBOL 75,13,28,25,19,18,8,0,0
170 SYMBOL 76,128,128,128,0,0,0,0,0
180 SYMBOL 77,16,0,16,144,177,167,64,128
190 SYMBOL 78,69,1,1,1,194,246,20,20
200 SYMBOL 79,128,0,0,0,0,0,0,0
210 SYMBOL 80,128,128,128,128,128,64,0,0
220 SYMBOL 81,28,28,24,24,20,12,10,0
```

```
230 INK 0,9
240 INK 1,0
250 CLS
260 PLOT 1,300:DRAW 640,300,3
270 PLOT 1,100:DRAW 640,100,3
280 FOR T=1 TO 34
290 LOCATE T,12
300 PRINT " ABCD"
310 LOCATE T,13
320 PRINT " EFGH"
330 LOCATE T,14
340 PRINT " IJKL"
350 FOR H=1 TO 60:NEXT
360 LOCATE T,13
370 LOCATE T,13
380 PRINT " EMNO"
390 LOCATE T,14
400 PRINT " IPQ"
410 FOR H=1 TO 60:NEXT
420 NEXT
430 SYMBOL AFTER 255
```

A more complex form of animation can be achieved on the Amstrad using the INK colours. The next two programs demonstrate this admirably by drawing a cylinder whose coloured segments revolve smoothly.

Both programs consist essentially of two main parts. The first part draws the cylinder and colours the segments (four segments in the first program, and eight in the second) while the other performs the actual animation. It is the second part of the program that we are most interested in at the moment. Type in the first program and return to the text for an explanation of what's happening.

```
10 ' REVOLVING CYLINDER
20 '
30 GOSUB 240
40 GOSUB 60
50 '
60 ' PERFORM ANIMATION
70 '
80 D=1:H=1
90 FOR T=1 TO 4:READ C(T):NEXT
```

```
100 DATA 24,2,6,22
110 WHILE INKEY$=""
120 FOR F=1 TO 4
130 D=D+1:IF D=5 THEN D=1
140 INK F,C(D)
150 NEXT
160 ENT 1,5,-1,51:ENV 1,3,-2,85
170 SOUND 1,150,5,11,1,1,5
180 FOR G=1 TO 300:NEXT
190 H=H+1:IF H=5 THEN H=1
200 D=H
210 GOTO 120
220 WEND
230 '
240 ' DRAW
250 '
260 MODE 0
270 INK 1,24:INK 2,2:INK 3,6:INK 4,22
280 CLS
290 C=1
300 FOR T=1 TO 360
310 PLOT 320,100,0
320 DEG
330 IF T>90 THEN C=2:IF T>180 THEN C=3:IF T>270
THEN C=4
340 PLOT 320+100*COS(T),100+100*SIN(T),C
350 DRAWR 0,150,C
360 NEXT
370 RETURN
```

The colours required are all read into the INK commands one to four. The INKS are then changed by one in a loop; therefore, the quadrant that was INK 3 goes to INK 4, the one coloured INK 4 goes to INK 1, and so on. Thus, there is a shift of colours around the quadrant and, if a suitable pause is chosen, the effect is one of the cylinder revolving.

The second of the two programs creates a better effect by having eight coloured segments. Again, these INK colours are read into the first eight INK values and, using the same principle as before, a loop adds one to each INK value so that each colour shifts one place round the cylinder in an anti-clockwise direction.

Experiment with these programs by changing the shapes and colours of the picture and then, once you've fully understood the animation principle, why not try creating your own effect?

```
10 ' REVOLVING CYLINDER.2
20 '
30 GOSUB 220
40 GOSUB 60
50 '
60 ' PERFORM ANIMATION
70 '
80 D=1:H=1
90 FOR T=1 TO 8:READ C(T):NEXT
100 DATA 24,2,6,9,15,22,13,7
110 ON BREAK GOSUB 390
120 FOR F=1 TO 8
130 D=D+1:IF D=9 THEN D=1
140 INK F,C(D)
150 NEXT
160 FOR T=1 TO 30:NEXT
170 H=H+1:IF H=9 THEN H=1
180 D=H
190 GOTO 120
200 END
210 '
220 ' DRAW
230 '
240 MODE 0
250 INK 1,24:INK 2,2:INK 3,6:INK 4,9
260 INK 5,15:INK 6,22:INK 7,13:INK 8,7
270 CLS
280 C=1
290 FOR T=1 TO 360
300 PLOT 320,20,0
310 DEG
320 IF T>45 THEN C=2:IF T>90 THEN C=3:IF T>135
THEN C=4
330 IF C=4 AND T>180 THEN C=5:IF T>225 THEN C=6:
IF T>270 THEN C=7:IF T>315 THEN C=8
340 MOVE 320+200*COS(T),150+100*SIN(T)
350 DRAWR 0,150,C
360 NEXT
```

```
370 RETURN
380 '
390 ' RETURN TO NORMAL
400 '
410 MODE 1
420 INK 1,24:INK 3,6
430 END
```

In conclusion, this chapter should act as a springboard for you to create your own graphics effects and pictures. And now that you have the knowledge of commands such as DRAW, PLOT and TEST, you should be able to construct your own graphics pictures. A good starting-point is to draw a picture of a horse, using only straight lines. You can then progress on to more complex pictures and perhaps incorporate them into another program, perhaps an adventure of some kind.

The final program in this section is one that could have been featured in the chapter dealing with machine code. It is a machine-code routine that allows you to store an entire picture in the computer's memory and call it back on to the screen instantly. Various applications suggest themselves, but it could be used in an artist-type program, letting the user draw a picture on-screen, store it on tape, load it back into memory at any time and then display it in an instant on the screen. Games requiring complicated graphics screens could be drawn while the computer is displaying instructions on how to play and then, when the screen is needed, the computer would display it instantaneously!

Type in the following program and run it. When you have finished your screen picture, type in CALL 27488 to store the screen and CALL 27500 to display it.

This program is very tine and can thus be re-numbered at, say, line 20000 and used within any of your own programs.

```
10 MEMORY 27487
20 FOR T=27488 TO 27511
30 READ X:POKE T,X:NEXT
40 DATA 1,0,64,33,0,192,17,
122,107,237,176,201
50 DATA 1,0,64,33,122,107,17,
0,192,237,176,201
```

16
SOUNDS LIKE FUN

The Amstrad's sound features warrant a book in themselves, such is their complexity and potential. However, let's see what we can cover in the space available here.

First, let's look at the command allowing you to produce beeps and simple melodies. As you might have guessed, it is called SOUND and, for a simple tone, it is followed by two parameters:

SOUND CH, P

where CH is the channel chosen and P is the pitch specified. Let's start with the channel feature. The Amstrad has four channels — three sound channels and a noise channel that can be mixed in with the sound. However, for our purposes at the moment, let's just consider the three sound channels. As you probably know already, your Amstrad has a stereo sound facility, and this can be tested by connecting a stereo cassette recorder (or a proper Hi-Fi system) up to the stereo socket at the back of the Amstrad.

You have seven channels available with stereo and mono versions: channels one, two and four are all single channels, while channels three, five and six mix two channels together; channel seven gives you all three of the channels together, thus creating a form of stereo sound.

The pitch of the sound to be played can vary between zero and 4095. However, the extremes of this range are of little use; you will find that for most purposes values between 40 and 900 are sufficient. The demonstration sounds below show the simplest form of the SOUND command in action:

```
10 SOUND 1,100
20 WHILE INKEY$="":WEND
30 SOUND 1,500
40 WHILE INKEY$="":WEND
50 SOUND 1,300
60 END
```

The Amstrad first plays a short high-tone beep, and then waits for you to press a key. Once you've pressed a key, a much lower note is played and, if you press another key, a mid-range note is played that is between the two previous notes.

From the above demonstration, you can see that larger values for the pitch parameter create lower tones, and vice versa. But the SOUND command can also be used within a loop to create a rising or falling tone, as you will see if you type in the programs below:

```
10 FOR T=60 TO 250
20 SOUND 1,T
30 NEXT
```

```
10 FOR T=250 TO 60 STEP -1
20 SOUND 1,T,2
30 NEXT
```

The second of these programs features the number 2 after the SOUND 1,T command. This added parameter gives the duration of the note in hundredths of a second. Although the value in line 20 gives a very short note, when that value is used within a continuous loop, increasing or decreasing the pitch value by one each loop, then you will get just one continuous tone becoming higher or lower. If you make the loop in line 10 of the second program much shorter, say FOR T=120 TO 70, and shorten the duration of the tone still further to 1, you would then have a simple zap sound which you could use for a laser in an action game.

Of course, you might not want a continuous tone, but instead a series of distinct notes. If this is the case, try using the STEP command with a FOR/NEXT loop. The size of the STEP indicates the difference between the notes, as the short routine below shows:

```
10 FOR T=60 TO 400 STEP 20
20 SOUND 1,T
30 NEXT
```

Notice that the duration value in the above routine has been omitted; if no value is specified, then the computer chooses a value of a fifth of a second.

These separate notes, at present, bear no relation to proper music. But before we can construct a proper tune, we need to know the pitch values for all the musical notes; fortunately, Amsoft (the software division of Amstrad) has given these values in the back of the Amstrad manual. Here are a couple of note values just to illustrate the

point; at this stage, it might be a good idea to get your manual out and have a look at all the note values given.

Middle C =478 INTERNATIONAL A=284

From these note tables, it is very simple to build up a subroutine with several octaves of notes which can be used to play tunes. Let's have a look at an example:

