BASIC
PROGRAMMING ON THE AMSTRAD COMPUTERS CPC 464, 664 and 6128

Wynford James

MICRO PRESS

BASIC Programming on the Amstrad computers CPC 464, 664 and 6128

BASIC Programming on the Amstrad computers CPC 464, 664 and 6128

Wynford James

MICRO PRESS

First published in 1985 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

Reprinted 1986

British Library Cataloguing in Publication Data
James, Wynford
BASIC programming on the Amstrad CPC 664/464.
1. Amstrad CPC 664 (Computer) — Programming
2. Amstrad CPC 464 (Computer) — Programming
I. Title
001.64'2 QA76.8.A4/

ISBN 0-7447-0036-1

Typeset by Keyset Composition, Colchester, Essex
Printed and bound by Billing and Sons Ltd, Worcester

Contents

The CPC 6128

As this book was about to go to press, Amstrad launched the CPC 6128 home micro. Its advantages over the 664 are that it has twice as much RAM, a different keyboard (which isn't as good as the 664) and a new version of the CP/M business operating system which allows it to run a wider range of business programs.

The good news is that programs written in BASIC on the 464 and 664 will run without alteration on the 6128. This means that it should run all 664 disk based programs and 99.5% of 464 cassette programs. **It also means that you can use all of the programming hints and tips in this book to teach you how to use your new 6128.**

Although the 6128 is supplied with 128K of RAM, only 64K is available for BASIC programs. This is because the BASIC on the 6128 was originally designed for the 464 and 664 which only have 64K available. To help you get around this problem, Amstrad has included some extra BASIC commands which you can load from disk. These allow you to use the 'spare' 64K of RAM either as somewhere to store extra screen images or as a fast filing system. At the moment, you can't use the spare 64K to hold BASIC programs.

All these extra commands are installed by running the 'Bankmanager' machine code program supplied on your system disk. If you want to use the spare RAM to hold extra screen images, Bankmanager supplies you with two extra commands:

'SCREENCOPY' and 'SCREENSWAP' allow you to store up to four screen images in the extra RAM and then switch them onto the screen. This could be useful for games and animation, where you could set up new screens in the background and then switch them in when needed. Each screen image is given a 'Block' number from 1 to 5. To display an image you need to move it to Block 1.

SCREENCOPY copies a whole screen image from a source Block to a target Block. The previous contents of the target Block are always overwritten and lost. For example if you want to save your display for later use, you would type something like 'SCREENCOPY,1,4'. This copies the image from Block 1 (which is always the display screen) to Block 4 (which is part of the spare RAM).

SCREENSWAP just swaps the contents of two Blocks – it doesn't overwrite anything. For example SCREENSWAP,1,4 would put the current screen image in Block 4 and the contents of Block 4 on the screen. Another SCREENSWAP,1,4 would swap the images back to the way they started.

If you decide you want to use the spare 64K of RAM for storing data rather than screen images, the Bankmanager supplies you with four extra BASIC commands as follows:

BANKOPEN allows you to specify how many characters will be in each record. The maximum length is 255 characters. BANKWRITE lets you write a record, BANKREAD lets you read a record and BANKFIND does a string search to find a matching record and returns the record number so that you can then do a BANKREAD to recover the data.

All the commands apart from BANKOPEN need you to specify an integer variable to hold the status code and a string variable to hold the data to be read or written to the file.

Chapter One

Getting Started

Introducing the microcomputer system

You own an Amstrad microcomputer. You are in a rare position for a micro owner, because the Amstrad computer is sold as a complete microcomputer system. You do not need to buy anything else to be able to use your Amstrad minutes after you have unpacked it.

What is a microcomputer system? The Amstrad computer is housed within a grey plastic case, and contains many electronic components which together enable the Amstrad to draw pictures, carry out arithmetic, and do all the things we have come to expect personal micros to be able to do.

But the computer on its own is a dumb beast. It is as clever as a driverless car. To do anything remotely useful the Amstrad must receive instructions from a human being. Thus a keyboard is built into the Amstrad so that we can communicate with the microcomputer inside by typing in our instructions.

However, we humans do have one or two weaknesses compared to the Amstrad. We like to be able to see what we're typing for example. So the Amstrad comes with a monitor, a TV-like screen on which the computer displays every character which we type at the keyboard.

Now, we have a way of giving instructions to the Amstrad, via the keyboard, the input device. And the computer can give messages back to us by displaying them on the monitor, the output device. And we have the Amstrad itself to do all the computing. Surely we need nothing more? So why does the Amstrad CPC 464 also contain a cassette recorder, and the CPC 664 a disk drive? To understand why these devices are so useful, we need to look a little more closely at the electronic components within the Amstrad itself.

Although there are a great many of these components, the most

important areas within the computer from our point of view are those that make up the computer memory.

The forgetful computer

The Amstrad has two types of memory — ROM and RAM. It is probably easier to understand why we need ROM and RAM if we look at the sorts of things human beings remember.

We all remember our own names, where we live, which way is up, and how to turn the television on. This type of information is stored permanently in our brains, because it is vital for everyday life.

The Amstrad also needs to remember some instructions permanently. It needs to know how to carry out arithmetic, which shape to show on the monitor screen when the letter 'A' is pressed at the keyboard, how to draw lines, and many other things. The manufacturer of the Amstrad records instructions which enable the Amstrad to carry out all these tasks in ROM, Read Only Memory, within the computer.

The memory is known as Read Only because the Amstrad (and its owner) can only examine or 'read' the instructions in the ROM — these instructions are permanently recorded and cannot be changed. This is just as well — we would not want to accidentally affect the computer's ability to do arithmetic, or leave it displaying Q every time we pressed 'A' on the keyboard.

But we human beings do not permanently remember everything that we do. We are thankful to forget that awful Western film that was on TV last night, or the mark we got in the Maths exam in the 4th year, or how far it is from the Earth to the moon. A great deal of the information we use is only remembered for very short periods — the telephone number of the local computer shop, the name of the person who has just broken the UK 1500 metres record, etc.

In the same way, the Amstrad 'remembers' some information only temporarily. The instructions stored in the ROM would enable the Amstrad to multiply 12345 by 6789. But once the Amstrad has carried out this calculation and displayed it on the monitor, is there really much point in the micro storing the answer 8384415 forever?

Not really. So the Amstrad stores the instructions that we give

it in RAM, or Random Access Memory. This is a temporary form of memory. The RAM is empty when the Amstrad is switched on. Gradually the RAM will get filled up as we type instructions at the keyboard. That information will stay in the RAM until the Amstrad is switched off.

The RAM acts as a kind of blackboard on which the Amstrad can record information, and that blackboard is 'cleaned' if the machine is switched off. The RAM can hold the equivalent amount of information to about 30 pages of this book. Clearly the micro would not be much use if we could only store information permanently, because we would soon use up all the memory and the computer would be useless. So we can clear the RAM at any time either by asking the computer to do it for us, or simply by switching the machine off.

Unfortunately, the ease with which RAM can be wiped clean does pose another problem. Suppose we have been working away with the Amstrad for several hours. We have typed in many instructions at the keyboard. These instructions, known as the computer program, enable the Amstrad to draw a view of the Earth from space. But what is to become of this masterpiece now? If we leave the Amstrad switched on all the time, to preserve the program, we won't ever be able to use the micro again, because any new program typed in will probably need the RAM occupied by the old one. Yet if we switch the Amstrad off, the program stored in RAM will be lost, and the wonderful picture it drew will be just a memory.

At last, by a roundabout route, we have come to the reason why the CPC 464 needs a cassette recorder and the CPC 664 a disk drive. The micro can take any program stored in RAM and convert it into a series of electrical pulses. On the 464 these pulses are recorded on cassette tape as audible tones. On the 664 the program is stored by magnetising the surface of a floppy disk. The next time we want to impress Aunt Gladys with our artistic skills, we need only place the cassette or disk on which the program is recorded into the machine and 'load' the program back. The computer converts the stored program back into a program in RAM. We can command the Amstrad to obey the program instructions — 'run' the program — and once again the monitor screen will display that view of Earth from space.

It is probably worth while noting at this stage that different makes of micro cannot load or save each other's programs. Each

micro has a different ROM and understands a different set of instructions.

The Amstrad

Now that we have identified the main parts of a microcomputer system and why those parts are needed, let's look at the Amstrad itself.

The Amstrad keyboard is set up in the same way as a typewriter keyboard, with some additional keys which are in a different colour to the rest of the keyboard. We are going to begin by looking at the purpose of these keys, so switch your Amstrad on now.

If you have never used a micro before you may perhaps be worried that you might damage this expensive machine by typing the wrong thing at the keyboard. Fear not. Nothing you can do has any *permanent* effect on the Amstrad. If the screen turns bright red and nothing you type appears on the display, the worst that might happen is that you have to switch the Amstrad off and then back on again, at which point everything returns to normal.

Once your Amstrad is switched on you will see on your monitor or TV screen a few brief lines of blurb announcing that you are using the Amstrad computer, and then, for the CPC 464, 'BASIC 1.0'. BASIC is the computer language built into the Amstrad, a language with its own set of rules. There are many computer languages, but nearly all micros use BASIC as standard, although the BASIC dialect can vary greatly from machine to machine. (CPC 664 owners will find the message 'BASIC 1.1' displayed. This is an improved version of BASIC for the Amstrad, which includes some extra commands discussed later in the book.)

Further down the screen is the word 'Ready', and on the next line a small rectangle on the left-hand edge of the screen. This rectangle is called the *text cursor*, and is the computer equivalent of a pen-nib. The cursor is necessary so that you know the next position the computer is going to 'write' text on the screen.

When you switch the Amstrad on, the keys are set to produce lower case or small letters. The keyboard letters are all upper case or capitals, but if you press the A, B and C keys in succession, you

will find that the Amstrad displays:

abc[

Note that the cursor moves along as you press the keys — it always shows the *next* position at which the computer will print a character.

Now press the CAPS LOCK key at the left of the keyboard. Press the A, B and C keys again. The screen will now display:

abcABC[

Pressing the CAPS LOCK key switches all the alphabetic keys to display capital letters. Pressing it again will cause the keys to produce lower case letters once more, so press CAPS LOCK a third time and type A, B and C.

abcABCabc[

So CAPS LOCK acts as a switch controlling whether the alphabetic keys produce small or capital letters. Unfortunately it is impossible to tell which way the keys are set just by looking at the keyboard. The only way to find out is to press a key and see the result!

Making mistakes

However it doesn't really matter if you press the wrong key or get capitals when you didn't want them, because the Amstrad allows easy correction of errors. Put your finger on the key at the top right of the keyboard marked DEL, and hold the key down.

The cursor travels back to the start of the line, and erases all the characters you have typed so far. The DEL key is the DELETE key, and it erases the character immediately to its left. When you hold the DEL key down for long enough, it *auto-repeats*, and deletes further characters, back to the start of the line if you hold it down for long enough.

In fact most of the keys on the Amstrad keyboard auto-repeat after a short pause. Press the A key and hold it down until no more As are displayed. You should find you have over 6 lines of As, with the cursor coming to a sudden stop about one third of the way along the seventh line.

The reason that the Amstrad stops the As auto-repeating is that there are now 255 characters in succession, and the Amstrad will

not allow any line to have more than 255 characters (count them if you don't believe me!). The Amstrad doesn't consider the edge of the screen to be the end of a line, so although you can see 7 lines of As, the Amstrad still counts this as a single unbroken line (micros have some funny habits).

As far as the Amstrad is concerned, a line only ends when you tell the computer it has ended. You can tell the computer 'That's the end of this line' by pressing the large blue key marked ENTER. Press it now and see what happens.

You should have the following displayed after the lines of As:

```
Syntax error
Ready
[
```

Once you press the ENTER key, you are telling the Amstrad that the line you are typing is now finished, and that the computer should enter that line into its memory. At the moment you are using the Amstrad in *immediate* or *direct* mode, so-called because when you press the ENTER key the computer examines the line you have just typed and *immediately* obeys the instructions you have just given. However the Amstrad must be able to understand those instructions to obey them. In this case the Amstrad can make no sense of 255 letter As, and the words 'Syntax error' are the micro's way of saying "I don't understand". The word 'Ready' just shows that the Amstrad has finished obeying all the commands you typed in, and is now ready for more.

This book is devoted to looking at the instructions that the Amstrad *does* understand, but before we begin experimenting with those instructions let us finish this brief tour of the keyboard.

The two SHIFT keys are provided so that you can select the characters shown on the upper half of the keys that carry two symbols. Press the numbers 1234567890 (on the top row of the keyboard) and then, while holding down either of the SHIFT keys, press 1234567890 again.

```
1234567890!"£$%&'()_[
```

Any key which is marked with two characters will display the one on the upper half of the key if it is held down at the same time as either SHIFT key. If the CAPS LOCK is set so that the alphabetic keys give small letters, holding down either SHIFT key and an alphabetic key at the same time will produce a capital letter.

If you wished to use the " symbol (on the same key as 2 on the top row of the keyboard) you might find it tedious to keep pressing two keys every time you wanted one character. The keys with two characters normally display the character on the bottom half of the key, but they can be set so that they display the other character. This is done by holding down the CTRL key at the bottom right of the keyboard and pressing CAPS LOCK once.

Try it now, and then press the numeric keys on the top row and you should see:

!"£$%&'()_[

The alphabetic keys will now produce capital letters. You can still type in numbers because the Amstrad has a separate numeric keypad to the right of the main keyboard which you can use. Note that the only way to return the keyboard to normal is to hold down CTRL and press CAPS LOCK again.

The DEL key is all very well if you have made a mistake near the end of a line, but what if the error is right back at the start and you have no wish to get rid of nearly an entire line painstakingly typed in? Type in the following, *without* pressing the ENTER key:

Hear is a mistake

'Hear' is to be corrected to 'Here'. Now remember that the cursor tells you at which position the Amstrad will next display a character. The position of the cursor can be adjusted by using the arrowed keys just above the numeric keypad. For this reason these keys are known as the *cursor keys*. Hold down the ← cursor key. The key will move to the left of the line. Stop once the cursor is over the letter 'a' in 'Hear'. (Don't worry if you go past the 'a' — you can use the → key to bring the cursor back to the right place.)

The 'a' is visible inside the cursor. If you now press the CLR key, next to the DEL key at the top right of the keyboard, the 'a' disappears:

Her is a mistake

The rest of the characters on the line are shifted back one place, and the cursor is now over the 'r' in 'Her'. To delete any other characters on the line you would follow exactly the same procedure: move the cursor to the offending character, then press CLR. If several characters in succession are wrong, holding down CLR at this stage will cause it to auto-repeat and all the characters

will be erased. DEL erases the character to the *left* of the cursor, while CLR erases the character *within* the cursor.

In this example you also want to insert a new character in the word 'Her'. Move the cursor to the space after 'Her' and press E. The 'e' will be added on to 'Her' and the rest of the line pushed one character to the right.

```
Here_is a mistake
```

Mistakes in lines are easy to correct using the CLR, DEL and cursor keys, but note that this only works if you haven't pressed the ENTER key. Once you do this, the computer regards the line as finished and correcting and editing a completed line is done in rather a different way.

The CTRL, ESC and COPY keys have special functions which are more easily explained once you have used the Amstrad a little longer. The TAB key has no special use and merely produces a right arrow character.

Giving the Amstrad commands

We have just seen that the Amstrad cannot make sense of a line of 255 As. What sort of instructions *can* the Amstrad understand and obey?

There is a whole variety of them, and you shall be introduced to most of them in the rest of the book. There are commands that cause the Amstrad to draw lines on the screen, to change the colours shown, to carry out complex calculations, or to play notes of music. All of these commands have one thing in common — the Amstrad can obey them because the manufacturer has stored the details of these commands in the ROM and the Amstrad can therefore recognise them.

This leads to the computer seeming to be both very clever and extremely stupid. It can do all sorts of marvellous things, yet if you type a single wrong letter in an instruction the micro will mulishly refuse to obey it, and will display a message like 'Syntax error' to show its confusion. In a moment we shall look at one of the commands the Amstrad will obey, the instruction PRINT. A human would probably accept that PRONT really meant PRINT (after all the I and the O are next to each other on the keyboard and it would be easy to press the wrong one by mistake). The Amstrad would not accept PRONT.

When giving commands to the Amstrad you *must* obey the

rules or *syntax* of the computer language the Amstrad understands. Youcanstillunderstandthissentenceevenifithasnospaces, but the Amstrad wouldn't. It is rather like an old-fashioned uncle who is always correcting your grammar — it won't even listen to you unless you talk to it properly.

The screen display

At last we are ready to begin using the Amstrad seriously. The first thing to do is to *reset* the computer — return it to the state it is at on switch-on, with the RAM completely clear. Resetting the Amstrad can be achieved by switching the micro off and then on again. You can get the same effect less drastically by holding down CTRL and SHIFT at the same time and pressing the ESC key. The screen clears and the Amstrad blurb appears at the top.

When you are satisfied that any line you have typed in or *input* is correct, you must press the ENTER key to tell the Amstrad that you have finished and the command can now be obeyed. For the moment the ENTER key will be shown as one of the keys you must press and will appear as [ENTER] after the other keys. Now type the following:

```
mode2[ENTER]

You get the message:
Syntax error
Ready
[
```

The Amstrad has not understood the command. In Amstrad Basic spaces are used to identify where one word ends and the next starts. The Amstrad can understand 'mode' if a space follows it, but in this case it believes the word is 'mode2' which it does not recognise. Spaces are *very important* in Amstrad Basic, and so you *must* make sure that you do not leave these spaces out in any of the following examples. In some cases you will discover for yourself that spaces do not appear to matter for some instructions, but it really is safer to *always* space your instructions out. If the spaces aren't needed, the Amstrad will ignore them, but if they are needed and you leave them out, the Amstrad will object.

Now type:

```
mode 2[ENTER]
```

This time the Amstrad recognises the word 'mode' and obeys the instruction. The screen clears and 'Ready' appears in much smaller letters at the top left of the screen.

In the real world there are many varieties of paper for different uses. An architect does not design houses on a note-pad, and a novelist does not use foolscap paper to write stories. In computing, the screen display is equivalent to a sheet of paper, and it is useful to be able to change the display to suit the purpose. The command 'mode' followed by the number 0, 1 or 2 selects one of the screen displays allowed on the Amstrad. Each mode allows a different number of characters per line to be displayed on-screen.

Someone typing in a lot of text at the keyboard finds it useful to be able to see as much of it as possible on-screen. Mode 2 is best for this purpose — the Amstrad can print 25 lines with 80 characters in each line in mode 2.

Now type:

mode 1[ENTER]

Again the screen clears, and 'Ready' appears in the top left screen corner. This mode should look familiar — the Amstrad automatically reverts to mode 1 when reset or switched on. Mode 1 has 25 lines with 40 characters per line. Mode 1 gives the most easily readable characters, and you can consider it as the 'working' mode when you are giving commands to the Amstrad. In mode 1, it is much easier to read what you've typed!

Now input:

mode 0[ENTER] (the number is 0 — at the bottom left of the numeric key-pad)

Once more 'Ready' appears — but this time in much larger characters. Mode 0 gives 25 lines with just 20 characters per line. This mode is the best one to use if you want to draw colourful pictures, as you shall see later in the book.

Mode	Number of lines	Characters per line
0	25	20
1	25	40
2	25	80

Figure 1 The three screen modes available on the Amstrad.

Most micros are very fussy about whether you type commands in small or capital letters, but one of the clever features of the Amstrad is that it will accept mode 0, MODE 0, or even mOde 0 as valid commands. For reasons that you will see later it is best to use lower case letters all the time, although the Amstrad is quite happy with other variations. Throughout the rest of this book statements will be referred to in both upper and lower case to emphasise their interchangeability, but you should get used to using lower case when you are actually typing at the keyboard.

Finally, type:

mode 1[ENTER]

to return the Amstrad to the working mode.

The PRINT statement

Type:
print "Fred"[ENTER]
On the screen you get:
Fred
Ready
[

PRINT is used whenever you want to display anything on the screen. The Amstrad prints whatever comes within the inverted commas. Type:

print "7+5"[ENTER]
giving:
7+5
Ready
[

The Amstrad does not work out what 7+5 is, because it is within inverted commas. The computer prints anything in inverted commas exactly as it is. Let's use the Amstrad as a (very expensive) calculator and use PRINT to display the results. Type:

print 7+5[ENTER] (you will have to hold down SHIFT and press the key with ; on it to get the + sign)

The Amstrad prints:

The Amstrad does NOT print 7+5 because the numbers are not in inverted commas this time. It recognises + as an instruction it understands, and so it adds the numbers instead and displays the answer. The space at the start of the 12 is for its sign. Numbers larger than zero are printed with just a space in front rather than a + sign. Now type:

```
print 7−5[ENTER]
```

(the − sign is on the same key as the = sign)

The Amstrad prints:

```
 2
```

Now try:

```
print 5 − 7[ENTER]
```

giving:

```
−2
```

Multiplication is a little more peculiar. Because × is a letter to the computer, the Amstrad uses * for multiplication. Type:

```
print 9*7[ENTER]
```

(hold down SHIFT and press the key with : to get *)

giving:

```
 63
```

There is no division sign on the keyboard, so / (below the ?) is used. Type:

```
print 12/4[ENTER]
```

giving

```
 3
```

Simple arithmetic so far, but of course the Amstrad can cope with much more difficult calculations. Type:

```
print 12345*9876[ENTER]
```

giving:

```
 121919220
```

Or even:

```
print 12.34*5.67*8.9[ENTER]
```

giving:

622.71342

After the last calculation you will find that the whole screen display shifts up or *scrolls*. The statement print "Fred" which started this section off has now disappeared from the top of the screen. The Amstrad always scrolls the display if it has to print something when the cursor is on the bottom line of the screen.

The screen looks very cluttered now, so it is time to introduce another Basic statement. Type:

cls[ENTER]

This is short for CLear Screen, and its effect is self-evident. From now on type cls[ENTER] yourself whenever you feel that the screen needs clearing.

Tidying up the screen

You will use the PRINT statement so often when programming the Amstrad that you will probably be relieved to know that the ROM recognises ? as an abbreviation for print. Type:

? "The ? is a short way of saying print"[ENTER]

(you will need to hold down the SHIFT key and press / to get ?). Type:

? "If you add 7 and 6 you get";"the answer"[ENTER]

giving:

If you add 7 and 6 you get the answer

The ; tells the Amstrad to print whatever comes after the ; next to the last character printed. In this case the two parts to be printed have run together. Not very useful? Type:

? "If you add 7 and 6 you get";7+6[ENTER]

giving:

If you add 7 and 6 you get 13

The Amstrad has carried out the calculation and printed the result. But try:

? "If you work out 5 − 6 you get";5 − 6;"which is messy."
[ENTER]

and you get:

If you work out 5 − 6 you get −1 which is messy.

which shows us that you have to be careful with ;! If you are going to use ; make sure that whatever is printed just before ends in a space. Try:

? "If you add 7 and 6 you get ";"the answer."[ENTER]
? "If you work out 5 − 6 you get ";5 − 6;"which is not messy." [ENTER]

to see the difference. Type:

? 1;2;3[ENTER]

giving:

1 2 3

All numbers are printed with a space for the sign in front and a space immediately after, so you can't have a messy display with all the numbers running together. Type:

? 12;13;14[ENTER]

giving:

12 13 14

You should be able to see that these three numbers are not in line with those printed further up the screen. This would produce a very untidy display if you wanted a whole table of results on the screen — none of the numbers would be in line vertically. Type:

? 1,2,3

giving

1 2 3

Type:

? 12,13,14[ENTER]

giving:

12 13 14

This still has some flaws, (we might not want the numbers to be so far apart, for example) but at least the numbers are in line

vertically. The comma between the numbers tells the Amstrad to print each number beginning at a particular position on-screen. It can also be used with words:

```
? "Fred","Bill","Jenny"[ENTER]
```

giving:

```
Fred            Bill            Jenny
```

It might look as if these are in a different position to the numbers, but don't forget that numbers are printed beginning with a sign, and for + numbers this is shown by printing an invisible space.

The screen layout in mode 1

How does the Amstrad know where to print on the screen to keep everything in line vertically? You saw earlier that in mode 1 there are 25 lines of 40 characters.

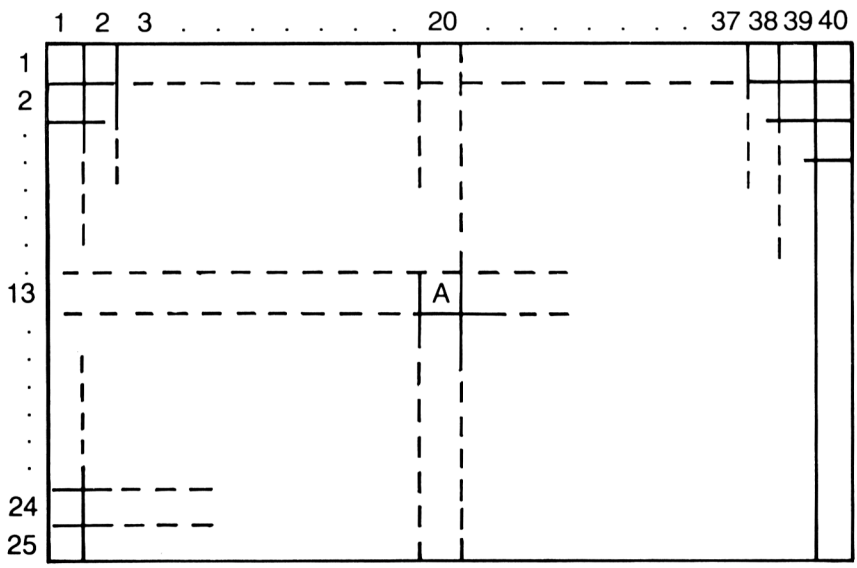

Figure 2 The screen display in mode 1.

A character can be printed anywhere on the screen, and its position can be identified by referring to the column and the line on which it lies. The letter A in Figure 2 lies in column 20 and on line 13, roughly the middle of the screen. Note that the column comes first when you are identifying a character position.

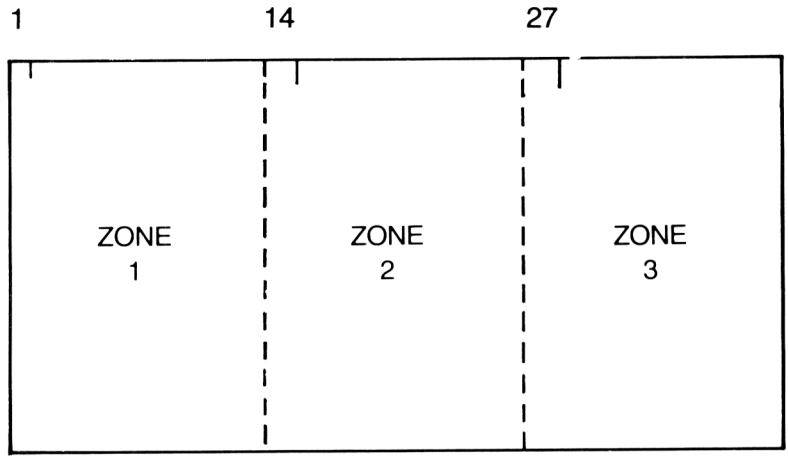

Figure 3 The print zones in mode 1.

The Amstrad divides the screen vertically into invisible *zones* 13 columns wide. If commas are used in a PRINT statement, the Amstrad prints the first item starting in column 1, the next in column 14, the next in column 27. As the next zone would begin in column 40, there is no room for another zone of 13 characters, so the Amstrad starts printing again on a new line. If any item is so long that it fills more than one zone, the computer moves to a new empty zone to print the next item.

Type:

```
? 1,2,3,4,5[ENTER]
```

and you will see the Amstrad print the last two numbers on a new line. Or try:

```
? "Just too many words","two words","just three words"
[ENTER]
```

to see how the Amstrad copes with printing something that needs more than just 13 columns.

The TAB and SPC statements

Although the zones are very useful for printing tables on the screen, it would be restricting if these were the only positions on a line you could print. Amstrad Basic also contains statements to

enable you to print to any position on a line. If you were printing a heading that you wanted centred on a line you would use the PRINT TAB statement, or ? TAB for short. Type:

```
cls[ENTER]
? tab(9)"This begins in column 9"[ENTER]
```

If you refer back to Figure 2 you can see that in mode 1 there are 40 columns in which characters can be printed. You can refer to any of those columns by its number. By using ? TAB(9) you are saying that what is printed should begin at column number 9. The number in the brackets after the TAB can be anything from 1 to 40 in mode 1. You might like to experiment to see what the Amstrad does if you use a number greater than 40, or a negative number!

You can use more than one TAB on a line:

```
? tab (9)"Hello"tab(20)"there"tab(30)"folks"[ENTER]
```

The TAB statement causes the Amstrad to print spaces up to the column where printing is to begin. After the Amstrad has printed "Hello", the cursor stays on the same line, and prints spaces up to column 20, where the printing of "there" begins, and so on. You might think:

```
? tab (30)"folks"tab(20)"there"tab(9)"Hello"[ENTER]
```

would give the same result. However, the Amstrad will not TAB backwards along a line, and if a new TAB is to an earlier position on the line, that TAB will be carried out on a completely new line.

Now type:

```
? "This"spc(4)"sentence"spc(4)"is"spc(4)"spaced"spc(4)
"out!"[ENTER]
```

The Amstrad prints 4 spaces between each of the words. SPC(10) would leave 10 spaces, and so on. Again, in mode 1 the number in brackets after SPC can be anything from 1 to 40. SPC and TAB can be mixed in a command:

```
? tab(10)"A mixture of TAB"spc(8)"and SPC."[ENTER]
```

You might have noticed that all the words in the TAB and SPC statements followed on directly from each other, as if semi-colons had been used. In fact both TAB and SPC automatically conclude with an 'invisible' semi-colon, so that any further TABs or SPCs continue on the same line.

Exercises

1) Try out the TAB and SPC statements in mode 0 and mode 2. Are there any differences in the size of numbers you can use in the TAB or SPC statements in the different modes?
2) Type in some PRINT statements using commas in modes 0 and 2. Why are the results different from what happens in mode 1?

Chapter Two

Programming

Simple programming

In the last chapter you were using the Amstrad in *immediate* or *direct* mode. Any commands typed in at the keyboard were obeyed instantly if the Amstrad understood them. Immediate mode has its uses because it enables you to experiment with commands and observe the effect straight away.

However you will spend most of your time using the Amstrad in PROGRAM mode. You can see the difference between the two modes by typing:

```
1 print"Tom"[ENTER]
```

When you press the ENTER key nothing happens. At the start of the line you typed the number 1. The Amstrad reads this *line number* and recognises that you do *not* want the computer to obey the instruction that follows. Instead the Amstrad stores the line in RAM, waiting for you to give the command to obey it.

You can confirm that the Amstrad now has your command tucked away in its memory by typing:

```
list[ENTER]
```

Your program of one line is listed on the screen. The Amstrad recognises the word 'print' as being one of the Basic language *key-words* that it understands, and so converts it to 'PRINT' in the listing. Type:

```
9 print "Harry"[ENTER]
5 print "Dick"[ENTER]
list[ENTER]
```

Although you typed in the lines in the order 1, 9 and then 5, the Amstrad lists the program in line number order, from the lowest number line to the highest. It will also obey the program instruc-

tions in that same order, and . . . how do we get the Amstrad to carry out the program? Type:

run[ENTER]

The computer RUNs the program, and obeys the program instructions in line number order. When it has finished, the program is still stored in the memory, as you can see by typing LIST[ENTER] again. Type:

5 print "Deirdre"[ENTER]
list[ENTER]

The new line 5 has replaced the old one, and poor Dick is gone. You can always change any line just by typing a new line with the same line number as the one you wish to replace. Type:

1[ENTER]
list[ENTER]

Line 1 has been replaced by an empty line, so the Amstrad does not even list it as part of the program. This is an easy way of getting rid of a line you no longer want.

You could now do several things with your program. You could extend it by adding new lines in any order. You could save it for future use by storing it on cassette or disk. Or you can wipe it from the RAM. Let's take the last course until we have a program that is really worth saving. Type:

new[ENTER]

This tells the micro you want to delete the program and start anew. Typing list[ENTER] at this stage will produce nothing as the program has gone.

Some help the Amstrad can give you

The Amstrad has several features which make programming much simpler.

It is wise to spread line numbers out when you are typing a program in, because you may later want to add a command which needs to come before some of those already input. Things might get tricky if you have numbered the lines 1, 2, 3, and 4 and now

you want to put a line after 2 but before 3! The standard practice is to use line numbers going up in tens, to leave plenty of free line numbers for any commands added later. The Amstrad will even do this for you, so let's use this facility as we write the next program. Type:

```
auto[ENTER]
```

The Amstrad prints '10' and waits for you to type a line in. Type:

```
10 mode 0[ENTER]
20 ?"This is in mode 0"[ENTER]
30 ?"Here comes a multiplication!!"[ENTER]
40 ?"73.45 times 5.769 gives ";73.45*5.769[ENTER]
```

After the last line the Amstrad prints the new line number 50, but we have no more lines to add to the program. You can escape from the automatic line numbering by pressing the ESC key at the top left of the keyboard. Type:

```
list[ENTER]
```

and you will see that line 50 is not included in the program. Type:

```
run[ENTER]
```

and you will see the effect as the Amstrad obeys your program line by line. Type:

```
mode 1[ENTER]
```

to get back to the 'working' mode.

AUTO can be used to begin numbering at any line number, with any interval in between, although generally it is used just as you have seen above. Type:

```
auto 15,1[ENTER]
```

and line numbering begins at 15, with intervals of 1. Type:

```
15 ?"Here is an extra line"[ENTER]
16 ?"And here's another"[ENTER]
```

and press [ESC] again. List the program and then run it and you will see the two new lines have been inserted into the correct position within the program. The Amstrad checks to see if the line number it generates has already been used in the program,

because this new line will replace the old one once [ENTER] is pressed.

On the CPC 464 the computer warns you if this might happen by printing a * after the line number just displayed. If you [ESC] at this point, the line will stay as it was. CPC 464 owners can see this by typing:

```
auto 10[ENTER]
```

The Amstrad prints:

```
10*
```

to tell you that you already have a line number 10 and if you continue this line will be wiped out when you press [ENTER]. On the CPC 664 the AUTO command has been improved, and 664 owners do not just get a '*' to show that the line number has already been used. The entire line is displayed again, and in this case the CPC 664 prints:

```
10 mode 0
```

The advantage here is that you can make changes or edit the line displayed (more of that in a moment!). By using AUTO on the 664 you can quickly step through all the lines in your program in succession, modifying any that contain errors.

If you add too many extra lines to a program you may end up having the problem we have tried to avoid — no room for new line numbers. If you list the present program you will see that we now have no room to put a line between 15 and 16. Luckily the Amstrad can help us out here as well. Type:

```
renum[ENTER]
list[ENTER]
```

The command RENUM renumbers all the program lines, with the first line as 10 and subsequent lines going up in tens. As with AUTO, you can choose any number as the first line number, and space the line numbers out by any amount you desire. Type:

```
renum 100,5[ENTER]
list[ENTER]
```

and the program will be renumbered beginning with line 100 and going up in 5s.

What to do when you make a mistake

Type *exactly* the following:

```
1 pront "This is the first line"[ENTER]
```

On this line we have deliberately made a mistake, by putting 'pront' instead of 'print'. Once you press [ENTER] in *immediate* mode the Amstrad will try to obey your command straight away. If you make a mistake the computer will give an *error message* to show that it doesn't understand the command.

In program mode things are very different. The Amstrad just stores away any numbered lines and does not check to see if they make sense until the program is run. Type:

```
run[ENTER]
```

The Amstrad tries to carry out line 1, and then abandons execution of the program because it does not recognise 'pront'. It tells you which line caused the trouble, and even prints the line for you. You can now *edit* line 1 to correct the error.

Notice the cursor is over the first character on the line, the number 1. Using the *right cursor* key (the arrow key next to the COPY key at the top right of the keyboard), move the cursor until it is over the 'o' in 'pront'. In the last chapter we saw that pressing CLR removed the character within the cursor. This is exactly what we want to do now, because the 'o' is incorrect. Press CLR once. The line is now:

```
1 prnt "This is the first line"
```

with the cursor over the n. We now need to add the character 'i'. The Amstrad automatically inserts any characters typed in *before* the character within the cursor. Type 'i'. The line is now:

```
1 print "This is the first line"
```

At the moment the only changes made have been to the line shown on the screen. Press [ENTER] and the Amstrad replaces the incorrect line with the line we have just edited. (It doesn't matter that the cursor is half way along the line in this case.) You can see the new line by listing the program again.

If you spot a mistake in your program before running it, you can correct the line yourself by typing:

```
edit 1[ENTER]
```

or whatever line number you wish to edit. You can then use the cursor keys, the CLR key and perhaps the DEL key to correct the line. Try it out now — edit a few changes into some of your program lines and then edit them out again.

Editing by copying lines

There is an alternative method of editing lines which you may prefer to use. This involves *copying* the line. Type:

```
1 pront "Here is another line with a mistake"[ENTER]
```

Now hold down either of the SHIFT keys and *at the same time* press the UP CURSOR key. The cursor splits in two, and one cursor covers the '1' in the line you have just typed. The upper cursor is the COPY cursor and shows what you are about to copy. The other cursor is the normal TEXT cursor which keeps your place on the screen, and anything you copy will be printed here. Press the COPY key once and you will see that the '1' is copied onto the new line. Press COPY three times more and you should see this on-screen:

```
1 pront "Here is another line with a mistake"
1 pr
```

At this stage we do not want to copy the 'o' because this is the part of the line which is incorrect. The lower text cursor is the normal cursor which shows where the next character typed will be printed, so if you type 'i' it will be printed here and the screen will now show:

```
1 pront "Here is another line with a mistake"
1 pri
```

We now want to copy the rest of line 1, but we don't want that 'o'! You might think that you can move the copy cursor just by using the cursor keys. Try it and see what happens.

The cursor keys move the text cursor — and the Amstrad bleeps at you if you try to move beyond the new line you are making! To move the copy cursor you must *always* hold down SHIFT and a cursor key. Do it now to move the copy cursor onto the 'n' in 'pront'.

Make sure the text cursor is *after* the i in '1 pri', because it is here that copying will start. Press the COPY key and hold it down

until you have reached the end of the line you are copying. Don't worry if you copy a lot of spaces as well as the Amstrad doesn't take any notice of them. Press [ENTER] and this new line 1 will replace the old one in the computer's memory, as you can see by listing the program.

This whole process may seem rather complicated at the moment, but you will quickly discover it becomes second nature. It's well worth spending some time getting the hang of editing, because you'll find it speeds programming up considerably.

101 uses for the COPY key

Although the COPY key is useful for editing, I also use it a great deal when inputting program lines. If several lines in a program are similar, I copy chunks of one line to another. Let's look at a simple example. Type:

```
new[ENTER]
auto[ENTER]
10 print"*******"[ENTER]
```

Now use the SHIFT and cursor keys to get the copy cursor onto the 'p' of 'print'. Press the COPY key and copy the rest of the line, then press [ENTER]. You should now have:

```
10 print"*******"
20 print"*******"
30
```

Repeat the copying process on lines 30, 40 and 50. Press ESC once the Amstrad prints line number 60. List and then run the program. You should get a square of *s. You can use this same technique to copy just part of a previous line or combine different parts of several other lines.