```
10 ' MUSIC SUBROUTINE
20 '
30 GOSUB 1000
100 ' PLAY NOTES
110 FOR T=1 TO 24
120 SOUND 1,M(T)
130 NEXT
990 END
1000 '
1010 ' NOTE STORAGE
1020 '
1030 DIM M(24)
1040 FOR T=1 TO 24
1050 READ M(T)
1060 NEXT
1070 DATA 478,451,426,402,379,358,
338,319,301,284,268,253
1080 DATA 239,225,213,201,190,179,
169,159,150,142,134,127
1090 RETURN
```

The data (the note values) is read into an array M(n) and then the playing routine simply plays each separate note in ascending order, rather like a scale. These notes can be played by the SOUND command by choosing the appropriate note and substituting the required value; thus, if you want to play the lowest C within the routine, you would play the first note SOUND 1,M(1) followed by whatever duration you required.

Playing a tune using this method would involve having a number of SOUND statements, each with the correct element of the music array. Needless to say, this is hardly the most efficient method as far as memory goes — a much better way is shown below.

Here, the computer has a second set of data read into an array, which corresponds to the notes in the music array. Therefore, if the

first three notes you want to play are the first, seventh and third in the music array, the play array must have the values 1, 7 and 3 as the first three elements of the array.

Type in the program below, run it and listen out for the Amstrad's version of *Twinkle, Twinkle, Little Star*:

```
10 ' MUSIC SUBROUTINE
20 '
30 GOSUB 240
40 ' PLAY NOTES
50 DIM P(100)
60 RESTORE 100
70 FOR T=1 TO 47
80 READ P(T)
90 NEXT
100 DATA 1,1,8,8,10,10,8,0,6,6,5,5,3,3,1
110 DATA 0,8,8,6,6,5,5,3,0,8,8,6,6,5,5,3
120 DATA 0,1,1,8,8,10,10,8,0,6,6,5,5,3,3,1
130 '
140 ' NOTE THAT 0 EQUALS A REST
150 '
160 FOR T=1 TO 47
170 SOUND 1,M(P(T))
180 SOUND 1,0,10
190 '
200 ' NB. SOUND 1,0 CREATES A GAP BETWEEN NOTES
210 '
220 NEXT
230 END
240 '
250 ' NOTE STORAGE
260 '
270 DIM M(24)
280 RESTORE 320
290 FOR T=1 TO 24
300 READ M(T)
310 NEXT
320 DATA 478,451,426,402,379,358,338,
319,301,284,268,253
330 DATA 239,225,213,201,190,179,169,
159,150,142,134,127
340 RETURN
```

Now replace the play routine with the one given below and you have the tune *The Entertainer*. Remember that you must re-number the note storage routine so that it starts at line 290. This can be easily done by typing RENUM 290,240 before the new play routine is typed in.

```
10 ' MUSIC SUBROUTINE
20 '
30 GOSUB 290
40 ' PLAY NOTES
50 DIM P(120)
60 RESTORE 100
70 FOR T=1 TO 105
80 READ P(T)
90 NEXT
100 DATA 8,9,10,18,0,10,18,0,10,18,0,0
110 DATA 18,20,21,22,18,20,
22,0,17,20,0,18,0,0
120 DATA 8,9,10,18,0,10,18,0,10,18,0,0
130 DATA 15,13,12,15,18,22,0,20,17,15,20,0,0
140 DATA 8,9,10,18,0,10,18,0,10,18,0,0
150 DATA 18,20,21,22,18,20,22,
0,17,20,0,18,0,0
160 DATA 18,20,21,22,18,20,22,
0,18,20,0,22,18,20,22,0,18,20
170 DATA 0,22,18,20,22,0,17,20,0,18
180 '
190 ' NOTE THAT 0 EQUALS A REST
200 '
210 FOR T=1 TO 105
220 SOUND 1,M(P(T))
230 SOUND 1,0,4
240 '
250 ' NB. SOUND 1,0 CREATES A GAP BETWEEN NOTES
260 '
270 NEXT
280 END
```

The third and final tune is strictly for the winter months (I'm sure that you will recognize this simple refrain). Again, the note storage routine must be re-numbered — so, delete the play routine by typing DELETE 10–280 and then carry out the re-numbering operation with

RENUM 230,290.

```
10 ' MUSIC SUBROUTINE
20 '
30 GOSUB 230
40 ' PLAY NOTES
50 DIM P(120)
60 RESTORE 100
70 FOR T=1 TO 28
80 READ P(T)
90 NEXT
100 DATA 10,10,10,0,10,10,10,0,10,13,6,8,10
110 DATA 0,0,11,11,11,11,11,10,
10,10,13,13,11,8,6
120 '
130 ' NOTE THAT 0 EQUALS A REST
140 '
150 FOR T=1 TO 28
160 SOUND 1,M(P(T))
170 SOUND 1,1,8,0
180 '
190 ' NB. SOUND 1,0 CREATES A GAP BETWEEN NOTES
200 '
210 NEXT
220 END
```

Speaking volumes

The volume parameter is the fourth value on the SOUND command. Although there's a volume control on the Amstrad (it is positioned on the side of the case on the right, next to the ON/OFF switch) you can control the volume of the sound through software. The volume parameter can lie between one and 15, with a value of 15 being the loudest; the computer defaults to a middle value and, to be honest, this is one of the least used of the SOUND parameters. However, it is often best to use a middle value for the majority of sounds within a program — that way, one still has enough leeway to create a loud, surprising sound (such as an explosion) or a quiet tone (such as a rustle). Below is a rather interesting effect that can be created using a variable volume level:

```
10 ' FADE TO PLAY
20 '
30 FOR T=7 TO 1 STEP -1
40 SOUND 1,200-CINT(3.5*T),20,T
50 NEXT
```

Compose your own

The following program allows you to construct your own tunes for the computer to play back to you while showing you how it looks on a musical stave. You enter the notes in letter form, with a space for each pause. A sample tune may look like AB DCEFDC CDFEGABCC. You can enter up to 40 notes, though this number can be easily extended using Mode 2 or by setting up two staves on one screen.

After the computer has accepted your tune (skipping over any invalid letters), you will be asked to enter the tempo of the piece and the volume at which you would like your musical masterpiece to be played.

Each note is displayed on a treble stave as it is played by the computer. If you're not happy with the final result, why not change the length of the pauses or look ahead to the sound effects section of this chapter and transform the Amstrad into a synthesizer. The possibilities for this program are numerous, it really is up to you!

```
10 ' SIMPLE COMPOSER
20 '
30 MODE 1
40 WINDOW #1,3,37,22,25
50 INK 3,24
60 PAPER #1,3
70 PEN #1,1
80 INK 1,0:INK 0,26
90 Y=0
100 GOSUB 890
110 CLS
120 PRINT TAB(10);"THE COMPOSER"
130 PRINT TAB(10);"============"
140 PRINT:PRINT:PRINT
150 CLS #1
```

```
160 PRINT #1," Bottom C to B, type letter in"
170 PRINT #1," Top C, type H. Top D, type I"
180 PRINT #1," Top E, type J. Top F, type K"
190 INPUT "YOUR COMPOSED PIECE";A$
200 A$=UPPER$(A$)
210 INPUT "TEMPO (BETWEEN 1 AND 40)";P
220 IF P>40 OR P<1 THEN PRINT:GOTO 210
230 INPUT "VOLUME (BETWEEN 1 AND 15)";V
240 IF V>15 OR V<1 THEN PRINT:GOTO 230
250 GOSUB 790
260 X=1
270 IF LEN(A$)>38 THEN A$=LEFT$(A$,38)
280 IF LEN(A$)>19 THEN F=1 ELSE F=2
290 FOR T=1 TO LEN(A$)
300 IF ASC(MID$(A$,T,1))=32 THEN FOR J=1 TO
10*P:NEXT:SOUND 1,0,P*2,0:GOTO 370
310 IF ASC(MID$(A$,T,1))<64 OR ASC(MID$(A$,T
,1))>75 THEN 390
320 ON ASC(MID$(A$,T,1))-64 GOSUB 450,480,
510,540,570,600,630,660,690,720,750
330 LOCATE X,Y:PRINT CHR$(237);
340 Z=Y-6:Z=13-Z
350 FOR J=1 TO 10*P:NEXT
360 SOUND 1,M(Z),P*2,V
370 FOR J=1 TO 10*P:NEXT
380 X=X+F
390 NEXT
400 CLS #1
410 INPUT #1," DO YOU WANT TO HEAR IT AGAIN (R) OR
DO YOU WANT TO START AGAIN (S) ";J$
420 IF J$="S" THEN RUN
430 IF J$="R" THEN GOSUB 790:GOTO 260
440 GOTO 400
450 ' A
460 Y=12
470 RETURN
480 ' B
490 Y=11
500 RETURN
510 ' C
520 Y=17
```

```
530 RETURN
540 ' D
550 Y=16
560 RETURN
570 ' E
580 Y=15
590 RETURN
600 ' F
610 Y=14
620 RETURN
630 ' G
640 Y=13
650 RETURN
660 ' TOP C
670 Y=10
680 RETURN
690 ' TOP D
700 Y=9
710 RETURN
720 ' TOP E
730 Y=8
740 RETURN
750 ' TOP F
760 Y=7
770 RETURN
780 '
790 ' SCREEN
800 '
810 CLS
820 PRINT TAB(14);"TREBLE STAVE"
830 LOCATE 1,6
840 FOR T=1 TO 5
850 PRINT:PRINT STRING$(40,"_");
860 NEXT
870 PRINT:PRINT
880 RETURN
890 ' NOTE STORAGE
900 '
910 DIM M(12)
920 FOR T=1 TO 12
930 READ M(T)
```

```
940 NEXT
950 DATA 478,426,379,358,319,284,253,
239,213,190,179,159
960 RETURN
```

So far, all the sound features discussed have been based around varying the length and pitch of various notes. The resultant sound is much the same as you would expect to hear from a small electronic organ. However, your Amstrad is equipped with much greater sound facilities than this — using the ENT and ENV commands, it is possible to create all sorts of synthesized sounds.

Sound synthesis is a complex subject indeed, and it would be impossible to cover all the features of your Amstrad's sound chip — so we will only concentrate on a few of the major features that you are most likely to want to use at first.

Let's start with a bit of theory. Assuming that you know a little about music, such as a bit of musical notation and so on, we'll now start investigating the noise channel with a view to creating some very realistic sound effects.

First, you must switch off the pitch parameter of the SOUND command; this allows you access to 'noise' as opposed to musical tones. Noise can be used to create many realistic effects, from gunshots and explosions to the different sounds of musical instruments. There are some fifteen different types of noise available on the Amstrad: each type is referred to as an *envelope*.

The noise envelope is the seventh parameter on the SOUND command. To listen to a noise envelope, type SOUND 1,0,20,6,0,0,N — where N is a value between one and 15. After you have experimented with various values of N, you may find that the envelopes do sound very similar, particularly values one to five. However, when envelopes are used as the basis for a sound effect, the differences are much more apparent. Have a listen to the seventh, eighth and ninth sound effects examples on the following pages if you want to hear noise envelopes in a FOR/NEXT loop which create some really interesting effects.

Creating more complex sounds requires the use of volume and tone envelopes. The loudness (or amplitude) of a sound gives it its distinctive quality. The sounds that we have dealt with up to now have all had a constant loudness (the *Fade to Play* program being an exception). Consider, for a moment, the sound of a piano note — its loudness is far from being constant; there is a very sharp rise as the key is struck, which then dies rapidly away as the string damper does

its job. An electric guitar, on the other hand, has a slower rise but a longer tail-off. Thus, we can tell the difference between a large number of instruments and sounds, just because of their different volume *envelopes*.

The changing volume of a sound can be drawn on a graph creating a 'sound shape'. Typically, a volume envelope can be separated into four parts: the *Attack*, when the loudness rises; the *Decay*, when the sound loses a little of its amplitude; the *Sustain*, when the loudness remains constant; and the *Release*, when the sound's amplitude decreases to zero. Figure 2 is a graph showing a typical volume envelope.

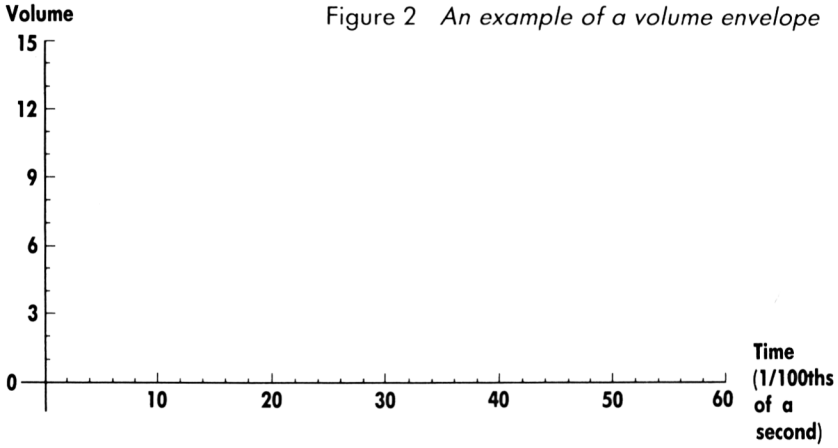


Figure 2 *An example of a volume envelope*

This volume envelope can be controlled by the ENV command; this command can appear very confusing, with up to 16 parameters following it, although it's not that difficult to understand. We are going to use the ENV command to create sound waveforms with the four amplitude features already mentioned (attack, decay, sustain and release).

First, we have to give this volume envelope a number, so that it can be specified in a program where more than one ENV command is present. Following on from the envelope's number are four sections of the ENV command, one for each of the four amplitude features. Each section consists of three parameters: the number of volume steps, the size of each step and the time that each step lasts for (measured in hundredths of a second). Therefore, a possible ENV command could be:

```
                ATTACK              SUSTAIN
              ~~~~~~~~~~          ~~~~~~~~~~
ENV 1, 3, 5, 2, 7, -1, 1, 1, 0, 8, 4, -2, 4
                          ~~~~~~~~~~      ~~~~~~~~~~
ENVELOPE NUMBER   DECAY                   RELEASE
```

Remember, there are 15 volume steps, so never over-step the maximum or minimum values. To help you design your own envelopes, Figure 3 shows a volume envelope graph on which you could plot a volume waveform.

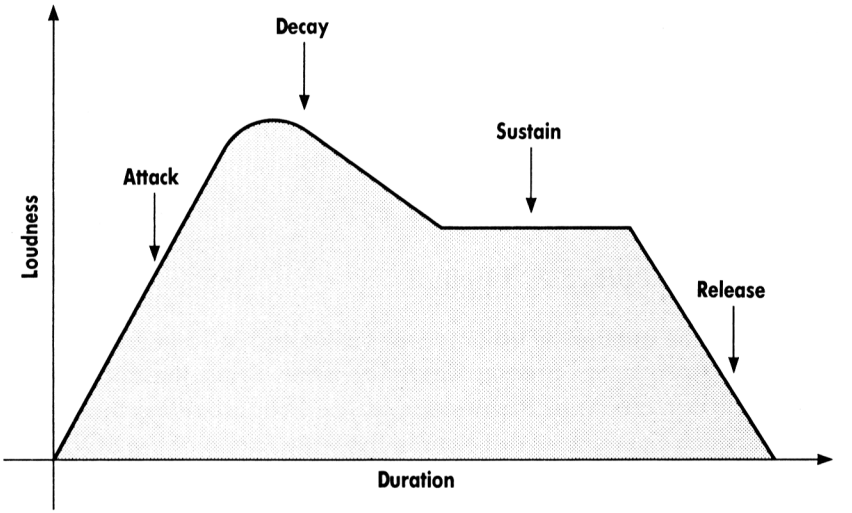


Figure 3 *A volume envelope graph*

Now you have created an envelope, how do you incorporate it into the SOUND command? Well, it's rather straightforward as the fifth parameter of the SOUND command is reserved for the ENV command. All you have to put in this place is the number of the volume envelope; in the example given earlier, the number of the envelope was 1. Thus, a SOUND command calling our pre-defined volume envelope might look like:

```
SOUND 1, 150, 30, 12, 1, 0, 0
```

We have already discussed what the seventh and final parameter of the SOUND command does. All that is left to discuss is the use of the sixth parameter, and for that you'll have to know a little bit about tone envelopes and the ENT command.

The tone envelope is similar to the ENV command, although it does not use attack, decay, sustain and release to help create the envelope shape. The tone envelope is concerned with the rise and fall in pitch of a note and is made up of an envelope number, followed by groups of parameters consisting of the number of steps, and the size and duration of each. Only one group is needed if the envelope is only going to rise or only going to fall in pitch. Where an envelope requires both rise and fall, a group of parameters is needed for each change. One rise and one fall would require two groups, whereas two rises and one fall would require three groups. This can be seen much more clearly from the example given below:

```
10 ' Sample ENT Envelope
20 '
30 ENT 1,25,-3,3
40 ' Envelope number 1
50 ' 25 steps at -3 each
60 ' Lasting for 3/100ths each
70 '
80 SOUND 1,200,80,15,0,1
```

Although the number of volume steps is limited to 15 in the volume envelope, there is far greater flexibility when using the ENT command. You can create as many steps as you like, providing that your eventual pitch figure does not exceed the wide range of the SOUND command (one to over 4000). The above example makes the pitch change in just one direction; the figure is negative, indicating that the pitch is rising (thus, the note played gets higher and higher).

A tremolo effect can be achieved by making the pitch rise and fall; this is done using at least two groups of parameters. The short program below creates a tremolo effect — type it in, run it and then return to the text for an explanation of how the program does what it does.

```
10 ' TREMOLO
20 '
30 ENT 1,12,-3,1,12,3,1
40 SOUND 1,250,200,15,0,1
```

You may have been quite disappointed with the end result of the above program: all you got was a tiny tremolo and then a tone. There is a good reason for this — the length of the envelope is only 24/100ths of a second, yet the note played by the SOUND command

lasted for 2½ seconds. If you want a tremolo to last for the whole length of the note, you'll have to go about it in one of two ways. One way would be to add a number of groups of parameters, raising and lowering the pitch until the note's duration is reached — there would then be eleven groups of parameters. Obviously, this is impractical; so, the designers of the Amstrad's Basic Locomotive have created a feature so that if you want a tone envelope to last the whole length of the notes played (in other words, if the envelope duration is shorter than the note(s) duration, then the computer repeats the envelope again and again, if necessary, until the note(s) duration is equalled), you just add a minus sign to the envelope's number. Thus, in the tremolo routine above, the ENT command would read:

```
ENT -1,12,-3,1,12,3,1
```

Figure 4 shows an envelope graph planner for tone envelopes.

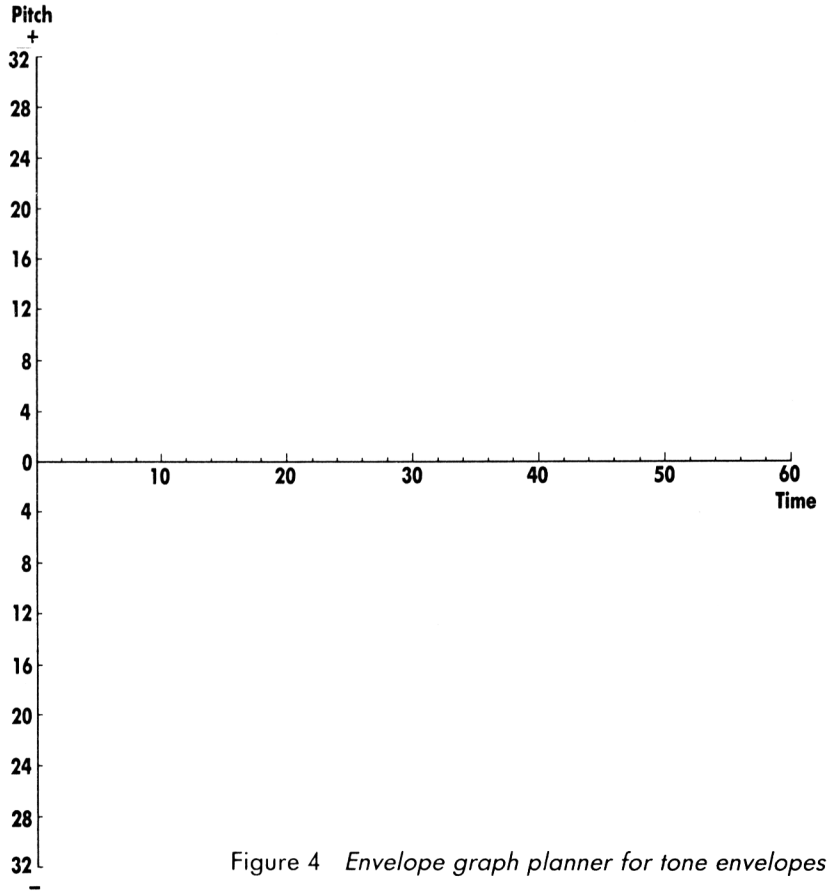


Figure 4 *Envelope graph planner for tone envelopes*

Here are a number of effects generated, either using the ENT or ENV command, or creating noise envelopes. Type each program in, and experiment by changing a value here and a value there. You never know, you might just come up with the sound effect you have been looking for!

```
10 ' EFFECT.1
20 '
30 ENT -1,5,5,1,10,-5,1,5,1,1
40 SOUND 1,500,10000,7,0,1
50 FOR T=1 TO 5000:NEXT
60 PRINT CHR$(7)
```

```
10 ' EFFECT.2
20 '
30 ENT -2,5,-1,2,1,5,3
40 SOUND 1,100,400,14,0,2
50 SOUND 2,108,400,14,0,2
60 SOUND 4,92,400,14,0,2
```

```
10 ' EFFECT.3
20 '
30 ENT -2,20,-1,1,1,20,1
40 SOUND 1,280,400,14,0,2
```

```
10 ' EFFECT.4
20 '
30 ENV 1,100,3,1
40 SOUND 1,200,100,1,1,1,5
50 GOTO 30
```

```
10 ' EFFECT.5
20 '
30 ENV 1,40,6,2
40 SOUND 1,220,40,0,1,1,5
50 GOTO 30
```

```
10 ' EFFECT.6
20 '
30 ENT 1,10,-4,20
40 ENV 1,16,24,2
50 SOUND 1,150,100,1,1,1,7
60 WHILE INKEY$="":WEND
70 GOTO 40
```

```
10 ' EFFECT.7
20 '
30 FOR T=15 TO 1 STEP-1
40 SOUND 1,0,1,7,0,0,T
50 NEXT
60 GOTO 30
```

```
10 ' EFFECT.8
20 '
30 FOR T=15 TO 1 STEP-1
40 SOUND 1,0,7,5,0,0,T
50 SOUND 2,0,21,6,0,0,T
60 NEXT
70 GOTO 30
```

```
10 ' EFFECT.9
20 '
30 FOR T=15 TO 1 STEP-1
40 SOUND 1,0,7,5,0,0,T
50 SOUND 2,0,13,6,0,0,T
60 NEXT
70 GOTO 30
```

Synchronization

Before we get too heavily involved with synchronization, here are a couple of programs that show this process in practice. The larger of the two programs plays a well-known tune with a synthesizer effect, while the much shorter routine uses an effect seen earlier but increases it by using all three sound channels.

```
10 ' SYNTH'ED STARS
20 '
30 ' MUSIC SUBROUTINE
40 '
50 GOSUB 300
60 ' PLAY NOTES
70 DIM P(100)
80 RESTORE 120
90 FOR T=1 TO 47
```

```
100 READ P(T)
110 NEXT
120 DATA 1,1,8,8,10,10,8,0,6,6,5,5,3,3,1
130 DATA 0,8,8,6,6,5,5,3,0,8,8,6,6,5,5,3
140 DATA 0,1,1,8,8,10,10,8,0,6,6,5,5,3,3,1
150 '
160 ' NOTE THAT 0 EQUALS A REST
170 '
180 FOR T=1 TO 47
190 SOUND 1,M(P(T)),20,15
200 IF P(T)>2 THEN SOUND 2,(M(P(T)))-2,20,15
ELSE SOUND 2,M(P(T)),20,15
210 SOUND 4,(M(P(T)))+2,20,15
220 SOUND 1,0,10
230 SOUND 2,0,10
240 SOUND 4,0,10
250 '
260 ' NB. SOUND 1,0 CREATES A GAP BETWEEN NOTES
270 '
280 NEXT
290 END
300 '
310 ' NOTE STORAGE
320 '
330 DIM M(24)
340 RESTORE 380
350 FOR T=1 TO 24
360 READ M(T)
370 NEXT
380 DATA 478,451,426,402,379,358,
338,319,301,284,268,253
390 DATA 239,225,213,201,190,179,
169,159,150,142,134,127
400 RETURN
```

```
10 ' FADE TO PLAY.2
20 '
30 FOR T=7 TO 1 STEP -1
40 SOUND 1,200+(5*T),20,T
50 SOUND 2,198+(5*T),20,T
60 SOUND 4,202-(5*T),20,T
70 NEXT
```

This synchronization of the three channels provides a 'thicker' note and, when the pitches of the three channels are all slightly different, the resultant sound gives a phased effect.

Synchronization can be used to play a melody while another channel (or possibly several channels) plays a rhythm backing. A simple example program is given below; note how the EVERY command is used to interrupt the main program and divert the Amstrad to play a bass beat. Why not modify one of the earlier tunes in this chapter so that it is played with a rhythm?

```
10 ' SYNCHRONISE
20 '
30 Z=0
40 EVERY 20 GOSUB 90
50 FOR T=340 TO 80 STEP-20
60 SOUND 1,T,20,10
70 NEXT
80 GOTO 40
90 Z=Z+1:IF Z=2 THEN Z=0:GOTO 120
100 SOUND 2,1000,10,13
110 RETURN
120 SOUND 4,0,8,13,0,0,4
130 RETURN
```

Complex synchronization is achieved via the SQ command. This creates a formalized 'sound queue', so that all the notes to be played are sorted into a playing order.

Briefly, SQ(x) — where x is the channel number being checked (either channels one, two or four) — can return a value between zero and 255, and this value should be broken down to obtain the information that one may need; each of the eight bits are concerned with a different piece of information. (These pieces of information are detailed in the section of the Amstrad manual entitled *A Sound Primer*, and basically it tells you the situation within the sound queue, such as which sound channel is playing and how many free spaces, if any, there are present.)

For synchronization of different channels, we use a special value for the SOUND channel, and this can be one of the five listed below:

8 Synchronize with channel one
16 Synchronize with channel two
32 Synchronize with channel four

64 Hold sound playing at present
128 Clear sound queues

To conclude this chapter, here is an example of these values in action:

```
10 SOUND 10,300,100,15
20 WHILE INKEY$="":WEND
30 SOUND 17,380,100,15
```

You can see that the first note is not played until the second note has been played, and that the second note is only played once a key is pressed. The channel number in line 10 is derived by using channel two and synchronizing it with channel one (a special value of eight — thus 8+2=10). The channel number in line 30 is achieved by using channel one and synchronizing it with channel two (a special value of 16 — thus, 16+1=17).

17
THE RUDIMENTS OF MACHINE CODE

In this short chapter, we will be looking to explain the meaning of the commands PEEK, POKE and CALL.

PEEK and POKE may seem more frightening than they really are. Here is a simplistic account of what they do. Imagine the computer's memory as a suitcase filled with thousands of matchboxes. Every matchbox is numbered, so at any time we can look inside any specific one and see what is inside. If there's nothing already inside the matchbox, then we can put something in it and store it away.

The number on the side of the matchbox is known as the *address*, and to see what is inside we PEEK the address. To place a number into a specific matchbox, we use the POKE command. Below is a demonstration of this principle. Input the words on the left as direct commands:

PRINT PEEK (32098) This examines address 32098 in the computer's memory and finds a zero; that is, it's empty
POKE 32098,24 The POKE command looks up address 32098 and deposits the number 24 there
PRINT PEEK (32098) If address 32098 is examined again using the PEEK command, you should find the number 24.

Using the PEEK and POKE commands, you can do all sorts of wonderful things, as long as you have a good knowledge of machine code and know where the addresses are that you want to change.

The CALL command is a little more advanced than PEEK or POKE, but simply it is very similar to GOSUB, in that it goes to a subroutine; the only difference is that the routine it goes to is one of the programs that go together to make up the Amstrad's ROM. When you become proficient with the CALL command, you may find CALL &BD19 extremely useful as it smooths the movement of animated graphics.

To finish off, we have a little program that checks the computer's memory between the locations zero and 600. This may not seem that fantastic, but it certainly becomes more interesting when you know

that your Amstrad stores BASIC programs from address 368 onwards. Therefore, if you run this program you can see what the *Memory PEEKer* program looks like in terms of addresses. The program could be useful if it is placed at the end of another program, and then called to check the address contents.