We have concentrated so much on editing that we have rather neglected the programming side of things. Let's use some of the statements that were introduced in the last chapter. Type:

```
new[ENTER]
auto[ENTER]
10 ?"Here's a semi-colon ";[ENTER]
20 ?"and look what it does!"[ENTER]
30 [ESC]
```

Run the program. Remember what the semi-colon does? It causes whatever is printed next to immediately follow the present characters. Edit the ; out of line 10 and run the program again.

This time the printing takes place on two lines. Whenever the Amstrad comes to a PRINT statement, it will always begin printing on a new line, unless the previous PRINT ended with a semi-colon. This also applies to PRINT TAB and PRINT SPC:

```
new[ENTER]
auto[ENTER]
10 ?tab(9)"This is";[ENTER]
20 ?tab(17)"the Amstrad."[ENTER]
30 [ESC]
```

Run the program and you will see all the printing takes place on one line. Edit the ; out of line 10, and run the program again. This time two lines are used.

Use the COPY key to help you as you type in the next program:

```
new[ENTER]
auto[ENTER]
10 ? tab(20);"A"[ENTER]
20 ? tab(19);"A A"[ENTER]
30 ? tab(18);"A";spc(3);"A"[ENTER]
40 ? tab(17);"A";spc(5);"A"[ENTER]
50 ? tab(16);"AAAAAAAAA"[ENTER]
60 ? tab(15);"A";spc(9);"A"[ENTER]
70 ? tab(14);"A";spc(11);"A"[ENTER]
80 [ESC]
```

Run the program. Notice the use of SPC. It is much easier to use this than to carefully count spaces and have lines like:

```
60 ? tab(15);"A          A"
```

The LOCATE statement

You can print anywhere on the screen by using the LOCATE statement:

```
new[ENTER]
auto[ENTER]
10 mode 1[ENTER]
20 ? "Let's print beginning at (10,12)."[ENTER]
```

```
30 locate 10,12[ENTER]
40 ?"Here it is!"[ENTER]
50 [ESC]
```

The LOCATE command at line 30 moves the cursor to column 10 of line 12 on the screen. Printing begins at that position when line 40 is obeyed.

Run the program again, changing line 10 to:

```
10 mode 0[ENTER]
```

The message is printed beginning in the middle of the screen. There are only 20 columns in mode 0, and column 10 is half way along. Similarly, running the program in mode 2 will print the

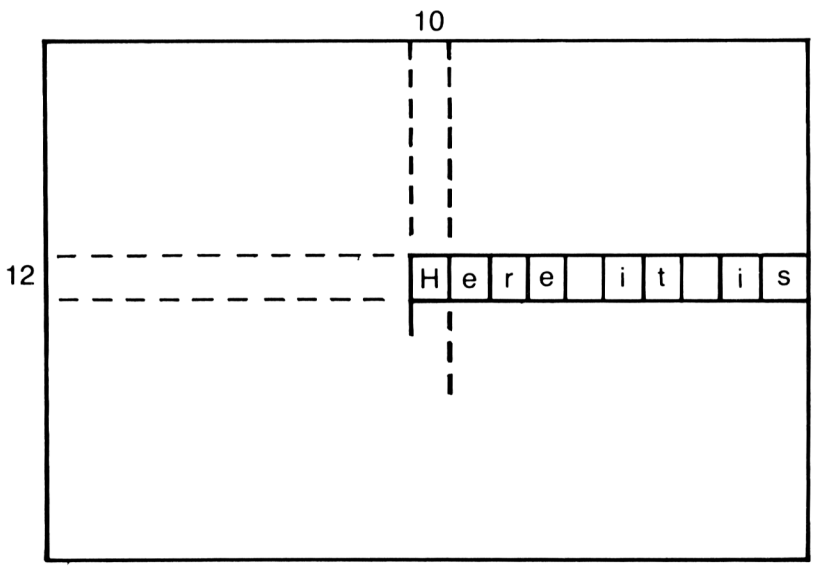

Figure 4 Cursor control.

message much closer to the left-hand end of the screen. There are 80 columns in mode 2, and column 10 is only an eighth of the way along a line.

You can still use SPC or TAB to get to other positions after a LOCATE statement, although again you can't TAB backwards to an earlier position on a line if you've already used LOCATE to go further along the line.

Exercises

(You can use the COPY key to make the programming easier in all the following questions. Make sure you type new[ENTER] to get rid of any previous program which is no longer needed.)

1) Write a program to print the following pattern:

```
**********
*        *
*        *
**********
```

2) Print this pattern beginning at (10,20) in mode 1:

```
*****
****
***
**
*
```

3) Use TAB or SPC in a program to produce this pattern:

```
*****
 ****
  ***
   **
    *
```

4) Use PRINT TAB or PRINT SPC to produce this display in mode 0:

```
This        message
  is        for
Amstrad     users
      only
```

Using variables

Let's write a brief program to make the Amstrad print a message. Type:

```
new[ENTER]
auto[ENTER]
10 ?"Hello there my friend"[ENTER]
20 ?"My name is Amstrad"[ENTER]
30 ?"and I think you're wonderful!"[ENTER]
40 [ESC]
```

Run the program and you will find that the Amstrad prints the message for you. Fantastic. Isn't the micro an amazing machine?

But this program isn't very exciting. If you run it again, you will get exactly the same message. Wouldn't it be nice if we could get the Amstrad to address whoever was using the program by their name? Now that *would* be an improvement. Type:

```
5 let name$="Wynford"[ENTER]   (put your name within the
                                inverted commas)
```

Edit line 10 to read:

```
10 ?"Hello there my friend ";name$
```

List the program, then run it. This time the Amstrad gives you a personal message. This may not seem much better than the previous effort, but there is an important difference between the two programs. Line 5 reads:

```
5 let name$="Wynford"
```

The Amstrad's memory consists of thousands of memory locations, which can be considered as a series of empty pigeon holes. Each of the holes can be used to store information. Line 5 of the program tells the Amstrad to take the word 'Wynford', find any one of those holes that is empty, label it with 'name$', and store the word 'Wynford' in that hole.

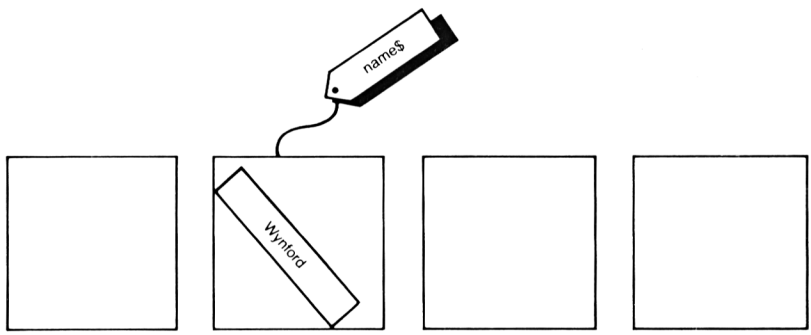

Figure 5 Storing a string in memory.

The '=' sign in line 5 is rather misleading, because we generally take '=' to mean that two things are exactly the same. In computing, a line like 5 should be read as:

> take the word 'Wynford' and store it in the memory location labelled as 'name$'.

When there is a reference to name$ in the program at line 10, the Amstrad goes to the 'hole' labelled with 'name$', and fetches the word it finds there. That word is then used in the print statement,

as you can see from the effect of line 10 when you run the program.

If you change the value of name$ by typing:

```
5 let name$="Attila the Hun"[ENTER]
```

and run the program again, you will see that the Amstrad has now stored 'Attila the Hun' at the memory location name$, and so it uses 'Attila the Hun' when it obeys line 10. In fact you can store *anything* in name$, and for this reason name$ is known as a *variable*. This means its value varies and depends on what you decide it should be. The '$' at the end of the variable shows that name$ is a *string* variable. This means what is stored there is a string of characters, which might be alphabetic, numeric or a mixture of the two. Type:

```
5 let name$="123abc£$%"[ENTER]
```

and run the program again to see that name$ really can be used to store *anything*.

Do we have to use name$? Type:

```
 5 let DaftStringVariableName$="John McEnroe"[ENTER]
10 ?"Hello there my friend ";DaftStringVariableName$
   [ENTER]
```

and run the program. The Amstrad has no objection to the variable names in lines 5 and 10. However, when programming it makes good sense to use variable names that remind you what the variable is for. For example:

```
let makeofcar$="British Leyland"
let titleofbook$="The Tin Men"
let playerone$="Fred"
let city$="London"
```

Variables like 'jelly$' or 'xyz$' are not very helpful reminders as to what the program does if you look at it months later and try to correct the errors in it!

There are a few limitations on string variable names:

1) They must not begin with a number, although numbers can be used elsewhere in the name.
2) Only alphabetic or numeric characters can be used.
3) The variable cannot be more than 40 characters long.
4) You must not use a key-word like PRINT or LET as a variable name.

Although you can use lower or upper case alphabetic characters, I prefer to use lower case all the time. This makes the variables easier to pick out when the program is listed, as the Amstrad automatically capitalises Basic key words like PRINT or LET. This capitalisation is also why you should avoid using key-words as variable names. The Amstrad will convert the variables to the capitalised key-words and expect those key-words to indicate commands.

The LET in line 5 of the program is *optional* and you do not have to include it. Edit line 5 to omit the LET. You will find the Amstrad still runs the program. Having just introduced LET, I am afraid you will not see it again in this book — leaving it out during programming saves a lot of typing!

If you are not certain if a variable name is valid or not, try it out in *immediate mode* to see the Amstrad's reaction. Type:

```
thisisokay$="Strawberry jam sandwiches"
```

and the Amstrad gives you a 'Ready' message, showing the variable name is acceptable, but type:

```
thisoneis not okay$="10 fat sausages sizzling in a pan"
[ENTER]
```

and you get an error message, because a space is not allowed in a variable name.

Once you have given a variable a value, you can use it as many times as you want within a program:

```
new[ENTER]
auto[ENTER]
10 let nicefood$="ice cream"[ENTER]
20 let nastyfood$="chocolate ants"[ENTER]
30 ?"So you like ";nicefood$;" then?"[ENTER]
40 ?"Personally, I prefer ";nastyfood$[ENTER]
50 ?nicefood$;" is okay for humans"[ENTER]
60 ?"I'd like to see ";nastyfood$;" ";nicefood$[ENTER]
70 [ESC]
```

What happens if you store a new string in one of the string variables you have already used? Type:

```
15 let nicefood$="trifle"[ENTER]
```

and run the program again. The string "ice cream" has been lost,

because the new string "trifle" replaces it. Line 15 tells the Amstrad to store the word "trifle" in the memory location nicefood$. The Amstrad finds that it already has the words "ice cream" at this memory location. It discards this and puts "trifle" in its place.

Exercises

1) Which of the following are allowed string variable names?
   AVERYBIGWORD$
   PrimeMinister$
   10DowningStreet$
   TenDowningStreet$
   Ten-Downing-Street$
   A$
   t$
   $
2) Correct the errors in this program so it will work:
   10 let today'sdate$="July 24 1984"
   20 print "The date is";today'sdate$
3) Complete this program so that when run it prints:
   Hello, Mr. James
   Or may I call you Wynford?
   10 let surname$="James"
   20 let christianname$="Wynford"
4) Choose some suitable words to store in strings which the computer uses to print out a description of the town sheriff. Here is an example, with the strings being printed in capitals so you can see where they go:

   > He wore a BLACK HAT on his HEAD and his name was MUDD. KILLER MUDD, they called him, because of the way he handled his GUN and the number of MEN he had sent to BOOT HILL.

The INPUT statement

Our program has introduced variables to us, but it hardly seems much better than the original program. Every time the program is run it still gives the same old tired message, depending upon what value you give name$ in line 5. But didn't I say that we were

going to write a program that would give a personalised message to whoever used it? Indeed I did, and to do that you need to learn about the INPUT statement. Type:

```
new[ENTER]
auto[ENTER]
10 input "What do your friends call you"; name$[ENTER]
20 ?"And how are you today ";name$[ENTER]
30 [ESC]
```

Notice that nowhere in the program have we given name$ a value. So how will the Amstrad know what to print at line 20?

Run the program. The Amstrad will print:

```
What do your friends call you?
```

Type your name in (there is no need for inverted commas) and press [ENTER]. The Amstrad takes your name, stores it in name$, and uses it when it comes to line 20. Run the program a few times, inputting a different name each time. You now have a general-purpose program that will give a different message depending upon who uses the program.

The INPUT statement enables the Amstrad to ask you for information while the program is running. The message included within the inverted commas in line 10 is called a *prompt*, as it prompts you to type something at the keyboard and even reminds you what information the Amstrad requires. The prompt is optional, and you could just have:

```
10 input name$
```

but this just gives a very cold and unhelpful ? when you run the program. There is too much of this sort of programming around. Let's keep things friendly and always include a prompt in INPUT statements.

The semi-colon at the end of the prompt just tells the Amstrad that we want to keep the ? that it automatically generates for INPUT statements. If you substitute a comma in line 10 you will find the ? is omitted. This is really a matter of style. You may prefer a more formal line 1 without a question mark. Type:

```
1 input "Type in your name. ",name$[ENTER]
  run[ENTER]
```

to see the difference.

A single INPUT statement can be used to input several strings in one go:

```
new[ENTER]
auto[ENTER]
10 input "Give your name and age, with a comma between
   ",name$,age$[ENTER]
20 ?"Really, ";name$;"! You don't look ";age$;" years old."
   [ENTER]
30 [ESC]
```

The Amstrad insists that the values input must be typed in separated by commas. Failing to do this gives a 'Redo from start' message. You will get the same message if you only input 1 of the 2 strings required, or if you input 3 strings instead of two. This is another reason to include the prompt in the INPUT statement. It reminds you how many inputs are needed, and what those inputs are.

Exercises

1) You may have received one of the dreaded 'personal' letters from a computer offering you a chance of winning a prize in a competition. Usually you are also urged to buy a useful book like 'The World Atlas of Worms'. Write a program to produce such a letter, after you have input details like name, address, today's date, etc. I offer my own example below, with the information which has been input shown in capitals:

The Leatherbound Book Company
Greek Street
Lower Ecclestone
JULY 24 1984

Dear MR. JAMES,

I was just strolling past 20 SYCAMORE ROAD the other day when I thought of you. MR. JAMES is one of the wisest men in EAST BLUNSTONE, I thought to myself. I am sure that he and everyone else in 20 SYCAMORE ROAD would see what wonderful value 'Investment in Lower Ecclestone' is, and at only £24.95!! And a chance for MR. JAMES himself to win an all expenses paid trip to the Leatherbound Book Company in Greek Street! Don't delay

MR. JAMES — you have just 7 days from JULY 24 1984 to take advantage of this golden opportunity.

Yours sincerely,
A good friend

2) Write a program that accepts as input the endings for 3 lines of a limerick, and then prints out the limerick. The following may inspire (or disgust) you, the input strings being printed in capitals:

There was a young fellow called JIM
Whose legs were exceedingly SLIM
He went for a run
In the hot midday sun
And then saw HIS LEGS LOOKED QUITE GRIM

3) Write a program which enables you to input your name and address, and prints out an address label like the following:

```
.....
.                                                   .
.       NAME:       Wynford James, B.Sc.            .
.       ADDRESS:      20 Sycamore Road              .
.                     East Blunstone                .
.                     Yorks.                        .
.....
```

The LINE INPUT statement

You may have had trouble with the last problem if your input string included a comma. There are occasions when this happens. For example, someone may type their name as 'Fred Adams, M.B.E.'. The ordinary INPUT statement is unusable here, as the Amstrad keeps on giving the 'Redo from start' message if a comma is typed:

```
new[ENTER]
auto[ENTER]
10 input "Your full title, Fred. ",title$[ENTER]
20 [ESC]
```

If you run this mini-program, you will find that inputs like "HRH, the Queen" produce a 'Redo from start' message. The command suggests to the Amstrad that your input is finished after "HRH", and then you go on to type "the Queen"! The Amstrad thinks you've INPUT two strings when it only wants

one, so it protests.

Here we must use the LINE INPUT statement, which accepts a whole line of text, commas and all. Change line 10 to:

```
10 line input "Your full title, Fred. ",title$[ENTER]
```

and you'll find that you can now type *anything* and it will be acceptable as an input. You can only use LINE INPUT to input *one* string, of course, because the Amstrad will no longer take a comma to mean that you have finished the first string and now want to input a second. One other point to remember about LINE INPUT is that it won't give the ? that you get automatically with INPUT. If you want a question mark you'll have to include it yourself in the prompt string.

Numeric variables

You should use string variables whenever you want to store a string of alphabetic characters, or a mixture of alphabetic and numeric. It seems logical that you should also be able to store numbers. Well, you can — but it depends what you want to do with the numbers. Type:

```
new[ENTER]
auto[ENTER]
10 firstnumber$="123"[ENTER]
20 secondnumber$="345"[ENTER]
30 ?"The first number is ";firstnumber$[ENTER]
40 ?"The second number is ";secondnumber$[ENTER]
50 [ESC]
```

Run the program. Works fine doesn't it? And if all you ever want to do with those numbers is print them, then go ahead and use string variables to store their values. But if you want to do any arithmetic, type:

```
50 ? "Multiplying them together gives ";
   firstnumber$*secondnumber$[ENTER]
```

Run the program. The Amstrad prints:

```
Type mismatch in 50.
```

A new error message. It tells us that the Amstrad won't do arithmetic with strings. Quite sensible really. After all,

firstnumber$ might be "Jellybabies" and secondnumber$ might be "The Atlantic Ocean" and you can hardly multiply those together! Before it carries out arithmetic on any variables the Amstrad needs to be absolutely sure it's dealing with numbers. So if you have any numbers which are going to be used in calculations they must be stored not in string variables but in *numeric variables.*

Numeric variables are remarkably like string variables. The only difference is that they lack the '$' sign, and when we store a value in them, we do not use inverted commas. For example:

```
length=27
height=7.6
distance=38.45
petrolused=5.5
temperature=-10.5
```

are all acceptable and reasonable variable names. Also acceptable, but not very helpful for programming, are:

```
y33=5.6
A=75
```

Numeric variables can hold integer (whole) numbers like 27, or decimal numbers like −10.5. But they won't hold strings — try it.

If you list the original program, you will see that with a few minor changes you can make the program work. All you need to do is edit a few lines so that the numbers are stored in numeric variables rather than as strings. Edit your program so that it looks like this:

```
10 first number=123
20 secondnumber=345
30 ?"The first number is ";firstnumber
40 ?"The second number is ";secondnumber
50 ?"Multiplying them together gives ";firstnumber*
   secondnumber
```

Run your program. The last line should now read:

```
Multiplying them together gives 42435
```

Let's add a few more lines. Type:

```
auto 60[ENTER]
```

```
60 ?"Adding them gives ";firstnumber+secondnumber
   [ENTER]
70 ?"Dividing them gives ";firstnumber/secondnumber
   [ENTER]
80 ?"Subtracting gives ";firstnumber-secondnumber
   [ENTER]
run[ENTER]
```

In lines 50 to 80 the Amstrad fetches the numbers from their locations in memory and uses them to carry out the arithmetic. We can change lines 10 and 20 to INPUT statements so that the value of firstnumber and secondnumber can be chosen while the program is running:

```
10 input "What is the first number";firstnumber[ENTER]
20 input "What is the second number";secondnumber
   [ENTER]
```

The Amstrad now carries out all the calculations on the numbers you input.

Exercises

1) You are going to visit Rurislovakia, where the currency is in ruris, and there are 2.15 ruris to the pound. Write a program which converts pounds to ruris after you have input the number of pounds.
2) Write a program that will find the area of a carpet after you input its length and width.

Arithmetic with variables

Although you have so far only learnt a few Basic statements, the use of variables vastly increases the power of programs you can write. For example, in just 3 lines you can write a program that converts kilometres into miles:

```
new[ENTER]
auto[ENTER]
10 onemile=1.6093[ENTER]   (a mile is 1.6093 kilometres)
20 input "What is the distance in kilometres ";
   distanceinkilometres[ENTER]
30 ?"This is";distanceinkilometres/onemile;"miles."
   [ENTER]
```

40 [ESC]

Now suppose we extend this program. Your car does about 30 miles to the gallon, and it would be useful to know how many gallons you'll need to cover a particular distance in kilometres. We already work out the number of miles at line 30. Rather than work it all out again on another line, and then divide by 30 to find out how many gallons we need, it is much simpler to save the result of the calculation carried out in 30 and store it in another variable. So add lines to your program until it is like this:

```
10 onemile=1.6093
20 input "How far is the distance in kilometres ";
   distanceinkilometres
25 distanceinmiles=distanceinkilometres/onemile
30 ? "This is";distanceinmiles;"miles."
50 ? "It would take";distanceinmiles/30;"gallons."
```

Line 25 takes the result of dividing the distance in kilometres by 1.6093 and stores it in the variable distanceinmiles. This is used again in line 50 to work out how many gallons of petrol are required.

Petrol costs £1.80 a gallon, and you're very keen to know how much the journey will cost. Let's try to make this program more general now. After all, you will probably get a new car some time, and it may not do 30 miles to the gallon. The price of petrol is unlikely to stay the same either. Let's put all these numbers into variables:

```
10 onemile=1.6093
12 milestothegallon=30
14 costofpetrol=1.80
20 input "How far is the distance in kilometres ";
   distanceinkilometres
25 distanceinmiles=distanceinkilometres/onemile
30 ? "This is";distanceinmiles;"miles."
40 gallonsforjourney=distanceinmiles/milestothegallon
50 ? "It would take";gallonsforjourney;"gallons."
60 costforjourney=gallonsforjourney*costofpetrol
70 ? "It would cost";costforjourney;"pounds."
```

In lines 25, 40 and 60 we are using variables whose values we know to calculate the value of new variables, which are themselves used in later calculations. If you change your car, or the cost

of petrol goes up, you only need to edit line 12 or 14 to the new value to find the program of use again. But that's the hard way to do it, isn't it? The best program of all would be one where you could input the number of miles your car did to the gallon, and the cost of petrol — then you would never need to edit any lines. Therefore change lines 12 and 14 so that these values are input by the user, just like line 20.

Exercises

1) You wrote an earlier program that accepted as input the length and width of a room, and calculated the area to be carpeted. Extend the program to find the cost of carpeting if the cost per square yard is input by the user.
2) A fish-and-chip shop owner has just bought a computer to help him. He wants to be able to type in the cost of an order, and then have the computer work out the 15% VAT for the order. The VAT is then added on to the cost to give the total price. Write the program for him.
3) The ReadEasy Book Club sends a letter out to each member every month in the following form:

Member's name: WYNFORD JAMES
Membership no.: 12345678

You owe us	£12.50
This month's book costs	+£ 5.95
Total	£18.45
Last month's payment — thanks	−£ 5.00
Outstanding	£13.45

Write a program which accepts as input the member's name, number, the amount owed, the cost of this month's book, and the payment made by the member last month. The program then prints out details of the money owed as shown above.

Does 2=2+1??

The heading of this section is intended to remind you that the '=' sign in programming does not always mean 'equals'. Type:

```
new[ENTER]
auto[ENTER]
10 input "Type in any number. ",number[ENTER]
```

```
20 number=number+1[ENTER]
30 ? "The number is now ";number[ENTER]
40 [ESC]
```

Run the program and type in any number. You will find that the value printed at the end of the program is 1 more than the number you typed in. How has this happened?

Line 20 seems to be nonsense, and it is if you read the '=' as meaning 'equals'. Let's go through the program from the start and see what happens to the number you input. At line 10, the value you type in is taken by the Amstrad and stored in a 'pigeon hole' in memory, which is given the label 'number':

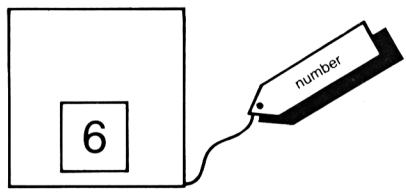

Figure 6 Storing an input value of 6 in memory.

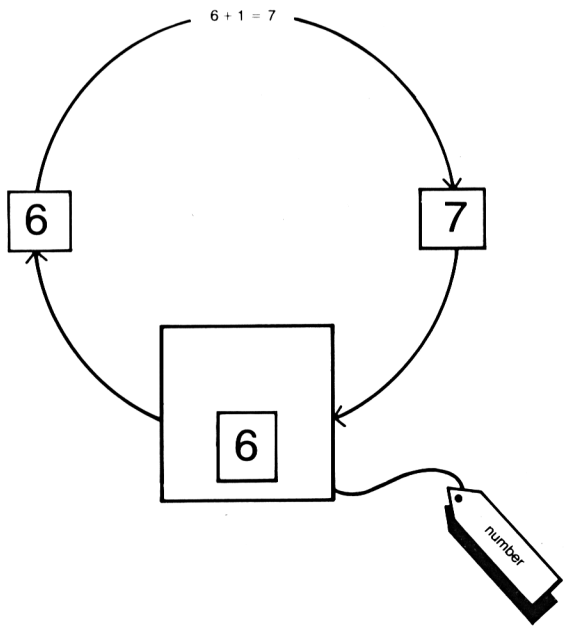

Figure 7 Number=number+1 — what it means to the Amstrad.

In line 20, you are telling the Amstrad:

> Take the value from 'number', add 1 to it, and put the result back in 'number'.

You can see the same effect here:

```
new[ENTER]
auto[ENTER]
10 input "What are your savings";savings[ENTER]
20 input "How much have you spent recently";spent[ENTER]
30 savings=savings-spent[ENTER]
40 ? "Now your savings are ";savings[ENTER]
50 [ESC]
```

Suppose your savings came to £60 and you spent £20:

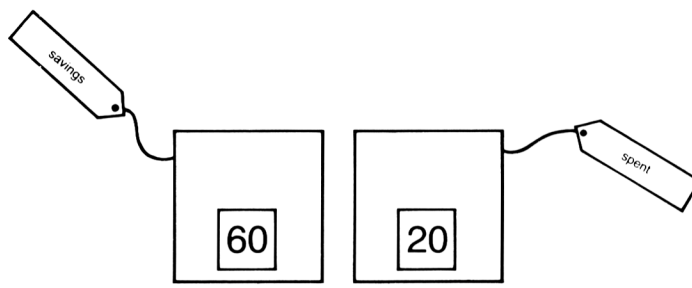

Figure 8 Savings and expenditure.

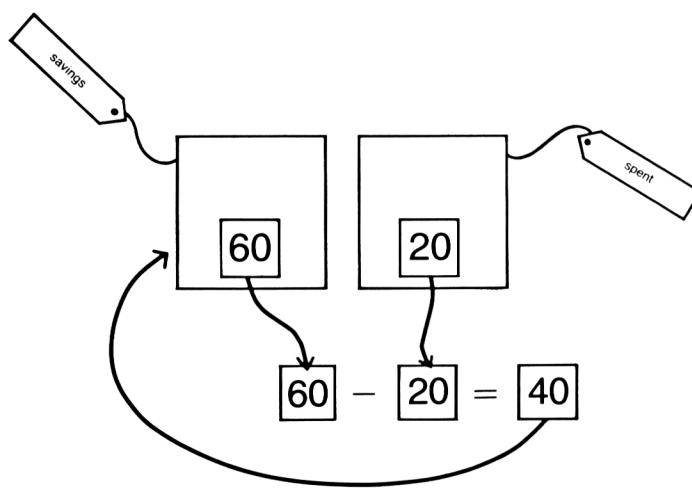

Figure 9 Savings and expenditure.

Line 30 causes the Amstrad to take the value from 'savings' (60), subtract 'spent' (20), leaving 40, which is then stored back in 'savings':

It is this value which is printed out in line 40. The value stored in 'spent' is unaffected.

This capability of variables gives us an easy way of counting up to a certain value:

```
new[ENTER]
auto[ENTER]
10 input "Please type in a starting value. ",value[ENTER]
20 value=value+1[ENTER]
30 ? "It is now ";value[ENTER]
40 value=value+1[ENTER]
50 ? "It is now ";value[ENTER]
60 value=value+1[ENTER]
70 ? "It is now ";value[ENTER]
80 [ESC]
```

Does this repetition of lines seem rather tedious? There is a shorter way of writing this program. But you will have to wait until later in the book to find out how to do it!

Exercises

1) The fish-and-chip shop owner is now using the computer to add up his bills. He inputs the separate cost of 4 items and the computer adds the costs together to give a total. The VAT is calculated and the bill printed out as, for example:

Item 1	£0.95
Item 2	£1.20
Item 3	£0.35
Item 4	£0.50
Total	£3.00
VAT	£0.45
Overall cost	£3.45

There are still a lot of problems with this program. If there are only 2 items, 0 must be input for the cost of the remaining items. If there are more than 4 items, another bill must be printed. Sometimes the VAT works out as 0.4275, and the overall cost then

contains small fractions of a penny. The fish-and-chip shop owner (and you!) still have a lot to learn.

Saving and loading programs

You may well feel that you are now beginning to produce programs worthy of being saved for future use. If that's the case, then refer to the User Instructions, which explain the business of loading and saving programs very clearly.

Whether you own a cassette-based 464 or a disk-based 664 micro, make sure you clearly label each cassette or disk with the names of the programs it contains. It is very easy to forget where you saved the latest programming masterpiece — or even what it was called!

On with the programming!

These first two chapters have been an introduction to computing. You have learnt about RAM and ROM, and you have also learnt about variables, which are fundamental to computing. You have been introduced to a few Basic keywords like PRINT, LET, and INPUT, and written some programs of your own.

In the succeeding chapters the pace will quicken, so don't attempt to 'do' the entire book in a day. Don't miss anything out, as the programs in later chapters use Basic statements introduced in earlier ones.

From now on I will not remind you to press [ENTER] when inputting program lines, and it will be up to you to use AUTO, RENUM, and the editing facilities of the Amstrad to make your task easier.

Lastly, do remember that there are many different ways to write a program to carry out a particular task. There are no correct answers to the programming exercises in this book. Provided your program works, be satisfied. Later in the book we will be seeing how writing longer programs can be made easier by pre-program planning, but for the moment, go your own way and enjoy yourself.

Chapter Three

Drawing Pictures

The graphics screen

In Chapter 2 we saw that the Amstrad can position text on the screen by using *text coordinates*. The micro can also draw lines on the screen, but to do this the Amstrad has to know where the lines begin and end. The graphics screen is divided up into 640 points horizontally and 400 points vertically.

We can identify the position of any point on the screen by describing how far along and how far up the screen the point is.

The position of the point in the figure is (200,300). The first number is called the X coordinate of the point (how far along it is) and the second number is the Y coordinate (how far up it is).

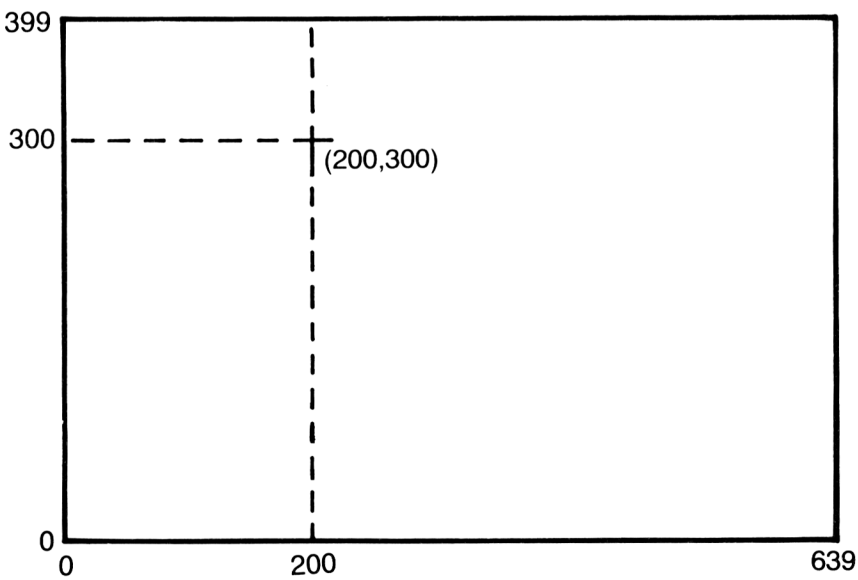

Figure 10 The graphics screen, showing the point (200,300)

Notice that these *graphics coordinates* are measured from the bottom of the screen, and that the *bottom* left-hand point on the screen has the coordinates (0,0). This can be a bit confusing, because text coordinates work in a completely different way, with the *top* left-hand character position having the coordinates (1,1)! Notice also that because the numbering of points begins with 0, the top right-hand point has the coordinates (639,399) and *not* (640,400) as you might imagine.

Run the following program:

```
10 MODE 1
20 MOVE 124,156
30 DRAW 300,300
40 DRAW 200,400
50 DRAW 124,156
```

When we use the graphics commands on the Amstrad, we are using the *graphics cursor* to draw lines on the screen. Normally the graphics and text cursor remain together, but as soon as we use a graphics command the invisible graphics cursor is used.

The MOVE command in line 20 causes the graphics cursor to move invisibly to the point (124,156). The DRAW command makes the cursor move from its position at (124,156) to the new coordinates (300,300) drawing a line between the two points. The remaining DRAW commands in lines 40 and 50 draw the two other sides of a triangle.

In general terms, we can say that MOVE x,y causes the graphics cursor to move to the point x,y without drawing a line. DRAW x,y causes a line to be drawn from the last point visited by a MOVE or a DRAW to the point x,y.

The different modes

Although the graphics screen is divided into 640 horizontal and 400 vertical points, the Amstrad cannot really tell all these points apart. We saw in Chapter 1 that there are 3 screen modes, each with a different-sized text screen. The graphics screen is the same for all the modes, but in some of the screen modes the Amstrad is better able to tell points apart than others. Run the above program again after you have edited line 10 to be:

```
10 MODE 0
```

The drawing remains the same, but the lines are much thicker and the picture looks more 'chunky'. Now try:

```
10 MODE 2
```

This time the lines are very fine. Mode 2 is called the *high resolution mode*, because when using mode 2 the Amstrad can distinguish between 640 points horizontally and 200 points vertically, which results in very fine lines when you use DRAW.

In mode 2 the Amstrad cannot tell the difference between points that are vertically too close. It would treat the points (10,10) and (10,11) as exactly the same. In fact both mode 1 and mode 0 have the same vertical resolution of 200 points as mode 2, but their horizontal resolutions are much worse. Type:

```
10 MODE 1
```

and run the program again. Mode 1 is the *medium resolution mode*, and in mode 1 the Amstrad can only show 320 separate horizontal points. This means that, for example, (200,300) (200,301) (201,300) and (201,301) are all treated as the same point. Now type:

```
10 MODE 0
```

and run the program for the third time. Mode 0 is the *low resolution mode*, and can only identify 160 different horizontal points.

You may wonder why on earth anyone would choose to use a screen mode that produces 'chunky' drawings when the high resolution mode 2 is available. The main reason is that although mode 0 is low resolution, it can display drawings in up to 16 different colours on the screen at the same time. Modes 1 and 2 are much worse:

	Graphics resolution	Number of colours on-screen simultaneously
Mode 0	160 × 200	2
Mode 1	320 × 200	4
Mode 2	640 × 200	16

Figure 11 The different graphics resolutions and number of colours available in the different modes.

You mustn't forget that the Amstrad has only a limited amount of memory. It can only record a certain amount of information about

the screen in the RAM. As with many things in computing, there is a trade-off here. The RAM can be used to record details of many points of two possible colours, fewer points of 4 possible colours, or very few points with 16 possible colours. The Amstrad gives you the choice and you must select the mode which seems best suited to your purposes.

Exercises

1) Try to write a brief program using MOVE and DRAW commands to draw a rectangle on the screen.
2) Using the MOVE and DRAW statements write a program that draws this picture of a robot head:

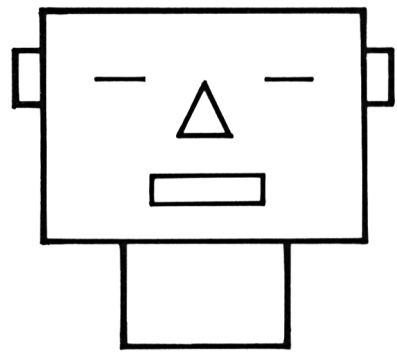

Figure 12 Robot head.

3) Write a program to draw the following shape:

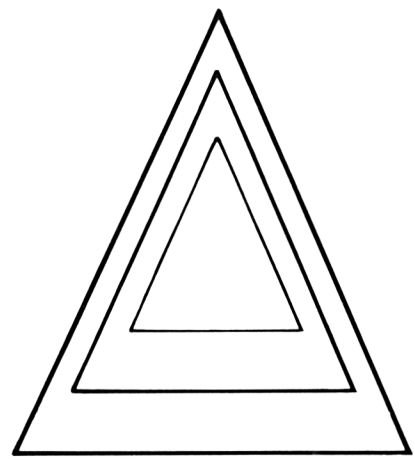

Figure 13 Triangles.

The PLOT statement

The Amstrad can also display individual points on the screen, although a single point is rather difficult to see in mode 2! This program plots 6 separate points on the screen:

```
10 MODE 0
20 PLOT 160,200
30 PLOT 320,200
40 PLOT 324,200
50 PLOT 328,200
60 PLOT 332,200
70 PLOT 480,200
```

In mode 0, the resolution is so low that the four points plotted in lines 30 to 60 merge to form a line. In mode 1 all the points can be seen, and the points are so fine in mode 2 that you may not be able to see them at all.

PLOT works in the same way as MOVE or DRAW. The PLOT command must be followed by the x and y coordinates of the point to be plotted. In fact the 'point' is really made up of a number of points, because none of the modes is accurate enough to identify every point on the screen. The smallest 'block' of points on screen that can be located in the different modes is called a *pixel*:

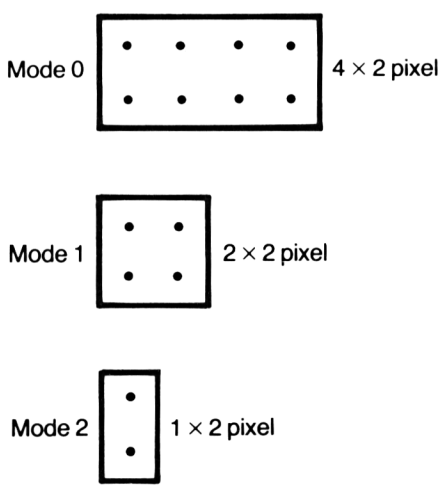

Figure 14 The size of a graphics pixel in each of the modes.

Variables and graphics programs

This program draws the outline of a house, with a roof:

```
10 MODE 1
20 REM draw a rectangle as the house fr
30 MOVE 120,100
40 DRAW 500,100
50 DRAW 500,300
60 DRAW 120,300
70 DRAW 120,100
80 MOVE 120,300
90 DRAW 185,380
100 DRAW 435,380
110 DRAW 500,300
```

The above program introduces the REM statement. This is short for REMark, and is included to make the program easier to follow. Now that the programs are getting longer, I will include REM statements to explain the different parts. The Amstrad does nothing when it finds a REM statement — REMs are included purely as notes for the programmer's benefit.

Longer programs pose another problem because when they are listed they will not fit completely onto the screen. You can list any part of a program by typing, for example:

```
list 10–30
```

which lists lines starting with line 10 and going up to and including line 30.