```
10 ' MEMORY PEEKER
20 '
30 FOR T=0 TO 600
40 IF PEEK(T)>31 THEN PRINT T,PEEK(T),CHR$
(PEEK(T))
50 IF PEEK (T)<=31 THEN PRINT T,PEEK(T)
60 WHILE INKEY$="":WEND
70 NEXT
```

18
SYSTEM INTERRUPTS ON YOUR AMSTRAD

Your Amstrad is blessed with an unusual feature, that of being able to use system interrupts from BASIC. An interrupt is what its name suggests, a method of allowing the computer to divert from the main program to perform separate operations almost simultaneously. This leads to a form of *multi-tasking*.

Usually, multi-tasking can only be achieved through the use of complex machine-code techniques. However, the Amstrad has two BASIC commands to do this, and a number of other commands that deal with the timers in the machine that are linked with interrupts.

The AFTER command

The AFTER command is used in the following format:

AFTER (how many fiftieths of a second), the number of the delay timer (between zero and three), GOSUB (line number)

Let's explain this a little. After a set time, the AFTER command directs the computer to a certain subroutine and the delay timer used can be any one of the four available on the Amstrad. (If you are using a number of interrupts in a long program, you must be careful not to mix them up and use the wrong timer.)

Here's two example programs showing interrupts in use: one using AFTER (which we have already discussed) and the other employing the EVERY command. EVERY has two numbers after it (just like AFTER) and they stand for the same elements as the numbers following AFTER; the only difference is that the interrupt is repeated every time the period of time (specified in the first variable) is exceeded.

The first routine is little more than a demonstration showing AFTER in use. Note how, despite the GOTO in line 90, the program jumps to the subroutine specified in the AFTER command.

The second routine is potentially much more useful, and you will see variants of it in Chapter 16. The main loop is concerned with playing a series of notes increasing in pitch. Every couple of notes, the program diverts to the subroutine that plays the bass note, effectively providing a bass backing. It is obvious that this could be used to great effect in far more complicated music routines, providing three-voice accompaniment to the basic melody. Why not try and simulate this in a program of your own?

```
10 ' INTERRUPTS.2
20 '
30 AFTER 150 GOSUB 100
40 ' CHANGE BACKGROUND TO BLUE
50 PAPER 0
60 CLS
70 LOCATE 12,12
80 PRINT "THE BACKGROUND IS BLUE"
90 GOTO 90
100 ' CHANGE BACKGROUND TO RED
110 PAPER 3
120 CLS
130 LOCATE 12,12
140 PRINT "THE BACKGROUND IS RED"
150 END
```

```
100 ' INTERRUPTS
110 '
120 EVERY 40 GOSUB 180
130 FOR T=250 TO 70 STEP -15
140 SOUND 1,T,20,11
150 NEXT
160 FOR T=1 TO 2000:NEXT:GOTO 130
170 '
180 ' PLAY BASS BEAT
190 '
200 SOUND 2,500,20
210 RETURN
```

You can disable the interrupts easily on the Amstrad by simply typing DI; to re-enable the interrupts, you must type EI. In a music program where, say, the computer was playing a bass beat every other musical note using the EVERY command, you would need to

disable the interrupts at the end of the piece and during any long pauses.

The TIMER feature

The Amstrad is gifted with a real-time timer that can be accessed by the TIME command. Printing TIME will give you (in 300ths of a second) the length of time that the machine has been switched on. From this, it is easy to calculate a playing time of a game or even construct a digital clock.

Printing TIME, however, after your computer has been switched on for a fair time will give you a horrendously large figure — how can you work with that? Well, if you divide the figure by 300, you'll be given seconds (or by 30 if you want tenths of a second). Secondly, for most programs, the time must be set to zero; after all, if the machine was switched on for 30 minutes before a game was played, the player would be most distraught if he finished the program in under a minute, only to find the computer reporting that it had taken 30 minutes and 57 seconds to complete the game! We cannot set TIME to zero — not through BASIC at any rate — so how do we do this? The answer is that we make a variable equal to TIME at the start of the game and then, when the game is over, the time that has elapsed since it started is easily calculated by subtracting the variable from TIME. This may seem rather complex but if you type in the little game below, you will see the principle much more clearly.

```
10 ' REACTION TESTER
20 '
30 RANDOMIZE TIME
40 FOR T=1 TO 4:D$=D$+CHR$(INT(RND*26)+65)
:NEXT
50 FOR T=1 TO 1000:NEXT
60 CLS:LOCATE 18,12
70 SOUND 1,180,40
80 START=TIME
90 PRINT D$
100 LOCATE 15,24
110 INPUT A$
120 IF A$=D$ THEN 150
130 GOTO 110
```

```
140 '
150 ' WIN
160 '
170 SCORE=TIME-START
180 CLS
190 LOCATE 10,10:PRINT "YOU'RE RIGHT!!"
200 LOCATE 7,12:PRINT "AND IT TOOK YOU";SCORE/
300;"SECS"
210 LOCATE 10,20
220 PRINT "ANOTHER GO? PRESS Y"
230 F$=INKEY$:IF F$="Y" THEN RUN
240 GOTO 230
```

In this game, you must type the four letters in and press ENTER as fast as possible. The Amstrad then tells you how long it took you. (My best time was 1.65 seconds — can you beat that?)

The timing feature starts in line 80 where a variable, appropriately called START, is made equal to the TIME just before the four letters are printed on the screen. As soon as the right answer is entered, the computer stops the clock and makes another variable SCORE equal to the time at the end minus the starting time. The reaction time held in the variable SCORE is finally divided by 300 before it is printed on the screen.

The final command we are going to look at in this section will not be one that you will use frequently, but it is important that you know of its existence. The REMAIN command disables a timer if it was being used in an interrupt (in a line such as EVERY 8, 2 GOSUB 80 the timer being used is 2) and prints how many 1/300ths of a second were left before the timer reached the figure where the program was interrupted (when the EVERY or AFTER command came into play). Thus, in the example above, if the program printed REMAIN (2) (the figure in brackets being the delay timer specified) after only 5/300ths of a second had passed, then the computer would disable timer two and give the figure three, because there were only 3/300ths of a second to go before the EVERY command would be executed and the computer would GOSUB to line 80.

The example program below waits for you to press a key and then tells you how much time there was before the next EVERY command would be enacted (an interrupt occurs every time the tone sounds). You can use this short program as a reaction timer in itself, seeing how close you can get to 100 or any other number within the EVERY command's range of one to 100.

```
10 ' REMAIN
20 '
30 EVERY 100,3 GOSUB 70
40 WHILE INKEY$="":WEND
50 PRINT REMAIN (3)
60 END
70 SOUND 1,100,10,15
80 RETURN
```

19
WINDOWS

The Amstrad has the very useful feature of allowing windows to be created; windows are like a screen within a screen. The command to define a window is shown below:

```
WINDOW #A, X, X1, Y, Y1
```

As already mentioned, the Amstrad has a number of stream outputs. Streams zero to seven are used for text output on the screen in the form of windows, while stream eight is output to the printer and stream nine to the cassette. The A value is the number stream, while the X and Y values are the co-ordinates of the window; X and X1 are the horizontal values, and Y and Y1 are the vertical values. Therefore, (X,Y) represent the co-ordinates of the top left-hand corner of the window, and (X1,Y1) define the bottom right-hand corner.

The CLS, PAPER and PEN commands all work with windows. You can set the background colour of a window using PAPER #A, C, where A is the stream number and C is the colour. PEN #A, C and CLS#A all work in the same way to the PAPER command, though CLS has no colour — the window being cleared to the PAPER value.

The following program sets up seven windows and fills them with different colours:

```
10 ' WINDOWS EVERYWHERE
20 '
30 INK 0,0:MODE 0:CLS
40 PRINT " WINDOWS EVERYWHERE"
50 INK 8,6:INK 14,24:INK 6,7
60 BORDER 1
70 WINDOW #1,1,5,3,7
80 WINDOW #2,8,12,3,7
90 WINDOW #3,15,19,3,7
100 WINDOW #4,1,5,10,14
110 WINDOW #5,8,12,10,14
120 WINDOW #6,15,19,10,14
```

```
130 WINDOW #7,8,12,17,21
140 FOR T=1 TO 7
150 PAPER #T,(T*2)
160 CLS #T
170 PEN #T,0
180 PRINT #T," "
190 PRINT #T," NO. ";T
200 NEXT
210 WHILE INKEY$="":WEND
220 MODE 1
230 END
```

The program is pretty straightforward — lines 70 to 130 set up the windows, while lines 140 to 200 set up a loop to set the paper colour, clear the window and print the window's number in black ink.

There are a couple of further points that you should know about. Firstly, you can list a program through a window — try listing the above program through window number seven. This is achieved simply by typing LIST 7. Secondly, it should be remembered that each new window overwrites an old window. Therefore, if the last window was the size of the whole screen, it would overwrite any pre-defined window already on-screen.

There is one other command that is specifically to do with windows, the WINDOW SWAP command. This allows you to swap the text of one stream to another. The program following demonstrates this; this demonstration is very similar to the previous program, except for the extra command to shift the contents of each window around. The format for this command is:

```
WINDOW SWAP X, Y
```

where X is one stream and Y is another — the text and window details are transferred between the two windows. Try the program below:

```
10 ' WINDOW SWAP DEMO
20 '
30 INK 0,0:MODE 0:CLS
40 PRINT " WINDOWS SWAP DEMO"
50 INK 8,6:INK 14,24:INK 6,7
60 RANDOMIZE TIME
70 BORDER 1
80 WINDOW #1,1,5,3,7
```

```
90 WINDOW #2,8,12,3,7
100 WINDOW #3,15,19,3,7
110 WINDOW #4,1,5,10,14
120 WINDOW #5,8,12,10,14
130 WINDOW #6,15,19,10,14
140 WINDOW #7,8,12,17,21
150 WHILE INKEY$=""
160 FOR T=1 TO 7
170 PAPER #T,(T*2)
180 CLS #T
190 PEN #T,0
200 PRINT #T," "
210 PRINT #T," NO. ";T
220 NEXT
230 A=INT(RND*7)+1
240 B=INT(RND*7)+1
250 IF A=B THEN 230
260 WINDOW SWAP A,B
270 WEND
280 MODE 1
```

Another rather tidy way to use windows is as graphics creators. A coloured window is by far the quickest and, indeed, the easiest way to construct a filled-in square or rectangle. That principle has been taken one step further with the following graphics routine; windows that increase and decrease rapidly in size produce the effects that you see on-screen.