We can make the program much more interesting by using variables, whose values can be changed easily:

```
10 MODE 1
20 REM set up coordinates for house front
30 houseleftx=120
40 houseboty=100
50 houserightx=500
60 housetopy=300
70 REM draw the house front
80 MOVE houseleftx,houseboty
```

```
90 DRAW houserightx,houseboty
100 DRAW houserightx,housetopy
110 DRAW houseleftx,housetopy
120 DRAW houseleftx,houseboty
130 REM set up coordinates for roof
140 roofleftx=185
150 rooftopy=380
160 roofrightx=435
170 REM draw the roof
180 MOVE houseleftx,housetopy
190 DRAW roofleftx,rooftopy
200 DRAW roofrightx,rooftopy
210 DRAW houserightx,housetopy
```

Notice that because many of the coordinates are used twice, we do not need as many variables as you might think. You can now make the house lower just by changing line 60 to give the house top a lower y coordinate, such as 200. Or you can raise the roof by increasing rooftopy in line 50. Once again this shows the power of variables. If the house was drawn using specific numbers you would need to change many lines to get the same effect. How can you make the house narrower? Why do you have to change two

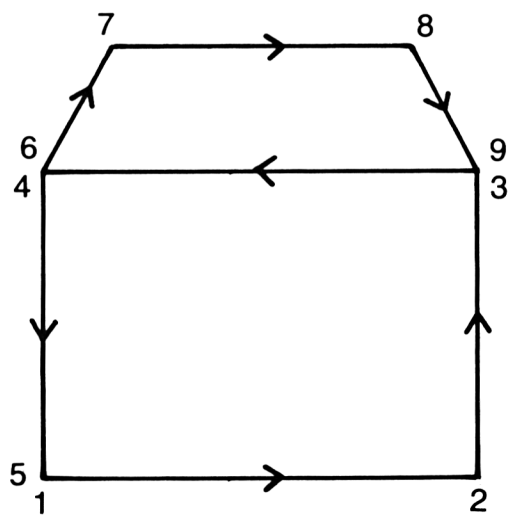

Figure 15 House.

variables rather than one? What does the Amstrad do if you have a coordinate which is too big, such as setting rooftopy to 500 in line 150?

When we use graphics to draw pictures, there are many ways of producing the same result. It is worth spending a bit of time looking at the shape you want to draw, because you can make the program a lot shorter if you visit all the 'corners' in your shape in the right order. The house was drawn by visiting the corners in the order shown in Figure 15.

Exercises

1) The program to draw the house took 9 MOVE and DRAW statements. Can you shorten this to 8 by drawing the lines in a more sensible order?
2) Write a program that draws this shape on-screen:

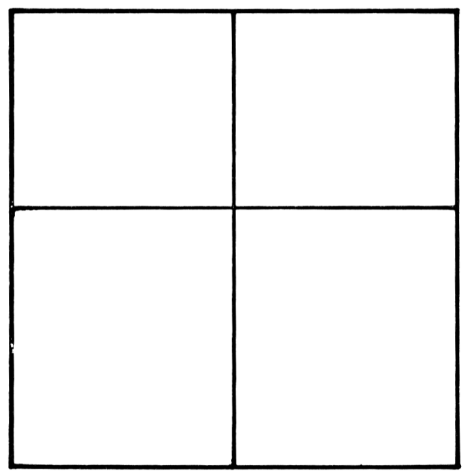

Figure 16 Window.

INPUT and graphics

Using the INPUT statement gives us a problem, as you can see if you delete lines 30 to 60 and put:

```
30 INPUT "What is the left x coordinate for
the house";houseleftx
```

```
40 INPUT "What is the bottom y coordinate for
 the house";houseboty
50 INPUT "What is the right x coordinate for
the house";houserightx
60 INPUT "What is the top y coordinate for th
e house";housetopy
```

The prompt and the input values make a mess of the screen display. We can avoid this by telling the Amstrad to print text only to a certain part of the screen. This is done by using the WINDOW statement:

```
10 MODE 1
20 REM set up coordinates for house front
25 WINDOW 1,40,20,25
30 INPUT "What is the left x coordinate for t
he house";houseleftx
40 INPUT "What is the bottom y coordinate for
 the house";houseboty
50 INPUT "What is the right x coordinate for
the house";houserightx
60 INPUT "What is the top y coordinate for th
e house";housetopy
70 REM draw the house front
80 MOVE houseleftx,houseboty
90 DRAW houserightx,houseboty
100 DRAW houserightx,housetopy
110 DRAW houseleftx,housetopy
120 DRAW houseleftx,houseboty
130 REM set up coordinates for roof
140 roofleftx=185
150 rooftopy=380
160 roofrightx=435
170 REM draw the roof
180 MOVE houseleftx,housetopy
190 DRAW roofleftx,rooftopy
200 DRAW roofrightx,rooftopy
210 DRAW houserightx,housetopy
```

The 'window' inside which the text is printed is identified by giving the left-most column to be within the window, followed by the right-most column, the top-most line, and the lowest line. Because we are talking about the position of *text* here, we must use *text* coordinates to describe the window:

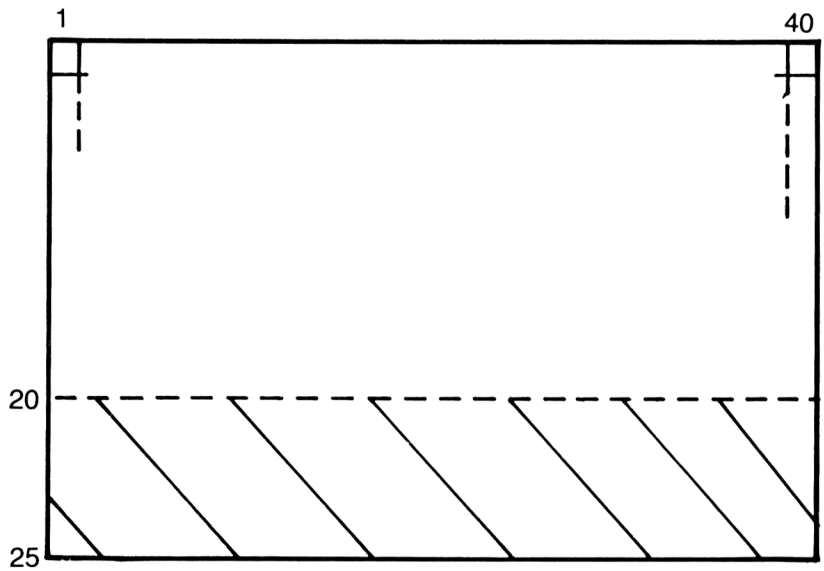

Figure 17 The shaded area contains the characters that would lie within the text window set up by WINDOW 1,40,20,25

The WINDOW continues to work even when the program has finished, as you can see if you list the program. You can return everything to normal by typing in a mode command. You might like to complete the program by changing lines 140 to 160 so that the roof coordinates are also input.

We could set the window to the top of the screen using:

```
25 WINDOW 1,40,1,2
```

Try to adjust the window so that it is to the right of the picture drawn by the program.

One thing you will have noticed is that choosing the wrong-size window can make the text a bit difficult to follow. The way to avoid this is to plan ahead and decide what portion of the screen you need for graphics and what for text, and adjust your INPUT prompts accordingly.

Note that setting a text window has no effect on the graphics screen. You can still draw lines through the text window if you have positioned the window in the wrong place, such as the middle of the screen.

Exercises

1) Write a program to enable you to input 3 sets of coordinates which the computer then uses to draw a triangle.
2) Write a program that accepts as input 4 sets of graphics co-ordinates and then draws an arrow on-screen.

Adding colour

When you switch the Amstrad on, the micro is set to print yellow text and graphics on a blue background in all the modes. In fact there are 27 different colours which can be displayed on the screen, although some of them are a bit difficult to tell apart (especially for those of you with green-screen monitors). Each colour has a number, called the INK number, and whenever we refer to a colour we use this number rather than the name of the colour itself: see Figure 18 overleaf.

At this stage it is important to realise that the computer does not actually use the whole of the screen while it is printing or doing graphics. You may have noticed already that the Amstrad actually works within a large rectangle around which there is a border of unused screen: see Figure 19 overleaf.

Although the Amstrad does not use this border, it is kept the same colour as the rest of the screen. The border is not really part of the computer memory, because it always stays the same colour and is never used by the Amstrad for printing or drawing graphics. Within the main screen, the Amstrad has to use a great deal of memory to record what letters are currently printed on the screen, what lines have been drawn, and what colours have been used. Although the screen looks as if it never changes, the Amstrad is actually redrawing it all many times a second. (The same redrawing process takes place with the pictures shown on an ordinary television.)

The border around the screen is a very different matter. As the Amstrad never uses the border, it need only record one piece of

Ink number	Colour
0	Black
1	Blue
2	Bright blue
3	Red
4	Magenta
5	Mauve
6	Bright red
7	Purple
8	Bright magenta
9	Green
10	Cyan
11	Sky blue
12	Yellow
13	White
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

Figure 18 The 27 INK colours that can be used on the Amstrad.

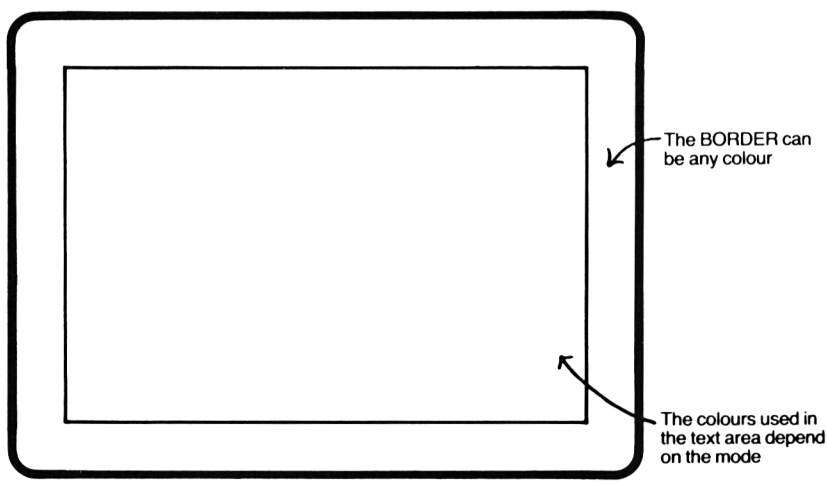

Figure 19 The BORDER area on your monitor or TV.

information about it — its colour. That is all the Amstrad needs to be able to redraw the border area. Within the rectangle the Amstrad uses much more memory, because it has to record information about every possible point. After all, the main screen area might contain hundreds of lines in different colours.

I have gone to some length to point out why the border is not the same as the rest of the screen. It is because of this difference that the border is treated in a different way when it comes to colours. The border can be set to *any* colour in *any* mode. There are never any restrictions on the colour of the border. Mode 2 can only display 2 colours at once *within* the main screen rectangle, but its border can be *any* colour. Type:

```
MODE 2
BORDER 0
```

If you refer back to Figure 18 you will see that 0 is the INK number for the colour black. By typing BORDER 0 you are telling the Amstrad to set a black border. Set the border to a few other colours – you can use any number from 0 to 26, so there are 27 possible border colours altogether. BORDER 26 gives you a white border for example.

The border can be set in modes 0 and 1 in exactly the same way. You will find that if you set a border and then change mode, the border remains set to its new colour. When the Amstrad is switched on or reset the border becomes blue, BORDER 1. You could add a BORDER command to any of the programs you have written so far. It won't affect the program in any way, except perhaps to make it more colourful.

PEN and PAPER colours

You can also change the colours used within the main screen rectangle. Here the question of RAM becomes important, and there are restrictions on the number of colours you can have simultaneously on the screen at any one time.

You can change the colour the Amstrad 'writes' with by using the PEN command. Type:

```
MODE 0
PEN 4
```

You might expect from Figure 18 that this will give you magenta

characters, but I did tell you that the colours in the border worked differently to those for the main screen! Choosing PEN 4 actually causes the Amstrad to print in white. You can think of PEN 4 as being filled with white ink. Typing:

```
PEN 5
```

chooses a pen full of black ink. You can even have:

```
PEN 14
```

which gives you flashing blue/yellow ink!

There are 16 pens available for use in any mode and Figure 20 shows the colour number for the INK that the pens use.

PEN or PAPER number	Mode 0	Mode 1	Mode 2
0	1	1	1
1	24	24	24
2	20	20	1
3	6	6	24
4	26	1	1
5	0	24	24
6	2	20	1
7	8	6	24
8	10	1	1
9	12	24	24
10	14	20	1
11	16	6	24
12	18	1	1
13	22	24	24
14	1/24	20	1
15	16/11	6	24

Figure 20 The PEN and PAPER colours for the different modes. In mode 0, choosing PEN or PAPER 14 gives a flashing colour which alternates between the two colours shown.

Note that the *same* pen can write with a *different* ink in another mode. This means that a program that works perfectly well in mode 0 may well give you a blank screen in mode 2! The pen you have chosen may have the same colour as the background in mode 2. As you can see, the 16 pens aren't much use in mode 2, because 8 of them write in yellow and the other 8 in blue. We will

see later how to change the inks that can be used in each mode, but for the moment switch to mode 0, and type:

```
PEN 11
```

to see the pink ink in action. One frequent problem when playing around with the pen colours is that you can end up being unable to read anything on the screen, because you are using a pen with the same colour ink as the background. If you are confident of your typing ability, just change the pen number by typing in a new pen command in invisible characters. Otherwise you can *reset* the machine to its normal ink and text colours by holding down [CTRL] and [SHIFT] and pressing [ESC]. (*A word of warning*. If you do this it clears the memory and you will lose any program you have input.)

It is worth mentioning here that it is possible to write a program that for one reason or another the computer cannot finish. If you ever find that the micro seems to have got 'stuck' in a program, you can make the computer abandon the program by pressing [ESC] twice in succession. If you press [ESC] once the computer will pause and if you then press any key other than [ESC] the computer will continue with the program as if nothing has happened.

You can change the background colour as well by using the PAPER command. Reset the micro by holding down [CTRL] and [SHIFT] and pressing [ESC]. Then switch to mode 0. Type:

```
PAPER 3
```

and the next characters printed will be printed on a red background. You can change the whole of the inner screen area to this new colour by typing:

```
CLS
```

The Amstrad clears all of the main screen to the new paper colour.

PAPER in mode 0 comes in the same 16 colours as the pens. PEN 14 gave us flashing blue/yellow ink, and PAPER 14 gives a flashing blue/yellow background. Figure 20 can be used to help you select both the pen and paper colours. For example, to get red characters on a white background in mode 0, type:

```
INK 3
PAPER 4
CLS
```

PEN and PAPER commands can, of course, also be used in programs:

```
10 MODE 0
20 LOCATE 4,7
30 PEN 3
40 PAPER 5
50 PRINT "Red on black"
60 LOCATE 4,13
70 PEN 6
80 PAPER 3
90 PRINT "Blue on red"
100 LOCATE 4,19
110 PEN 5
120 PAPER 6
130 PRINT "Black on blue"
140 REM turn pen and paper back to normal
150 PEN 1
160 PAPER 0
```

If you run this program in the other two modes by changing line 10, you'll find you get some funny results, because the PENs contain different INKs in the other modes. You can also use variables:

```
10 MODE 0
20 redpeninmode0=3
30 blackpaperinmode0=5
40 PEN redpeninmode0
50 PAPER blackpaperinmode0
60 CLS
70 LOCATE 8,12
80 PRINT "Done!"
90 REM turn pen and paper back to normal
100 PEN 1
110 PAPER 0
```

The last two lines restore normal PEN and PAPER colours so you are not left with some unreadable mixture like yellow on white.

Exercises

1) Change the program on page 60 so that it prints the three words in green on white, red on yellow, and white on black. Make the border cyan.
2) You can even print every letter of a word in a different colour. Use the LOCATE statement to move to the right place and then change pens before printing each letter. Print the word RAINBOW in mode 0, with every letter a different colour.
3) Write a program that allows you to input the PEN and PAPER numbers to be used, and then clears the screen to the new colour and prints "The new colours" in the centre. (You must input the PEN and PAPER numbers to numeric variables and not string variables, as the Amstrad is expecting a number to go with both PEN and PAPER statements.)

Graphics and colour

Drawing pictures in colour is remarkably easy on the Amstrad. You have already used the commands MOVE and DRAW in programs. These commands, used on their own, always result in lines drawn using PEN 1 for whatever mode you are in. Thus all the graphics programs we have looked at so far have produced lines drawn with PEN 1. PEN 1 contains INK number 24 in all modes, so the lines have all been bright yellow (see Figures 18 and 20 if you're not sure about this).

If you want to have a different colour when drawing a line, you must use an extension of the DRAW command. Reset the Amstrad, and type:

```
MOVE 100,100
DRAW 300,300,2
```

The Amstrad draws a line from (100,100) to (300,300) using PEN 2, which contains INK number 20, bright cyan, in mode 1. Type:

```
DRAW 400,0,3
```

and a red line is drawn with PEN 3 from (300,300) to (400,0). PEN 3 uses INK number 6, red, in mode 1. (You may find this line difficult to see against the blue background.)

The commands are just as easy to use in a program. Again, remember that a program that works in one mode may not work

in another because of the different INKs the PENs have in different modes. This program draws a rectangle in mode 1, with one side in yellow, one in cyan, and the other two in red:

```
10 MODE 1
20 MOVE 100,100
30 DRAW 400,100
40 DRAW 400,300,3
50 DRAW 100,300,2
60 DRAW 100,100,3
```

Notice that at line 30 no PEN is specified, so the Amstrad automatically uses PEN 1, which draws a yellow line. After you have run the program once, run it again. You may be surprised to find that there is no longer a yellow line!

If no PEN number is given, as in line 30, the Amstrad uses PEN 1 only if this is the *first* draw command it has obeyed. Otherwise, the PEN used is the same as the one used in the *last* draw command. After you run the program the first time, the last PEN used is PEN 3. When the Amstrad runs the program the second time, it has just used PEN 3 in its last DRAW command, so it uses PEN 3 again at line 30 where no PEN is specified.

The advantage of this is that once you have set the PEN in a draw command, all lines drawn after that are automatically drawn in that same colour unless you introduce a new PEN number:

```
10 MODE 1
20 MOVE 200,100
30 DRAW 400,200,2
40 DRAW 100,350
50 DRAW 200,100
```

You might like to try running this program in mode 2, where PEN 2 holds a different colour INK.

CPC 664 owners should note that they have an alternative command which can be used to set the colour in which lines are drawn, the GRAPHICS PEN command. On the 664, the following two lines could be used in place of line 30:

```
25 graphics pen 2
30 draw 400, 200
```

Mode 0 is by far the best mode to use if you want a colourful graphics display and you are not too concerned about the resolution:

```
10 REM draws a flag in 10 different colo
20 MODE 0
30 PAPER 1
40 CLS
50 REM set up flagpole coordinates
60 poleleftx=100
70 polerightx=120
80 poleboty=50
90 poletopy=350
100 knobleftx=80
110 knobrightx=140
120 knobtopy=370
130 REM draw flagpole
140 MOVE poleleftx,poleboty
150 DRAW poleleftx,poletopy,0
160 DRAW knobleftx,knobtopy
170 DRAW knobrightx,knobtopy
180 DRAW polerightx,poletopy
190 DRAW polerightx,poleboty
200 REM setup flag coordinates
210 flagrightx=500
220 flagtopy=240
230 MOVE polerightx,flagtopy
240 DRAW flagrightx,flagtopy,2
250 DRAW flagrightx,flagtopy-100,2
260 MOVE flagrightx,flagtopy
270 flagtopy =flagtopy-10
280 DRAW polerightx,flagtopy,3
290 flagtopy =flagtopy-10
300 DRAW flagrightx,flagtopy,4
310 flagtopy =flagtopy-10
320 DRAW polerightx,flagtopy,5
330 flagtopy =flagtopy-10
```

```
340 DRAW flagrightx,flagtopy,6
350 flagtopy =flagtopy-10
360 DRAW polerightx,flagtopy,7
370 flagtopy =flagtopy-10
380 DRAW flagrightx,flagtopy,8
390 flagtopy =flagtopy-10
400 DRAW polerightx,flagtopy,9
410 flagtopy =flagtopy-10
420 DRAW flagrightx,flagtopy,10
430 flagtopy =flagtopy-10
440 DRAW polerightx,flagtopy,11
450 flagtopy =flagtopy-10
460 DRAW flagrightx,flagtopy,12
470 REM turn paper back to normal
480 PAPER 0
```

You can lower the flag to half-mast just by changing line 220!

Exercises

1) Draw a picture of a door in green on a white background. Print the number 12 on the door in blue, and draw a letter box in black. Use mode 0.
2) Modify your previous program so that it draws the same scene in different colours in mode 1. (You will have to use different pens here, because the pens you have chosen for your first program may be filled with the same colour INK in mode 1).
3) Take the house program from earlier in the chapter and modify it so that you can input the colours of the lines to be used in the drawing.

Stained glass WINDOWs

We saw a little earlier in the chapter how useful a text window could be when we were using INPUT statements. The WINDOW statement has other more valuable functions when we are dealing with graphics. It is possible to set up more than one text window at once:

```
10 MODE 1
```

```
20 REM set up first window at top left
30 WINDOW 1,20,1,12
40 REM set up second window at bottom right
50 WINDOW #1,21,40,13,25
60 PRINT "This goes to the main text window"
70 PRINT #1,"And this goes to the other."
```

The 'main' text window is always chosen as the one for which all commands like PRINT and CLS are intended. If you now type:

```
? "hello"
```

this will be printed automatically to the main text window. If you type:

```
? #1, "hello"
```

the message will be printed within the other window. Each window has a number — the main window is number 0. In line 50 of the program, window number 1 is set up using WINDOW #1. A message is printed to this window in line 70 using PRINT #1. The main window can be printed to using either PRINT #0 or just PRINT. Try:

```
? #0, "This is window #0"
? "This is window #0, too"
```

The #0 is OPTIONAL when you are dealing with the main text window. If you don't mention a WINDOW number the Amstrad always assumes you are referring to WINDOW #0, the main window.

Before the windows send you cross-eyed, you can always get the normal screen back by doing a mode command. Try it now.

The clever thing about WINDOWs is that as well as printing to them, we can set the windows to use different PAPER and PEN colours. Add these lines to the program:

```
51 REM print to the main window
52 PAPER 3
54 CLS
56 PEN 1
60 PRINT "This goes to the main text window"
61 REM now print to the second window
```

```
62 PAPER #1,2
64 CLS #1
66 PEN #1,0
```

WINDOW #0, the main window, now prints in yellow text on a red background, while WINDOW #1 prints in blue text on a cyan background! When choosing the PEN and PAPER colours for a window, you must give the WINDOW number to which the commands apply. This is why all the statements in lines 62 to 70 contain #1, like PEN #1,0 at line 66. This chooses pen 0, which holds blue ink, to be used for printing in WINDOW #1 ONLY. Again, the #0 for the main window is *optional* for the PEN and PAPER colours. Lines 52 to 56 could be written:

```
52 PAPER #0,3
54 CLS #0
56 PEN #0,1
```

with exactly the same result when the program is run. All the commands you have met so far that deal in any way with text can also apply to text windows. Try:

```
CLS #1
```

and you will see WINDOW #1 cleared to cyan. Notice that although the CLS applies to WINDOW #1, the Amstrad still prints messages like 'Ready' to the main window, WINDOW #0. You can even list your program in WINDOW #1 with:

```
LIST #1
```

Type:

```
PEN #1,3
```

Nothing seems to have happened. We have changed the pen used in WINDOW #1 to PEN 3, filled with red ink. To see the effect of this change we must first print something to WINDOW #1, so type:

```
PRINT #1, "Printing in red at #1"
```

We could change the background with:

```
PAPER #1,1
```

and again nothing will appear to happen until something is printed to WINDOW #1, so type:

PRINT #1, "A bright yellow background."

and:

CLS #1

clears the whole of WINDOW #1 to the new PAPER 1 colour, which is bright yellow.

Exercises

1) Set up two text windows for the top and bottom half of the screen, and print your first name in the top window and your surname in the lower.
2) Set up two text windows in mode 0, and print the message WINDOW #0 in the main window in green on a white background, and WINDOW #1 in the other window in black on a red background.

Graphics and windows

Windows can even be set to overlap:

```
10 MODE 1
20 REM set up main window
30 WINDOW 1,20,8,16
40 PAPER 3
50 CLS
60 REM set up second window to overlap
70 WINDOW #1,11,30,10,18
80 PAPER #1,2
90 CLS #1
```

Notice that WINDOW #1 is in front of WINDOW #0. When windows overlap, the window cleared last always overlaps the one cleared first. If you add:

100 CLS

and delete line 50, you will find that because WINDOW #1 is cleared first, it is covered by WINDOW #0. WINDOW #0 is not permanently in front, as you can see if you try CLS[ENTER] and CLS #1[ENTER] a few times.

As if overlapping windows are not mind-boggling enough, the Amstrad allows you to set up as many as 8 separate text windows,

numbered from 0 to 7!

```
10 MODE 0
20 WINDOW 1,20,23,25
30 INPUT "Give 4 pen colours (0-16)";pen1,pen2,pen3,
   pen4
40 INPUT "Now 4 paper colours";paper1,paper2,paper3,
   paper4
50 REM set the windows up, create PEN and PAPER
   colours for them
60 WINDOW #1,6,15,13,22
70 CLS #1
80 PEN #1,pen1
90 PAPER #1,paper1
100 CLS #1
110 REM move roughly to centre of window for printing
120 LOCATE #1,5,5
130 PRINT #1,"1"
140 WINDOW #2,1,8,7,16
150 PEN #2,pen2
160 PAPER #2,paper2
170 CLS #2
180 LOCATE #2,4,5
190 PRINT #2,"2"
200 WINDOW #3,7,14,1,10
210 PEN #3,pen3
220 PAPER #3,paper3
230 CLS #3
240 LOCATE #3,4,5
250 PRINT#3,"3"
260 WINDOW #4,13,20,6,15
270 PEN #4,pen4
280 PAPER #4,paper4
290 CLS #4
300 LOCATE #4,4,5
310 PRINT#4,"4"
```

The last program shows how valuable the WINDOW statement can be. It gives us a fast and easy way of filling a rectangular area

with a colour.

In line 120 the LOCATE command is used for the first time with a window. The text coordinates used in the LOCATE statement are based on the new window. The top left corner of each window is taken to have the coordinates (1,1), and each rectangle is about 7 characters wide and 9 lines deep, so the centre of each window is at (4,5).

This program demonstrates once again the advantages of variables, and reminds us that wherever we use numbers, we can make the program more flexible by using variables instead.

The graphics programs we looked at earlier suffered because there was no way to fill an area with a particular colour, although we could draw lines in different colours. We can now combine MOVE, DRAW and the WINDOW statements to produce some very effective displays. (Remember that the graphics screen is unaffected by any text windows set, and still covers the whole screen. So the coordinates used here are just as they would be if no text windows were set.)

```
10 MODE 0
11 REM create red roof
20 WINDOW 3,18,1,5
30 PAPER 3
40 CLS
41 REM create brown house front
50 WINDOW #1,3,18,6,20
60 PAPER #1,9
70 CLS #1
71 REM create green door
80 WINDOW #2,10,12,15,20
90 PAPER #2,12
100 CLS #2
101 REM create blue window
110 WINDOW #3,6,8,9,12
120 PAPER #3,6
130 CLS #3
131 REM create second blue window
140 WINDOW #4,14,17,9,12
150 PAPER #4,6
160 CLS #4
```

```
161 REM draw panes for window
170 MOVE 208,208
180 DRAW 208,272,1
190 MOVE 160,240
200 DRAW 256,240
201 REM draw panes for other window
210 MOVE 416,240
220 DRAW 544,240
230 MOVE 480,208
240 DRAW 480,272
241 REM draw outline within door
250 MOVE 296,88
260 DRAW 296,168,5
270 DRAW 372,168
280 DRAW 372,88
290 DRAW 296,88
```

One problem that does arise when WINDOWs and DRAWs are combined is that two different sets of coordinates are being used.

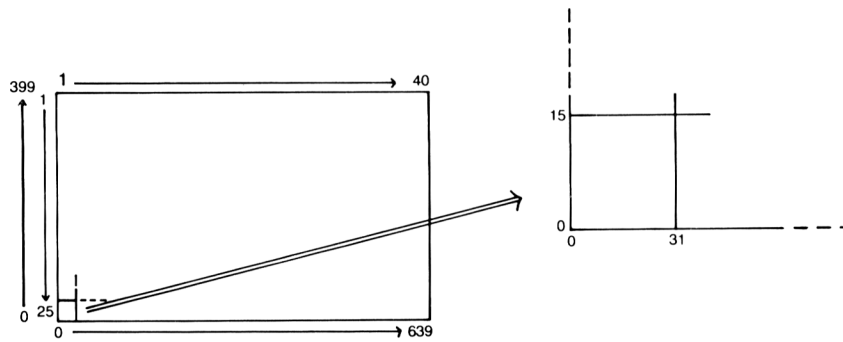

Figure 21 Each text coordinate in mode 0 is 32 points wide and 16 points high. The diagram shows the graphics points at the same position as the character position (1,25).

To draw accurately within a window we need to calculate at what graphic coordinates that text window starts. If we superimpose the graphics and text coordinates in mode 0, it is clear that each text character position is 32 points wide and 16 points high:

From this we can work out that the graphics coordinates for the bottom left corner of any character in mode 0 can be found from:

graphicsx = (textx − 1)*32
graphicsy = (25 − texty)*16

For example, the green door in the program above is created by a WINDOW statement at line 80, and the character at the bottom left of the door has the text coordinates (10,20). From this we can see that the extreme bottom left point within the door is:

graphicsx = (10 − 1)*32 = 9*32 = 288
graphicsy = (25 − 20)*16 = 5*16 = 80

or (280,80). Knowing that the bottom left of the door lies at (280,80) enables us to plan what coordinates the MOVE and DRAW commands need. For example the drawing of the outline within the door begins on line 250 with a MOVE to the point (296,88). This was chosen because we know (296,88) is just to the right and above the bottom corner of the door at (280,80).

If this all seems rather complicated, don't worry about it for the moment. Converting from text to graphics coordinates is only necessary if you intend to use both sorts of commands to produce a screen display. We shall see later that this conversion *is* important if you plan to draw any diagram or graph accompanied by text, but we shall also see that there are much simpler ways to do this than the one suggested above!

Filling areas with colour

Earlier in the chapter I stated that there is no way we can fill a particular area with colour. Although no command is available on the CPC 464 to allow colouring of graphics areas, the BASIC for the CPC 664 does cater for this with a new FILL command. This is remarkably rapid and very effective, as is demonstrated in this brief program:

```
10 MODE 1
20 MOVE 200,100
30 DRAW 400,100,2
40 DRAW 100,350
50 DRAW 200,100
60 MOVE 200,200
70 FILL 3
```

The FILL command fills an area of the screen beginning from the current graphics cursor position. Line 60 is necessary because otherwise the graphics cursor lies on the line it has just drawn, and FILL will not work under these circumstances. The number after the FILL specifies the PEN to be used when the area is coloured. As you can see by running the program, colour fill stops

as soon as it reaches a line drawn using the current graphics pen ink.

CPC 664 owners might like to modify some of the earlier programs so that they use the FILL command. A word of warning — failure to position the cursor correctly prior to the FILL command can result in the colour 'leaking out' through holes in a drawing. Sometimes this results in the whole screen being filled with a single colour!

Exercises

1) Set up 4 windows in different colours to produce this display:

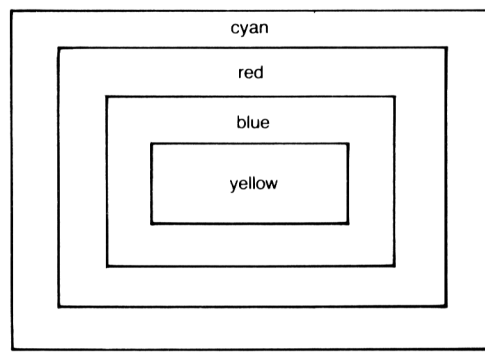

Figure 22 Colour windows.

2) Using mode 0, draw a picture of a red car with the windows in blue and the square 'tyres' in black with a white hubcap:

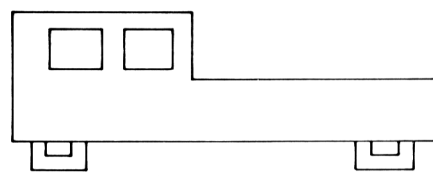

Figure 23 Car.

See if you can add more detail by using MOVE and DRAW commands.

3) Extend your previous program so that you can input the colour of the car and the character position at which the car is to be printed on screen.
4) Create a picture of a stick man waving from one of the windows of a house. Use the WINDOW, MOVE and DRAW statements.

Chapter Four

Loops

Over and over again

So far all the programs we have looked at have had one thing in common. When the program is run, the Amstrad obeys the instructions in line number order, from the lowest numbered line to the highest number. Yet you may already have noticed that in some circumstances this is a handicap. Quite often in a program we give a series of instructions which we would like to repeat.

Any program you write contains *sequences* of instructions which are obeyed in line number order. In this chapter you will find out how to include *repetitions* of instructions in your programs. A program containing repeated instructions is much more powerful than one which uses a simple sequence of instructions, as you will see if you run this program:

```
10 MODE 1
20 FOR count=1 TO 100
30 PRINT count
40 NEXT
```

Line 20 sets up a *loop* of instructions. The variable 'count' is used to count the number of times the loop will be carried out. In this case the loop begins with 'count' being set to 1. Line 30 prints out the value of 'count' — 1 the first time around the loop.

When the Amstrad gets to line 40 the NEXT statement tells it to add 1 on to 'count'. It will now do the next repetition of the statements within the loop, providing that 'count' has not reached the stopping value of 100 as given in line 20. 'Count' is known as the *control variable* for the loop, because its value controls when the loop stops.

All the statements between the FOR at line 20 and the NEXT at

line 40 are obeyed 100 times. This is clear if you add another statement between the two ends of the loop:

```
10 MODE 1
20 FOR count=1 TO 100
30 PRINT count
35 PRINT "Hello!"
40 NEXT
```

Each time around the loop the Amstrad prints the value of 'count' at line 30 and then prints 'Hello!' at line 40. If you change line 20 to:

```
10 MODE 1
20 FOR count = 5 TO 9
30 PRINT count
35 PRINT "Hello!"
40 NEXT
```

you can see that the loop now begins with 'count' being 5, and continues until 'count' is 9. Lines 30 and 40 are obeyed 5 times altogether before the loop ends. To make sure you've got the idea, edit line 20 so that the loop begins with 'count' at 3 and runs until its value is 15.

Let's look at a short program which shows the power of the loop:

```
10 MODE 1
20 PRINT
30 INPUT "What is the multiplication table";table
40 PRINT
50 FOR count = 1 TO 10
60 PRINT count;" times ";table;" is ";count*table
70 NEXT
```

This program prints out a multiplication table for *any* number you care to input at line 20. The control variable can always be used in calculations or statements within the loop:

```
10 MODE 2
20 FOR xcoordinate=1 TO 60
```

```
30 PRINT TAB(xcoordinate);"Tab within a loop"
40 NEXT
```

(The peculiar effect in this program occurs because the line at the top of the screen is scrolled off, and all the other lines are shifted up one as a new line is printed.) Similarly, the control variable might be used in a LOCATE statement:

```
10 MODE 1
20 FOR x = 10 TO 30
30 LOCATE x,8
40 PRINT "*";
41 LOCATE x,17
42 PRINT "*";
50 NEXT
60 FOR y = 9 TO 16
70 LOCATE 10,y
80 PRINT "*";SPC(19);"*"
90 NEXT
```

Not exciting enough for you? Try:

```
10 MODE 1
20 LOCATE 20,1
30 PRINT "*";
40 FOR y = 2 TO 10
50 LOCATE 21-y,y
60 PRINT "*";
70 LOCATE 19+y,y
80 PRINT "*";
90 NEXT
100 LOCATE 10,11
110 FOR count = 1 TO 21
120 PRINT "*";
130 NEXT
```

Or for a colourful example:

```
10 MODE 0
20 FOR colour=0 TO 15
```

```
30 PEN colour
40 PRINT "Amstrad"
50 NEXT
60 REM pen back to normal
70 PEN 1
```

Line 30 chooses a PEN with a different number every time through the loop, as the value of the variable 'colour' automatically increases by 1. Loops give especially impressive results in graphics programs. This one draws a succession of triangles inside one another:

```
10 MODE 1
20 leftx=100
30 boty=0
40 rightx=500
50 midx=300
60 topy=300
70 FOR count = 1 TO 40
80 MOVE leftx,boty
90 DRAW rightx,boty
100 DRAW midx,topy
110 DRAW leftx,boty
120 leftx = leftx+10
130 boty = boty+10
140 rightx = rightx-10
150 topy = topy -10
160 NEXT
```

Loops provide an easy means of keeping a running total:

```
10 MODE 1
20 totalspent = 0
30 FOR month = 1 TO 12
40 PRINT "What did you spend in month ";month;
50 INPUT spent
60 totalspent = totalspent+spent
70 NEXT
80 PRINT "You spent ";totalspent;" pounds altogether"
```

Notice that an INPUT statement with a prompt is not used at line 40. This is because the variable 'month' is included as part of the printout, and the Amstrad would have mistaken 'month' as the variable in which it should store the input value.

Exercises

1) Write a program to print your name until it fills the screen in mode 1.
2) Write a program to print a large letter 'M' in asterisks on the screen.
3) Extend the 'totalspent' program so that it also takes account of your savings in each month, and calculates the total savings for the year.
4) Write a program that draws a succession of pentagons inside each other. The colour of the pentagons and the rate at which they 'shrink' is input at the start of the program.
5) Write a program to print 'Amstrad' in yellow on the 16 different PAPER backgrounds in mode 0.

Controlling the loop

There is no reason why the start and stop values for a loop should not themselves be variables under your control. A loop gives us an easy way of generating a series of WINDOWs:

```
10 MODE 0
11 WINDOW 1,20,20,25
20 INPUT "How many windows";numberofwindows
30 FOR  count = 1 TO numberofwindows
40 WINDOW #count,count,count+5,count,count+4
50 PAPER #count,count
60 CLS #count
70 NEXT
```

Line 30 produces the WINDOWs. You might like to extend the program so that the windows drawn at line 30 are of different sizes. Here is another example where the stop value for the loop is input:

```
10 MODE 1
20 INPUT "How many lines do you want";numbero
flines
30 FOR count=1 TO numberoflines
40 MOVE count*10,0
50 DRAW 320,350
60 NEXT
```

You can use any number you like at 20, but anything above a few hundred means a long wait while the program finishes! I chose a random position in line 50, although you could even make that dependent on the control variable. Try an input value of 50 with this program:

```
10 MODE 1
20 INPUT "How many lines do you want";numbero
flines
30 FOR count=1 TO numberoflines
40 MOVE count*10,0
50 DRAW 500,count*10
60 NEXT
```

When we are dealing with graphics coordinates, a loop with a control variable which changes in steps of one is not very useful. In the above program, the control variable had to be multiplied by 10 to ensure that the lines drawn were visibly separated on-screen. If you edit lines 40 and 50 so that 'count* 10' becomes 'count', and run the program again, you will find that the lines merge together because they are too close.

We can change the rate at which the control variable increases by including a STEP value at the start of the loop:

```
10 REM draws 9 rectangles
20 MODE 0
30 FOR count=0 TO 500 STEP 60
40 MOVE count,0
50 DRAW count+50,0
```

```
60 DRAW count+50,200
70 DRAW count,200
80 DRAW count,0
90 NEXT
```

Line 30 gives 'count' the starting value of 0, and at the end of the loop at line 90 that value is increased not by 1 but by the STEP value of 60. A point to note is that the stopping value for the loop does not have to be an exact number of STEPs above the starting value. 'Count' is first 0, then 60, 120, 180, 240, 300, 360, 420, 480, and the loop then ends because 'count' becomes 540 which is greater than the stopping value of 500. The STEP value itself can be a variable. Add an input statement to the above program so that you can select the STEP, and experiment with different STEPs to see the results.

This program is similar, but produces more interesting results. It draws a series of (possibly) overlapping squares. Try input values of 5 and 50, for example:

```
10 REM draws squares of different sizes
20 MODE 1
30 INPUT"Distance between squares"; distance
40 INPUT"Length of side for square";length
50 FOR count=0 TO 400 STEP distance
60 MOVE count,count
70 DRAW count+length,count
80 DRAW count+length,count+length
90 DRAW count,count+length
100 DRAW count,count
110 NEXT
```

Exercises

1) Write a program to draw a rectangular grid on the screen. (You will need two loops, one which first draws the parallel vertical lines, and the other which draws the horizontal ones.)

2) Write a program that accepts as input the sales figures for a number of months, and then draws a bar chart, with a bar to show each month's sales in different colours. Use mode 0. (You can either draw the bars using DRAW commands to give an outline only, or use text WINDOWs in which case your bars cannot be so accurately drawn.)
3) The fish-and-chip shop owner of Chapter 2 has now learnt about loops, and wants to write a program which accepts as input the number of items bought, and the cost of each item. It then calculates the total cost and prints out this cost, the 15% VAT, and the total of the two. Write the program to help him out.
4) Write a program that accepts as input the text coordinates of a box and the character to be used when drawing the sides, and then draw the box. You can use the program that draws a rectangular box of asterisks as a basis. Add colour if you wish.
5) Write a program that allows you to print a triangle of asterisks anywhere on the screen in a colour that is input.

Different STEPS

The STEP used in a FOR . . . NEXT loop can be negative and can even be decimal. If negative steps are used, the start value in the FOR . . . NEXT loop must be higher than the stopping value:

```
10 MODE 1
20 FOR countdown=10 TO 1 STEP-1
30 PRINT countdown
40 NEXT
50 PRINT "BLASTOFF!!!!!!!!!!!"
```

Decimal steps are used a lot in graphics programs where the figure being drawn is not a simple triangle or rectangle, but a polygon (many-sided figure). The coordinates for the corners of the figure are easily calculated using sines and cosines, which have values less than 1.

The following program enables you to draw any polygon, from a triangle up to a circle. You can choose where the figure is centred, its 'radius' (i.e. the distance from its centre to its corners), and the number of sides the figure will have. This program is extremely useful, although if you are not familiar with sines and cosines you may find it difficult to follow. The variable

PI is always necessary for these sorts of calculations, and so this has been built into the Amstrad and does not need to be given a value in the program:

```
10 MODE 1
20 INPUT "Radius of figure"; radius
30 INPUT "X and Y coordinates for centre"; centrex,
   centrey
40 INPUT "Number of sides"; sides
50 CLS
60 stepsize=2*PI/sides
70 MOVE centrex,centrey+radius
80 FOR angle = 0 TO 2*PI STEP stepsize
90 DRAW centrex+radius*SIN(angle),centrey+radius*
   COS(angle)
100 NEXT
110 DRAW centrex,centrey+radius
```

Exercises

1) Write a BLASTOFF!!! program in mode 0, with the countdown numbers being printed within successively smaller bars as the countdown progresses.
2) Extend the polygon program so that the Amstrad draws 2 different-coloured polygons, one inside the other.

READ and DATA

Suppose we wish to write a program that takes the names and examination marks for a succession of schoolchildren, and calculates and prints out the total mark as a percentage. We may well be looking at the marks for 100 different children, and it is obviously not going to be an easy job inputting 100 names and their associated marks! When we need to input a lot of data to enable a program to run, there is a much better way of doing it than using INPUT statements or setting a large number of variables to the chosen values. We can instead READ the information from DATA statements:

```
10 MODE 1
20 FOR pupil = 1 TO 5
```

```
30 READ pupilname$,mark1,mark2
40 mark1percent=mark1/60*100
50 mark2percent=mark2/80*100
60 average=(mark1percent+mark2percent)/2
70 PRINT pupilname$;mark1percent;mark2percent;average
80 NEXT
90 DATA Arkwright,40,20,Busby,10,16
100 DATA Cuthbert
110 DATA 7,34,Jones,55
120 DATA 76,Knight,58,71
```

The output from this program is a real shambles, but we shall sort that out in a moment! Let's concentrate on the new features introduced in the program. At line 30 there is a READ statement. This tells the Amstrad to search through the program from the start until it finds a line beginning with a DATA statement. The first DATA statement is at line 90. The Amstrad now reads the first item after the word DATA, and stores that item in the variable given at line 30. So 'Arkwright' is stored in the variable 'pupilname$'.

But there are two other variables given in line 30. The Amstrad carries on reading after 'Arkwright', and stores 40 in the variable 'mark1' and 20 in the variable 'mark2'. It then uses these variables to calculate the average percentage mark for the child (mark1 is a mark out of 60, and mark2 is out of 80), and prints out the results at line 70.

Line 80 is a NEXT, so the computer returns to the start of the loop at line 20 to repeat the instructions again. This time line 30 is treated a little differently. The Amstrad keeps track of how many items it has READ from a DATA statement, and it now reads the next item it finds — in this case 'Busby' — and stores that in 'pupilname$'.

The whole process is repeated until all the names and marks have been read in and the loop has been completed. If the Amstrad comes to the end of a DATA statement, it just searches for the next data statement and carries on reading there. The items in the DATA statements must be separated by commas, but the program gives exactly the same results if items are kept in the same order but split over the DATA lines in a more sensible way:

```
90 DATA Arkwright,40,40
95 DATA Busby,10,16
100 DATA Cuthbert,7,34
110 DATA Jones,55,76
120 DATA Knight,58,71
```

It's much easier to spot and correct errors if the DATA lines are kept fairly short. The DATA lines can be placed anywhere in the program. The computer will ignore the lines until it needs to find them because of a READ statement. You can even place the lines right at the start of the program — try it for yourself. It's best to keep the DATA lines together because it makes them easier to locate and change, so you should aim to collect your DATA lines at the start or end of the program.

Sometimes the data need to be used again later in the program. For example, in the exam-marks program the pupils' names and marks might be printed out twice at different stages of the program. It would be pointless to duplicate all the data, because the Amstrad can be told to begin reading the DATA from any particular data line by using the RESTORE statement. If you add this line to the exam-marks program:

```
25 RESTORE 95
```

and run it again, you will find that only Busby's marks are printed out! You have told the Amstrad to READ the DATA beginning at line 95, and every time the computer begins the loop again, it is forced to start reading the data at the same place as before. If you delete 25 and put:

```
15 RESTORE 100
```

the Amstrad goes into the loop at line 20 and reads the DATA from line 100 the first time through the loop, the data from 110 the next time, then the data from 120, and on the next occasion there is no data line left to read and the Amstrad gives an error message.

Using RESTORE without a line number automatically causes the computer to start reading data at the first data line. This program uses RESTORE to draw a series of stick men to different places on the screen.

```
10 MODE 1
20 FOR count = 100 TO 500 STEP 100
30 RESTORE
40 READ legx,legy,groinx,groiny,legx1,legy1
50 READ armx,army,midx,midy,armx1,army1
60 READ headx,heady,earx,eary,earx1,eary1
70 MOVE legx+count,legy:DRAW groinx+count,gro
iny
80 MOVE legx1+count,legy1:DRAW groinx+count,g
roiny
90 DRAW midx+count,midy:DRAW armx+count,army
100 MOVE armx1+count,army1:DRAW midx+count,mi
dy
110 DRAW headx+count,heady:DRAW earx+count,ea
ry:DRAW earx1+count,eary:DRAW headx+count,hea
dy
120 NEXT
130 DATA 35,105,70,150,90,110
140 DATA 60,170,80,180,95,160
150 DATA 80,190,65,205,85,210
```

Tidying-up arithmetic

Returning to the program on page 82, there are several ways in which the program could be improved. First, it's clear that we need some way to round off the marks. There are several 'rounding' commands on the Amstrad. We could use INT, which rounds off numbers to the nearest smaller integer (whole number):

```
40 mark1percent=INT(mark1/60*100)
50 mark2percent=INT(mark2/80*100)
60 average=INT((mark1percent+mark2percent)/2)
```

Notice the value to be rounded is bracketed after the INT. Using INT for exam marks seems a bit unfair, because now all the marks are rounded down. A better expression to use is CINT, which converts numbers to the NEAREST integer. Thus 3.4 would be rounded down, and 3.5 rounded up:

```
40 mark1percent=CINT(mark1/60*100)
50 mark2percent=CINT(mark2/80*100)
60 average=CINT((mark1percent+mark2percent)/2)
```

Exercises

1) Improve the exam marks program by tidying-up the printout. Use PRINT TAB to organise the marks into columns, and give each column a heading.
2) Take your sales figures program from the previous exercise and modify it so that the sales figures are read from DATA statements before the bar chart is drawn.
3) Write a program that reads in the text coordinates for 7 windows and their paper colours, and sets up the windows on-screen.
4) Write a program that reads in the x and y coordinates and the PEN colour to be used to draw a series of lines which build up to make a picture of a ship.

Random numbers

Now that you have met INT and CINT, it is worth looking at a function which is immensely useful in games — the RND statement which generates a random number. Type:

```
? rnd
```

The Amstrad prints a random number greater than 0 but less than 1. If you type the same thing again, you will get a completely different random number. Unfortunately random numbers less than 1 aren't much use in games. If we write a program that 'throws' two dice, we want the computer to produce random numbers from 1 to 6 for the 2 dice, so we have to play about with RND and INT to get the range of numbers we want. In general, this is how we convert the random number from having a value between 0 and 1 to having a value in the range we want:

1) Generate the random number.
2) Take the lowest number of the range from the highest and add 1.
3) Multiply the random number by this difference.

4) Add the lowest number of the range to this result.
5) Use INT to round off the value.

If this seems hard to follow, don't be too concerned! Provided you know *how* to get random numbers, there's no need to worry about the way it works. This program demonstrates the process in reality:

```
10 MODE 1
20 INPUT "What is the lowest random number";lowest
30 INPUT "What is the highest random number";highest
31 REM print 20 random numbers in this range
40 FOR count = 1 TO 20
50 randomnumber=INT(RND*(highest-lowest+1)+lowest)
60 PRINT randomnumber;
70 NEXT
```

Random numbers are the basis for countless games. Here's a program that rolls those two dice:

```
10 MODE 1
20 INPUT "How many times shall I throw the 2 dice";thr
30 FOR count = 1 TO throws
40 dice1=INT(RND*6+1)
50 dice2=INT(RND*6+1)
60 dicetotal=dice1+dice2
70 PRINT "Throw";count;"gave";dicetotal
80 NEXT
```

We can also have great fun with graphics. This program draws a random number of triangles in random positions on the screen:

```
10 MODE 0
20 numberoftriangles=INT(RND*100+1)
30 FOR count = 1 TO numberoftriangles
40 leftx=INT(RND*640)
50 rightx=INT(RND*640)
60 midx=INT(RND*640)
70 lefty=INT(RND*400)
```

```
80 righty=INT(RND*400)
90 midy=INT(RND*400)
100 pencolour=INT(RND*16)
110 MOVE leftx,lefty
120 DRAW rightx,righty,pencolour
130 DRAW midx,midy
140 DRAW leftx,lefty
150 NEXT
```

Note that because the lowest number in our range is the graphics coordinate 0, there is nothing to add at stage 4) of our calculation, and the random coordinates are produced by a slightly shorter expression in lines 40 to 90.

Exercises

1) Write a program to print out 100 random numbers from 1 to 10 on the screen. You should have a fair idea if your program works, because you should be able to find at least one occurrence of each number in the output.
2) Extend the random triangles program so that random rectangles or some other shape are also drawn on the screen. You might like to add WINDOWs, but don't forget that these use text coordinates and will require numbers in a different range to the graphics statements.
3) Use random numbers in the LOCATE statement to print 100 asterisks in random colours in mode 0.

Random maths

Another popular use of random numbers is to generate arithmetic problems, in this case on the multiplication tables up to 12:

```
10 MODE 1
20 FOR questions = 1 TO 10
30 number1=INT(RND*12+1)
40 number2=INT(RND*12+1)
50 PRINT "What is";number1;"times";number2;
60 INPUT answer
70 NEXT
```

It would obviously be better if the computer could tell us if the answer given is right or wrong. We might decide to repeat the question if the answer is wrong, to give the user another chance. Repetition — sounds like an opportunity to use a loop, doesn't it? Unfortunately, the sort of loop we've met so far is useless in these circumstances. The FOR . . . NEXT loop always ends as the result of a *count*. The Amstrad counts the number of times the loop has been obeyed, and when this count reaches the stop value for the loop, the loop ends.

However, there are many occasions when we don't know beforehand how many times a loop may have to be repeated. Repeating a question which has been answered wrongly is one of them. It would be ludicrous to expect anyone to know beforehand how many times a question will have to be repeated before they get it right. We need something different here . . . a loop, but not a FOR . . . NEXT loop.

The WHILE . . . WEND loop

The WHILE . . . WEND loop is the answer to our problem. The difference between this and the FOR . . . NEXT loop is that FOR . . . NEXT always terminates as the result of a count, but WHILE . . . WEND terminates as the result of a *condition* being satisfied:

WHILE the answer is wrong
ask the question again
WEND

Let's look at a short program which asks a single multiplication question repeatedly until the correct answer is given:

```
10 MODE 1
20 response=0
30 number1=INT(RND*12+1)
40 number2=INT(RND*12+1)
50 WHILE response<>number1*number2
60 PRINT"What is";number1;"times";number2;
70 INPUT response
80 WEND
```

Line 50 includes < >, which you may not be familiar with. It means 'not equal'. So line 50 basically says 'While the response

given is not equal to the two numbers multiplied together, repeat the commands that follow'. The end of this WHILE loop is given by the WEND (short for WHILE END) at line 80. Notice the difference from FOR . . . NEXT. We have no idea when this loop will end, but we do know the conditions under which it will end — when the response given *is* equal to the two numbers multiplied together. Line 50 tells the Amstrad "WHILE the response is wrong, keep asking the question".

The WHILE loop is valuable where we want to restrict the input from the keyboard to a particular range of values. One of the problems with computing is that you can't trust people! Someone using a program may be asked to type in a number less than 10, but respond by typing 10. A well-designed program will reject a number in the wrong range and prompt the user to try again. This is very easy to achieve with the WHILE loop:

```
10 MODE 1
20 response=11
30 WHILE response>10
40 INPUT "Please give a number less than 10. ",respons
50 WEND
```

This program raises a number of points. Line 30 introduces >, which means 'greater than', so the line reads 'While the response is greater than 10, repeat the commands that follow'. When you run the program, the Amstrad repeatedly asks for a number less than 10, until you cooperate, when the loop ends.

At line 20, the variable 'response' is set to 11. It is *very important* when using WHILE that you make sure that you have set up a condition so that the Amstrad will carry out the WHILE loop at least once. If you delete line 20 and run the program again you will get the 'Ready' message, and nothing else happens. This is because the Amstrad treats any variables which have not been given a value as being 0. The computer obeys line 10, then it comes to line 30, a WHILE loop to only be carried out while 'response' is greater than 10. The Amstrad checks the value of the variable 'response', finds it has not been given a value and is hence 0, and so does not carry out the WHILE loop at all. As far as the computer is concerned, the condition for ending the WHILE loop has been satisfied even before the loop has been carried out once!

We can also use WHILE as a way of getting an endless loop which we can terminate in our own time by inputting an appropriate value. For example, using a FOR . . . NEXT loop we could construct this program to let us add up our expenses for the month:

```
10 MODE 1
20 totalexpense=0
30 INPUT "How many expense items";items
40 FOR count=1 TO items
50 PRINT"How much was spent on item ";count;
60 INPUT expense
70 totalexpense=totalexpense+expense
80 NEXT
90 PRINT "The total expense was ";totalexpense;"pound
```

We can use WHILE to avoid having to input the number of items:

```
10 MODE 1
20 totalexpense=0
30 answer$="Y"
40 WHILE answer$="Y"
50 INPUT "Please input the expense. ",expense
60 totalexpense=totalexpense+expense
70 INPUT "Any more (Y/N)";answer$
80 WEND
90 PRINT "The total expense was ";totalexpense;"pound
```

If we are reading values from a *data* statement, and we have no idea how many values there are, we can add a WHILE loop which relies on a *data terminator* to bring it to an end:

```
10 MODE 1
20 READ name$,telephonenumber$
30 WHILE name$<>"XXX"
40 PRINT name$,telephonenumber$
50 READ name$,telephonenumber$
60 WEND
```

```
70 DATA Albert,01 234 5657,Betty,0734 2105,Cu
thbert,92 4165,Deirdre,21 4358
80 DATA Egbert,87 5502,Francis,27148,Gilbert,
33 3333,XXX,YYY
```

The data terminator is a value we include in the data to indicate that the data has now reached its end. In this case the terminator is 'XXX'. Line 30 causes the computer to carry on reading names and telephone numbers from data while the name is not 'XXX'. Notice that because we are always reading two values at a time from DATA in line 50, we need to include some sort of value for telephonenumber$ after 'XXX'. I have used 'YYY', but it is really irrelevant what you put for this value. It plays no part in the program and is only included because the Amstrad would object if it found no value for telephonenumber$ in the data.

You should always choose a data terminator value that can't possibly occur otherwise. When reading in exam marks out of 100, it would be unwise to have a data terminator of 0. For example, Bloggs of 4C might have managed to get 0, and the reading of data would end part way through. In a case like this a more appropriate terminator would be −99, or something similar.

You might be a little confused by some features of the program. Why the READ statement at line 20, outside the WHILE loop? This takes care of two problems. First, it is unlikely but possible that there is no data at all except for the data terminator. Line 20 ensures that in this case the WHILE loop is never carried out. The Amstrad would read the first two values at line 20, find that name$='XXX', and so not obey the loop beginning at line 30. We 'read ahead' to check what the data is before we do anything with it.

The second problem with not carrying out a 'read ahead' on the DATA is that the data terminator may well be printed out in the list of names and telephone numbers! You can see this if you try a simpler version of the program, which may seem at first sight to be just as good as the other:

```
10 MODE 1
30 WHILE name$<>"XXX"
40 READ name$,telephonenumber$
```

```
50 PRINT name$,telephonenumber$
60 WEND
70 DATA Albert,01 234 5657,Betty,0734 2105,Cu
thbert,92 4165,Deirdre,21 4358
80 DATA Egbert,87 5502,Francis,27148,Gilbert,
33 3333,XXX,YYY
```

This points to one feature of WHILE loops which you must be careful about. If you are going to end the WHILE loop as a result of a particular value being reached, you must make sure that the value occurs as the result of statements at the *end* of the WHILE loop. In this case name$ is read at the *start* of the loop, and so line 50 prints out 'XXX' as being a name. (Contrast this with the previous program, where the new string name$ was only READ at the *end* of the loop.)

Exercises

1) Write a program that reads the names, ages and birthdays of your friends from DATA and prints a list on the screen. Include a suitable data terminator.
2) Write a program which enables you to draw pictures from the keyboard. Set up a text window to allow the user to input x,y coordinates to a program which draws lines from the last coordinates given to the new ones. (You will only be able to draw pictures that do not involve the 'pen' being lifted off the 'paper'.) Use WHILE to end the program when you input a 'N' response to the question 'Any more coordinates?'.

AND and OR

Sometimes it is useful to end a WHILE loop if *any* of several different conditions occur. In our multiplication test, we might decide that rather than repeat the question endlessly if the answer is wrong, we will repeat it WHILE the answer is wrong AND the number of tries at a correct answer is 3 or less. This part-program illustrates the idea:

```
10 tries=0
20 response=0
```

```
30 firstnumber=INT(RND*12+1)
40 secondnumber=INT(RND*12+1)
50 WHILE response<>firstnumber*secondnumber A
ND tries<3
60 PRINT "What is ";firstnumber;"times ";seco
ndnumber;
70 INPUT response
80 tries=tries+1
90 WEND
```

We have set up a count of tries at the answer here, rather like the one the computer uses in FOR . . . NEXT loops, and the lines dealing with that count are 10, 50, and 80. Line 50 tells the computer the conditions under which it should carry out the WHILE loop. In this case, it is WHILE both the response is wrong AND the number of tries is less than 3. *Both* conditions must be true for the Amstrad to obey the WHILE loop. Run the program a few times and you will find that the loop ends either if you get the answer right or you make 3 wrong tries.

Lines 10, 50 and 80, which are involved in the count for the number of tries, are very important. It is easy to make mistakes when setting up a count, and find that it ends too early, too late, or not at all! In this case, the counting of the tries begins at 0, line 10, and each try is counted *after* the try has been made, line 80. After the third try, tries=3, and the WHILE loop ends. The variable tries is no longer less than 3, which is one of the conditions given in line 50 for the loop to continue.

If this seems obvious, you might care to discover what happens if line 10 sets the value of tries to 1, which, after all, does not seem an unreasonable value, as you are just about to have your first try at the answer. You will find that you have to change the terminating conditions in line 50.

Here's a simple guessing game which gives you 8 tries at guessing a random number from 1 to 100 that the computer has generated:

```
10 MODE 1
20 tries=1
```

```
30 randomnumber=INT(RND*100+1)
40 guess=0
50 WHILE guess<>randomnumber AND tries<9
60 PEN 1
70 PRINT
80 PRINT"This is guess number ";
90 PEN 3
100 PRINT tries
110 PEN 2
120 PRINT
130 INPUT "What is your guess";guess
140 IF guess<randomnumber THEN PRINT"Too low"
150 IF guess>randomnumber THEN PRINT"Too high
"
160 tries=tries+1
170 WEND
```

Lines 140 and 150 are a preview of the next chapter. I have added a bit of colour to liven things up. This time tries is set to 1 at line 20, rather than 0. This isn't deliberate awkwardness — if you can't see the reason, set tries to 0 and see what happens. I might add that, difficult though this may be to believe, the random number can *always* be guessed in 8 tries, provided you use the right approach.

We can also set up a WHILE loop so that it ends under one condition OR another. For example, a program that plays a game of Noughts-and-Crosses might contain lines like the following:

```
500 WHILE win=0 OR moves<9
510 PRINT "What is your move";
```

where win is set to 0 at the start of the game and will be set to 1 if either side wins. The loop here is:

WHILE neither of us have won
OR
there are any places left to move
carry on playing the game
WEND

If we had the first condition only, we might have to carry on playing the game even when there was nowhere left to move. If we had the second condition only, we would have to carry on playing the game even when someone had won, because there would still be places left to move. We need to end the game EITHER if someone wins OR if there's nowhere left to play.

We can use OR in a WHILE statement to ensure that the input falls into a given range. The previous example program that did this only ensured that the input value was *below* 10. Now we can fix the input so it falls *between* two numbers:

```
10 MODE 1
20 INPUT "Type in a number from 5 to 10";numb
er
30 WHILE number<5 OR number>10
40 PRINT "That won't do."
50 INPUT "Please pick a different number.",nu
mber
60 WEND
70 PRINT number;" is just right!"
```

Line 30 starts the loop, and will continually ask for a number WHILE the number input is less than 5 OR the number input is above 10. There are many games such as Noughts-and-Crosses or Battleships where one of the restrictions on a player's move is that the move must be on the board. (Not unreasonable!) A WHILE loop like the above can enable the computer to reject moves which fall outside the range of the board.

Testing that input is reasonable is known as *data validation*, and you can see from the above that such validation is as vital in games as it is in business or educational programming.

Exercises

1) Write a program that accepts as input a graphics x coordinate, and rejects any coordinate that lies off-screen. (The x co-ordinates run from 0 to 639.)
2) Write a program to print a line of asterisks anywhere on-screen in mode 1. The program accepts as input the x and y text coordinates and the length of the line. The program rejects

invalid x and y coordinates, and invalid lengths for the line, and will not print the line if it cannot fit in the space remaining on that line. (You will need to use separate WHILE loops to test the validity of each of the 3 input items.)

Chapter Five

Making Decisions

Making choices

In the last chapter we began to develop a simple program that asked multiplication questions, but we were handicapped because there was no way we could get the Amstrad to make a *choice* within the program. We would really like the Amstrad to give us the right answer if we keep getting the question wrong. We want to be able to say:

1) If the answer's wrong then give the right answer.
2) Otherwise, just carry on to the next question.

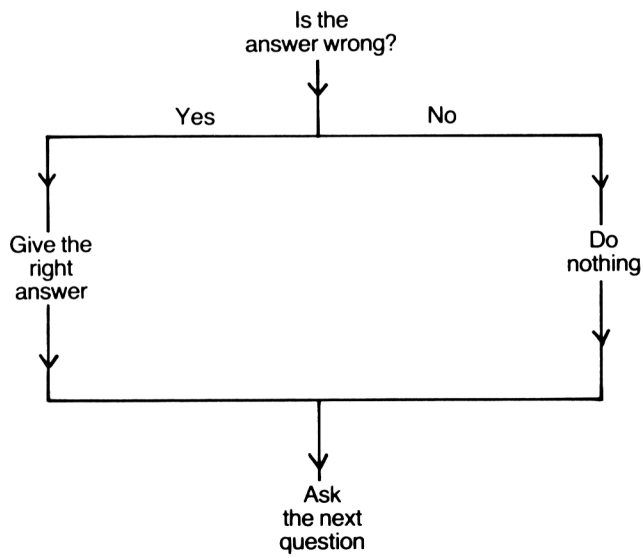

Figure 24 Flow diagram.

We can choose between two different courses of action by using the IF . . . THEN statement:

```
10 MODE 1
20 INPUT "How old are you";age
30 IF age>18 THEN PRINT"You must have left school
   by now."
```

If you run this program and input various ages you will find that line 30 is *only* carried out if you input an age greater than 18. The Amstrad looks at the statement immediately after the IF, and if this turns out to be *true*, the rest of the line is obeyed. If the statement after the IF is *false* (i.e. the age is less than 18), the computer just ignores the statements after the THEN and carries on to the next line of the program. We could easily extend the program by adding a variety of other IF . . . THEN statements which would apply for different age ranges:

```
10 MODE 1
20 INPUT "How old are you";age
30 IF age>18 THEN PRINT"You must have left school
   by now."
40 IF age<=11 THEN PRINT"So you're at primary school."
50 IF age>=65 THEN PRINT"I see you're retired."
60 IF age>100 THEN PRINT"Over 100! Congratulations!"
```

Line 40 tests if the age is less than or equal to 11, and line 50 if the age is greater than or equal to 65. Notice that if you input 67, two of the IF . . . THENs are true, and you get two messages, and if you input 110 you get three messages!

The statements after the THEN don't have to be PRINTs, they can be any valid Basic statement. For example, to return to our multiplication test, we can use IF . . . THEN to add up the number of wrong answers given:

```
10 MODE 1
20 wrong=0
30 FOR questions = 1 TO 10
40 number1=INT(RND*12+1)
50 number2=INT(RND*12+1)
60 PRINT "What is ";number1;"times ";number2;
70 INPUT answer
```

```
80 IF answer<>number1*number2 THEN wrong=wrong+1
90 NEXT
100 PRINT
110 PRINT "You had ";wrong;"wrong."
```

Line 80 checks if the input answer is not equal to the two random numbers multiplied together — i.e. whether the answer is wrong or not. If the answer *is* wrong, 1 is added to the running total of wrong answers held in the variable 'wrong'.

The program would be better if, when your answer was wrong, the computer told you so and gave the correct answer so that you could learn from your mistakes. It might seem that we need another two IF . . . THEN statements:

```
80 IF answer<>number1*number2 THEN wrong=wron
g+1
85 IF answer<>number1*number2 THEN PRINT"No t
hat's wrong."
86 IF answer<>number1*number2 THEN PRINT"The
answer is ";number1*number2
```

This seems tedious and repetitive. After all we've checked if the answer's right in line 80, why should we have to do it again? In practice we can use a *multi-statement line* at line 80:

```
80 IF answer<>number1*number2 THEN wrong=wron
g+1:PRINT"No that's wrong.":PRINT"The answer
is ";number1*number2
```

If the answer's wrong then the Amstrad obeys *all* the statements after the THEN in line 80. The statements must be separated by colons, and they *must* be on the same line as the IF . . . THEN statement. If they are on other lines, the Amstrad will obey them regardless of the result of the IF . . . THEN.

In fact you can have multi-statement lines on *any* line:

```
10 MODE 1:wrong=0:FOR questions = 1 TO 10:num
ber1=INT(RND*12+1):number2=INT(RND*12+1):PRIN
T "What is ";number1;"times ";number2;:INPUT
answer
```

```
20 IF answer<>number1*number2 THEN wrong=wron
g+1:PRINT"No that's wrong.":PRINT"The answer
is ";number1*number2
30 NEXT:PRINT:PRINT "You had ";wrong;"wrong
."
```

This program is the same as the one at the end of page 99. Unless you are working on an enormous program which uses up a lot of memory, I would suggest you avoid using multi-statement lines like those above. They make a program difficult to read and difficult to edit. Stick to one or two statements per line, except when you use IF . . . THEN.

The program we've just looked at serves as the basis for innumerable question-and-answer programs. With a few minor additions, it can be tailored to ask geography questions, test knowledge of French, or even see if you know how many days there are in the months:

```
10 MODE 1
20 wrong=0
30 FOR questions = 1 TO 12
40 READ month$,days
50 PRINT "How many days are there in ";month$
;
60 INPUT response
70 IF response<>days THEN wrong=wrong+1:PRINT
"No that's wrong.":PRINT month$;" has ";days;
"days."
80 NEXT
90 PRINT
100 PRINT "You had ";wrong;"wrong."
110 DATA January,31,February,28,March,31,Apri
l,30,May,31,June,30
120 DATA July,31,August,31,September,30,Octob
er,31,November,30,December,31
```

The data for the test in this case is just the names of the months and the number of days in each month. In a general knowledge

quiz you would probably make the data the entire question, because each question would be phrased differently, and your data section might look like this:

```
200 DATA What is the capital of the U.K., Lon
don
210 DATA Who is the President of the United S
tates, Ronald Reagan
```

Exercises

1) Write a program to accept as input someone's height and weight, and make appropriate comments on the figures.
2) Write a program to read a list of names and exam marks from data statements, and print out the names with PASS next to those who have 45 or more, and FAIL next to the others. Print PASS and FAIL in different colours.
3) Extend the multiplication program using IF . . . THEN statements so that after answering 10 questions you are given a message which depends on the number of wrong answers you gave. Print the messages using different PENs.
4) Devise your own question-and-answer program and include a count for right answers only. Print a message at the end depending upon the number of right answers, and convert the number of right answers to a percentage of the total.

IF . . . THEN . . . ELSE

In the multiplication test program we could easily add a line to print a suitable message every time a question was answered correctly:

```
10 MODE 1
20 wrong=0
30 FOR questions = 1 TO 10
40 number1=INT(RND*12+1)
50 number2=INT(RND*12+1)
60 PRINT "What is ";number1;"times ";number2;
70 INPUT answer
```

```
80 IF answer<>number1*number2 THEN wrong=wron
g+1:PRINT"No that's wrong.":PRINT"The answer
is ";number1*number2
85 IF answer=number1*number2 THEN PRINT"That'
s right. Well done."
90 NEXT
100 PRINT
110 PRINT "You had ";wrong;"wrong."
```

We have here a situation which occurs quite commonly in computing — there are just two possible outcomes when an answer is input. Either the answer is right, or it is wrong — there is no other possibility. In these circumstances where there are just two outcomes, we can use an extension of the IF . . . THEN statement to take care of both outcomes at once:

```
80 IF answer<>number1*number2 THEN wrong=wron
g+1:PRINT"No that's wrong.":PRINT"The answer
is ";number1*number2 ELSE PRINT"That's right.
 Well done."
```

Line 80 now says 'IF the answer's wrong THEN do this ELSE it must be right so do this'. Notice that you *cannot* extend the IF . . . THEN statement any further as it stands. A program to cope with many different outcomes requires more than a single IF . . . THEN . . . ELSE statement to take care of all the possibilities. For example, the IF . . . THEN . . . ELSE statement is ideal for a program drawing a monthly temperature bar chart for the U.K. After all, temperatures can only be above or below zero, can't they?

```
10 MODE 0
11 REM set up axes
20 yzero=100
30 MOVE 35,0
40 DRAW 35,399
50 MOVE 35,yzero
60 DRAW 600,yzero
```

```
61 REM draw bar chart
70 FOR month = 1 TO 12
80 READ temperature
90 IF temperature<0 THEN pencolour=6 ELSE pen
colour=3
100 scaledtemperature=temperature*12
110 MOVE month*40,yzero
120 DRAW month*40,yzero+scaledtemperature,pen
colour
130 DRAW month*40+35,yzero+scaledtemperature
140 DRAW month*40+35,yzero
150 NEXT
160 DATA -5,-1,0,5,11,15,20,22,14,12,10,-3
```

Running the program shows us that there are in fact 3 possibilities here: a temperature can be above, equal to, or below zero. If we want to show that a temperature of 0 belongs neither to the above zero or below zero group we need another line:

```
95 IF temperature=0 THEN pencolour=12
```

You might like to try the following, which is an amusing little program using IF . . . THEN . . . ELSE. This gives an insight into how computers can often seem intelligent when in fact they are doing remarkably little:

```
10 MODE 1
20 INPUT "What's your name";name$
30 PRINT "Pleased to meet you, ";name$
40 PRINT "I'm thinking of an animal and I wan
t you to try and guess its name."
50 PRINT "You can ask me any questions you li
ke   about what the animal looks like, but
you only get one guess at its name."
60 PRINT "If you think you know the animal, t
ype y when you're asked if you want to make
your one guess."
```

```
70 PRINT "Okay, what's your first question?"
80 response$="n"
90 guess=0
100 WHILE response$<>"y"
110 PRINT
120 guess=guess+1
130 LINE INPUT question$
140 IF RND   >0.5 THEN PRINT"Yes" ELSE PRINT"
No"
150 INPUT "Are you ready to make a guess yet
(y/n)";response$
160 IF response$<>"y" THEN PRINT"What's your
next question then?"
170 WEND
180 INPUT"What do you think the animal is";an
imal$
190 PRINT
200 IF RND(1)>0.5 THEN PRINT"Yes!!! You guess
ed it in";guess;"guesses." ELSE PRINT"No, I'm
 sorry, you're wrong. And you've had";guess;"
guesses!"
```

Exercises

1) Improve the exam-marks program you did earlier, by using IF . . . THEN . . . ELSE to select those who fail and those who pass the exam.
2) Write a program which reads a series of names and ages from a data statement, and prints the names in two columns, one for those old enough to vote, and another for those too young to vote.
3) Write a program that allows you to input your savings and expenditure via the keyboard, with expenditure being input as negative numbers. Set up a text window on the screen, and print the savings and expenditure to two different columns, with totals at the end. Print expenditure in red and savings in another colour.

Input from the keyboard: another way

In cases where there are many options it is often simpler just to use a series of IF . . . THEN statements. Here is a simple program that enables you to draw on the screen from the keyboard:

```
10 MODE 1
20 direction$=""
30 xcoord=320
40 ycoord=200
50 WINDOW 1,40,24,25
60 WHILE direction$<>"y"
70 INPUT "Which direction (L/R/U/D)";directio
n$
80 IF direction$="d" THEN ycoord=ycoord-1
90 IF direction$="u" THEN ycoord=ycoord+1
100 IF direction$="l" THEN xcoord=xcoord-1
110 IF direction$="r" THEN xcoord=xcoord+1
120 PLOT xcoord,ycoord,1
130 WEND
```

You can end the program by typing 'y' at the keyboard.

There are several points made by this program. First, because we want to be able to move in any of 4 different directions, left, right, up or down, we have little choice but to use several IF . . . THEN statements. Second, the program does not work with the CAPS LOCK on, because all the keys then produce upper case letters, and the IF . . . THEN statements check only for lower case letters. Lastly, the continual need to press a key and then press ENTER is irritating and makes the program almost unusable. Is there no better way of arranging for input from the keyboard?

There is. Instead of using INPUT we can use INKEY$:

```
10 MODE 1
20 direction$=""
30 xcoord=320
40 ycoord=200
60 WHILE direction$<>"y"
70 direction$=INKEY$
```

```
80 IF direction$="d" THEN ycoord=ycoord-1
90 IF direction$="u" THEN ycoord=ycoord+1
100 IF direction$="l" THEN xcoord=xcoord-1
110 IF direction$="r" THEN xcoord=xcoord+1
120 PLOT xcoord,ycoord,1
130 WEND
```

The program is transformed. INKEY$ continually scans the keyboard and picks up the state of the keys from the keyboard *without* the ENTER key being pressed. If you want to draw a line to the left, you need only hold the L key down. It will auto-repeat, and send a whole stream of characters to INKEY$, which will pick them up one by one. Line 70 stores the character currently in INKEY$ into the variable 'direction$'. It is this variable which is used in the IF . . . THEN statements on lines 80 to 120 to decide in which direction the line should be drawn or whether the program should finish.

The INKEY$ statement is valuable in any program where time is a factor and a single-key response is appropriate. You should be aware though that INKEY$ does *not* wait for a key depression. It is continually scanning the keyboard, even when no key is pressed. You can see this if you run this short program:

```
10 MODE 1
20 response$="n"
30 WHILE response$<>"y"
40 PRINT"Are you ready to stop yet (y/n)?"
50 response$=INKEY$
60 WEND
```

The computer continually prints 'Are you ready to stop yet?' because INKEY$ scans the keyboard, gets no response, and the program continues. As the next line is WEND, the whole loop repeats until you press 'y'. Use INKEY$ only when you are prepared for your program to continue even if there is no response from the keyboard. If you want a response, use INPUT or LINE INPUT.

INKEY$ provides a means for someone using a program to give an instant response. We can now include a time factor in tests or

games, and see how long it takes the user to respond. This is the basis for the keyboard familiarisation program included on the Amstrad Welcome tape. We haven't yet learnt how to generate random letters, so the following program is a crude version which tests you on one letter read from data:

```
10 MODE 1
15 PEN 1
20 PRINT "I am going to print a letter"
30 PRINT "on the screen, and I want you to fi
nd"
40 PRINT "that letter as quickly as possible
on"
50 PRINT "the keyboard, and press it."
60 PEN 3
70 PRINT:PRINT"You will be timed!"
80 PRINT "Press R when you are ready"
90 response$=INKEY$
100 WHILE response$<>"R" AND response$<>"r"
110 response$=INKEY$
120 WEND
130 starttime=TIME
140 READ letter$
150 PRINT "The letter is ";
160 PEN 2
170 PRINT letter$
180 response$=INKEY$
190 WHILE response$=""
200 response$=INKEY$
210 WEND
220 totaltime=TIME-starttime
230 PRINT:PRINT "That took ";totaltime/300;"s
econds."
240 IF response$=letter$ THEN PRINT"At least
you got it right." ELSE PRINT"And you got it
wrong!"
250 DATA q
```

Lines 90 to 120 scan the keyboard until you press the 'R' key. Notice that the WHILE loop set up in line 100 continues WHILE the response is not R AND not r. (A bit mindboggling!) This means that the loop will end when R is pressed, whether an upper case R or lower case r is produced as a result of CAPS LOCK being on or off.