```
10 ' GRAPHIC EFFECTS USING WINDOWS
20 '
30 MODE 2
40 X=40:Y=13
50 B=0
60 INK 1,6:INK 0,1
70 PAPER #1,3
80 FOR A=1 TO 40
90 IF A/3=INT(A/3) AND A>5 THEN B=B+1
100 IF X-A=0 THEN X=X+1
110 WINDOW #1,X-A,X+A,Y-B,Y+B
120 CLS #1
130 SOUND 1,450-A*10,10
140 NEXT A
```

```
150 FOR T=1 TO 1500:NEXT T
160 PAPER 1:CLS
170 PAPER #1,0
180 FOR A=40 TO 1 STEP -1
190 IF A/3=INT(A/3) THEN B=B-1
200 IF A/2=INT(A/2) THEN J=J+1:IF J=2 THEN J=0
210 WINDOW #1,X-A,X+A,Y-B,Y+B
220 PAPER #1,J
230 CLS #1
240 SOUND 1,450-A*10,10
250 NEXT A
260 FOR T=1 TO 1500:NEXT T
270 ON BREAK GOSUB 290
280 GOTO 70
290 MODE 1
300 END
```

20
THE ERROR COMMANDS AND ON BREAK

Locomotive BASIC includes two less common commands: ON BREAK GOSUB and ON ERROR GOTO. After each of these commands, there should be a number corresponding to the line that the program jumps to if the condition (if the Break key is pressed or an error is created) is fulfilled.

There are several other error commands you should know about, such as ERR and ERL; these are variables used in the error messages — ERL is the line number where the last error was created, while ERR is the type of error message. A list of error messages are given in Appendix C.

The ERROR command is followed by a number (not in brackets) corresponding to the error number. If such a command is executed, the computer will take the appropriate error action. Thus, if ERROR 8 was entered, the computer would display a 'Line not found' error.

The ON BREAK STOP command is the last of this group of commands and is rather self-explanatory. Below are several examples of these commands in action:

```
10 ' DEMO SHOWING ON BREAK
20 '
30 ON BREAK GOSUB 80
40 FOR T=1 TO 100
50 PRINT T
60 NEXT T
70 END
80 RETURN
```

```
10 ' DEMO SHOWING ON ERROR
20 '
30 ON ERROR GOTO 90
40 FOR T=1 TO 12
50 PRINT A(T)
60 NEXT
```

```
70 END
80 '
90 ' PRINT ERROR TYPE AND NUMBER AND END
100 '
110 PRINT:PRINT ERR;" IN ";ERL
120 PRINT
130 GOTO 70
```

```
10 ' DEMO SHOWING ON BREAK STOP
20 '
30 J=1
40 ON BREAK GOSUB 60
50 FOR T=1 TO 10000:NEXT
60 J=J+1:IF J=3 THEN 90
70 RETURN
80 FOR T=1 TO 10000:NEXT
90 ON BREAK STOP
```

The first program prevents you from breaking the sequence of numbers, from one to 100, that is being printed on the screen. This can be used easily in your own programs, perhaps twinned with a caustic comment telling the user that trying to break into the program gets you nowhere!

The second program deliberately contains an error: an array that has not been dimensioned is printed on the screen. This works fine until the computer tries to print the eleventh element in the array; it then comes up with an error. Instead of printing the error in English and breaking the program, the computer prints up the error number using ERR and the line that it occurred in using ERL.

The third program constructs a situation where you must press the Break (ESC) key four times before it breaks. Each time the Break key is pressed, one is added to the variable J. When J reaches three (when the ESC key has been pressed three times) the program jumps to line 90 where, if the Break key is pressed once more, the program is stopped.

When an error has been found by the ON ERROR GOTO command, the program can be continued by using one of the following three forms of the RESUME command:

RESUME on its own — carries on from the line where the original error was found
RESUME followed by a line number — carries on from the specified line number

RESUME NEXT — continues the program from the command or statement immediately following the one in which the original error was found.

If you modify the ON ERROR demonstration (given a couple of pages earlier) so that line 130 now reads RESUME 40, you can see the difference that the new command makes. As soon as an error is found in the program, the FOR/NEXT loop starts all over again.

All these commands are useful when protecting finished programs, and error-trapping and improving programs under development.

21
A POT POURRI OF COMMANDS

The range of commands on a computer often defy simple classification and a chapter such as this becomes necessary. Here we discuss a few commands that, although quite important, don't really fit well into any of the other sections.

We have already come across the function FRE, which we have used to determine the amount of spare user memory left. But there is another use for this command, and that is 'garbage collection'. No, it doesn't carry a dustbin-full of worn-out REM statements to the dump; instead, it gets rid of memory-consuming string variables. What happens is that every time a string is used it is re-stored in memory — therefore, it could take 10K or even 20K if it was the subject of a large FOR/NEXT loop, altering the string each time round the loop. And, of course, the Amstrad is only interested in the last version of the string.

Using the FRE ("") command helps to eradicate this appalling waste of memory. The following program shows FRE("") in action and shows, very clearly, the amount of memory that a string can waste when it is being constantly re-defined:

```
10 ' USING FRE TO RELEASE MORE MEMORY
20 '
30 CLS
40 PRINT "ORIGINAL AMOUNT OF SPARE MEMORY:";
FRE(0)
50 LOCATE 3,12
60 PRINT "AMOUNT OF MEMORY NOW SPARE:"
70 FOR T=1 TO 200
80 A$=A$+CHR$(INT(RND*26)+65)
90 LOCATE 30,12
100 PRINT FRE(0)
110 NEXT
120 LOCATE 2,18
```