At line 130 we set the time for the start of our one letter test. TIME is a variable that the Amstrad uses to store the time in 1/300ths of a second since the machine was switched on. Try:

```
? TIME
```

and the Amstrad will show you how long it's been switched on! The Amstrad updates TIME continually, as you can see if you print its value out again; you will find it has increased. We store the TIME at which the test begins in the variable starttime.

Lines 180 to 210 perform a similar waiting exercise to lines 90 to 120, only this time we're waiting for ANY key to be pressed and not a particular one. As soon as a key is pressed, the test is over, so line 220 calculates the difference between the present TIME and the starttime. This tells us the total time it took to answer the question. Line 230 divides that time by 300 to turn it into seconds.

The program we've just looked at introduces several new ideas, although it is incomplete as it stands — we'll finish it off properly in the next chapter. TIME is a handy variable and can be used just as easily where INPUT is involved, if timing seems appropriate.

Exercises

1) Take the random number multiplication test program you wrote earlier and add a timing element to it. Print the total time at the end of the test, together with a comment depending on whether that time was fast, average, or slow.
2) Write a program to continually print TIME in minutes and seconds until you press 'N' for 'no more'.
3) Extend the keyboard drawing program so that lines can also be drawn diagonally by the depression of a single key. Include an option to plot points in the background colour. This will enable you to erase lines and to move about from one part of the screen to the other without leaving a line behind. (One problem with this is that you can no longer see where you are

on the screen! You will have to plot two points every time — one in the PAPER colour to erase the old position, and one in a PEN colour so that you can see where you are.)

Compound conditions with IF . . . THEN

We can also use AND and OR with IF . . . THEN statements. Our multiplication test with timing might include lines like:

```
100 IF totaltime>1000000 AND rightanswers<5 T
HEN PRINT "You're slow as well as stupid!"
110 IF totaltime<5000 AND rightanswers<5 THEN
 PRINT "There's no point in being fast and ge
tting them wrong!"
```

A program to test whether you are a fit person to go to a famous university might have the line:

```
200 IF income>200000 OR IQ>130 THEN PRINT"Wel
come to the University!"
```

This program 'bounces' a single point around the screen in a colourful and very organised manner:

```
10 MODE 0
20 finish=0
30 xcoord=300
40 ycoord=100
50 xchange=4
60 ychange=8
70 WHILE finish<1
80 IF xcoord+xchange<0 OR xcoord+xchange>639
THEN xchange=-xchange
90 IF ycoord+ychange<0 OR ycoord+ychange>399
THEN ychange=-ychange
100 xcoord=xcoord+xchange
110 ycoord=ycoord+ychange
120 colour=INT(RND*15+1)
130 PLOT xcoord,ycoord,colour
140 WEND
```

The point begins with coordinates (300,100), lines 30 and 40, and is then moved by the amounts in lines 50 and 60. The WHILE loop beginning at line 70 is a perpetual one, and you can only get out of it by pressing ESC twice. Lines 80 and 90 work out what the new x and y coordinates will be once xchange and ychange have been added on. If the new coordinate is outside the screen area, the IF . . . THEN statement reverses the direction of movement by making the change minus whatever it was before. This makes the point 'bounce' off the screen sides. You can confine the point to 'bounce' within any area you like by changing the 'edge' co-ordinates as given in lines 80 and 90. Substitute 100 for 0 and 400 for 639 in line 80, for example.

Exercises

1) Modify the 'bounce' program so that the point plotted only changes colour when it bounces off one of the screen sides.
2) Write a program which reads in from data the number of times a criminal has been convicted and his age. Print out the names and ages of all criminals with more than 10 convictions who are under the heading PERSISTENT OFFENDERS.
3) Bank customers are credit-rated as 1 (safe), 2 (average), or 3 (very risky). Write a program which reads in from data customers' names, credit ratings and their current bank balance, and prints out the details for any who are credit-rated as 3 or owe more than £500.

Chapter Six

Strings

Strings and things

In the previous chapters we have tended to concentrate on numeric variables and the commands we can use to manipulate them, but there are equally potent commands for dealing with string variables. Surprisingly, some statements which you might expect to work exclusively for numeric variables also work for strings:

```
10 MODE 1
20 INPUT "What's your first name";name$
30 INPUT "What's your surname";surname$
40 wholelot$=name$+surname$
50 PRINT "Put them together and you get ";who
lelot$
```

Strings can be joined together or *concatenated* simply by putting a '+' sign between them. We can also *compare* strings using <, >, and =, just as we can compare numbers:

```
10 MODE 1
20 INPUT "Type in the first word.",first$
30 INPUT "Type in the second word.",second$
40 IF first$=second$ THEN PRINT"The two words
 are identical.":END
50 IF first$<second$ THEN PRINT first$;" come
s before ";second$;" alphabetically." ELSE PR
INT second$;" comes before ";first$;" alphabe
tically."
```

With strings, the '=' sign means that the two strings must be absolutely identical, character for character. If you type 'Apple' and 'apple' the Amstrad will *not* regard these words as being the same. After all, one begins with a capital letter! Indeed, the computer has its own ideas about the alphabet, and *all* capital letters are counted as being 'earlier' in the alphabet than the lower case ones.

As you have probably discovered from running the program, the Amstrad counts '<', (smaller than), as meaning 'earlier in the alphabet' when dealing with strings. Line 40 ends with a statement we have not met before, END. This just tells the Amstrad to finish running the program. The Amstrad does this automatically anyway, once it runs out of lines, but it is convenient to include END here, as it prevents the Amstrad from going on to line 50. You might like to remove the END to see what effect this has when you input two identical strings!

The ability to compare strings is useful for sorting purposes. It is quite common for names to be put into alphabetical order, for example, and the computer can take the donkey work out of tasks like this. We may set up a *database* of information about a particular topic, and the computer can rapidly search through that data and extract the information we want. For example, you could have a program which recorded all your friends' names and birthdays. The Amstrad can quickly find the birthday of your friend, once you input the name:

```
10 MODE 1
20 INPUT "What is your friend's name";friend$
30 WHILE name$<>"XXX" AND name$<>friend$
40 READ name$,birthday$
50 IF name$=friend$ THEN PRINT friend$;" has
a birthday on ";birthday$
60 WEND
70 IF name$="XXX" THEN PRINT "I'm sorry, but
I don't have that name in my data."
80 DATA JANE JAMES,October 29,FRED BLOGGS,May
 16,GERALD SPUD,June 3
90 DATA XXX,XXX
```

This program may seem a bit silly in its present form, as it's a lot

quicker just to list the program and look at the names and birthdays. We shall see later that it is possible to store data like this outside the program itself, and of course in practice such a program would only be used when many hundreds of data items were involved.

One problem you may have uncovered with the program was identified at the start of the chapter. Unless you type in your friend's name *exactly* as it is in the data section, the computer doesn't realise the two strings are the same and announces that it can't find the name. On some micros the only way around this is to extract the individual characters from the word input and convert them all into lower or upper case as required. On the Amstrad things are simpler. Type:

```
? UPPER$("Fred")
```

and you will see the entire word printed in upper case letters. UPPER$ converts whole words to upper case — you can probably guess what LOWER$ does:

```
? LOWER$("Fred")
```

We can improve the birthdays program by adding this line:

```
25 friend$=UPPER$(friend$)
```

This tells the Amstrad to take the string from friend$, convert it to upper case, and put the new string back into friend$. If you run the program again you'll find it doesn't matter how you type in your friend's name now — if it's in the data the Amstrad will find it. One consequence of using UPPER$ or LOWER$ to convert input is that it's much more sensible to make DATA items consistently either upper case or lower case. Converting the input is no help if the strings in the DATA are themselves mixtures of upper and lower case characters!

Exercises

1) Write a program that accepts as input your surname, first name, and title (Mr., Mrs., etc.), concatenates them, and prints out your full name with your title in front.
2) Write a foreign language dictionary program that finds the equivalent foreign word for an English word that you input.

3) Write a program that accepts as input 3 names and prints them out in alphabetical order. (This is more tricky than you might think.)

How long is a string?

On many occasions in programming it is useful to know the length of a string. An input string may have to be used as a column heading in a table being printed out, and we need to make sure that the string is not too long to fit in the column. Alternatively, the string may have to be centred on a line, and we can only do this if we know its length. The length of a string can be found by using LEN:

```
? LEN("Fred")
```

The Amstrad prints 4. We can use LEN in a centring program:

```
10 MODE 1
20 INPUT "What's your name";name$
30 CLS
40 length=LEN(name$)
50 xposn=(40-length)/2
60 LOCATE 18,11
70 PEN 3
80 PRINT"Hello"
90 LOCATE xposn,13
100 PEN 2
110 PRINT name$
```

Line 40 first subtracts the length of the string from the line length (40 for mode 1) to find out how many spaces are left on the line. To centre a word, these spaces need to be placed equally to the right and left of the word, so this result is divided by 2 to give the x coordinate for the string on the line. Strictly speaking, the value of xposn in line 50 should be rounded using INT, but the LOCATE command at line 90 doesn't seem disturbed by the fact that xposn is partly decimal.

You may find that some input strings don't seem centred very well. This is because words with an odd number of letters leave

an odd number of spaces on the rest of the line. The spaces are inevitably shared unequally to the right and the left.

Here's another example of LEN in action:

```
10 MODE 1
20 INPUT "What length words are you searching
 for";length
30 READ word$
40 WHILE word$<>"XXX"
50 IF LEN(word$)=length THEN PRINT word$
60 READ word$
70 WEND
80 DATA One,feature,of,computing,became,appar
ent,that,still,applies,today.,Computers,were,
becoming
90 DATA faster,smaller,"and,",above,"all,",ch
eaper,with,each,passing,year.,The,use,of,tran
sistors,resulted,in
100 DATA switching,speeds,of,millionths,of,a,
second.,XXX
```

You might notice a slight flaw in this program — try looking for words of 6 letters, for example. We shall be seeing how to get around this problem shortly. Line 90 may be a bit confusing. As the Amstrad recognises commas as dividing one data item from another, it can only accept commas as part of a string when the whole string is placed within inverted commas, hence the use of "and," and "all,".

The Amstrad can also generate a string of any required length composed of a single repeated character:

```
? STRING$(20,"A")
```

This is useful as it means that we no longer need a loop to produce a line full of repeated characters:

```
10 MODE 1
20 INPUT "How long is the rectangle";length
30 INPUT "How high is it";height
40 CLS
```

```
50 LOCATE 1,1
60 PRINT STRING$(length,"*")
70 FOR yposn=1 TO height-2
80 PRINT"*";SPC(length-2);"*"
90 NEXT
100 PRINT STRING$(length,"*")
```

We need to use height-2 in line 70 because the corner asterisks also count towards the height, so we must subtract the contribution of two asterisks that will be made by lines 60 and 100.

Exercises

1) Modify the program that finds words of a given length so that all the words are printed to the screen to make up lines of text. The words of the chosen length are printed in a different colour so they stand out from the rest of the text.
2) A typical feature of word-processors is the search-and-replace facility, which allows the user to get the computer to search through the text and replace occurrences of one word by another. Write a program that allows you to specify a search word and a replacement word. The program then reads text from DATA and prints it to the screen, substituting the replacement word wherever it finds the search word. (The town sheriff description from Chapter 2 provides an amusing piece of text to experiment with.)
3) Write a program that accepts as input your name, and then prints it centred within a box of asterisks.

Little bits of strings

The program that searched for words of a requested length suffered from one obvious weakness. The program considered strings like "today." as having 6 letters because the full stop was included.

We can eliminate the full stop by checking the last character of each word and removing it from consideration if it is not alphabetic. To do this we use one of three string-handling commands which allow you to extract *substrings* (shorter strings) from a larger string. Let's look at an example. Type:

```
? LEFT$("Hello there",4)
```

This tells the Amstrad to print the *LEFTMOST* 4 characters of the string given, so we get 'Hell'. Or we could have:

```
? LEFT$("Aircraft-carrier",8)
```

giving us 'Aircraft'. You can probably guess what this does:

```
? RIGHT$("Start",4)
```

It takes the *rightmost* 4 characters from the string. The third command is a little more versatile, as it can carry out both the LEFT$ and RIGHT$ functions as well as extracting substrings from the *middle* of longer strings:

```
? MID$("Gigantic",4,3)
```

MID$ produces a substring beginning at the 4th character of the string and continuing for 3 characters, so we get 'ant'. The second number need not be included:

```
? MID$("Elasticity",7)
```

This gives 'city'. The substring begins at the 7th character and carries on to the end of the string.

As with everything else in programming, LEFT$, RIGHT$ and MID$ work equally well with string variables.

To return to our original problem: we need to strip off the last character from a string and ignore it if it's not alphabetical. We can modify the program like this:

```
40 WHILE word$<>"XXX"
41 endofword$=RIGHT$(word$,1)
42 wordlength=LEN(word$)
43 IF endofword$="." OR endofword$="," THEN w
ordlength=wordlength-1
50 IF wordlength=length THEN PRINT word$
60 READ word$
70 WEND
```

Line 41 takes the rightmost character out of the word. If this character is a comma or a full stop, line 43 reduces the word length by one, as punctuation shouldn't count towards that length.

MID$ can help us to eliminate characters we don't want from input. In an earlier program we saw how the computer would not accept "Smith" and "SMITH" as being the same thing to we

humans. We can get round that problem by using UPPER$ or LOWER$ on the input string. But another common problem is that many people inadvertently type extra spaces before, during, or after strings. The computer ignores spaces before or after an input string, but it can do nothing about extra spaces *within* a string. It does not recognise "J. SMITH" or "J. SMITH" as being the same string as the "J.SMITH" it has stored as data. How can they be the same? As far as the computer's concerned, the string it has stored is even a different length to the other two!

This program strips out the spaces anywhere within an input string. It could easily be used at the start of the program where you input your friend's name and the computer searches for and prints out the friend's birthday. In order to make the action of the program visible, the original string is printed flanked by characters to make the spaces clear. The new string produced by stripping out all the spaces is then printed in the same way:

```
10 MODE 1
20 INPUT "Please type in your word. ",word$
30 wordlength=LEN(word$)
40 newword$=""
50 FOR character=1 TO wordlength
60 letter$=MID$(word$,character,1)
70 IF letter$<>" " THEN newword$=newword$+let
ter$
80 NEXT
90 PRINT
100 PRINT "The old word was:"
110 PRINT
120 PRINT STRING$(10,"*");word$;STRING$(10,"*
")
130 PRINT
140 PRINT
150 PRINT "The new word stripped of spaces is
:"
160 PRINT
170 PRINT STRING$(10,"*");newword$;STRING$(10
,"*")
```

We could also use MID$ to produce an anagram of an input word:

```
10 MODE 1
20 PRINT "I will produce an anagram of any wo
rd   you input."
30 PRINT
40 INPUT "Type in the word.  ",word$
50 anagram$=word$
60 length=LEN(word$)
70 FOR muddle=1 TO length
80 random=INT(RND*length+1)
90 rightbit=length-random
100 anagram$=LEFT$(anagram$,random-1)+RIGHT$(
anagram$,rightbit)+MID$(anagram$,random,1)
110 NEXT
120 PEN 1
130 PRINT:PRINT "The anagram is ";
140 PEN 3
150 PRINT anagram$
160 PRINT
170 REM restore normal pen colour
180 PEN 1
```

The loop from lines 60 to 110 muddles up the letters of the word that has been input. The loop could be set to run, say, 5 times, but this would mean long words would not be well muddled, so the loop is set to run a number of times equal to the length of the original word. The anagram is produced by first selecting a random position within the word, line 80. We then work out how many letters are to the right of this position, line 90. Finally line 100 rearranges anagram$ so that the left and right halves are closed up and the letter at the random position is extracted and placed at the end of the word.

Exercises

1) Write a program that accepts as input a word and prints out the letters of the word on separate lines.

2) Extend the previous program so that the word is now printed on a single line, but with all the letter 'e's printed in a different colour.
3) Write a program that accepts as input your first name, and then prints out your initial.
4) Write a program that will read words from DATA and print out those beginning with a capital letter in a different column to those beginning with a lower case letter. (Remember that the Amstrad considers all upper case letters to be earlier in the alphabet than lower case ones.)
5) Produce a program which allows you to input your name in full and then prints out *all* your initials.

Using INSTR

In the earlier programs in this chapter, any text used was held in DATA as individual words, rather than as sentences. The easiest way to locate words in a long string such as a sentence is by using the INSTR statement:

```
? INSTR("Unbeatable","be")
```

The Amstrad searches the string "Unbeatable" for the *first* occurrence of the string "be", and then prints the character position at which the shorter string begins. Try:

```
? INSTR("Unbeatable","table")
```

and you get 6. What if the second string doesn't occur at all in the first, or the second is longer than the first?

```
? INSTR("Jam","marmalade")
```

The Amstrad prints 0 if it can't find a matching string.

It is just as easy to use INSTR with a much longer string:

```
? INSTR("there is the theatre","the")
```

The Amstrad prints 1, because the substring "the" first appears at character 1, as part of "there". Once we know the position of the first occurrence of a substring, we can optionally ask the Amstrad to continue the search for the substring by choosing the character position at which the search will begin. We know that "the" first occurs at character position 1, so let's carry out a second search

beginning at character position 2:

```
? INSTR(2,"there is the theatre","the")
```

The Amstrad tells us that "the" occurs again beginning at position 10. Let's carry on the search at position 11:

```
? INSTR(11,"there is the theatre","the")
```

The micro prints 14. We could continue the search from position 14, until we reached the last position in the string where the substring occurred. We would then know that we had found all the occurrences of the shorter string within the longer. (If we were only really interested in the word "the" on its own, how could we make sure that we found *only* this word and not its occurrences in longer words like "there", "theatre", or "other"?)

The following program uses INSTR to discover how many letter 'e's there are in an input word:

```
10 MODE 1
20 INPUT "Type in the word.  ",word$
30 start=1
40 numberofes=0
50 continue=1
60 WHILE continue=1
70 position=INSTR(start,word$,"e")
80 IF position>0 THEN numberofes=numberofes+1
: PRINT"There is an 'e' at position ";positio
n:start=position+1 ELSE continue=0
90 WEND
100 PRINT "There are ";numberofes;" 'e's in y
our word."
```

Line 80 ensures that the variable 'start' is reset to the character position one after that where a match was discovered. If no match is made, the ELSE sets 'continue' to 0 so that the WHILE loop will terminate.

We could easily use an approach like the above for checking letters in a game of Hangman:

```
10 MODE 1
11 REM choose a random word
```

```
20 numberofwords=10
30 chosenword=INT(RND*numberofwords+1)
40 FOR words=1 TO chosenword
50 READ word$
60 NEXT
61 REM set up the variables
70 length=LEN(word$)
80 answer$=STRING$(length,"-")
90 tries=1
100 guess$=""
101 REM give 10 tries
110 WHILE tries<11 AND answer$<>word$
120 PEN 3
130 PRINT
131 REM show what letters have been correctly
 guessed so far
140 PRINT"The word is ";answer$
150 PRINT
160 PEN 1
170 PRINT"This is try number ";tries
180 PRINT
190 INPUT"Guess a letter. ",letter$
191 REM check if that letter occurs in t
he word
200 start=1
210 continue=1
220 WHILE continue=1
230 position=INSTR(start,word$,letter$)
231 REM if letter occurs, place it in answer
at right position
240 IF position>0 THEN answer$=LEFT$(answer$,
position-1)+letter$+MID$(answer$,position+1):
start=position+1 ELSE continue=0
241 REM continue letter check until we don't
find any matches
250 WEND
```

```
251 REM give another try
260 tries=tries+1
270 WEND
280 PEN 2
290 PRINT
300 IF answer$=word$ THEN PRINT"You've got it
!" ELSE PRINT"It was ";word$
310 PEN 1
320 END
330 DATA jackal,lynx,antelope,elephant,tiger
340 DATA rhinocerous,iguana,ostrich,panda,wha
le
```

This program contains two WHILE loops, one inside the other. These 'nested loops' will be discussed in more detail in the next chapter, but for the moment let us look at the program as a whole. The words to be used in the game are held in DATA in lines 330 and 340. If you want to use some of your own words, just add more DATA statements and change line 20 to show how many words there now are.

The first part of the program randomly selects a word by reading through a random number of words. Once the word has been chosen, line 80 sets up another string of exactly the same length, but filled with dashes. The outer WHILE loop beginning at 110 allows you to have 10 tries at guessing the word. The inner loop, from 220 to 250, checks the word to see where the letter occurs, if at all. When the program discovers the position of the letter guessed, line 240 places that letter at exactly the same position in answer$. In other words, a correctly guessed letter replaces one of the dashes in answer$. Eventually, either 10 tries have been made, or answer$ and word$ are identical (i.e. all the letters of word$ have been correctly guessed), so the game ends.

You may find this program difficult to follow at present, but it should serve to illustrate that the Basic statements so far introduced already allow us to write programs of some sophistication. The Hangman program above performs the same function as the one on the Welcome cassette, although it clearly lacks the graphical frills.

Exercises

1) The Hangman program has two weaknesses. First, a player can type in more than a single letter at a time as a guess. Second, the player is not told what guesses have already been made. Improve the program so that these problems are overcome. (*Hint*: for the second weakness, how could concatenation help you keep a record of a player's guesses?)

The Amstrad character set

In an earlier program we saw how we could eliminate non-alphabetic characters like , or . from the end of a string by checking the last character position. If a comma or full stop was present, it could then be stripped off the end of the word. Clearly this approach is not of much general use — there are a host of other non-alphabetic characters which might be at the end of a string, such as ",!,?,',(, or). Are we going to have to check in case each of these might be present as well? Is there no way we can immediately identify *any* non-alphabetic character which may be at the end of a word?

Indeed there is. Every character on the keyboard has associated with it a code number, called the ASCII code. The codes range from 0 to 255, with codes 0 to 31 having special meanings to the computer, like 'Switch the printer on' or 'Turn the screen off'. The codes from 32 to 255 each represent a particular character. The advantage of the ASCII codes is that alphabetic characters are collected together over a particular range of codes, and so it is very easy to spot a non-alphabetic character because its ASCII code lies outside that range. For example, codes 65 to 90 are the codes for the upper case characters:

```
10 MODE 1
20 FOR code=65 TO 90
30 PRINT CHR$(code);
40 NEXT
```

Line 30 introduces CHR$, which tells the Amstrad 'Print the character that has the code that follows in brackets'. Code 65 stands for the letter A, 66 for B, and so on. Codes 97 to 122 are for the lower case letters. Edit line 20 so that the loop starts at 97 and

finishes at 122 and run the program again to see that this is so.

The remaining codes are for the numbers, punctuation, mathematical symbols, and a whole series of other characters mainly used in games. Edit line 20 so the loop runs from 32 to 255 and you will see most of the Amstrad character set, which includes grinning faces, crotchets and quavers, and even a variety of stick men!

A warning here. The codes under 32 give the Amstrad all sorts of commands, and if you use any of them by mistake you can get some very funny effects. As the program is short, you might like to try printing the characters from 0 to 255. You can restore things to normal with a mode change, or you can reset the computer with [CTRL] [SHIFT] and [ESC] if you don't feel up to that!

There are characters with codes 0 to 31, but they can only be printed if you precede them by printing the character with an ASCII code of 1. Character 1 tells the Amstrad 'Print the character that comes straight after, even if it's in the range 0 to 31 — and none of that funny business!'

```
10 MODE 1
20 FOR code=0 TO 31
30 PRINT CHR$(1);CHR$(code);
40 NEXT
```

You can actually get most of these characters directly by holding down [CTRL] and at the same time pressing the non-numeric keys on the keyboard. Run through the alphabet. The characters printed are the same as those just produced by the program.

Most of the character set is described in detail in Appendix 3 of the User Instructions, but the following table identifies the most useful ASCII codes:

Characters	ASCII codes
Various special codes	0–31
A space	32
0–9	48–57
A–Z	65–90
a–z	97–122

Figure 25 Some of the more useful ASCII codes to know.

There are many advantages to the ASCII codes. One is that they provide us with an easy means of restricting input from the keyboard to a particular range of characters. The following program will print *only* any upper case characters you input and ignore all other inputs. Press the CAPS LOCK to make the keys produce upper case letters before you run the program, otherwise you will find yourself unable to print anything! The program uses a non-terminating loop, so you will have to press [ESC] twice to escape from it:

```
10 MODE 1
20 continue=1
30 WHILE continue=1
40 code$=INKEY$
50 IF code$<>"" THEN code=ASC(code$)
60 IF code>64 AND code<91 THEN PRINT code$;
70 WEND
```

Line 40 uses INKEY$ to scan the keyboard. If a key is pressed, the character produced is saved in code$. Line 50 introduces a new Basic key word. ASC takes a single character and produces the ASCII code for that character. Unfortunately we need the IF . . . THEN part, because if no key has been pressed, code$ is " ", and the Amstrad gives an error message if it's asked to find the ASCII code of nothing! Line 60 prints code$ if the code for the character is greater than 64 and less than 91. From Figure 25 we can see that the effect of this is to print only letters which are upper case.

We could use the ASCII codes to write a simple encoding/decoding program:

```
10 PRINT
20 INPUT "What is the code";change
30 PRINT
40 LINE INPUT "What is the message? ",message
$
50 length=LEN(message$)
60 codedmessage$=""
70 FOR count=1 TO length
80 letter$=MID$(message$,count,1)
90 letterascii=ASC(letter$)
```

```
100 codedletter$=CHR$(letterascii+change)
110 codedmessage$=codedmessage$+codedletter$
120 NEXT
130 PRINT
140 PRINT"The coded message is:"
150 PEN 3
160 PRINT
170 PRINT codedmessage$
180 PEN 1
```

Type in any whole number of 120 or less as input. The computer accepts your message at line 40. The loop from 70 to 120 takes the letters one at a time from your message and adds the number you input to the ASCII codes to produce a new ASCII code. This is turned back into a character at 100 and added to the coded message in 110.

I have not included the usual mode command at the start of the program. This gives you the chance to try decoding your encoded message, and if the screen is not cleared you can copy the encoded message using the COPY key, rather than trying to remember what may be a string of gibberish! To decode a message you must input minus the number you used to encode the message, i.e. if you used 23 for encoding you must use −23 to decode. Using high numbers for encoding will give you a message consisting completely of non-alphabetic characters, and you may end up with a mixture of those stick men and quavers we mentioned earlier!

VAL and STR$

On some occasions it is easier to treat a number as part of a string, for example, when a date is input. Yet later that number may be needed for arithmetic, and we saw much earlier in the book that the Amstrad quite rightly refuses to do arithmetic with strings. We need to be able to convert the string version of the number into a numeric variable. To do this we use VAL, which converts a numeric string into a numeric variable:

```
10 MODE 1
20 INPUT "Type in any characters, but begin w
ith a number. ",characters$
```

```
30 number=VAL(characters$)
40 PRINT "The number was ";number
```

VAL only looks at the first character position of the string to see if there's a number there. If you input 'June 6th', for example, line 30 will produce 0, because the first character is not a number and VAL looks no further. The '6' could be extracted from the date:

```
10 MODE 1
20 INPUT "Input a date (eg June 6th). ",date$
30 continue=1
40 position=1
50 WHILE continue=1
60 letter$=MID$(date$,position,1)
70 code=ASC(letter$)
80 IF code>47 AND code<58 THEN continue=0 ELS
E position=position+1
90 WEND
100 number$=MID$(date$,position)
110 number=VAL(number$)
120 PEN3
130 PRINT"You typed in day ";number;"of the m
onth."

140 PEN1
```

The program examines the letters of the string one by one until an ASCII code indicating a number is found (line 80). Line 100 removes the part of the string containing the number, and this is converted to a number by VAL in 110. A simpler solution is to insist that the first characters of the date are the number, i.e. '6th June' rather than 'June 6th'!

STR$ performs the opposite function to VAL, converting a numeric variable into a string variable:

```
10 MODE 1
20 INPUT "Type in today's date in the form 31
/6/83",date$
21 REM convert day and month to numbers
30 day=VAL(date$)
```

```
40 monthposition=INSTR(date$,"/")
50 month$=MID$(date$,monthposition+1)
60 month=VAL(month$)
70 yearposition=INSTR(monthposition+1,date$,"
/")
80 year$=MID$(date$,yearposition+1)
81 REM find out name of month
90 FOR count=1 TO month
100 READ nameofmonth$,numberofdays
110 NEXT
111 REM find next week's date
120 day=day+7
121 REM if this is more than the number of da
ys in the month
122 REM find the next month
130 IF day>numberofdays THEN day=day-numberof
days:READ nameofmonth$,numberofdays
140 weekfromtoday$=nameofmonth$+STR$(day)+",
19"+year$
150 PRINT
160 PRINT"One week from today the date will b
e:"
170 PEN 3
180 PRINT weekfromtoday$
190 PEN 1
200 DATA January,31,February,28,March,31,Apri
l,30,May,31,June,30
210 DATA July,31,August,31,September,30,Octob
er,31,November,30,December,31
```

This program works out a date one week after today's date. Both the day and the month are needed as numbers, the day because arithmetic is performed on it, line 120, and the month because it is used to set a loop to read the right number of data items to find the month's name, lines 90 to 110. In line 140 the numeric variable 'day' is converted back to a string using STR$.

Although the exercise of converting string variables to

numerics may seem an academic one, it is worth studying the techniques used above to extract the numeric values from a string. Converting numbers to strings and subsequently concatenating the strings provides an easy way of keeping a series of associated values together without having to describe them individually. The original numbers can be rapidly obtained by using MID$ to remove the required part of the string.

Exercises

1) Write a program that will only allow you to type numeric values at the keyboard.
2) The program that calculates next week's date fails if next week is in a new year. Improve the program so that it will give the correct date even for inputs such as 25/12/84.
3) Another flaw in the program is that it does not take into account leap years. Leap years are those years divisible by 4. Centuries such as 1700, 1800, etc., are only leap years if they are divisible by 400. For example 1600 is a leap year but 1700 is not. Improve the program so that it adjusts the number of days in February depending on whether the year is a leap year or not.
4) Write a program that finds how many shopping days there are until Christmas after today's date has been input.
5) For a real challenge, write a program that will tell you how many days you have lived after you have input your birthdate and today's date.

Chapter Seven

Loops and Lists

Nested loops

In the last chapter we had an example of a program using two loops, one of which was inside the other. These *nested loops* enable us to simplify programs that might otherwise require many more separate loops. Let us look at a simple example which demonstrates the nature of nested loops:

```
10 MODE 1
20 FOR outerloop=1 TO 3
30 PEN 3
40 PRINT "Outer loop is ";outerloop
50 FOR innerloop=1 TO 4
60 PEN 2
70 PRINT "Inner loop is ";innerloop
80 NEXT
90 NEXT
100 PEN 1
```

The output from the program looks like this:

```
>
Outer loop is 1
Inner loop is 1
Inner loop is 2
Inner loop is 3
Inner loop is 4
Outer loop is 2
Inner loop is 1
Inner loop is 2
Inner loop is 3
Inner loop is 4
Outer loop is 3
Inner loop is 1
Inner loop is 2
Inner loop is 3
Inner loop is 4
```

Figure 26 Program output.

When you run the program, the Amstrad begins the outer loop at line 20 and prints its value, 1, at line 40. A second, inner loop begins at line 50, and yet the Amstrad is still within the first loop, because it hasn't reached a NEXT to tell it that the first loop is finished. So the micro sets up a second loop, and prints its value, 1, at line 70.

At line 80 the program contains the first NEXT statement. Now you might imagine that the Amstrad would take this to mean 'Now do the next step in the outer loop'. This is not the case. The first time the Amstrad finds a NEXT, it is associated with the most recent FOR encountered. Here the most recent FOR came at line 50, so the Amstrad proceeds to count through this inner loop and print 2, 3, and 4. Once the inner loop has reached its stopping value, the computer carries out line 90 — another NEXT, and this is associated with the most recent FOR that still doesn't have a NEXT — i.e. the outer loop beginning at line 20.

So the Amstrad goes back to line 20 to begin the next circuit of the outer loop, prints 2, comes to the inner loop, prints 1, and proceeds to run through the inner loop again, printing 2, 3, and 4. And so on. The inner loop 'nests' completely within the outer loop:

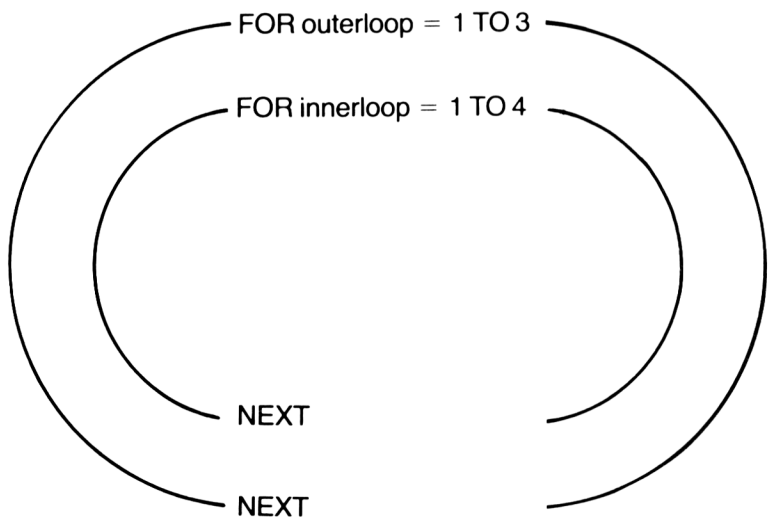

Figure 27 Nested loop.

Within a nested loop, the start and end of each loop can always be joined *without* crossing a line joining the start and end of another part of the nested loop:

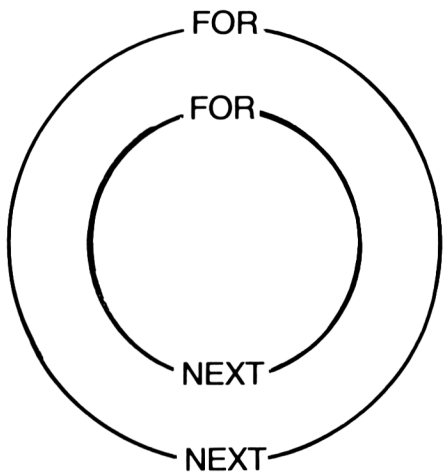

Figure 28 Nested loop.

The loops can begin and end with any value, and you may like to play about with the nested loops in the program above to make sure you understand what's happening.

Enough of the example. Let's look at a real program, extraordinarily brief, that prints out all the multiplication tables from 1 to 10:

```
10 MODE 1
20 FOR tables=1 TO 10
30 PEN 3
40 PRINT tables;" times table"
50 PRINT
60 PEN 2
70 FOR number=1 TO 10
80 PRINT number;" times ";tables;" = ";number
*tables
90 NEXT
100 PRINT
110 NEXT
120 PEN 1
```

This program would have been much longer but for nested loops, as it would have involved setting up 10 separate loops to count through each multiplication table! If the tables whizz by a bit too quickly for you, you might like to consider the place to put an INPUT statement that asks if you're ready for the next table, so that you get a good chance to look at each of them.

This program draws a series of triangles to fill the screen:

```
10 MODE 1
20 FOR xcoordinate= 0 TO 500 STEP 100
30 FOR ycoordinate= 0 TO 350 STEP 50
40 MOVE xcoordinate,ycoordinate
50 DRAW xcoordinate+90,ycoordinate+10
60 DRAW xcoordinate+40,ycoordinate+40
70 DRAW xcoordinate,ycoordinate
80 NEXT
90 NEXT
```

The STEP size and the coordinates of the triangle itself are arbitrary, and you might like to experiment with different values and see the results.

In Chapter 3 there was a short program that drew the outline of a house. With a few modifications to the size of the house, we can use a nested loop to produce a whole street full of houses:

```
10 MODE 1
20 REM set up coordinates for house front
30 FOR houseleftx=0 TO 500 STEP 100
40 houseboty=200
50 houserightx=houseleftx+90
60 housetopy=250
70 REM draw the house front
80 MOVE houseleftx,houseboty
90 DRAW houserightx,houseboty
100 DRAW houserightx,housetopy
110 DRAW houseleftx,housetopy
120 DRAW houseleftx,houseboty
130 REM set up coordinates for roof
140 roofleftx=houseleftx+15
```

```
150 rooftopy=270
160 roofrightx=houseleftx+75
170 REM draw the roof
180 MOVE houseleftx,housetopy
190 DRAW roofleftx,rooftopy
200 DRAW roofrightx,rooftopy
210 DRAW houserightx,housetopy
220 NEXT
```

In the last chapter one of the programs produced an anagram of any word input. We can now extend that program using a nested loop so that it produces a series of anagrams which have to be guessed as quickly as possible:

```
10 MODE 1
20 starttime=TIME
30 FOR anagrams=1 TO 10
40 READ word$
50 anagram$=word$
60 length=LEN(word$)
70 FOR muddle=1 TO length
80 random=INT(RND*length+1)
90 rightbit=length-random
100 anagram$=LEFT$(anagram$,random-1)+RIGHT$(
anagram$,rightbit)+MID$(anagram$,random,1)
110 NEXT
120 PEN 1
130 PRINT:PRINT "The anagram is ";
140 PEN 3
150 PRINT anagram$
160 PEN 2
170 PRINT:INPUT "What is the word";guess$
180 PRINT
190 IF guess$=word$ THEN PEN 3:PRINT"Right!"
ELSE PEN 1:PRINT"Wrong, it's ";word$
200 NEXT
210 totaltime=TIME-starttime
```

```
220 PRINT"That took you ";totaltime/300;" sec
onds."
230 DATA ready,enormous,kangaroo,jellybaby,fu
rniture
240 DATA electricity,window,polystyrene,funne
l,spaghetti
```

In this case the outer loop asks the questions, while the inner loop is used to muddle up the words and produce the anagram. We could choose to use a slightly different form of nested loop, with the outer loop being a WHILE, so that we can impose a time limit. Add these lines:

```
11 timelimit=100
12 PRINT "You have a time limit of ";timelimi
t;"seconds to solve 10 anagrams."
13 timesecs=0
14 right=1
15 countofwords=0
.
.
.
31 countofwords=countofwords+1
.
.
.
191 timenow=TIME-starttime
192 timesecs=timenow/300
193 PRINT:PRINT"You have used ";timesecs;"sec
onds."
.
.
.
225 IF right>1 THEN PRINT"You got ";right;"ri
ght." ELSE PRINT"And you didn't get one right
!"
```

WHILEs can also be nested:

```
10 MODE 1
20 READ question$,answer$
30 WHILE question$<>"XXX"
```

```
50 PEN 3
60 PRINT:PRINT question$
70 PEN 2
80 response$=""
90 tries=1
100 WHILE response$<>answer$ AND tries<4
110 LINE INPUT response$
120 tries=tries+1
130 IF response$<>answer$ AND tries<4 THEN PR
INT"Wrong, try again."
140 WEND
150 IF response$<>answer$ THEN PRINT "The ans
wer was ";answer$
160 READ question$,answer$
170 WEND
180 PEN 1
190 DATA How many letters are there in the al
phabet?,26
200 DATA What is the capital of the UK?,Londo
n
210 DATA Who won the Wimbledon's Mens Singles
 Title this year?,John McEnroe
220 DATA Who won the last World Cup?,Italy
500 DATA XXX,YYY
```

In this case the first WHILE loop running from line 30 to 170 ensures that questions are asked until the data terminator 'XXX' is reached. The inner WHILE loop allows three tries at the correct answer. The main advantage of using an outer WHILE loop rather than just a FOR . . . NEXT loop to read in the 4 questions is that the WHILE loop enables us to add new questions to the DATA without having to change the program anywhere else. Every time we add a new question we just make sure the DATA statement has a line number less than 500, so that it comes before the data terminator is read. If a FOR . . . NEXT loop was used, we would need to change the stopping value for the loop every time we added a new question.

Exercises

1) Print a rectangle completely made up from asterisks using a nested loop. (This can be done with just a single loop and the use of STRING$, but try it with a nested loop for the practice.)
2) Print an asterisk to all the text coordinate positions which contain an odd coordinate, such as (1,1), (2,7), (5,4), etc.
3) Write a program that reads in the names of pupils and their marks from 12 exams and prints them on the screen together with the average mark. (Use an outer loop to read in the pupil's name, and an inner loop to read in the 12 exam marks and find their average.)
4) Extend the program that draws a row of houses so that it draws several rows, each in a different colour. Add doors and windows to the houses.
5) Write a program that draws a large building with 3 rows of windows, 5 windows in each row, all the windows being of identical size.

Lists

Both of the last two programs of the previous section suffered from a weakness, because exactly the same questions are asked every time the program is run. We can get around this difficulty by reading through a random number of questions before asking each question, but this only creates a different problem — how can we make sure we don't repeat a question?

In one of the questions in the last exercise you were asked to read the names and examination marks for some pupils from data and print them on the screen. If this were a program being used in real life, the data might well be required for further use. It would be helpful, for example, if the computer could sort all the Maths marks into order and print out the pupils' names and their marks from highest to lowest. The micro could well repeat this exercise with the other 11 sets of marks. Yet to sort the marks into order the computer needs to be able to compare marks, and to compare marks it needs to have access to *all* the marks at once, and not just to the marks which have been read in for one pupil.

The above problems show that there is a need at times for the computer to hold *lists* of data, so that it can compare items in

those lists, and, if necessary, rearrange them. With our present knowledge this is an incredibly tedious business. Suppose we just wanted to sort the Maths marks for 15 pupils. The computer needs to have all 15 marks at once so that it can compare and sort them. The first part of the program would just read the names and marks in from data:

```
10 MODE 1
20 READ name1$,mark1
30 READ name2$,mark2
40 READ name3$,mark3
50 READ name4$,mark4
60 READ name5$,mark5
70 READ name6$,mark6
80 READ name7$,mark7
90 READ name8$,mark8
100 READ name9$,mark9
110 READ name10$,mark10
120 READ name11$,mark11
130 READ name12$,mark12
```

. . . and so on, and we haven't even started sorting the marks yet! This approach is hopeless, and thankfully there is a much simpler way.

Instead of reading the data items into separate variables so that all the data is available to the computer at any time, we can choose to read the data into an *array* (a fancy word for a list!):

```
10 MODE 1
20 DIM name$(15),mark(15)
30 FOR count=1 TO 15
40 READ name$(count),mark(count)
50 NEXT
400 DATA Adams,40,Bunter,56,Crane,77,Digby,12
,Easterby,84
410 DATA Fernham,45,Goddard,9,Humphrey,31,May
,92,Norcot,94
420 DATA Reed,11,Stanley,54,Terence,63,Venabl
es,44,Wallace,18
```

Line 20 tells the Amstrad that there are going to be 15 names in our list and 15 marks. We are setting the dimensions of the list, hence the new Basic key-word DIM. Notice that we can have string arrays, such as name$( ) or numeric arrays, such as mark( ). The loop from 30 to 50 reads the 15 names and marks into the two lists. The value in the brackets on line 40 is called the *subscript*, and because the count changes each time through the loop, the subscript also changes. The net result of this is that when the loop finishes, all the data has been stored in the form shown in Figure 29.

Any of the elements in the list can now be referred to by their subscript. Try, for example, the following in immediate mode after running the program:

```
? name$(15)
```

The Amstrad prints 'Wallace', the 15th name in the array. Try:

```
? mark(5)
```

The computer prints 84, the 5th mark in the list. As the names and marks are now held in the list, we can direct the Amstrad to do all sorts of things without having to READ the DATA again, as we can see by adding these lines to the previous program:

```
60 PRINT "I will print out the names of those
 with a mark lower than any mark you give."
70 PRINT
80 INPUT "What is the high mark";highmark
90 PEN 3
100 PRINT:PRINT "Pupils with marks less than"
;highmark
110 PRINT:PRINT"Name","Mark"
120 PEN 2
130 FOR count=1 TO 15
140 IF mark(count)<highmark THEN PRINT name$(
count),mark(count)
150 NEXT
160 PEN 1
```

The loop from 130 to 150 enables the Amstrad to examine the 15

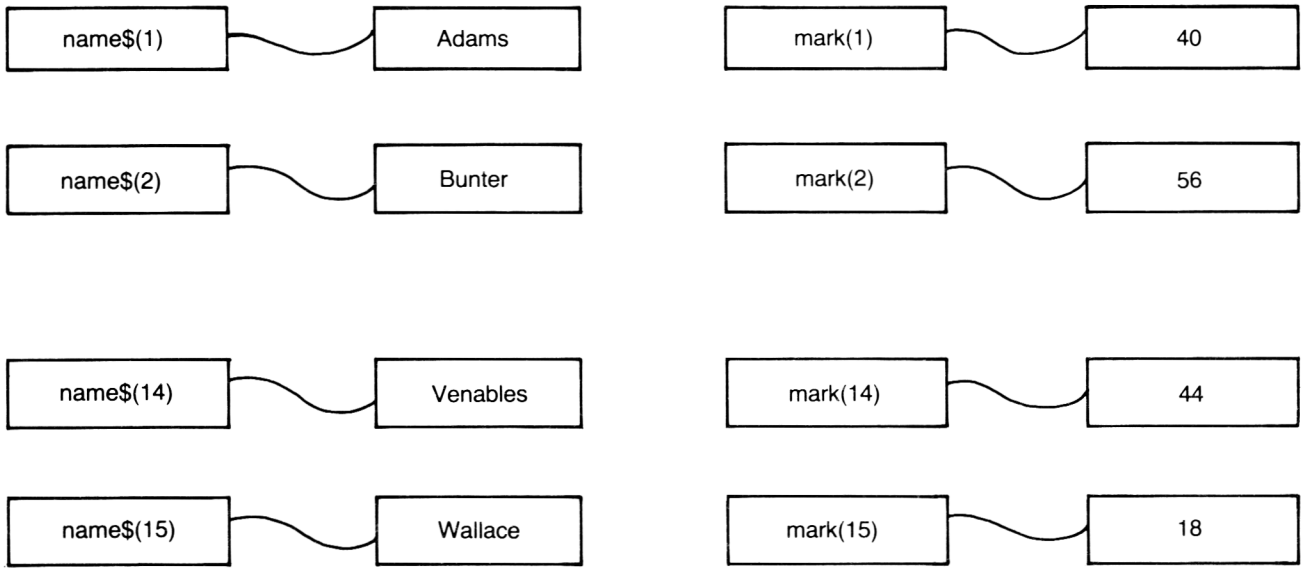

Figure 29 Data storage.

marks in the array one by one, and print out the name and mark for anyone whose mark is lower than the input value. This underlines the link between the two lists. For example, name$(14) refers to the 14th name in the array, and we can find the mark for that pupil by looking at the value in the marks list with the same subscript, mark(14). Individual parts of an array like name$(11) or mark(7) are termed the *elements* of the array.

Although arrays must be referred to by their subscript, the elements of an array are fundamentally exactly the same as normal string or numeric variables. *Any* action we can carry out on a string variable can similarly be carried out on a string array element, and *any* action we can carry out on a numeric variable we can carry out on a numeric array element. Try these, for example:

```
? LEFT$(name$(11),2)
? mark(7)+mark(12)
? name$(1)+name$(6)
```

The array elements behave exactly as if they were ordinary variables. Their only difference lies in the way we refer to them.

We can also use arrays to store a list of results. In an earlier program we threw two dice by generating random numbers. We can now store the result of each throw in an array, and use the array values to set up a display of the results as the program runs:

```
10 MODE 1
20 WINDOW 1,40,1,1
30 WINDOW #1,1,40,2,25
40 INPUT "How many throws";throws
50 DIM diceresult(12)
60 FOR dicetotal=2 TO 12
70 LOCATE#1,1,dicetotal*2
80 PRINT#1,dicetotal
90 NEXT
100 FOR count= 1 TO throws
110 dice1=INT(RND*6+1)
120 dice2=INT(RND*6+1)
130 total=dice1+dice2
140 diceresult(total)=diceresult(total)+1
```

```
150 LOCATE#1,4,total*2
160 PRINT#1,diceresult(total)
170 NEXT
```

Whenever an array is set up as in line 50, all the elements of the array are set to zero. What we are doing here is creating 12 'boxes' which are going to hold the number of times each total of the dice has occurred. (We have an extra box diceresult(1) which we don't use, as you can't get 1 if you throw two dice!)

Lines 60 to 90 print the numbers 2 to 12 spaced one line apart. This just makes it clearer what's happening as each throw takes place.

Every time a new random dice total is created by lines 110 to 130, the contents of that box are increased by 1 in line 140. Figure 30 overleaf shows what happens if a total of 7 was the first throw. The new value of the box is printed at the right position in line 160. If you think the display is rather uninspiring, you might like to change line 160 to:

```
160 PRINT#1,STRING$(diceresult(total),"*")
```

This prints a string of asterisks, the string containing one asterisk for every time that total has occurred. An even more striking display arises if we choose to display the totals as rectangular bars drawn using the graphics commands:

```
100 FOR count= 1 TO throws
110 dice1=INT(RND*6+1)
120 dice2=INT(RND*6+1)
130 total=dice1+dice2
140 diceresult(total)=diceresult(total)+1
141 graphicsy=(24-total*2)*16
142 graphicsx=50
143 MOVE graphicsx+diceresult(total),graphics
y
144 DRAW graphicsx+diceresult(total),graphics
y+16
170 NEXT
```

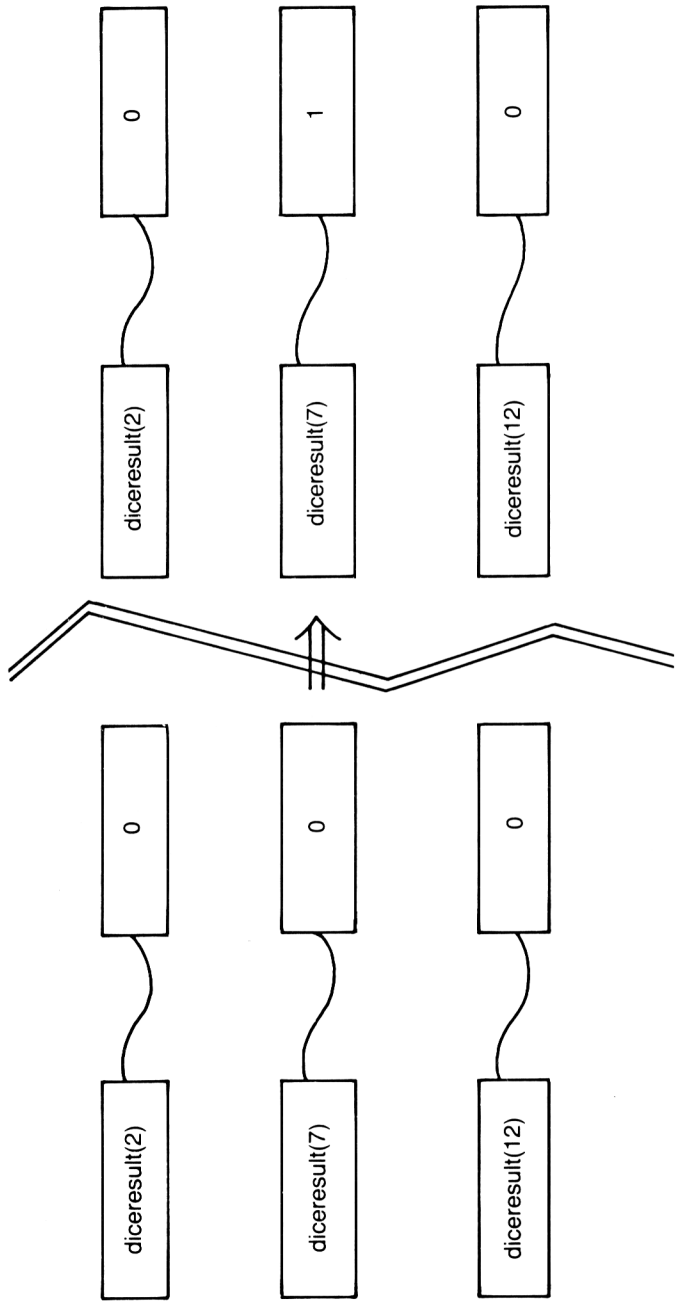

Figure 30 Die throw.

Each time a total occurs, a new graphics line is drawn at lines 143 and 144, and this makes the bars grow as the program runs. The graphicsy position is calculated using the formula introduced in Chapter 3, although a slight adjustment is necessary because the text positions are within a 24 line high window.

Because the graphicsy resolution remains unchanged even when you use a different mode, you can get an even more impressive display in colour by changing just these lines:

```
10 MODE 0
:
142 graphicsx=150
:
144 DRAW graphicsx+diceresult(total),graphics
y+16,total
```

Each bar is drawn in a different colour. The graphicsx coordinate at line 142 has to be moved to the right, because the numbers printed down the left of the screen are wider in mode 0 and would otherwise overlap the start of the bars.

In an earlier chapter we observed the attempts of a fish-and-chip shop owner to use his computer to help calculate customers' bills. This now becomes easier to implement through the use of arrays. First, we must read in the name of the item and its cost:

```
10 MODE 1
20 numberofitems=12
30 DIM nameofitem$(numberofitems),cost(number
ofitems)
40 FOR count=1 TO numberofitems
50 READ nameofitem$(count),cost(count)
60 NEXT
400 DATA "chips,small",0.22,"chips,medium",0.
28,"chips,large",0.35,"cod,small",0.45,"cod,m
edium",0.55,"cod,large",0.65
410 DATA sausage,0.20,pie,0.36,chicken leg,0.
76,beefburger,0.20,pasty,0.28,sausage roll,0.
25
```

Notice that we can use a variable in the DIM statement at 30. Once the data has been read in, the fish-and-chip shop owner can leave his computer running for the rest of the day while it processes customers' orders. This program is complicated enough for it to be worth while planning it out beforehand. The program will need 3 nested WHILE loops:

WHILE the fish shop is open
ask for the first item in the order
WHILE the item in the order is not "xxx"
search for the price for the item
WHILE there are still items to look at in the list
keep examining the list
IF you find the item THEN work out the cost
WEND
If you can't find the item THEN complain
ask for the next item in the order
WEND
print out the overall cost of the order
WEND

The above outline is written in *pseudo-code*. It isn't exactly a program, but it's certainly not English either! This leads to the following program, which continues indefinitely until the [ESC] key is pressed:

```
10 MODE 1
20 numberofitems=12
30 DIM nameofitem$(numberofitems),cost(number
ofitems)
40 FOR count=1 TO numberofitems
50 READ nameofitem$(count),cost(count)
60 NEXT
61 REM set up windows for inputs and bill
70 WINDOW 1,40,21,25.
80 WINDOW #1,1,40,1,20
81 REM first while loop - never-ending!
90 continue=1
100 WHILE continue=1
110 CLS
```

```
120 PEN 1
130 PRINT "Input name of item, end with xxx"
140 PRINT
160 INPUT "Input item, number of portions. ",
item$,portions
170 CLS#1
180 PEN #1,3
190 PRINT #1,"Item"," Number"," Cost"
200 PRINT #1
201 REM second while loop to process each ite
m
210 WHILE item$<>"xxx"
220 count=1
230 found=0
231 REM search for cost while there are any i
tems left in the list and we still haven't fo
und the item
240 WHILE count<=numberofitems AND found=0
250 IF item$=nameofitem$(count) THEN cost=por
tions*cost(count):PRINT #1,nameofitem$(count)
,portions,cost:totalcost=totalcost+cost:found
=1
260 count=count+1
270 WEND
271 REM ask for item again if we couldn't fin
d it
280 IF count>numberofitems AND found=0 THEN P
EN 2:PRINT"I can't find ";item$:PRINT"Please
type it in again.":PEN 1
290 INPUT "Input item, number of portions. ",
item$,portions
300 WEND
301 REM work out overall cost
310 PEN #1,2
320 PRINT #1, "Total cost:",,totalcost
330 PRINT #1,"VAT:",,0.15*totalcost
```

```
340 PRINT #1,"Overall cost:",totalcost*1.15
341 REM and do the next order!
350 WEND
400 DATA chips-small,0.22,chips-medium,0.28,c
hips-large,0.35,cod-small,0.45,cod-medium,0.5
5,cod-large,0.65
410 DATA sausage,0.20,pie,0.36,chicken leg,0.
76,beefburger,0.20,pasty,0.28,sausage roll,0.
25
```

This program serves as a good example of the use of both nested loops and arrays. One feature makes it more difficult to use and also serves to make the program more complicated. Each item on an order must have its name typed in full *exactly* as it has been stored in the list, because the computer uses the name of the item to locate its cost in the cost array. The program can be much simpler if the fish-and-chip shop owner refers to the items by their subscript, i.e. 'chips-small' is 1, and 'pie' would be 8. Edit 160 and delete lines 210–350, substituting these:

```
160 INPUT "Input item number, number of porti
ons. ",item,portions
210 WHILE item<>-99
220 IF item<>-99 AND item<=numberofitems THEN
 cost=portions*cost(item):PRINT #1,nameofitem
$(item),portions,cost:totalcost=totalcost+cos
t
230 IF item>numberofitems THEN PEN 2:PRINT"Th
at's too big as an item number.":PRINT"Type i
t again.":PEN 1
240 INPUT "Input item, number of portions. ",
item,portions
250 WEND
260 PEN #1,2
270 PRINT #1, "Total cost:",,totalcost
280 PRINT #1,"VAT:",,0.15*totalcost
290 PRINT #1,"Overall cost:",totalcost*1.15
300 WEND
```

In this case the number input is used to find the name and cost of the item directly, and there is no need to search through the entire array to see if there is a matching name for the one typed in. This method is obviously more efficient, because we can go to the element in the array *directly*, rather than as previously, when we had to look through the entire list to find the element we were interested in. Of course, the whole exercise is much simplified if we ignore the names of the items completely, and only use their costs in the program. One advantage of including the names and printing them on the bill is that it is easier to see if the wrong number has been input — 7 for sausage instead of 8 for the more expensive pie, for example.

It is worth studying the above program, even though at the moment it may appear to be of only limited interest. The technique used to find an element within the array is of general use, as we shall see in the chapter on files.

Exercises

1) Improve the anagrams program from earlier in the chapter by reading the words to be anagrammed into a string array and choosing the word to be used at random from the array. If you feel able, try to ensure that no word is selected more than once during a run of the program.
2) Write a program to read in the names and telephone numbers of your friends into two string arrays. The program then accepts as input a friend's name and will find and display the telephone number, or give an appropriate message if the name cannot be found.
3) Write a program that 'throws' a coin by generating a random number 1 or 2. If the number is 1, count the throw as a head, otherwise count it as a tail. Display the results of the throws, either by printing 'H' or 'T' at an appropriate position every time the coin is thrown, or alternatively by drawing a graphics bar chart.
4) Read a number of nouns and verbs into two string arrays, and use them to produce brief random sentences of the form 'The (NOUN) (VERB) the (NOUN)', e.g. 'The lion ate the giraffe'.
5) If you are really ambitious, try to create a random limerick!

Chapter Eight

Games and Graphics

Making up your own characters

In Chapter 6, you had a chance to view the entire Amstrad character set. Although there are a wide range of characters provided, it is obviously impossible to cater for all situations, and so the Amstrad allows us to define our own characters using the SYMBOL statement.

All characters are built up on an 8×8 grid, as you can see if you look at Page 2, Appendix III of the User Instructions. To design a new character, it is best to take a piece of squared paper and shade in the grid until you get the effect you want. Below is a tentacled creature which might appear in a game:

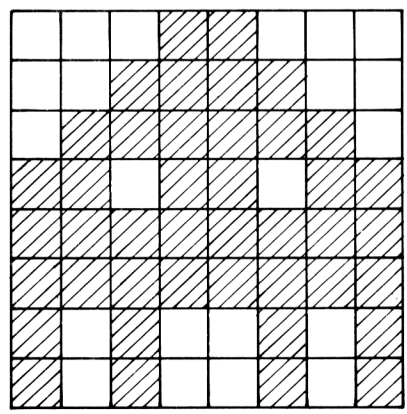

Figure 31 Game character.

Each row of the character can be described by a *hexadecimal number* (a number produced by counting in 16s, rather than the more familiar denary numbers we get by counting in 10s). The hexadecimal system is explained in detail on page 5, Appendix II of the User Instructions. In practice, good old denary numbers

serve just as well, so we shall use these to describe our tentacled friend.

We can get the numbers by adding together the figures at the top of the column for any squares in a row that are shaded:

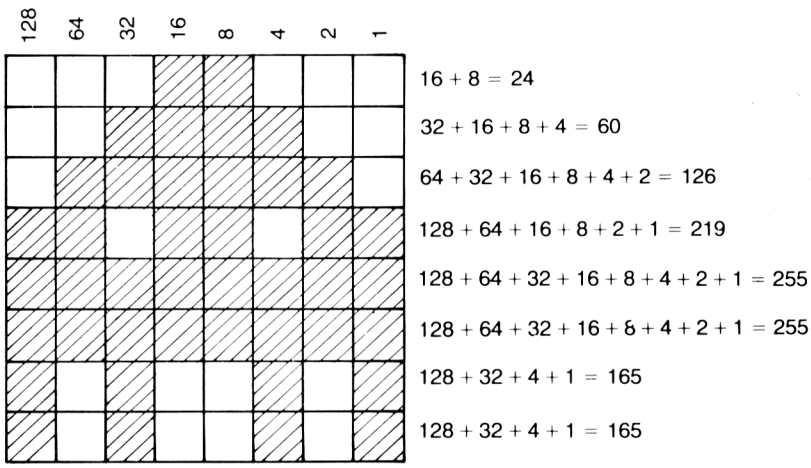

Figure 32 Game character (numbered).

If you understand binary and hexadecimal, you will know why the column headings have the value given, but otherwise you will have to take it on trust that these values always apply.

We can define the character with the 8 numbers and then use it in a program:

```
10 MODE 1
20 SYMBOL 240,24,60,126,219,255,255,165,165
```

Line 20 tells the Amstrad that we want to define a character. The first number after the SYMBOL key-word gives the ASCII code the new character will have, in this case 240. If you type:

```
? CHR$(240)
```

you will see that the redefined character remains after the program ends. Appendix III of the User Instructions tells us that character 240 is an arrow pointing up, but this character has now been redefined. The Amstrad automatically allows us to define 16 new characters using the codes 240 to 255.

We can redefine more than 16 ASCII codes, but only after using another statement:

SYMBOL AFTER 60

This tells the Amstrad that we want to be able in this case to redefine any ASCII codes after 60. (In fact the statement is slightly misleading, because we can also use ASCII code 60 itself.) We could now define nearly 200 characters of our own!

Any previous character definitions are lost, as you can confirm by printing CHR$(240) again. It is back to its original form as an arrow. Having used the SYMBOL AFTER statement, it is now possible to redefine some of the keyboard characters. We could redefine the 'A' as our creature:

SYMBOL 65,24,60,126,219,255,255,165,165

Not to be recommended — it makes typing a program very difficult!

In practice the 16 characters automatically provided are enough for most purposes. We can use the characters in games of our own. The following program demonstrates how easy it is to place the tentacled creature under keyboard control (make sure the CAPS LOCK is off for this program):

```
10 MODE 1
20 PRINT"You can now guide the creature using
 the 'a' and 'z' keys to move it up and down,
 and the ',' and '.' keys to move it left and
 right."
25 PRINT:PRINT"Press 's' to stop."
30 INPUT "Press ENTER when you're ready.",rep
ly$
40 MODE 0
50 SYMBOL 240,24,60,126,219,255,255,165,165
51 REM set up start position
60 xcoord=10
70 ycoord=12
80 newxcoord=10
90 newycoord=12
100 PEN 6
```

```
110 PAPER 1
120 CLS
130 response$=""
140 continue=1
150 WHILE response$<>"s"
151 REM scan keyboard
160 response$=INKEY$
161 REM update creature's position - check it
's not off-screen
170 IF response$="a" AND ycoord>1 THEN newyco
ord=ycoord-1
180 IF response$="z" AND ycoord<25 THEN newyc
oord=ycoord+1
190 IF response$="," AND xcoord>1 THEN newxco
ord=xcoord-1
200 IF response$="." AND xcoord<20 THEN newxc
oord=xcoord+1
201 REM if movement key has been pressed, pri
nt blank in old position
210 IF response$<>"" AND response$<>"s" THEN
LOCATE xcoord,ycoord:PRINT" ":xcoord=newxcoor
d:ycoord=newycoord
211 REM print creature
220 LOCATE xcoord,ycoord
230 PRINT CHR$(240);
240 WEND
250 PEN 1
260 PAPER 0
```

The above skeleton program can form the basis for many games, as we shall see later. A few points to note are that we need to check that the creature is not being moved off-screen, and we must rub the creature out from its last position before we print it to a new one, otherwise we leave a trail of creatures behind! The semi-colon after CHR$(240) is printed is vital. If you leave it out you will find that the screen scrolls when you move to the bottom right corner, which leads to some funny effects.

The keys I have used to control the movement are traditional in computer games, and a lot easier to use than the cursor keys. Some people prefer to use their hands the other way around, with 'z' and 'x' controlling left/right movement and ';' and '/' up/down movement. The program is easily modified to this if you wish.

As we noted a little earlier, CHR$(240) is normally an up-arrow. The character set in Appendix III of the User Instructions shows us that characters 240 to 243 are all arrows, pointing in different directions. We could elaborate on the program above by creating 4 creatures that 'look' in different directions, but for simplicity's sake let's use the characters already available, and change our program so that an arrow is moved around. Before you make the changes and run the program, just type SYMBOL AFTER 240 to reset CHR$(240) to its normal arrow form:

```
50 arrow$=CHR$(240)

170 IF response$="a" AND ycoord>1 THEN newyco
ord=ycoord-1:arrow$=CHR$(240)
180 IF response$="z" AND ycoord<25 THEN newyc
oord=ycoord+1:arrow$=CHR$(241)
190 IF response$="," AND xcoord>1 THEN newxco
ord=xcoord-1:arrow$=CHR$(242)
200 IF response$="." AND xcoord<20 THEN newxc
oord=xcoord+1:arrow$=CHR$(243)

230 PRINT arrow$;
```

Each time the character is moved, the appropriate arrow character is chosen in lines 170 to 200.

Bigger characters

Single characters can be rather small to work with, but it is easy to combine them to produce larger figures. We could define a rather uninspired giant version of the creature by using 4 characters joined together as shown in Figure 33.

One problem with larger figure is that its parts must be printed to 4 different character positions. We can simplify things a little by joining the two characters at the top to create one string, and do

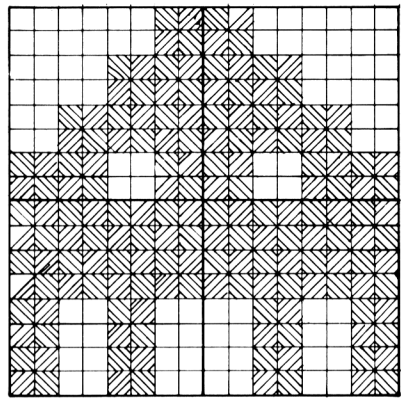

Figure 33 Game character using four characters.

the same for the two characters below to create a second string. We now only need to print using two different LOCATE statements:

```
10 MODE 1
20 PRINT"You can now guide the creature using
 the 'a' and 'z' keys to move it up and down,
 and the ',' and '.' keys to move it left and
 right."
25 PRINT:PRINT"Press 's' to stop."
30 INPUT "Press ENTER when you're ready.",rep
ly$
40 MODE 0
41 SYMBOL 240,3,3,15,15,63,63,243,243
42 SYMBOL 241,192,192,240,240,252,252,207,207
43 SYMBOL 242,255,255,255,255,204,204,204,204
44 SYMBOL 243,255,255,255,255,51,51,51,51
45 top$=CHR$(240)+CHR$(241)
46 bottom$=CHR$(242)+CHR$(243)
51 REM set up start position
60 xcoord=10
70 ycoord=12
80 newxcoord=10
90 newycoord=12
```

```
100 PEN 6
110 PAPER 1
120 CLS
130 response$=""
140 continue=1
150 WHILE response$<>"s"
151 REM scan keyboard
160 response$=INKEY$
161 REM update creature's position - check it
's not off-screen
170 IF response$="a" AND ycoord>1 THEN newyco
ord=ycoord-1
180 IF response$="z" AND ycoord<25 THEN newyc
oord=ycoord+1
190 IF response$="," AND xcoord>1 THEN newxco
ord=xcoord-1
200 IF response$="." AND xcoord<20 THEN newxc
oord=xcoord+1
201 REM if movement key has been pressed, pri
nt blank in old position
210 IF response$<>"" AND response$<>"s" THEN
LOCATE xcoord,ycoord:PRINT"  ":LOCATE xcoord,
ycoord+1:PRINT"  ":xcoord=newxcoord:ycoord=ne
wycoord
211 REM print creature
220 LOCATE xcoord,ycoord
230 PRINT top$;
231 LOCATE xcoord,ycoord+1
232 PRINT bottom$;
240 WEND
250 PEN 1
260 PAPER 0
```

There are a few weaknesses in this modified program, as you may have discovered! The difficulties arise because xcoord and ycoord are actually the text coordinates for the top left character of the

creature only. If we want to prevent it straying off the screen we must amend lines 170 to 200.

Exercises

1) Define a single character of your own, and print it to the screen using the 16 different PENs available in mode 0.
2) Improve the arrow program by defining four more arrow characters to represent the different diagonal movements possible. You will need to use four other keys on the keyboard to allow the user to move the arrow diagonally.
3) Modify the program that moves the large character around so that the creature does not behave in such a peculiar fashion when it is moved to the right-hand column or the bottom line.
4) Define your own larger character, such as a dog or a robot, and write a program to allow it to be moved about on-screen.

Bumping and shooting

We are well on the way to writing a games program. We have found out how to define characters, how to move them around the screen, and how to stop them going off the screen edge. There is one more thing we need to know. All games that involve on-screen movement also require checks for what is at nearby screen positions. If we have to shoot down hordes of attacking aliens, we need to detect if one of the enemy is in the path of our 'bullets'. If we are trying to get through a maze in a race against the clock, we need to know where the maze openings are, otherwise the game will lose all meaning as we move a character around oblivious to the 'walls' in front of it.

We can detect the pen used at a graphics position by using TEST (x,y). The program we wrote in the previous section used text coordinates, but you may recall that in Chapter 3 we worked out a connection between text and graphics coordinates in mode 0:

```
graphicsx = (textx - 1)*32    (use 16 for mode 1, and 8 for
                               mode 2)
graphicsy = (25 - texty)*16
```

This gave the graphics coordinates for the bottom left corner of

any text position. The graphics coordinates for the *centre* of a character position would be:

graphicsx = 16 + (textx − 1)*32 (8,16 for mode 1; 4,8 for mode 2)

graphicsy = 8 + (25 − texty)*16

We now have a means of testing any character position on screen to find its colour, but the text coordinates must first be converted to graphics coordinates. Let's look at an example based on the earlier arrow program:

```
10 MODE 1
20 PRINT "You must try to guide the arrow to
the bottom right as rapidly as possible."
30 INPUT "Skill level (1-10) (1,easy to 10,ha
rd)";skill
40 INPUT "Press ENTER when you're ready.",rep
ly$
50 MODE 0
60 PAPER 1
70 CLS
80 arrow$=CHR$(240)
90 REM set up start position for arrow
100 xcoord=1
110 ycoord=1
120 REM set up walls in red
130 PEN 3
140 FOR count=1 TO skill*12
150 randomxcoord=INT(RND*20+1)
160 randomycoord=INT(RND*25+1)
170 LOCATE randomxcoord,randomycoord
180 PRINT CHR$(233);
190 NEXT
200 REM set up destination in flashing colour
210 PEN 15
220 LOCATE 20,25
221 PRINT"*";
230 PEN 6
```

```
240 response$=""
250 REM loop continues until (20,25) is reach
ed
260 WHILE xcoord<>20 OR ycoord<>25
270 REM scan keyboard
280 response$=INKEY$
290 REM set coords to present position
300 newxcoord=xcoord
310 newycoord=ycoord
320 REM update arrow position - check it's no
t off-screen
330 IF response$="a" AND ycoord>1 THEN newyco
ord=ycoord-1:arrow$=CHR$(240)
340 IF response$="z" AND ycoord<25 THEN newyc
oord=ycoord+1:arrow$=CHR$(241)
350 IF response$="," AND xcoord>1 THEN newxco
ord=xcoord-1:arrow$=CHR$(242)
360 IF response$="." AND xcoord<20 THEN newxc
oord=xcoord+1:arrow$=CHR$(243)
370 graphicsx=16+(newxcoord-1)*32
380 graphicsy=8+(25-newycoord)*16
390 colourpen=TEST(graphicsx,graphicsy)
400 REM if it's a wall, don't move the arrow
410 IF response$<>"" AND colourpen<>3 THEN LO
CATE xcoord,ycoord:PRINT" ";:xcoord=newxcoord
:ycoord=newycoord
420 REM print arrow
430 LOCATE xcoord,ycoord
440 PRINT arrow$;
450 WEND
460 PEN 1
470 PAPER 0
```

This sets up a maze consisting of randomly placed red 'walls', the object being to guide your arrow as rapidly as possible to the bottom right of the screen. Note that you can vary the difficulty of

the game. The number of walls randomly placed by the loop from line 140 to 190 depends on the skill level chosen in line 30.

The walls are printed using PEN 3, and line 410 tests the colour at the new text position. Only if the colour has not been produced by PEN 3 is the arrow's position changed to the new text co-ordinates.

The game is incomplete as it stands, because it is possible to be unable to reach the destination due to the walls blocking all paths. There are several ways around this. One is to avoid printing walls within several character positions of the top left and bottom right corners, as this reduces the chances of a blockage. An alternative is to allow the player to 'blow up' a certain number of walls in the course of the game, by moving directly into them:

```
75 blowups=0
410 IF response$<>"" AND (colourpen<>3 OR blo
wups<2) THEN LOCATE xcoord,ycoord:PRINT" ";:x
coord=newxcoord:ycoord=newycoord
415 IF response$<>"" AND colourpen=3 THEN blo
wups=blowups+1
```

In this version the player can destroy two walls by running the arrow into them. This results in a rather complicated condition in line 410. IF a movement key has been pressed AND either the move is to a non-wall position OR the player can still blow up walls, move the arrow.

The same principles can be applied to a variety of games. The player may have to guide a car around a race track, avoiding slicks of black 'oil'; or try to direct a hungry caterpillar towards green 'leaves' while avoiding the poisonous red 'berries'. In each case TEST would be used to find the colour at the next character position, and the computer would take different courses of action depending on what that colour was.

Let's look at another simple program involving the use of TEST. This is a two-player game where each player must try to avoid bumping into the walls, or each other. As the characters move, they leave a trail behind, so the game gets harder the longer it goes on!

In a game like this it is much easier to break down the program

into shorter sections, so let's begin with setting up the game itself: giving the instructions, and so forth:

```
10 MODE 1
20 PEN 1
30 PRINT "Try to avoid hitting each other, or
 the walls."
40 INPUT"What's your name, left hand player";
player1name$
50 PRINT"Use the z and x keys to turn left or
 right."
60 PRINT"Use the d and c keys to go up or dow
n."
70 PRINT"Your character looks like this: ";:P
EN 3:PRINT CHR$(143)
80 PRINT:PEN 1
90 INPUT"What's your name, right hand player"
;player2name$
100 PRINT"Use the , and . keys to turn left o
r right."
110 PRINT"Use the / and ; keys to go up or do
wn."
120 PRINT"Your character looks like this: ";:
PEN 2:PRINT CHR$(143)
130 PRINT:PEN 1
140 INPUT"Press the ENTER key when you are re
ady.",enter
```

Now we need to draw the rectangular playing area and set up the start positions for both players:

```
150 PAPER 1
160 PEN 0
170 CLS
180 wall$=CHR$(233)
190 bang$=CHR$(238)
200 leftx=1
210 bottomy=20
```

```
220 rightx=40
230 topy=1
240 FOR count=topy TO bottomy
250 LOCATE leftx,count
260 PRINT wall$;
270 LOCATE rightx,count
280 PRINT wall$;
290 NEXT
300 LOCATE leftx,topy
310 PRINT STRING$(rightx-leftx,wall$)
320 LOCATE leftx,bottomy
330 PRINT STRING$(rightx-leftx,wall$)
340 player1x=13
350 player1y=7
360 player1xmove=1
370 player1ymove=0
380 player1$=CHR$(143)
390 player1hit=0
400 player2x=26
410 player2y=14
420 player2xmove=-1
430 player2ymove=0
440 player2$=CHR$(143)
450 player2hit=0
460 PEN 3
470 LOCATE player1x,player1y
480 PRINT player1$;
490 PEN 2
500 LOCATE player2x,player2y
510 PRINT player2$;
```

Finally, we have the game itself, which continues as long as neither player has hit a wall or the trail the other player leaves behind:

```
520 WHILE player1hit=0 AND player2hit=0
530 response$=INKEY$
```

```
540 IF response$="z" THEN player1xmove=-1:pla
yer1ymove=0
550 IF response$="x" THEN player1xmove=1:play
er1ymove=0
560 IF response$="d" THEN player1xmove=0:play
er1ymove=-1
570 IF response$="c" THEN player1xmove=0:play
er1ymove=1
580 player1x=player1x+player1xmove
590 player1y=player1y+player1ymove
600 graphicsx=8+(player1x-1)*16
610 graphicsy=8+(25-player1y)*16
620 colourpen=TEST(graphicsx,graphicsy)
630 LOCATE player1x,player1y
640 IF colourpen<>1 THEN PEN 0:PRINT bang$;:p
layer1hit=1 ELSE PEN 3:PRINT player1$;
650 IF response$="," THEN player2xmove=-1:pla
yer2ymove=0
660 IF response$="." THEN player2xmove=1:play
er2ymove=0
670 IF response$=";" THEN player2xmove=0:play
er2ymove=-1
680 IF response$="/" THEN player2xmove=0:play
er2ymove=1
690 player2x=player2x+player2xmove
700 player2y=player2y+player2ymove
710 graphicsx=8+(player2x-1)*16
720 graphicsy=8+(25-player2y)*16
730 colourpen=TEST(graphicsx,graphicsy)
740 LOCATE player2x,player2y
750 IF colourpen<>1 THEN PEN 0:PRINT bang$;:p
layer2hit=1 ELSE PEN 2:PRINT player2$;
760 WEND
770 LOCATE 1,24
780 IF player1hit=1 AND player2hit=1 THEN PRI
NT"It was a draw.":END
```

```
790 IF player1hit=1 THEN PRINT player2name$;"
 won." ELSE PRINT player1name$;" won."
800 PEN 1
810 PAPER 0
```

There are a few ways things can be speeded up. We could allow the player to turn only left or right which means we need only check for two keys instead of four:

```
540 IF response$="z" OR response$="x" THEN IF
 player1xmove<>0 THEN player1ymove=-player1xm
ove:player1xmove=0 ELSE player1xmove=player1y
move:player1ymove=0
550 IF response$="x" THEN player1xmove=-playe
r1xmove:player1ymove=-player1ymove
```

```
650 IF response$="," OR response$="." THEN IF
 player2xmove<>0 THEN player2ymove=-player2xm
ove:player2xmove=0 ELSE player2xmove=player2y
move:player2ymove=0
660 IF response$="." THEN player2xmove=-playe
r2xmove:player2ymove=-player2ymove
```

The resulting lines are over-complicated, and the time-saving minimal, but you might like to figure out why they work!