```
130 PRINT "MEMORY SPARE AFTER FRE() USED:";
FRE("")
```

The FRE("") command is invoked each time it is included within a program line, but the best way is to allocate FRE("") to an unwanted variable every now and then within a program. The above program PRINTs FRE(""), but you may not want the amount of free memory mysteriously appearing on the screen. It's up to you . . .

Also linked to the uses of memory, the HIMEM command gives the memory address of the highest byte of memory used by BASIC, and can be used in numeric expressions in the normal way. When the machine is switched on, the HIMEM value is usually 43903.

The MEMORY command is also used in conjunction with memory (how did you guess?). It sets the highest possible address that can be used by BASIC; it, in fact, sets the value of HIMEM. The primary use for such a command is to reserve memory out of reach of BASIC for the storage of machine-code routines.

Your Amstrad and a printer

There are three BASIC commands that you need to know about when using a printer with your Amstrad: WIDTH, PRINT #8 and LIST #8.

The WIDTH command simply controls the width of the copy printed out and is most useful. Following the WIDTH command there should be one figure, namely the number of characters across that you would like your print-outs to be.

PRINT #8 is similar to the normal print statement, except that it is outputting to the printer rather than the screen. Therefore, after a comma, you can add TAB, semi-colons, commas, SPC and most of the other commands used in tandem with PRINT.

LIST #8 is rather similar to normal LIST. If you want to list out portions of a program, then you use the format: LIST line number-line number, #8. This would print out the lines between and including the line numbers specified to the printer.

As regards printer codes to create emphasized print styles and different character sets, it is largely a matter of looking up the codes in your own printer manual. Any codes that I have used in this book apply to both the Epson and the Star printers — at the time of writing, the most popular printers used with the Amstrad (excluding Amstrad's own, of course).

22
THE AMSTRAD CPC 6128

Launched just over a year after the first Amstrad model, the CPC 6128 was originally going to be sold exclusively in the united States. The good news for British and European users was that Amstrad quickly changed their mind and made the machine available in Europe.

The machine essentially offers 128K memory as opposed to the 64K of the 464 and 664, an integral 3-inch disc drive, a later version of CP/M and a re-designed keyboard.

The ROM is the one found in all Amstrad machines. It is based around the Z80 chip and has exactly the same BASIC as found on the CPC 464 and 664. The new commands for accessing the extra memory and utilising the disc drive are all stored on disc. This means that the vast majority of software available for the 464 and 664 is compatible with the new machine. For this reason, the programs in this book can be run on all three machines.

The 6128 keyboard has all the keys found on the previous two models, the only difference being the size and placement of many of the keys.

The doubling of memory produces problems for a Z80-based machine. The Z80 chip can only address 64K of memory and various tricks are needed to make the second block of 64K accessible. Amstrad have made the second memory block useful for storing screen displays and making RAM data files, but not for writing actual programs. This is not a great drawback. Any programs you write which start to reach the limits of the program memory (43K or so) are likely to contain a lot of data. A large adventure game would need a substantial amount of memory for storing the details of each location, object and command. An accounts program would have to store all the transactions of each account. This data can be stored as a data file within memory as opposed to the usual method of storing on cassette or disc.

The commands for accessing the extra memory as a RAM data file are :BANKOPEN, :BANKWRITE, :BANKREAD and :BANKFIND.

:BANKOPEN initializes the current record number and sets the length of the record. :BANKWRITE and :BANKREAD are self-explanatory, allowing you to place data into and retrieve data from the file. :BANKFIND is a very handy search command. It returns the record number where the specified piece of data has been stored.

The extra block of memory can also be used to hold screen displays. When used in this way, it is split into four separate 16K chunks, each capable of holding one screen. The commands :SCREENCOPY and :SCREENSWAP move the contents of the screens around. To display a screen, you perform a :SCREENSWAP command moving the screen you wish to display into block one which equals the on-screen area. This has obvious potential in games programs with multi-screen arcade games switching smoothly and quickly from one screen to another, and also in adventures where screens of complicated graphic scenes can be displayed speedily while other screens are being constructed for display later.

CP/M is available on the CPC 664 and the 464 with disc drive and is detailed in its own chapter earlier in this book. The version of CP/M utilised by these two machines is 2.2 which is perhaps the most popular version around, but does lead to a problem when used with the Amstrad. CP/M relies on 64K of RAM and the 464 and 664 cannot offer a complete 64K. This means that many of the programs available through CP/M need to be altered or re-written to work with the Amstrad. The 6128 does not have this problem. It uses a later version of CP/M, version 3.0. This version has the advantage of being capable of utilising bank-switching memory systems and so can tap into the extra 64K block of memory on the 6128 machine. Therefore, most, if not all, CP/M programs will run without modification on the 6128. There is just one modification that will have to be made. Most CP/M machines use 5¼-inch discs and so CP/M programs will have to be made available on 3-inch discs. This should not be much of a problem.

23
HOW TO PROGRAM MORE PROFESSIONALLY

Some programming tricks and tips

(1) 'How can I disable the ESC key to stop people breaking into my programs?'
There are several ways to do this. One method involves re-defining the ESC key altogether using the KEY DEF command; for instance, KEY DEF 66,1,32 re-defines the ESC key to print a space. Secondly, you can use the ON BREAK GOSUB feature to make the program jump to a subroutine that prints up a warning message and then just returns without stopping the program at all. And thirdly, you can use a CALL figure to disable the ESC key — try CALL 47944. To re-enable the key, you must enter CALL 47947. Both of these calls will only work when the program is running, so they must be incorporated as part of the actual program.

(2) 'How can I prevent a program from being NEWed accidentally?'
Well, POKE 486222, 201 disables the CTRL-SHIFT-ESC function that wipes the computer clear of any program residing in the memory. To re-enable the CTRL-SHIFT-ESC function, type POKE 48622,195.

The above method will stop someone accidentally losing a program, but if you're worried that someone will type DELETE followed by a significant part of the program or type NEW and then press ENTER, you will need to be more clever. Typing the above commands require the computer to be in the direct mode so, if it is possible, try and keep the program running and use the tricks covered in the first question to prevent it from being broken into. However, some programs may not allow this; if this is the case, I suggest that if you don't need the letter E, you re-define it to another letter, thus creating a syntax error when the user tries to type NEW or DELETE.

Remember that, as a programmer, if you use either of these solutions to the above problems, you must leave yourself a 'back door'

— so that, when developing the program, you do not find yourself in a position where you have to switch off your Amstrad and start again as you have crashed the program.

(3) 'Is there any simple way of preventing people tampering with my program?'

If you do not want selective blocks on certain functions, but a blanket block on any way to list your program, you should use the protected program mode of the SAVE command: SAVE "Basic Program File Name",P will prevent your program being listed, and so on.

(4) 'I want to add a high score feature to my game that I am writing — how can I do it?'

There are a number of ways a simple high score feature can be created, using the MAX command to search through a list of scores and find the highest.

If you want to create a more complex high score feature, then it would be wise to use a numeric sort, and then output the information in a tidy table. Have a look at the program that follows:

```
10 ' HIGH SCORE TABLE
20 INK 2,1,24
30 DIM H$(8,2)
40 ' DUMMY SCORES READ INTO ARRAY H$
50 FOR T=1 TO 8
60 READ H$(T,1):READ H$(T,2)
70 NEXT
80 DATA ROBERT,1200,FRANK,990,GEORGE,2340
90 DATA ARNOLD,9910,MARK,780,TIM,1500
100 DATA SCOTT,2200,MARY,1760
110 ' DISPLAY HIGH SCORES
120 CLS
130 LOCATE 12,4
140 PEN 3
150 PRINT "HIGH SCORE TABLE"
160 PEN 1
170 PRINT TAB(10);"====================="
180 FOR T=1 TO 8
190 IF T=1 THEN PEN 2 ELSE PEN 1
200 PRINT TAB(10);"|";TAB(29);"|"
210 PRINT TAB(10);"|";TAB(12);H$(T,1);
TAB(24);H$(T,2);TAB(29);"|"
```