Another way to speed up the program would be to avoid the problem of having to convert from text to graphics coordinates. But how do we do that?

Exercises

1) Improve the maze program by adding timing, so that a player can see how rapidly he gets through the maze. Calculate a score based on the skill level chosen and the time taken.
2) Extend the maze program so that some of the walls are printed in a different colour. Every time the player bumps into these walls, the score is reduced.
3) Improve the two-player program by making the game continue until one player has won three games. Print the players' names and their current scores underneath the playing area.

4) Make the two-player program more difficult by designing a new playing area which is not rectangular. Define two new characters of your own to represent the two players in the game.
5) Design four racing car characters to represent the different directions the car might move on the screen. Write a program to enable you to guide the car around a track drawn in black on the screen. If the car moves off the track, it crashes. The car will be easier to control if you include a 'brake' key to bring it to a halt. Time the car, and bring the game to an end when the car reaches the finish line, which is printed in a different colour.
6) Extend your previous program by including obstacles on the track, and adding a second car under another player's control. You might like to send this car the other way around the circuit!

Printing to the graphics cursor

All this fiddling about with conversion from text to graphics coordinates is all very well, but it really points to a weakness in the approach we have used so far. Here we are, with a graphics resolution of at worst 160 × 200, and yet in the games we have so far produced the characters move from one text position to another. The best text resolution is in mode 2, and even here we only have 80 characters per line and 25 lines. What we need is to be able to print characters to a graphics coordinate position. This way we get the best of both worlds, and we will no longer have to worry about converting from text to graphics coordinates.

Text can be printed to the graphics position by using the TAG command (short for Text cursor Attached to Graphics cursor?). This makes the design of graphs and charts much easier, because any text can be printed at the right place by using the graphics coordinates. For example, this program draws a bar chart for 12 months' sales, printing the first letter of the month under the relevant bar:

```
10 MODE 1
20 barwidth=50
30 monthwidth=16
40 leftoverwidth=barwidth-monthwidth
```

```
50 monthposition=leftoverwidth/2
60 xzero=20
70 xmax=639
80 yzero=50
90 ymax=350
100 MOVE xzero,ymax
110 DRAW xzero,yzero,1
120 DRAW xmax,yzero
130 TAG
140 FOR month=1 TO 12
150 READ month$,sales
160 MOVE xzero+(month-1)*barwidth,yzero
170 DRAW xzero+(month-1)*barwidth,yzero+sales
,3
180 DRAW xzero+month*barwidth,yzero+sales
190 DRAW xzero+month*barwidth,yzero
200 MOVE xzero+(month-1)*barwidth+monthpositi
on,yzero-16
210 PRINT LEFT$(month$,1);
220 NEXT
230 DATA January,97,February,130,March,141,Ap
ril,155,May,210,June,276
240 DATA July,240,August,223,September,112,Oc
tober,99,November,84,December,76
```

A point to note is that the *top left* corner of a character is tagged to the graphics cursor, and so you must still be a little careful when deciding where to print text. This is the reason for the calculations in lines 20 to 50: they make sure that the single letter will be printed in the middle of the bar by adding on monthposition to the x coordinate when the bar is drawn. You might like to change barwidth at line 20 to 75 or 150 — the letter is still printed centred beneath the correct bar (although you will find that the bars are now too wide to all fit on the screen!).

TAG can be switched off by using TAGOFF, at which point the printing of text will resume at the position the text cursor was at prior to the TAG command. TAG switches off automatically at the end of a program or if it is interrupted.

One point to note if you intend to use TAG in games is that movement must take into account the limitations of the resolution in whatever mode you're using. It is not possible, for example, to move a single point vertically in any mode. Doing so will cause confusion, as the Amstrad will print the moved character to exactly the same position on-screen. The y movement must always be at least +2 or −2 in any mode. The horizontal movement varies from mode to mode, and you should be guided by Figure 14 as to the minimum x movement which will give a visible change in each mode. This program makes the process clear:

```
10 MODE 1
20 TAG
30 x=300
40 y=200
41 MOVE x,y:PRINT CHR$(249);
50 response$=""
60 WHILE response$<>"n"
70 response$=INKEY$
80 IF response$="z" THEN MOVE x,y:PRINT" ";:x
=x-1:MOVE x,y:PRINT CHR$(249);
90 IF response$="x" THEN MOVE x,y:PRINT" ";:x
=x+1:MOVE x,y:PRINT CHR$(249);
100 WEND
```

You will find that you need to press a key twice to get any visible movement on-screen. If you run the program in mode 0, the situation becomes even worse, because here the minimal visible x movement is +4 or −4, and it takes four key depressions to make the character move!

ORIGIN and relative movement

The sales program can be simplified even further by drawing each bar after setting a new graphics origin. We saw earlier in the book that we could set up text windows, each with their own coordinates relative to their top left corner. In a similar way, we can set up new graphics coordinates by shifting the origin, the point normally at (0,0):

```
10 MODE 1
20 barwidth=50
30 monthwidth=16
40 leftoverwidth=barwidth-monthwidth
50 monthposition=leftoverwidth/2
60 xzero=20
70 xmax=639
80 yzero=50
90 ymax=350
100 MOVE xzero,ymax
110 DRAW xzero,yzero,1
120 DRAW xmax,yzero
121 ORIGIN xzero,yzero
125 MOVE 0,0
130 TAG
140 FOR month=1 TO 12
150 READ month$,sales
170 DRAW 0,sales,3
180 DRAW barwidth,sales
190 DRAW barwidth,0
200 MOVE monthposition,-16
210 PRINT LEFT$(month$,1);
215 xzero=xzero+barwidth
216 ORIGIN xzero,yzero
220 NEXT
230 DATA January,97,February,130,March,141,Ap
ril,155,May,210,June,276
240 DATA July,240,August,223,September,112,Oc
tober,99,November,84,December,76
```

As you can see, the calculations involved in drawing each bar are greatly simplified. Through the loop beginning at 140, the origin is shifted each time to be at the bottom left corner of the next bar to be drawn. The DRAW commands needed to produce the bar are now very straightforward, as the graphics cursor only needs to be moved the distance 'sales' vertically to draw the height of the bar, and 'barwidth' horizontally to draw the width of the bar.

The advantage of using ORIGIN is that it easily enables us to draw figures of the same general shape to any position on the screen. In the sales program each of the bars was a different height, but changing the origin made the commands to draw each bar a lot simpler.

Often, the drawing of any figure can be simplified if that drawing can be described in *relative* rather than *absolute* co-ordinates. Take this program that draws a rectangle:

```
10 MODE 1
20 MOVE 0,0
30 DRAW 200,0
40 DRAW 200,100
50 DRAW 0,100
60 DRAW 0,0
```

The coordinates in the program are *absolute* — in line 30 the DRAW command refers to the position (200,0) on the graphics screen, and no matter how many times the program is run, these coordinates will not change. Yet although this is only a very simple figure, it is easy to see that having to work out all the coordinates for the DRAWs makes things much more difficult. You have probably already noticed this yourself when trying to write short graphics programs in earlier chapters. It would be better if a figure could be drawn by referring to the distance to be moved from the present position, rather than having to work out the absolute coordinates. The *relative* moves to draw the rectangle would look like this:

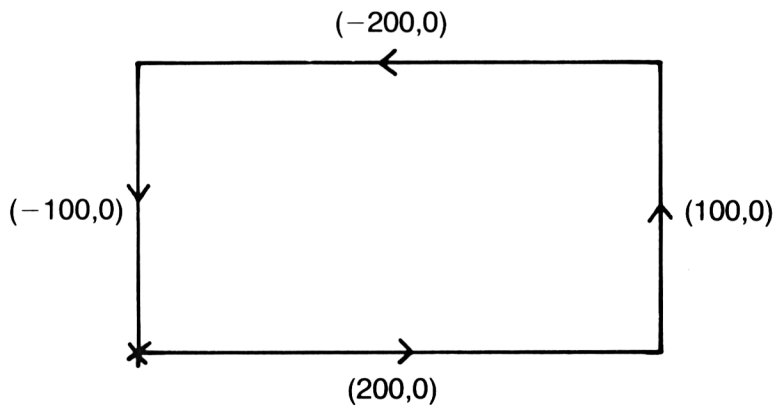

Figure 34 Rectangle.

In fact we can get relative MOVEs and DRAWs without setting up a new origin:

```
10 MODE 1
15 INPUT "Where do you want the rectangle";x,
y
20 MOVE x,y
30 DRAWR 200,0
40 DRAWR 0,100
50 DRAWR -200,0
60 DRAWR 0,-100
```

The DRAWR commands in lines 30 to 60 tell the computer to move relative to the previous position visited. Line 30 causes a line to be drawn from the last point visited to 200 points along and 0 points up. Line 60 causes a line to be drawn from the last point visited to a point 0 points along and 100 points down.

Adding a scale factor enables us to draw a whole range of similar rectangles:

```
10 MODE 1
15 INPUT "Where do you want the rectangle";x,
y
20 MOVE x,y
25 INPUT "How big is it going to be";scale
30 DRAWR 200*scale,0
40 DRAWR 0,100*scale
50 DRAWR -200*scale,0
60 DRAWR 0,-100*scale
```

Inputting a value of 1.5 for the scale gives a rectangle 1.5 times the size drawn before; a value of 0.1 gives a rectangle just a tenth the size. Although they are not demonstrated in the above program, MOVER and PLOTR work in the same way, moving and plotting relative to the last position. The building up of a picture composed of repetitive elements is made much easier by using ORIGIN and/or relative drawing commands.

Exercises

1) Rewrite the maze program from earlier in the chapter so that the arrow is moved to graphics coordinates rather than text coordinates.
2) Print a title for the sales chart in an appropriate position underneath it. Scale the left-hand axis so that it is clear how many sales there were in each month.
3) Write a program that displays the results of throwing two dice 200 times by drawing a bar chart. Label the axes of the graph horizontally and vertically.
4) Write a program that draws the outline of a figure using relative moves and draws:

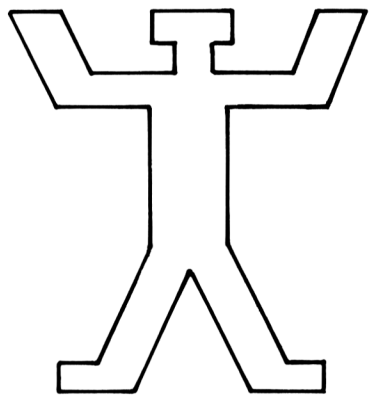

Figure 35 Figure.

5) Extend the previous program so that a series of figures is drawn to form a 'human pyramid', each row being in a different colour.

Changing the INK

So far we have only been able to see some of the colours that the Amstrad can produce. There are only 16 pens available, and yet Figure 18 in Chapter 3 showed us that there are 27 INKs we can use. The Amstrad allows us to change the INK in each PEN so that we can choose any combination of colours for a particular mode.

The number of colours that can be used on-screen at the same time in any mode does *not* change, however. Although we can

have bright red text on a white background in mode 2, these are the *only* colours we could have on-screen at that time. We are *always* limited to 2 colours in mode 2, 4 in mode 1, and 16 in mode 0.

When you switch on or reset the Amstrad, it reverts to mode 1 and uses PAPER 0, which is blue (INK number 1) in all the modes, and PEN 1, which is yellow (INK number 24) in all the modes. Reset the computer now, and type:

```
INK 1,6
```

All the text on-screen changes colour from yellow to bright red instantly. The INK command needs two numbers. The first number is the number of the PEN or PAPER whose ink is to be changed. The second number gives the colour INK which is to be used instead.

The command INK 1,6 told the Amstrad to change the INK in PEN 1 to INK number 6, bright red. *Anything* previously printed or drawn using PEN 1 has its colour changed from the old to the new INK. So to turn all the text blue you need only type:

```
INK 1,2
```

and what was bright red now becomes bright blue. How would we return the text to normal? Perhaps you can work it out for yourself. Type:

```
INK 1,24
```

Normally PEN 1 uses INK 24 in all the modes, as you can see if you look back at Figure 20 in Chapter 3 again.

It is equally easy to change the PAPER colour. At the moment the Amstrad is using PAPER 0, which is blue. Let's change this to white:

```
INK 0,26
```

Perhaps the text is a bit difficult to read. Try:

```
INK 0,6
```

or perhaps:

```
INK 0,0
```

The PAPER in all modes is usually blue, INK number 1. This time I'll leave it to you to turn everything back to normal.

We don't need to have already used a PEN or PAPER colour to change it. Reset the Amstrad and type:

```
INK 3,0
```

Nothing *seems* to happen. If we now go on to choose PEN 3 in mode 1, Figure 20 suggests that text will be printed using INK 6, bright red. But we have just used the INK command to change the INK used by PEN 3 to INK 0, black. Type:

```
PEN 3
```

and all the text is printed in black. Type:

```
INK 3,6
```

and now PEN 3 and all the text it printed is set to INK number 6, bright red. This colour change remains even if you change mode. Try it. It is even possible to set a colour so that it flashes between two different colours! Try:

```
INK 1,3,26
```

to see the text printed using PEN 1 changing from INK 3, red, to INK 26, white, and back again.

One obvious advantage of the INK command is that it enables us to choose any colour combinations from the 27 INKs. Even in a two-colour mode like mode 2 we can brighten things up by using red text on a white background, instead of being restricted to just the colours yellow and blue which are available at switch-on. This program lets you see the more than 700 combinations of colour you now have in mode 2:

```
10 MODE 2
20 FOR x=0 TO 27
30 CLS
40 INK 0,x
50 FOR y=0 TO 27
60 IF x<>y THEN INK 1,y:PRINT"INK ";y
70 response$=""
80 WHILE response$=""
90 response$=INKEY$
100 WEND
110 NEXT
120 NEXT
```

A less obvious advantage is that by setting the PEN colour to that of the background, messages or diagrams can be made to appear on the screen instantly, instead of seeing them as they are drawn:

```
10 MODE 0
20 REM print stick men using different pens
30 FOR x= 1 TO 17 STEP 4
40 FOR y= 1 TO 20 STEP 4
50 PEN 1
60 REM set to background colour
70 INK 1,1
80 LOCATE x,y
90 PRINT CHR$(248);
100 NEXT
110 NEXT
120 PEN 5
130 INK 5,24
140 LOCATE 1,22
150 PRINT"Press any key to see the men.";
160 REM wait for key depression
170 response$=""
180 WHILE response$=""
190 response$=INKEY$
200 WEND
210 INK 1,24
```

We can set a whole series of PENs so that they produce the background colour, which is in INK 1. This program takes the last idea a stage further, and produces a veritable ballet of stick men that prance across the screen as the PEN colours are switched from foreground to background and back again:

```
10 MODE 0
11 REM print stick men using different pens
20 FOR x= 1 TO 17 STEP 4
30 FOR y= 1 TO 20 STEP 4
40 PEN 1
41 REM set to background colour
```

```
50 INK 1,1
60 LOCATE x,y
70 PRINT CHR$(248);
80 PEN 2
81 REM set to background colour
90 INK 2,1
100 PRINT CHR$(249);
110 PEN 3
111 REM set to background colour
120 INK 3,1
130 PRINT CHR$(250);
140 PEN 4
141 REM set to background colour
150 INK 4,1
160 PRINT CHR$(251);
170 NEXT
180 NEXT
190 PEN 5
200 INK 5,24
210 LOCATE 1,22
220 PRINT"Press any key to see the men move."
;
221 REM cycle through colours
230 FOR colours=0 TO 50
231 REM wait for key depression
240 response$=""
250 WHILE response$=""
260 response$=INKEY$
270 WEND
271 REM work out colour to change to backgrou
nd
280 wipe=colours MOD 4
290 IF wipe=0 THEN wipe=4
300 INK wipe,1
301 REM work out colour to change to foregrou
nd
```

```
310 show=(colours+1) MOD 4
320 IF show=0 THEN show=4
330 INK show,24
340 NEXT
341 REM turn PEN 1 back to normal
350 PEN 1
360 INK 1,24
```

Line 230 runs through the INK colours from 0 to 50. We are actually interested only in INK colours 1 to 4, and we get numbers in this range in lines 280 and 310. The MOD command gives the remainder when the number is divided. For example, 17 MOD 4 yields 1, because there is a remainder of 1 when 17 is divided by 4. There is still a slight problem, because the range of numbers we get actually goes from 0 to 3, and not from 1 to 4. 8 MOD 4 gives a remainder of 0, so lines 290 and 320 are included to adjust the result to show a 'remainder' of 4. The net effect is that on each pass through the loop, one colour is switched to 1, the background, and the next colour is switched to 24, the foreground. The program cycles through INKs 1, 2, 3 and 4, and the eye is deceived into seeing the men move across the screen, when they are actually just being switched on and off like light bulbs.

Exercises

1) Write a program that resets the INK colours so that a message is printed in mode 1 on a white background with the text being in purple, black and green.
2) Add a few lines to the maze program from earlier in the chapter so that the screen display is drawn in the background colour and appears instantly once it is complete.
3) Draw a picture of a house against a blue sky, with green grass in front of it. On the touch of a key, switch the colours so that the same scene is shown at night, with light shining from the house windows.
4) Envelop the house in your previous program in flames, produced by drawing lines in a series of suitable flashing colours.

Chapter Nine

Planning a Program

How the Amstrad can help you

We have come some way in the last few chapters, and the programs we have looked at have been longer and more complex than those at the beginning of the book. By now you should be well used to the editing facilities of the Amstrad, but there is still a few other ways the micro can make programming easier that we have not yet discovered. One of the things we shall discuss in this chapter is how the Amstrad can help you track down the errors or 'bugs' that inevitably occur in any developing program.

However, the primary purpose of this chapter is to emphasise the importance of planning programs, and suggest ways in which you can avoid the problems which tend to occur in poorly organised programs. To this end, this chapter contains a games program from its conception to its final appearance as a working, bug-free program.

First, let's make life a little easier for ourselves and discover how the Amstrad can help lazy humans.

The function keys

You will already have found there are some commands that you seem to use excessively. For example, you have typed LIST and RUN with monotonous frequency in the course of this book. Help is at hand! You can LIST or RUN a program by pressing just a single function key. (Load or type in a brief program now so that you have something that can be listed or run!)

We saw in Chapter 6 that every character has associated with it an ASCII code. The codes from 0 to 31 have special meanings; the codes 32 to 127 are for the upper and lower case characters, the numbers, and various other characters; and the codes 128 to 159

can each be associated with a much longer string which will be produced when the appropriate key is pressed.

Although the separate numeric keys appear to be no different to the top row of numeric keys in the characters they generate, the Amstrad sees each of the keys on the numeric keypad as automatically associated with the special ASCII codes 128 to 140:

Key	ASCII code
0	128
1	129
2	130
3	131
4	132
5	133
6	134
7	135
8	136
9	137
.	138
ENTER	139

Figure 36 ASCII codes.

Each of the numeric keys can be set to produce a whole string when it is pressed by first setting the key with a command like:

```
KEY 128,"Hello there Amstrad user!"
```

From Figure 31 we can see that 0 on the numeric keypad is associated with the code 128, and if you now press 0, you will see 'Hello there Amstrad user!' printed at a single key-stroke. Although any key on the keyboard can be set to carry out this kind of action, only the keys on the numeric keypad can be programmed so easily, and so they are known as the *function keys*. Each of these keys can have its functions redefined to produce a particular result when it is pressed.

A more sensible use for the key might be to set it to LIST a program:

```
KEY 128,"LIST"
```

Notice that the key number must be followed by a comma, and anything we want printed must be in inverted commas. If you press 0 again you will find that LIST is printed, but the cursor

stops at the end of the word, waiting for you to press [ENTER]. Sometimes you may want the cursor to stop here, but in this case it would be nice if the Amstrad pressed its own [ENTER] key so that the listing could go ahead.

The action of pressing [ENTER] is represented by one of those ASCII codes from 0 to 31, in fact ASCII code 13. Type:

```
KEY 128,"LIST"+CHR$(13)
```

and press 0 again, and you will see the Amstrad prints LIST and moves the cursor to a new line, listing any program you currently have in the computer.

It's sensible to include an [ENTER] at the start of the key definition. If you've just input a line and you have forgotten to press the [ENTER] key, pressing the 0 at this stage will tack LIST onto the end of the program line — not what you really intended! If you define the key as:

```
KEY 128,CHR$(13)+"LIST"+CHR$(13)
```

the Amstrad will start a new line and then print LIST when you press 0.

The key definition can be extended to include other Basic key-words. It is a good idea to switch to mode 1 before listing a program. For example:

```
KEY 128,CHR$(13)+"MODE 1:LIST"+CHR$(13)
```

Because the numeric keys are still available on the top row of the keyboard, all the keys on the numeric keypad can be redefined so that they produce frequently used commands at the touch of a key. These definitions can be included in a program and saved for future use:

```
10 KEY 128,CHR$(13)+"mode 1:list"+CHR$(13)
20 KEY 129,CHR$(13)+"run"+CHR$(13)
30 KEY 130,CHR$(13)+"save"
40 KEY 131,CHR$(13)+"load"
50 KEY 138,CHR$(13)+"auto"
60 KEY 139,CHR$(13)+"cls"+CHR$(13)
70 KEY 132,"while"
80 KEY 133,"wend"
90 KEY 134,"for"
```

```
100 KEY 135,"next"
110 KEY 136,"read"
120 KEY 137,"data"
```

How you choose to define the keys is a matter of personal choice. The first few definitions in the program above are of general use, whereas lines 70 to 120 set the remaining keys to produce keywords I use a lot in programming. You might have your own preferences.

The great advantage of setting the definitions in a program is that the program can be loaded and run at the start of any programming session, thus your keys are ready for use! You can even have several sets of key definitions, the one you use depending on the type of program you're working on. For example, for graphics work you might define the keys like this:

```
10 KEY 128,CHR$(13)+"mode 1:list"+CHR$(13)
20 KEY 129,CHR$(13)+"run"+CHR$(13)
30 KEY 130,CHR$(13)+"save"
40 KEY 131,CHR$(13)+"load"
50 KEY 138,CHR$(13)+"auto"
60 KEY 139,CHR$(13)+"cls"+CHR$(13)
70 KEY 132,"move"
80 KEY 133,"draw"
90 KEY 134,"plot"
100 KEY 135,"pen"
110 KEY 136,"window"
120 KEY 137,"data"
```

One weakness of the function keys is that unfortunately there is no way of making it clear exactly what each key now does. The only way around this is to stick some labels onto the keys, or to cultivate a good memory! Don't let this put you off the function keys. Unless you are a touch-typist the keys are a valuable resource and you should use them.

There is a limitation on the number of characters that can be stored within the function keys. There can be no more than 120 characters shared between all the keys. You can see this if you edit the last line of the last program so that KEY 137 has a

definition covering 2 lines in mode 1. (Just hold down any key to fill up the line with characters.) When you run the program the Amstrad gives an error message, because the key definition, taken with the others, exceeds 120 characters. Unless you plan on using very long key definitions, this is unlikely to be a problem. You may even regret having no other keys to redefine if your present key definitions don't use all the 120 characters allowed.

Defining other keys

Other keys on the keyboard can also be redefined, but the procedure here is more complicated than the programming of the numeric keypad. First, we must select a key to redefine. Let's use 'A'. Next, we must decide what we wish to define the key to produce. We can *only* redefine it to another ASCII code, not a string of characters as we did for the other keys. Let's redefine 'A' to produce the ASCII code for 'Z':

```
KEY DEF 69,1,90
```

The first number in the definition is the *KEY NUMBER* for 'A'. A full list of all key numbers is given in the User Instructions. The second number is either 0 or 1, and indicates whether we want the key to auto-repeat or not, 1 allowing auto-repeat, and 0 switching it off. The third number is the ASCII code of the character we now want the key to produce. The ASCII code for an upper case 'Z' is 90.

So key 46, 'A', is now set to produce 'Z' every time it is pressed. You will find that this applies whether the CAPS LOCK is on or off. The key still generates an upper case 'Z'. To turn the key back we must redefine it to its lower case ASCII code:

```
KEY DEF 69,1,97
```

where 97 is the ASCII code for a lower case 'a'.

This whole exercise seems rather a waste of time. What advantage does it give us to be able to swap the ASCII codes around on the keyboard and confuse ourselves?

Earlier I said that the codes 128 to 159 can each be associated with a longer string. We have seen that codes 128 to 140 are automatically allocated to the numeric keypad. But what about the codes 141 to 159? How can we get at these codes and use them?

The answer is to use *both* the KEY and KEY DEF commands.

Let's select a key that we don't ordinarily use, and redefine it. I have chosen the key with the £ sign:

```
KEY DEF 24,0,141
```

The £ key will now produce the ASCII code 141 when we press it. Press it now and nothing happens, because 141 is one of those special codes, and we haven't defined it yet! Type:

```
KEY 141"read"
```

Now press the £ key again. 'Read' is printed. We have set up an indirect connection between the keyboard and one of the inaccessible special codes, 141.

In general, the way to extend function key use beyond the numeric keypad is as follows:

1) Select a key on the keyboard that is rarely used.
2) Redefine that key with KEY DEF so it produces one of the special ASCII codes in the range 141 to 159.
3) Define the special ASCII code using KEY to give the longer string you want printed when you press the key.

This principle has been adopted in the following program, which redefines 5 of the rarely used keys to the right of the keyboard:

```
10 KEY 128,CHR$(13)+"mode 1:list"+CHR$(13)
20 KEY 129,CHR$(13)+"run"+CHR$(13)
30 KEY 130,CHR$(13)+"save"
40 KEY 131,CHR$(13)+"load"
50 KEY 138,CHR$(13)+"auto"
60 KEY 139,CHR$(13)+"cls"+CHR$(13)
70 KEY 132,"move"
80 KEY 133,"draw"
90 KEY 134,"plot"
100 KEY 135,"pen"
110 KEY 136,"window"
120 KEY 137,"data"
121 REM pound sign
130 KEY DEF 24,0,141
140 KEY 141,"read"
```

```
141 REM 'at' sign
150 KEY DEF 26,0,142
160 KEY 142,"while"
161 REM left square bracket
170 KEY DEF 17,0,143
180 KEY 143,"wend"
181 REM right square bracket
190 KEY DEF 19,0,144
200 KEY 144,"for"
201 REM slash
210 KEY DEF 22,0,145
220 KEY 145,"next"
```

Another key you may care to redefine is the TAB key to the left of the keyboard, but after that it becomes difficult to find keys which could be spared from their normal duties. One way around this is to use the extended KEY DEF statement to redefine a key to more than one function:

```
KEY DEF 22,0,145,146,147
KEY 145,"if"
KEY 146,"then"
KEY 147,"else"
```

The slash key has now been redefined to produce ASCII code 145 when pressed on its own, 146 when pressed at the same time as [SHIFT], and 147 when pressed with [CTRL]. Try out the different combinations and you will find you can produce a whole IF . . . THEN . . . ELSE statement using just the slash key and a few others!

Exercises

1) Set up your own function keys, and save the program for future use. A few other functions you might like to include are: resetting of PEN and PAPER colours to your favourite combination; setting keys to produce KEY and KEY DEF to speed up redefinition of keys; setting keys to print frequently used variable names like 'count', 'continue', 'number', etc.

Bugs and debugging

Earlier in the book we mentioned the mistakes that are made in programs — the quaintly named 'bugs' that we have to seek out and correct before the program runs properly. Contrary to what you may believe, no program springs into being fully working the first time it is run, unless it is of a very trivial nature. Most programs are the result of pre-program planning, followed by a period of tinkering with the program to iron out problems that the programmer has failed to anticipate in the original program design. If the programmer has not spent enough time in planning, the changes that need to be introduced can be so major that the program needs to be virtually rewritten.

So don't be surprised by bugs — expect them, especially if you are new to computing and still unfamiliar with the commands. The Amstrad itself has facilities to help identify errors, and if you use these facilities wisely you will find it much easier to detect and correct bugs. Let's look at a very short program that demonstrates some of these programming aids:

```
10 ON ERROR GOTO 100
20 pront"Hello"
30 END
100 PRINT"The error occurred at line ";ERL
110 PRINT"The error number is ";ERR
120 END
```

Line 20 contains a deliberate mistake — PRONT instead of PRINT. Line 10 tells the Amstrad that whenever it finds an error anywhere in the program, it should jump to line 100 and obey the commands there. Line 100 prints the line number for the line containing the error, and 110 prints the error number. The latter is a number the Amstrad assigns to indicate the type of error it has encountered. If you run the program, you will find that the error number is 2. Appendix VIII of the User Instructions gives a list of the error numbers and the errors associated with them, and this reveals that error 2 indicates a *syntax error* — a mistake in the 'grammar' of the line.

The Amstrad also prints the offending line so that you can edit it. Curiously, you receive rather more information if you leave

line 10 out, because then the Amstrad actually describes the error rather than just giving the error number! Most other micros don't respond so positively when errors occur, and ON ERROR is more important. The Amstrad actually seems to deal with some errors better without ON ERROR, but it still has some features that make it useful as the following program shows:

```
10 ON ERROR GOTO 1000
20 PEN 0
30 papper 1
40 END
1000 MODE 1
1010 INK 1,24
1020 INK 0,1
1030 PEN 1
1040 PAPER 0
1050 PRINT"The error occurred at line ";ERL
1060 PRINT"The error number is ";ERR
1070 END
```

The problem with line 20 is that the error occurs after PEN 0 has been selected, so the normal error messages given by the Amstrad would be invisibly printed in the background colour! ON ERROR directs the program to line 1000, and lines 1010 onwards restore normal INK, PEN and PAPER values so that the error report is readable. Any error which occurs after a switch in PEN and PAPER colours can be difficult to read, as you may already have discovered!

The errors we have discussed so far have all been syntax errors, but there are many other sorts of errors which may occur in a program. Typing:

```
number="Hello"
```

gives a 'Type mismatch' message which means we are trying to store a string in a numeric variable. These errors are detectable by the Amstrad because they disobey the 'rules' stored in the ROM, and because they are detectable they are relatively easy to correct. Much more of a problem are the *logical errors* made by the programmer.

A logical error is one which occurs as a result of a mistake in the logic of the program. For example, this program adds two numbers together . . . or does it?

```
10 MODE 1
20 number1=5
30 number2=7
40 PRINT number1;" plus ";number2;" gives ";n
umber1*number2
50 END
```

The program runs perfectly well, as none of the lines breaks any of the grammatical 'rules' that the Amstrad follows. Yet it still gives the wrong result, because the programmer has made a logical error on line 40, where the two numbers have been multiplied instead of added. In this case the error is obvious, but suppose the result had not been printed on the screen, but used as the basis for another calculation, and then another, and then a third. . . . By the time it becomes clear that there is a bug somewhere in the program, the source of the problem would be many lines further back, making the original error very difficult to detect.

One typical logical error is where a non-terminating loop is set up:

```
10 ON ERROR GOTO 1000
20 continue=0
30 WHILE continue=0
40 WEND
50 END
1000 MODE 1
1010 INK 1,24
1020 INK 0,1
1030 PEN 1
1040 PAPER 0
1050 PRINT"The error occurred at line ";ERL
1060 PRINT"The error number is ";ERR
1070 END
```

Fortunately it is easy to escape from the program by pressing [ESC] twice, because this even takes precedence over an ON

ERROR statement. But what happens if we have something like this:

```
22 PEN 2
24 INK 2,1
```

where lines 22 and 24 are part of the set-up procedure before the program proper begins with the loop at line 30? Pressing [ESC] now leaves us in a similar position to before, with everything unreadable. Fortunately, a minor change to line 10 solves the problem:

```
10 ON BREAK GOSUB 1000
```

Pressing [ESC] twice breaks from the program, and line 10 tells the Amstrad what to do in these circumstances. It is to jump to line 1000 and carry out the instructions there. You will notice that the messages you get in lines 1050 and 1060 are not very appropriate any more, because the Amstrad has no errors to report, as you have forcibly interrupted the program. Lines like this would be deleted, but lines 1000 to 1040 serve a useful purpose in restoring everything to normal.

Why GOSUB in line 10 and not GOTO as we had for ON . . . ERROR? We shall come to that shortly.

Another useful debugging command is STOP:

```
10.MODE 1
20 FOR x=1 TO 100
30 IF x MOD 5=0 THEN STOP
40 NEXT
```

STOP enables us to stop a program running, have a look at the state of any variables we are interested in, and then continue the program run. If you run the above program the Amstrad prints 'Break in 30'. At this stage you could type 'PRINT x' and the Amstrad will give you the present value of the numeric variable x, 5. Now type CONT and the program continues from where it had left off. The next time 'Break in 30' is reported, x is 10, and so on. The program is a contrived example, but it is often useful to stop a program part-way through a run to try to discover which variable it is that is set to the wrong value.

When debugging a program it is best to set up an ON ERROR

statement right at the start, to ensure any detectable errors are caught and reported. STOP should be inserted at appropriate points if it is clear that a logical error is involved, and the function keys should be set up to print the values of any variables you are interested in.

Organising your program

In the last section we met the ON BREAK GOSUB command for the first time. We failed to use the GOSUB part of the command correctly. A GOSUB command tells the computer to jump to the line number given and carry out all the commands on the lines following, until a RETURN is encountered:

```
10 MODE 1
20 GOSUB 100
30 PRINT"Now out of subroutine."
40 END
100 FOR count=1 TO 10
110 PRINT"Inside subroutine loop count is ";c
ount
120 NEXT
130 RETURN
```

Line 20 directs the computer to the subroutine beginning at line 100. The Amstrad obeys the loop from lines 100 to 120, and at line 130 the RETURN statement tells the computer to jump back to the instruction following the original GOSUB, which in this case occurred at line 20. The computer prints the message at line 30 and the program ends.

Line 40 is rather important since without it the Amstrad will come to the subroutine at line 100 and try to obey it again!

GOSUB is the first instruction we have met that breaks up the progression of a program from first line to last. Admittedly, we have used loops which cause the computer to repeatedly obey some commands, but up until now all the programs we have looked at have begun at the lowest-numbered line and progressed through to the highest-numbered line.

In the last few chapters the programs we have been looking at have been getting more complex. Once a program grows longer

than about 10 lines, it is much easier to cope with its development if the program is broken into much smaller chunks. The use of subroutines enables us to plan a program as a series of sub-programs, each designed to carry out a particular task. Rather than trying to write an entire program and then struggling through the whole thing wondering where the bugs began, we can write the program as several subroutines, and test and debug each one before proceeding to the next.

This may all sound a bit abstract, so let's proceed to an example. We are going to write a program to play a game. It's a popular game, and there have been versions written for most micros.

Planning the program

In the game, the player controls an aeroplane which flies from left to right across the screen, dropping down a line and beginning its flight again whenever it reaches the right-hand side. The plane flies above a number of tall buildings of differing heights. Whenever the player presses the 'b' key, a single bomb is dropped. If it hits a building, the top of the building is blown off. Otherwise the bomb continues to fall until it hits the ground, where it explodes. While a bomb is in flight the player cannot drop another bomb. The object of the game is to land the plane safely by destroying all the buildings. If the player fails to do this, the plane's descending flight eventually causes it to crash into one of the buildings, and the game ends.

From a programming point of view, we can view the game as falling naturally into several separate sections:

1) The instructions for playing the game, the setting of the skill level, etc.
2) The initialisation of the variables to be used in the game, where each variable is given its starting value, and the drawing of the screen display with the buildings and plane in their start positions.
3) The game itself.
4) The after-game comments, plus the player's score, resetting of INKs to normal, etc.

The above division is artificial, and you might well divide 2) into two parts, for example. At this stage we are not too concerned with the detail of the program, more with its overall shape.

We can now begin to write our program:

```
1 ON ERROR GOTO 70
10 MODE 1
11 REM instructions
20 GOSUB 1000
21 REM set up start position for game
30 GOSUB 2000
31 REM play the game
40 GOSUB 3000
41 REM comments and score
50 GOSUB 4000
60 END
70 INK 0,1
80 INK 1,24
90 INK 2,20
100 INK 3,6
110 PEN 1
120 PAPER 0
130 PRINT"Error at line ";ERL
140 PRINT"Error number ";ERR
150 END
1000 REM we'll fill this up later
1100 RETURN
2000 REM we'll fill this up later
2100 RETURN
3000 REM we'll fill this up later
3100 RETURN
4000 REM we'll fill this up later
4100 RETURN
```

That was easy, wasn't it? True, the program doesn't do anything yet, but we have set up its skeleton, and we can now add flesh to the bones. The high numbers for the subroutines are so that we have plenty of spare line numbers once we begin programming. Keeping the major sub-programs at thousand-line intervals makes it easy to remember which line numbers to list when you are working on a particular subroutine. The 'empty' subroutines are ready to be filled, but by setting them up we can ensure that

we can run the program having written very little of it! Run it now.

Notice that we have already set up the ON ERROR statement at line 1 to guard against any disasters with colour changes, and that we have sprinkled REMs liberally about our few lines. When you are working on a very long program, it is easy to forget over several weeks what is the function of different parts of the program. REMs are a valuable reminder. Do note that once the program is complete, REMs make it longer and will slow it down. It's often useful to save two versions of a 'completed' program, one *with* and one *without* the REMs. If any bugs turn up, you still have the REMed version to remind you of the purposes of different parts of the program. The shorter version is the working version, which you will usually want to run as quickly as possible!

Okay, let's tackle subroutine 1000 first. This is going to give the instructions and set the skill level. Let's liven things up by using red text on white:

```
1000 PEN 3
1010 INK 0,26
1020 PRINT"In this game you must try to bomb
all"
1030 PRINT"the buildings so that your plane c
an"
1040 PRINT"land safely. Press b to drop a bom
b."
1050 PRINT"Once a bomb starts falling, you ca
n"
1060 PRINT"only launch another bomb when that
 one"
1070 PRINT"has landed.
1080 PRINT:PRINT"What is your skill level (1,
 easy"
1090 INPUT "to 10, hard)";skill
1100 RETURN
```

If you run the program now, you will receive the instructions and have a chance to choose your skill level. The program then ends. This may not seem very impressive, but the point is that we can

now forget about this part of the program. We have tested it and, on its own, it works.

It may be that a subroutine developed later will reveal an error in this earlier routine. We might have used 'skilll' instead of 'skill' at line 1090, for example. At the moment we don't do anything with 'skill', so an error like that wouldn't come to light until later. But any errors that do occur will be as a result of this subroutine's interaction with another subroutine. We have tested this subroutine and, as far as we can see at the moment, it does the job for which it was written.

Now let's look at subroutine 2000. There are a lot of decisions to be made here, before the program proper begins. We must select an appropriate mode. We must decide if this is the sort of game that requires TAG or whether text coordinates are more appropriate. We must design characters for the plane and the buildings, and decide whether these should be single characters or multiple ones. We must decide what colours we should use, and how much of the screen should be 'sky' and how much 'ground'. We must devise a way of drawing the buildings in their various heights, and pick out a suitable starting position for the plane.