```
220 NEXT
230 PRINT TAB(10);"|";TAB(29);"|"
240 PRINT TAB(10);"====================="
250 SOUND 1,180
```

(5) 'Is there any way of being able to use the play button on my cassette machine to locate a program on a piece of tape without putting the computer into SAVE, LOAD or CAT mode?'

When you are positioning tape to find the correct place on the cassette, the Fast Forward and Rewind controls can be a little too fast to do the job that you want. Using the Play key would allow for much finer tape position control. To enable the Play key to be used at times other than when one of the standard cassette-handling commands had been chosen, one must enter OUT 512,16. To disable the Play key again, type OUT 512,0.

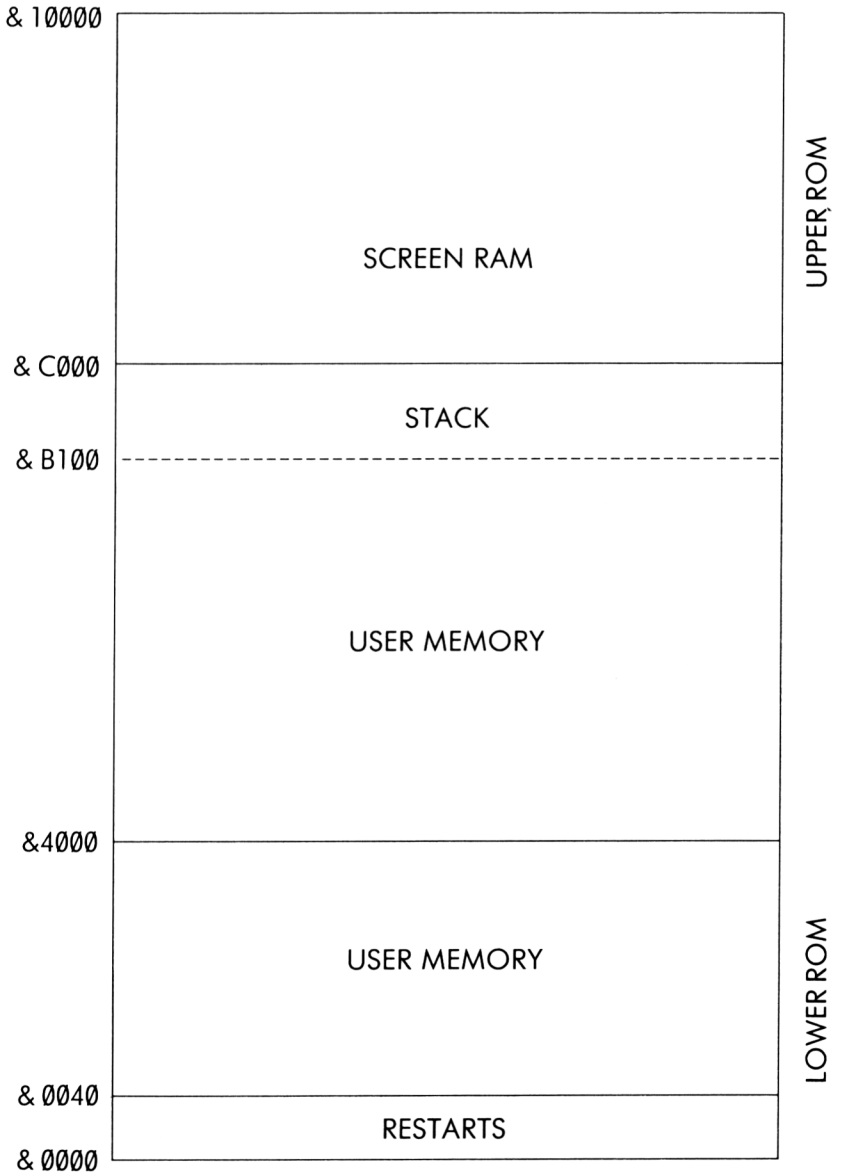
My final piece of advice for those trying to write a games program of some sort is to concentrate on originality. There are many copies of classic games available, but creating your own original is far more impressive and, if you are looking for commercial gain, the software houses and publishing companies are more interested in something that is novel, new and entertaining. After all, there is no point in creating a marvellous new game using some stunning technical innovations if the end result will soon be discarded after a few plays.

Appendix A
ADDRESSES OF SUPPLIERS

Below are a list of addresses of suppliers of Amstrad computers and peripherals, at the time of writing.

Indescomp, Po. Casetllana, 179, 1. – Madrid – 16, Spain, (279)3105
Reynolds Electronics Ltd, Finnabair Industrial Park, Dundalk, Co. Louth, Eire, 342 31281
Supremia Co Ltd, 122 Gloucester Avenue, London NW1 8HX, England, 01–722 2231 (TURKEY)
Microstar s.r.l., 20125 Milano, via Cagliero, 17, Italy, 02 68 87 604
Dinamicro aps, 26 Flintholm Alle, DK 2000, Copenhagen F, Denmark, 01 88 02 88
Informasjonssystemer as, Lilleakerveien 2D, Oslo 2, PO Box 74, 1324 Lysaker, Norway, 472 50 10 80
Toptronics Ltd, Nuppelantie 35, 20310 Turku 31, Finland, 921–392 766
Compumak Ltd, 9 Asklipiou St, 106 79 Athens, Greece, 36 09 846
Logica Imports Exports, 2 Limassol Ave, Nicosia, PO Box 537, Cyprus, 021 42336
D. Tanti & Co. Ltd, St John Square, Valletta, Malta, 25931
Holland Micro BV, Mathenesserlaan 332, 3021 HZ Rotterdam, Netherlands, 010 77484
Schneider Rundfunkwerke, Silvastrasse 1, 8939 Turkheim/ Unterallgau, Federal Republic of Germany, 08245 51–0
AWA-Thorn Consumer Products pty. Ltd, 348 Victoria Road, PO Box 11, Rydalmere, NSW 2116, Australia, 638 9022
EPIC, 20 Ashmount Street, (Salah el Din), Heliopolis, Egypt, 661767

Appendix B
MEMORY MAP



Appendix C
ERROR CODES AND DEFINITIONS

UNEXPECTED NEXT
A NEXT command has been found for which there appears to be no matching FOR.

SYNTAX ERROR
The program line does not make sense according to the Amstrad's BASIC. Always check your spelling when this error code appears.

UNEXPECTED RETURN
Similar to the UNEXPECTED NEXT error; the computer has found a RETURN which does not come within a subroutine.

DATA EXHAUSTED
The computer has executed a READ command, but there's no data left to read.

IMPROPER ARGUMENT
Part of one of the commands is invalid — the value used is out of the range of the command's parameter.

OVERFLOW
Some numeric operation has resulted in a figure greater than 1.7 to the power of 38.

MEMORY FULL
This is a pretty straightforward error title. Your program is too large for the computer's memory to handle. If this happens and the program is only a little larger than the size that the Amstrad can handle, then go through it eliminating any superfluous spaces, REM statements and any bit of code that can be taken out without spoiling the program as a whole.

LINE DOES NOT EXIST
Usually, a GOTO or a GOSUB will have tried to direct the program to a line number that does not exist for this error to occur.

SUBSCRIPT OUT OF RANGE
Check through your arrays, as an element is being referred to that does not exist within your array.

ARRAY ALREADY DIMENSIONED
Your program is trying to re-dimension an array that is already present.

DIVISION BY ZERO
SC=1000/INT(P) may yield this error, if P is less than one. You cannot divide by zero in any situation!

INVALID DIRECT COMMAND
Some of the Amstrad's commands can only be used with a program, and not in direct mode. Take DEF FN, for example.

TYPE MISMATCH
You cannot mix up string and numeric values — if you do, then a TYPE MISMATCH error appears.

STRING SPACE FULL
So many strings have been created that there is no further room left for the creation of more.

STRING TOO LONG
Strings can only be up to 255 characters in length — if a longer one is required, use several strings and add them together.

STRING EXPRESSION TOO COMPLEX
String expressions can generate intermediate string values. If the number of these values exceeds a reasonable limit, this error occurs.

CANNOT CONTINUE
If a program has been STOPped (or broken by a double pressing of ESC) and has been altered in any way, it cannot be continued with the CONT command.

UNKNOWN USER FUNCTION
A function has been referred to that as yet does not exist; that is, it hasn't yet been defined using DEF FN.

RESUME MISSING
The program has reached its end while still in an ON ERROR routine.

UNEXPECTED RESUME
The RESUME command can only be used at the end of an ON ERROR routine; if used anywhere else in the program, this error command results.

DIRECT COMMAND FOUND
A program line without a line number has been found while the computer was attempting to load a program.

OPERAND MISSING
The computer has come across an expression in BASIC that is incomplete. Typing LOAD and then pressing ENTER without putting quotation marks after the LOAD command would yield this error.

LINE TOO LONG
A line, when converted to BASIC internal form, which becomes too big gives this error message.

EOF MET
The computer is trying to read past the end of a file when inputting data from cassette.

FILE TYPE ERROR
There are several types of cassette file, and each requires individual commands to operate and manipulate them. If you use the wrong type of file command, you'll end up with this error message on-screen.

NEXT MISSING
A NEXT has been found for which there is no matching FOR.

FILE ALREADY OPEN
Another file is being created before a previous file has been closed.

UNKNOWN COMMAND
BASIC cannot find a taker for an external command.

WEND MISSING
The computer has found a WHILE command that doesn't match a WEND.

UNEXPECTED WEND
Essentially, the opposite error to the one above — a WEND has been found that does not have a matching WHILE command.

INDEX

ABS 30
AFTER 149–53
AMSDOS 71
Amsword 96
AND 46
array 38, 39, 40
ASC 50
ASCII 50
ATN 29
attack 138, 139
AUTO 62

BANKFIND 163
BANKOPEN 163
BANKREAD 163
BANKWRITE 163
BASIC 4
binary system 1
boolean algebra 2
BORDER 103, 104
BREAK 15

CALL 26, 147
CAT 65
CHAIN 65
CHAIN MERGE 66
CHKDISC 75
CHR$ 50
CINT 30
CLEAR 52
CLG 122
CLS 11
CP/M 73, 163, 164

CONT 15
COPYDISC (!) 75
COS 29
CREAL 30

DATA 34, 35, 36
decay 138, 139
DEF FN 32
DEF INT 33
DEF REAL 33
DEF STR$ 33
DEG 29
DELETE 61, 62
DI 150
DIM 38, 39
DIR 74
disable the ESC key 163
DISCCHK 75
DISCCOPY 74
DRAW 117
DRAWR 117

EDIT 63
EI 150
ELSE 45, 46, 47
END 26
ENT 140, 141
ENV 138, 139
envelopes 138
ERASE 40
ERL 158
ERR 158
ESC 15

EVERY 149–53
EXP 29

figurative functions 30
FIX 30
FOR 7, 21
FOR/NEXT 22, 23, 24
FRE 62

GOSUB 17, 18
GOTO 16

HIMEM 162

IF/THEN 45, 46, 47
INK 105, 106, 124, 125
INKEY 56
INKEY$ 25, 26
INPUT 55
INSTR 51
INT 30, 42

JOY 57

KEY 60
KEY DEF 59

LEFT$ 49
LEN 48
LINE INPUT 55
LIST 5, 8
LIST#8 162
LOAD 64
LOCATE 15, 102
LOG 29
Logo 76, 77, 78
LOG10 29
LOWER$ 50

mathematical functions 29
MAX 31
MEMORY 162
MERGE 65
MID$ 49
MIN 31
Mindstorms 77
MOVE 118
MOVER 118
multi-tasking 149

nested loops 21, 23
NEW 5
NEXT 21
NOT 46
numeric variables 9

ON BREAK GOSUB 158
ON BREAK STOP 158
ON ERROR GOTO 158, 159
ON . . .GOSUB 18
ON . . . GOTO 16, 18
OR 46
ORIGIN 121, 122

PAPER 11, 105
Papert, Seymour 77
PEEK 147
PEN 11, 105, 106
PI 29
pixel 101, 114
PLOT 114, 115, 116
PLOTR 116
POKE 147
POS 102
PRINT 5
PRINT AT 15
PRINT @ 15
PRINT #8 162

RAD 29
random numbers 41
RANDOMIZE 41

RANDOMIZE TIME 41
READ 34, 35, 36
release 138, 139
REM 12
REMAIN 152
RENUM 61
RESTORE 34, 35, 36
RESUME 159
RETURN 17
RIGHT 49
RND 41, 42
ROUND 31
RUN 5
RUN" " 65

SAVE 64
SCREENCOPY 164
SCREENSWAP 165
SGN 31
SIN 30
SOUND 128, 129, 130
SPACE$ 51
SPEED INK 104, 105, 106
SPEED KEY 59
SPEED WRITE 64
SQ 145
SQR 30
STEP 22
STOP 26
string arrays 40
string variables 40
STRING$ 51
STR$ 48
sustain 138, 139
SYMBOL 107, 108
SYMBOL AFTER 107
system interrupts 149–53

TAB 7
TAG 119
TAGOFF 119
TAN 30
TEST 118
TESTR 118
TIME 151
'trace' commands 62
TRANSPARENT INKS 106
TROFF 62, 63
TRON 62

UDG 109, 110
UNT 31
UPPER$ 50
user-definable graphics 107

VAL 48
VPOS 102

WEND 21, 24, 25
WHILE 21, 24, 25
WIDTH 162
WINDOW 154
WINDOW SWAP 155
wrap-around 119

XPOS 119

YPOS 119

The Amstrad Programmer's Guide is a must for all owners of the Amstrad CPC 464, 664 and 6128 computers. With this new and exciting guide you can . . .

* learn how to master BASIC
* improve programming skills
* manage finances
* write programs with a professional touch
* play games
* write letters
* teach the children
* solve problems

. . . and much more. The Amstrad Programmer's Guide will provide hours of fun and teach you how to program the Amstrad like a professional!

ISBN 0-273-02447-7

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/