With regard to the rest of the program, we need to have a variable which we can set when a bomb is dropped, so that we know whether to allow the player to drop another bomb or not. We need to set a variable to indicate when the plane has hit a building, so that we know when the game's over.

All of these decisions must be made before we write any lines of the subroutine, and trying to decide at the keyboard will only make the program take much longer to write. There are many possible solutions to these problems, and the following is only an example:

```
1999 REM set up 'sky'
2000 WINDOW 1,40,1,20
2010 INK 0,2
2020 PAPER 0
2030 CLS
2031 REM set up 'grass'
2040 WINDOW #1,1,40,21,25
2050 INK 2,19
```

```
2060 PAPER #1,2
2070 CLS #1
2071 REM define back and front of plane
2080 SYMBOL 240,0,0,192,192,255,0,0,0
2090 SYMBOL 241,0,128,192,224,254,224,192,128
2100 plane$=CHR$(240)+CHR$(241)
2101 REM define one 'block' of building
2110 SYMBOL 242,221,221,255,221,221,221,255,2
21
2120 building$=CHR$(242)
2130 bomb$=CHR$(252)
2140 bang$=CHR$(238)
2141 REM number of buildings depends on skill
2142 REM but add 5 because we must have some!
2150 numberofbuildings=INT(RND*skill+5)
2151 REM print them
2160 FOR number=1 TO numberofbuildings
2161 REM pick random x text coordinate
2170 x=INT(RND*20+10)
2171 REM height depends on skill
2172 REM but add 1 because they must be at le
ast this high!
2180 height=INT(RND*(skill+3)+1)
2181 REM print building block by block
2190 FOR count=0 TO height
2200 y=20-count:PEN 3
2210 LOCATE x,y
2220 PRINT building$;
2230 NEXT
2240 NEXT
2241 REM place plane and print
2250 planex=1
2260 planey=1
2270 INK 1,0
2280 PEN 1
2290 LOCATE planex,planey
```

```
2300 PRINT plane$;
2301 REM set variables to register bomb dropp
ing, plane hit, score
2310 planehit=0
2320 bomb=0
2330 score=0
2340 RETURN
```

Again, we can run the program and confirm that the first two subroutines work. If all the buildings were drawn in the 'sky', or the 'grass' was red, we would tinker with these logical errors until the subroutine worked properly.

We now come to subroutine 3000, which is really the crux of the whole program. The game itself is likely to be so complicated that it may have to be broken down into further subroutines, so for the moment we will only seek to identify the most important aspect of the game — when does it end? Clearly, the game must continue while the plane has not hit a building or 'landed' on the grass. We can choose any position as the final landing position which the plane must reach. I have chosen the bottom right character position:

```
2999 REM keep game going while plane not land
ed or crashed
3000 WHILE (planex<>39 OR planey<>20) AND pla
nehit=0
3010 GOSUB 5000
3020 WEND
3021 REM keep final screen on display for 2 s
econds
3030 oldtime=TIME
3040 WHILE TIME<oldtime+600
3050 WEND
3060 RETURN
```

The condition at line 3000 is rather complicated, and it seems we can really only test whether it's correct or not once subroutine 5000 has been developed. Let's leave that for later, and continue

by completing the last of our main subroutines, that beginning at line 4000:

```
4000 MODE 1
4010 INK 0,1
4020 INK 1,24
4030 INK 2,20
4040 INK 3,6
4050 IF planehit=0 THEN score=score+skill
4060 PRINT"Your score at skill level ";skill
4070 PRINT"was: ";
4080 PEN 3
4090 PRINT score*skill
4100 PEN 1
4110 RETURN
```

This subroutine resets normal INK and PEN colours and gives the score. Unfortunately we can't reach this subroutine at the moment, because the loop in subroutine 3000 never terminates when the program is run. But wait. . . there is a way we can make the loop terminate, simply by setting the variable planex, planey or planehit to the right value. This also gives us the opportunity to make sure the conditions at line 3000 are correct. Let's pop a line inside subroutine 5000 which should terminate the loop:

```
5000 planehit=1
5100 RETURN
```

Run the program. It seems to work, doesn't it? And it proves that subroutine 4000 works, too. Let's just make sure the loop also terminates if planex and planey are set to their 'landing position' values:

```
5000 planex=39:planey=20
5100 RETURN
```

That works too. We have now completed the main body of the program, and can proceed with subroutine 5000, which will control each cycle of the game. Looking at the screen display at the start of the game, it is clear that the first thing we must do is

erase the plane from its old position. Subsequently we will want to:

1) Move the plane along and check its new position will not be on a building.
2) If the plane has hit a building, do a suitable explosion, set planehit to 1, and end the game.
3) If a bomb is dropping, move it down one and do explosions if necessary.
4) If the 'b' key is pressed and no bomb is dropping, then drop one.

I have added a further stage to increase the range of skill needed in the game:

5) Pause for a time dependent on the skill level.

Stages 1) and 5) will take place every time, so we will confine these to subroutine 5000. The remaining stages may or may not occur, and they are each complex enough to warrant a separate subroutine. In the previous chapters some of the IF . . . THEN statements have been very complex, because all the statements following the IF have to be on the same line. If we put all the statements into a separate subroutine the lines will be shorter, and the program easier to understand and debug:

```
4999 REM main game routine - start by erasing
 plane
5000 LOCATE planex,planey
5010 PRINT"  ";
5011 REM move plane along
5020 planex=planex+1
5021 REM move plane down if it's reached righ
t edge
5030 IF planex>39 THEN planex=1:planey=planey
+1
5031 REM calculate graphics coords for new po
sition of front of plane
5040 planegx=8+planex*16
5050 planegy=8+(25-planey)*16
5060 colourpen=TEST(planegx,planegy)
```

```
5061 REM blow plane up if it's a building
5070 IF colourpen<>0 THEN GOSUB 6000:RETURN
5071 REM move bomb if it's dropping
5080 IF bomb=1 THEN GOSUB 7000
5081 REM check keyboard
5090 response$=INKEY$
5091 REM drop bomb only if one isn't dropping
5100 IF response$="b" AND bomb=0 THEN GOSUB 8
000
5101 REM print plane at new position
5110 PEN 1
5120 LOCATE planex,planey
5130 PRINT plane$;
5131 REM pause from 0 to 3/100ths second
5140 oldtime=TIME
5150 WHILE TIME<oldtime+10-skill
5160 WEND
5170 RETURN
6000 REM we'll fill this up later
6100 RETURN
7000 REM we'll fill this up later
7100 RETURN
8000 REM we'll fill this up later
8100 RETURN
```

Notice line 5070 — the first time we have used more than one RETURN in a subroutine. What we are really saying here is that if the plane has hit a building, we want to go to subroutine 6000, do an explosion, and once this is carried out we want the present subroutine to end. After all, the game will be over, and there seems little point in going through stages 3) to 5).

Line 5040 doesn't seem the same as the line we have used before to find the graphics x coordinate. The plane is made up of two characters, and the planex coordinate is actually the text position for the back part of the plane only. If we used TEST to find the colour at the centre of the next text position, we would end up looking at the front of the plane! So line 5040 examines the colour of the character one text position further along.

Run the program again, and you will see that we now have an opportunity to test and debug this plane-move routine before going any further. The plane seems to travel across the screen well enough, but when it hits a building in line 5070 it carries right on through! This is because we have not yet written subroutine 6000, which will set planehit to 1. Let's just add that line now to make sure that this subroutine works:

```
6000 planehit=1
```

Run the program again. The plane hits a building, the game ends, and you get the right sort of score for this very poor effort. We should also make sure that the program behaves properly if we manage to land the plane. The easiest way to do this is to run the program with no buildings being drawn. If you add this line:

```
2143 GOTO 2250
```

the Amstrad will jump to the stated line number avoiding the lines that print the buildings, and we can test the program again. Many programs use the GOTO statement quite indiscriminately, and it makes them hard to follow and difficult to debug. GOTO does have its uses, as we've just seen here, but in general GOTOs should be used sparingly. The one above is one of the few in this book, and here we are using the GOTO primarily for test purposes anyway, and can delete it once we confirm that subroutine 5000 works correctly.

We have begun to develop subroutine 6000, so let's complete it:

```
5999 REM blow plane up
6000 LOCATE planex,planey
6010 PRINT bang$;
6011 REM flash colours
6020 INK 3,6,26
6021 REM indicate plane is hit
6030 planehit=1
6040 RETURN
```

Running the program again confirms that this section works. Now let's turn our attention to subroutine 8000. This drops the

bomb, and we must obviously get this routine working before we can move on to the main bomb-drop routine at 7000:

```
7999 REM 'b' pressed - drop bomb
8000 bombx=planex
8001 REM bomb must be one line further down s
creen than plane
8010 bomby=planey+1
8011 REM don't drop it if the plane's on the
'grass'
8020 IF bomby>20 THEN RETURN
8021 REM indicate the bomb is dropping
8030 bomb=1
8031 REM print bomb to its start position
8040 LOCATE bombx,bomby
8050 PEN 1
8060 PRINT bomb$;
8070 RETURN
```

Run the program to confirm that you can drop a bomb. It remains frozen in the 'sky', and no further bombs can be dropped because 'bomb' is set to 1 to show the bomb is dropping. This shows that line 5100 has the conditions correctly set to prevent us from dropping further bombs while one is in flight, and that subroutine 8000 seems to be working correctly.

Subroutine 7000 moves the bomb down the screen, and will be very similar to subroutine 5000 which moves the plane:

1) Move the bomb down and check its new position will not be on a building.
2) If the bomb has hit a building, do a suitable explosion, set bomb back to 0, and end the subroutine.
3) Otherwise print the bomb at its new text position.

As with 5000, this contains stages 1) and 3) which occur most often, so we will confine these stages to this subroutine but use a separate one for stage 2) which only occurs on some occasions:

```
6999 REM bomb dropping routine - start by era
sing bomb
7000 LOCATE bombx,bomby
```

```
7010 PRINT" ";
7011 REM move bomb down a line to new positio
n
7020 bomby=bomby+1
7021 REM calculate graphics coords for new po
sition of bomb
7030 bombgx=8+(bombx-1)*16
7040 bombgy=8+(25-bomby)*16
7050 colourpen=TEST(bombgx,bombgy)
7051 REM blow bomb up if it's a building
7060 IF colourpen<>0 THEN GOSUB 9000:RETURN
7061 REM print bomb at new position
7070 PEN 1
7080 LOCATE bombx,bomby
7090 PRINT bomb$;
7100 RETURN
9000 REM we'll fill this up later
9100 RETURN
```

You can see by comparing this with subroutine 5000 how similar the two routines are. Run the program again. The bomb is no longer frozen in the sky, but continues to drop until it hits a building or the ground. We can't drop any more bombs because we haven't written subroutine 9000 yet. This will set 'bomb' back to 0 to indicate that the bomb presently dropping has just exploded and we are now free to drop another. Let's just add this line to subroutine 9000:

```
9000 bomb=0
```

Subroutine 7000 seems to work correctly, although the bomb remains on the screen, because its erasure will be part of the routine at 9000. What else do we need to do in this routine?

We want to print an explosion to the bomb's final position, then erase it. If a building has been hit, we need to update the score. The variable 'bomb' must be set to 0 to show that there is no longer a bomb dropping:

```
9000 REM we'll use this line in a moment!
9001 REM print the explosion and quickly flash it
```

```
9010 LOCATE bombx,bomby
9020 INK 1,0,6
9030 PEN 1
9040 PRINT bang$;
9041 REM turn pen back to normal
9050 INK 1,0
9051 REM bomb no longer falling - reset 'bomb

9060 bomb=0
9061 REM add to score if a building was hit
9070 IF colourpen=3 THEN score=score+1
9071 REM erase explosion
9080 LOCATE bombx,bomby
9090 PRINT" ";
9100 RETURN
```

Running the program proves we have a 'bug'. Whenever a bomb hits the 'grass', the screen scrolls upwards! This is because the 'sky' makes up the main text window, and once bomby becomes 21 we are moving out of this window in our attempt to print the explosion on the 'grass'. The Amstrad does its best to accommodate us by scrolling the main window up a line and then printing the explosion!

We can easily get round the problem by treating this as a special case, and adjusting bomby so that the explosion takes place within the main text window rather than just outside it:

```
8999 REM move bomb back into window if it's g
one outside
9000 IF bomby>20 THEN bomby=20
```

A few runs of the program provide convincing evidence that there are no major bugs. It is at this stage that it is easy to be tempted into thinking that the program is bug-free and that it has been fully tested under all possible conditions that can occur. However, this is not the case.

You may already have discovered a logical error in the program. When the plane is on the character position above a building, a dropped bomb blows up several blocks of the

building rather than just one. The flaw lies in subroutine 8000, where we have failed to cater for this possibility. We should check the colour at the bomb's new position even when it has just been dropped.

We could cater for this by duplicating lines 7030 to 7060, which check the colour of the bomb's new position, within subroutine 8000. But there is a better way.

One of the strengths of subroutines is that they provide us with the means of breaking down a program into manageable pieces. Their other great strength is that they give us an easy way of isolating routines in the program which may need to be used many times at different points. Instead of needlessly repeating these lines, we can place them in a subroutine and use them over and over again whenever they are required. This is exactly the situation we have here. We can identify two routines with similar needs:

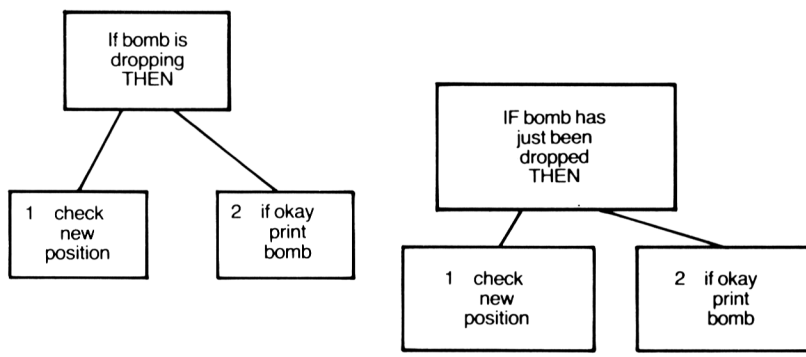

Figure 37 Subroutines.

We will remove the lines that calculate the graphics coordinates and place them in a subroutine beginning at 10000. We also need to call the new subroutine from within subroutines 7000 and 8000:

```
7020 bomby=bomby+1
7021 REM calculate graphics coords for new po
sition of bomb
7030 GOSUB 10000
```

```
7051 REM blow bomb up if it's a building
7060 IF colourpen<>0 THEN GOSUB 9000:RETURN

8030 bomb=1
8031 REM print bomb to its start position
8035 GOSUB 10000
8036 IF colourpen<>0 THEN GOSUB 9000:RETURN
8040 LOCATE bombx,bomby

9100 RETURN
10000 bombgx=8+(bombx-1)*16
10010 bombgy=8+(25-bomby)*16
10020 colourpen=TEST(bombgx,bombgy)
10030 RETURN
```

The quickest way to check that this has cleared up the bug is to add this line, which causes all the buildings to be drawn 19 blocks high:

```
2182 height=18
```

Run the program and drop a series of bombs. Only one block is blown up, so the problem has been eliminated. We should now thoroughly test the whole program again, because correcting one mistake often introduces new ones! Although subroutine 10000 has solved our problem, you can probably see from Figure 38 that subroutine 7000 and 8000 have more in common than just the calculation of the graphics coordinates. See if you can extend subroutine 10000 so that it includes *all* the similar lines that subroutines 7000 and 8000 use.

Making life easy

Unfortunately, in a book devoted to explaining the commands available on the Amstrad, it is difficult to give the business of program planning the space it deserves. The main lesson to be drawn from our example program is that planning the program beforehand makes it easier to identify the separate routines that the program will require. The routines themselves can then be written one by one, each routine being thoroughly tested and

debugged before the next one is developed. Any bugs which subsequently surface will be easier to isolate and correct.

Drawing a diagram often helps program planning. There are many formal methods of laying out a program design, but it is often just as valuable to break the program down into separate stages, box each stage, and connect them with a few lines as a reminder of which routine calls which others. It is a good idea to try to identify where the program lines will simply be a sequence of instructions, where repetition occurs, and where choices must be made, because these are often appropriate points to introduce another routine. (You can see this from the description of the plane move routine which was broken down into five stages and involved three subroutines to cater for plane crashes, bomb drops, etc.)

Accept that errors inevitably occur. The program we have just looked at didn't work first time, but its division into subroutines made it much simpler to debug.

Exercises

1) Improve the game by adding a routine to print the score as the game progresses.
2) Limit the number of bombs the player can use. Print a running total of the number of bombs left at the bottom of the screen. (You will have to base the number of bombs available on the height of the buildings to be destroyed, otherwise it may be impossible to land the plane!)
3) Add routines to the program to enable the player to drop a single 'mega-buster' bomb during the course of the game, by pressing the 'm' key. The mega-buster bomb completely destroys any building it drops on. (Don't forget that you will need to adjust the score according to how many 'blocks' there are in the building.)
4) Rewrite the two-player game from Chapter 8 so that it is made up of a series of subroutines.

Chapter Ten

Sound and Music

Sounds amusing

One feature has been sadly lacking from our games in the earlier chapters and that is the use of sound. The sound facilities on the Amstrad are quite comprehensive. If you're interested in playing music on the micro, you will find that there is plenty of information about the use of sound in the Appendices, and it is relatively straightforward to produce a note of a particular frequency. The following program enables you to experiment with the basic SOUND command:

```
1 ON BREAK GOSUB 500
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
200 GOSUB 500
300 END
500 MODE 1
505 INK 1,24
510 PEN 1
520 PAPER 0
530 END
1000 PAPER 1
1010 CLS
1020 PEN 3
1030 PRINT"This program gives you a chance to
"
1040 PRINT"experiment with the basic sound co
mmand."
```

```
1050 PRINT
1060 PRINT"Press any of the keys 1 to 7 to re
set"
1070 PRINT"the channel."
1080 PRINT
1090 PRINT"Press 'h' to make the note higher,
 and"
1100 PRINT"'l' to make it lower."
1110 PRINT
1120 PRINT"Press 'e' to end."
1130 PRINT
1140 PRINT"The SOUND command will be printed
on"
1150 PRINT"the screen so you can get an idea
of"
1160 PRINT"its effects."
1170 PRINT
1180 INPUT"Press ENTER when you're ready.",en
ter
1190 RETURN
2000 PEN 1
2010 PAPER 0
2020 CLS
2030 CLS
2040 response$=""
2050 tone=478
2060 channel=1
2070 x=4
2080 channelx=10
2090 tonex=12
2100 y=12
2110 INK 1,0
2120 PEN 3
2130 LOCATE x,y
2140 PRINT"SOUND";
2150 WHILE response$<>"e"
```

```
2160 response$=""
2170 WHILE response$=""
2180 response$=INKEY$
2190 WEND
2200 code=ASC(response$)
2210 IF code>48 AND code<56 THEN channel=code
-48
2220 IF code=108 AND tone<4000 THEN tone=tone
+1
2230 IF code=104 AND tone>100 THEN tone=tone-
1
2240 PEN 1
2250 LOCATE channelx,y
2260 PRINT channel;
2270 PEN 3
2280 LOCATE tonex,y
2290 PRINT tone;
2300 SOUND channel,tone
2310 WEND
2320 RETURN
```

The sound is generated in line 2300. The two variables after SOUND are both *compulsory parameters* and they must always be included. The first variable tells the Amstrad which *channel* to use for the sound. There are 3 channels available, each with a slightly different 'voice', known as the A, B and C channels. Each channel is represented by a number in the SOUND command. The number can range up to 255:

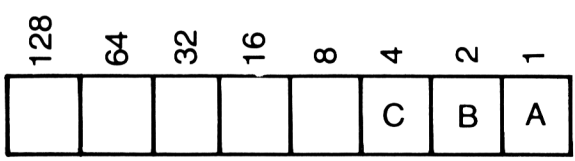

Figure 38 Sound command numbers.

You may recognise some similarities between this layout and that for using the SYMBOL command. We can tell the Amstrad to play a note on channel A by using 1 in the SOUND command. If

we want the note to play on channels A and C, we would use the number 5 instead, adding together the numbers above the channels in Figure 39, just as we did for the SYMBOL statement. Playing a note on all 3 channels, A, B and C would require the number 7.

Higher numbers enable us to synchronise the channels if notes reach them at different times. A note might be sent to channel A, and if a particular channel number was chosen the Amstrad would not play that note immediately, but wait for another SOUND command that sent a note to channel B, so that the notes could be played together. The synchronisation and queuing of notes is relatively straightforward, but this look at the complex subject of sound is intentionally brief, and I shall leave it to those of you with a keen interest in the subject to pursue it further.

The program confines itself to allowing us to play a note on 1, 2 or 3 channels simultaneously. The second part of the sound command sets the *tone* of the note. The tone is indicated by a number in the range 0 to 4095. The lower the number, the higher the pitch of the note, and vice versa. Appendix VII of the User Instructions gives a very comprehensive list of the tone numbers to use to produce notes in a particular octave, and you should look to this for guidance if you have musical inclinations. The value of tone is set to 478 in line 2050 — this is equivalent to middle C.

You can vary the channels and the tone while the program is running. The tone is varied in steps of 1 in lines 2220 and 2230. You may care to adjust these figures so that the pitch changes more rapidly. At present lines 2220 and 2230 only allow tone numbers in the range 100 to 4000, to ensure that if you edit the lines and make a large change in tone, the tone will not move out of the range 0 to 4095. Again, you could change these limits so that you can hear the sound for tone number 4095, for example. Each note automatically sounds for 0.2 seconds.

The program constantly scans the keyboard, line 2200, and adjusts the channel and tone if appropriate keys are pressed, lines 2210 to 2230. The ASCII codes are used because it is simpler to check for the numbers 1 to 7 in line 2210 by seeing if the code is in the correct range. An alternative approach which avoids the ASCII codes is to use a line of the form:

```
2210 IF INSTR(response$,"1234567")>0 THEN
        number=VAL(response$): channel=number
```

This line ensures that the key pressed is valid, and converts it into a numeric value if it is an allowed value. This just shows once again that there is always more than one way of solving a programming problem!

The current values of the channel and tone are constantly displayed so that you can get an idea of how these values change the resulting sound. As with most of the examples in this chapter, it is easier to understand what varying the channel does when you have run the program a few times and listened to the results.

Exercises

1) Write a program that READs the channel and tone for a simple tune from DATA statements, and plays the tune by repeatedly using the SOUND command.
2) Add suitable SOUND commands to the two-player game from an earlier chapter. Use a different pitch of sound to show when each player moves. You might also like to vary the pitch depending on the direction of movement.
3) Extend the previous program by making the computer play a tune once a player bumps into a wall or other obstacle.

The optional parameters

The SOUND command can be extended by the inclusion of up to 5 *optional parameters*. The extra parameters specify how long the note should last (its duration); how loud it should be (its volume); whether a volume envelope should be used to vary the volume; whether a tone envelope should be used to vary the tone; and how much 'noise' should be added to the tone. We shall investigate the volume and tone envelopes later. For the moment we will set them both to zero so that we can observe the effects of varying the other parameters. Add these lines to the previous program:

```
1121 PRINT
1122 PRINT"Press 'i' to increase duration."
1123 PRINT"Press 'd' to decrease duration."
1124 PRINT
1125 PRINT"Press 'q' for quiet (lower volume)
"
```

```
1126 PRINT"Press 'r' for rowdy (higher volume
)."
1127 PRINT
1128 PRINT"Press 'n' for more noise."
1129 PRINT"Press SPACE BAR for less."
2061 duration=20
2062 volume=12
2063 noise=0
2070 x=4
2080 channelx=10
2090 tonex=12
2091 durationx=16
2092 volumex=20
2093 noisex=27
2141 LOCATE 23,y
2142 PRINT"0 0";
2231 IF code=105 THEN duration=duration+1
2232 IF code=100 THEN duration=duration-1
2233 IF code=113 AND volume>0 THEN volume=vol
ume-1
2234 IF code=114 AND volume<15 THEN volume=vo
lume+1
2235 IF code=110 AND noise<31 THEN noise=nois
e+1
2236 IF code=32 AND noise>0 THEN noise=noise-
1
2320 PEN 1
2330 LOCATE durationx,y
2340 PRINT duration;
2350 PEN 3
2360 LOCATE volumex,y
2370 PRINT volume;
2380 PEN 1
2390 LOCATE noisex,y
2400 PRINT noise;
2410 SOUND channel,tone,duration,volume,0,0,n
oise
```

```
2420 WEND
2430 RETURN
```

You can now change the other parameters while the program is running, and their values will be displayed on the screen. I tried to choose suitable keys to make it easy to change the parameters, but inspiration failed me, and you may prefer to list the keys to use on the screen so you know which ones to press!

The duration of the note is measured in hundredths of a second, so line 2061 sets the duration initially to the standard duration of 0.2 seconds which normally applies if the optional duration parameter is not used. The volume can vary from 0 (silence) to 15, with the standard volume of 12 being set in line 2062. The noise is an extra which adds some crackle to the note. The addition of noise is indicated by a number in the range 0 to 31, with the standard being 0, no noise. Adding noise might seem rather counter-productive, but it's useful for games.

Exercises

1) Introduce some sound effects into the bombing program from the last chapter. Use a note with a suitable duration and addition of noise to give a bomb explosion. Play a note every time the plane moves, but make the duration of the note dependent on the skill level, so that the game is not slowed down. Increase the tone every time the plane drops down a line on the screen.

The volume envelope

The volume envelope can be used to change the volume of a sound as it is played. Up to 15 envelopes can be defined, and once an envelope has been set up, the computer stores the details of that envelope and it can be used at any time up until the computer is turned off. There is therefore usually no need to use an envelope command for a particular envelope more than once in a program, because once that envelope has been created it can be used over and over again.

Normally the SOUND command contains all the parameters affecting the sound, but including an envelope number of 1 to 15

means that the sound varies according to the envelope previously defined. For example, SOUND 1,478,20,12 would play middle C on channel A for 0.2 seconds at volume 12. But SOUND 1,478,20,12,3 would vary the volume of that note depending upon volume envelope 3, which would have been defined earlier.

Add these lines to the program, and you can set up a volume envelope at the start and observe its effects on the notes:

```
15 GOSUB 600
600 PRINT"This subroutine enables you to set
up"
610 PRINT"a volume envelope before going on t
o"
620 PRINT"try different notes with it."
630 PRINT
640 INPUT"Envelope number (1-15)";volenvnumbe
r
650 INPUT"Step count (0-127)";volstepcount
660 INPUT"Step size (-128 - 127)";volstepsize
670 INPUT"Pause (0-255)";volpause
680 ENV volenvnumber,volstepcount,volstepsize
,volpause
690 PRINT
700 INPUT "Press ENTER to continue",enter
710 RETURN
2121 LOCATE x,y-2
2122 PRINT"ENV ";volenvnumber;",";volstepcoun
t;",";volstepsize;",";volpause
2410 SOUND channel,tone,duration,volume,volen
vnumber,0,noise
```

The overall description of the volume envelope is:

ENV envelope number, step count, step size, pause

The volume envelope number allocates the definition that follows to a particular number so that it can be referenced from within the SOUND statement.

The step count indicates how many steps are to be involved in the variation of the volume.

The step size indicates how much the volume should change at each step.

The pause indicates what length pause in hundredths of a second there should be between each step.

For example, ENV 1,100,−5,1 would indicate that any sound using this envelope would have its volume falling in steps of 5 over 100 steps, with a pause of one hundredth of a second between steps. One consequence of using a volume envelope is that we must be careful with the SOUND definitions that use that envelope. If we used SOUND 1,478,20,12,1 we would lose part of the effect of the ENV command because the pauses alone take up 1 second (100 steps of 1/100th of a second). Yet we have specified a duration of 20/100ths of a second in the SOUND command, and as this takes precedence we would not hear the full effect of the ENV command.

The best way to get to grips with the ENV command is to run the program above. Define a single volume envelope and set up a suitable duration in the SOUND command so that you can get a good idea of how the volume envelope affects the resulting note. Then try changing just a couple of the parameters, say for the tone and noise. If you get an interesting sound, it's worth making notes of both the ENV and SOUND commands that produced it, so that you can use them at a later date. Alternatively, extend the program so that you can store any values in which you are interested in an array just by pressing a suitable key. The Amstrad could print out these values when you finish with the program. (In the next chapter you will learn how you might save this data on a file, so that you could build up your own sound library.)

The tone envelope

This operates in a similar way to the volume envelope, with a few minor variations. The overall description is:

ENT envelope number, step count, step size, pause

The envelope number again is a number from 1 to 15, but in this case negative numbers can also be used to show that the envelope is to be repeated.

The step count indicates into how many steps the tone is to be divided.

The step size shows how much the tone is to change at each step.

The pause is the time in hundredths of a second between each tone step.

Add these lines to the program to enable you to experiment with the tone envelope:

```
16 GOSUB 800
800 PRINT"This subroutine enables you to set
up"
810 PRINT"a tone envelope."
820 PRINT
830 PRINT"Envelope number (1-15, using negati
ve"
835 INPUT"values makes it repeat)";toneenvnum
ber
840 INPUT"Step count (0-239)";tonestepcount
850 INPUT"Step size (-128 - 127)";tonestepsiz
e
860 INPUT"Pause (0-255)";tonepause
870 ENT toneenvnumber,tonestepcount,tonesteps
ize,tonepause
880 PRINT
890 INPUT "Press ENTER to continue",enter
900 RETURN
2123 LOCATE x,y-4
2124 PRINT"ENT ";toneenvnumber;",";tonestepco
unt;",";tonestepsize;",";tonepause
2410 SOUND channel,tone,duration,volume,volen
vnumber,ABS(toneenvnumber),noise
```

It is worth keeping the volume envelope fixed, or not using it at all, until you are reasonably sure that you understand the type of effect the tone envelope can have on a sound.

Chapter Eleven

Files

Keeping records

In this chapter we will look at how you can use your computer to store and retrieve data in the form of a *file*. A computer file is much like a manual file — it consists of a series of *records* about related subjects. A file of customer names for a small business might contain a record for each customer, each record itself being broken down into *FIELDS* containing information about the customer: the name, address, what was ordered, how much it cost, and so on.

The file that the computer creates is saved, and can subsequently be loaded back at a later date and examined or amended. Typically there are only a few operations we can carry out:

1) creating or *WRITING* a record to the file;
2) amending or *UPDATING* a record on the file;
3) retrieving or *READING* a record from the file;
4) adding a record to the file;
5) deleting a record from the file.

The speed of disk drives makes it possible with the CPC 664 to load individual records into memory, modify them, and then save them back to disk. On the cassette-based 464 this is not a feasible approach. Each time we wished to change a record we would need to swap cassettes so the new record could be saved for future use, and this swapping of cassettes is far too tedious.

The simplest approach is to load the contents of an entire file into memory, look at or change any records we are interested in, and then, if any amendments have been made, create a completely new file containing the changed records.

On the CPC 464 this will involve the use of only two cassettes, one to hold the present file which is read into the computer, and

another to hold any new file created as a result of the deletion, insertion, or amendment of records. The cassettes will only be swapped once instead of many times if individual records were being loaded and saved.

On the CPC 664 we could make do with a single file. The original file could be loaded, modified, and then saved back to the disk with the same name as before. The new file would thus completely replace the old one.

Although on the face of it this seems a perfectly reasonable approach it does carry a hidden danger. If the new file contains any errors, be they human or computer-induced, the old file is no longer available for cross-checking purposes. If the file held the names of customers who owed money, the loss of accurate information could prove catastrophic.

Many firms keep a historical series of files as security against this type of problem. You may not be so concerned for the safety of your own files, but in general it is wise to retain at least the previous version of the file, perhaps naming it by the date on which it was created, and deleting it only when the 'new' file is itself modified and becomes the 'old' file.

These ideas relating to files will be easier to understand if we look at a simple example, involving a file containing the names and telephone numbers of your friends.

Creating the file

The first program we shall look at is used to create the file. (CPC 664 owners should modify the PRINT statements at lines 1130–1160 and 2160 so that they refer to disk rather than cassette.)

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
60 END
1000 PRINT"You can use this progam to create
a"
1010 PRINT"file of names and telephone number
s."
1020 PRINT"The technique is a general one, an
d"
```

```
1030 PRINT"you can substitute any data of you
r"
1040 PRINT"own that you want to save."
1050 PRINT
1070 PEN 2
1080 PRINT"In a moment you will be asked for
the"
1090 PRINT"names and telephone numbers of you
r"
1100 PRINT"friends. Type them in, and when yo
u"
1110 PRINT"want to finish, type xxx,xxx ."
1120 PRINT
1130 PRINT"The data will be recorded on casse
tte."
1140 PRINT"Please insert a blank cassette. Yo
u"
1150 PRINT"will be told when to press the REC
ORD"
1160 PRINT"and PLAY buttons."
1170 PRINT
1180 INPUT"Press ENTER when you are ready.",e
nter
1190 RETURN
2000 CLS
2010 INPUT"What do you want to call this file
";file$
2020 OPENOUT file$
2030 PRINT"Please type in a name and telephon
e"
2040 PRINT"number."
2050 PRINT
2060 PEN 3
2070 INPUT"Name, telephone number";name$,tele
phone$
2080 WHILE name$<>"xxx"
```

```
2090 WRITE #9,name$,telephone$
2100 PRINT
2110 INPUT"Name, telephone number";name$,tele
phone$
2120 WEND
2130 CLOSEOUT
2140 PRINT
2150 PEN 1
2160 PRINT"Your data has been saved onto tape."
2170 PRINT"You can now read the data from the
"
2180 PRINT"file using the next program."
2190 RETURN
```

Subroutine 1000 explains the procedure, and subroutine 2000 creates the file and writes the records to it. The process is straightforward enough. The file is named in 2010, and opened ready to receive output in 2020 with the OPENOUT command.

Any names and phone numbers you type at the keyboard are written to the file by the loop from 2080 to 2120. Line 2090 writes the name and phone number out to the file using the WRITE #9 command.

Line 2130 is very important. It tells the Amstrad that we have finished outputting data to the file, and we now want to close the file.

Once the file has been created, we can switch the Amstrad off, secure in the knowledge that the data that is wiped from the RAM has been saved and can be used again in the future.

Reading from the file

We can extend the previous program so that it allows us to create a file and then immediately read it back into memory. This is not likely to be done very often in practice, but at least it demonstrates that the file does actually exist! CPC 664 owners should modify the print statements at lines 3060 and 3070 so that they refer to disk.

```
40 GOSUB 3000
50 GOSUB 4000
3000 PEN 2
```

```
3010 PRINT
3020 PRINT"You can use this progam to read a"
3030 PRINT"file you have created back into me
mory."
3040 PRINT
3050 PEN 2
3060 PRINT"Please put the cassette containing
"
3070 PRINT"your file into the cassette record
er."
3080 PRINT
3090 INPUT"Press ENTER when you are ready.",e
nter
3100 RETURN
4000 CLS
4010 INPUT"What's the file called";nameoffile
$
4020 OPENIN nameoffile$
4030 PRINT"Here come all those names and phon
e"
4040 PRINT"numbers!"
4050 WHILE NOT EOF
4060 INPUT #9,friend$,phone$
4070 PRINT
4080 PRINT friend$,phone$
4090 WEND
4100 CLOSEIN
4110 PRINT
4120 PRINT"That's all folks!"
```

Before the computer can load the file, it needs to know the file name, line 4010. In this case we are reading from the file, and we want to input data from it, so we use OPENIN at line 4020.

It's not very likely that we will remember how many records are recorded on the file, so how can we tell when the file ends as the Amstrad reads the records in?

The computer takes care of this itself. It indicates that it has reached the end of the file by setting EOF to a certain value. We do

not have to concern ourselves with exactly how this works, but we exploit it in the program with lines 4050 to 4090. These lines are the equivalent of:

WHILE it's not the End Of File (EOF)
input a record from the file
print the record
WEND

INPUT #9 at 4060 tells the Amstrad to accept the input from the file, instead of from the keyboard as usual.

Having read the data from the file, we do face one problem — we've lost all the records except the last one! Each time a record is read in at 4060, the previous values for the variables friend$ and phone$ are erased by the new values. How can we arrange the program so that the input records are saved for examining at our leisure?

We have to read the records into arrays:

```
4000 CLS
4010 INPUT"What's the file called";nameoffile
$
4015 DIM friend$(100),phone$(100)
4016 count=1
4020 OPENIN nameoffile$
4030 PRINT"Here come all those names and phon
e"
4040 PRINT"numbers!"
4050 WHILE NOT EOF
4060 INPUT #9,friend$(count),phone$(count)
4070 PRINT
4080 PRINT friend$(count),phone$(count)
4085 count=count+1
4090 WEND
4100 CLOSEIN
4110 PRINT
4120 PRINT"That's all folks!"
```

Line 4015 sets up arrays big enough to hold 100 names and phone numbers. In practice the arrays could be much bigger, provided enough RAM is left for the program still to run.

The program uses a count to keep track of the number of records input and that same count is used to store each successive name and phone number in a different element in the arrays friend$( ) and phone$( ). When the file has finished loading, all the information in the file will have been transferred to the arrays in memory.

Earlier in the book we saw how an array could be searched for a particular item. In the fish-and-chip shop prices program, the user input the name of the item ordered, and the computer searched through the array until it found the item required. The price of the item came from the corresponding element in the array for the prices.

We could use a similar technique to find the phone number for a friend, once we have input the name. Rather than try to extend our program still further, let's reorganise it into a more coherent form.

A menu-driven program

The program that follows uses the phone numbers file to demonstrate how you can search a file or amend records in the file and subsequently create a new file to hold those records. Although the program is designed to operate only on the phone numbers file, it is easily adapted and extended so as to be usable with any file.

This program requires the user to make a choice from several courses of action, and in circumstances like this the simplest way to present those choices is in the form of a *menu*. The menu prints the possible actions to the screen and the computer invites the user to choose one:

```
10 MODE 1
20 GOSUB 1000
30 END
1000 response$=""
1010 fileload=0
1020 WHILE response$<>"5"
1030 PEN 3
1040 PRINT
1050 PRINT"Choose 1 - 5:"
```

```
1060 PEN 1
1070 PRINT:PRINT"1. Load file"
1080 PRINT:PRINT"2. Save file"
1090 PRINT:PRINT"3. Search file"
1100 PRINT:PRINT"4. Amend file"
1110 PRINT:PRINT"5. End"
1120 PRINT
1130 PEN 3
1140 PRINT"Which?"
1150 WHILE response$="":response$=INKEY$:WEND
1160 IF response$="1" THEN GOSUB 2000
1170 IF response$="2" THEN GOSUB 3000
1180 IF response$="3" THEN GOSUB 4000
1190 IF response$="4" THEN GOSUB 5000
1200 IF response$<>"" THEN CLS
1210 IF response$<>"5" THEN response$=""
1220 WEND
1230 RETURN
```

In a more complex program, the choice might be between a series of sub-menus, and after selecting a sub-menu the user would have to make further choices once that sub-menu was on display. Generally, choices are not made by laboriously typing in long sentences but by the depression of a single key. The choices are often numbered, as in this case.

Subroutine 2000 loads the file:

```
2000 CLS
2010 INPUT"What is the file name";file$
2020 count=1
2030 IF fileload=1 THEN ERASE name$,phone$
2040 DIM name$(100),phone$(100)
2050 OPENIN file$
2060 WHILE NOT EOF
2070 INPUT #9,name$(count),phone$(count)
2080 count=count+1
2090 WEND
2100 CLOSEIN
```

```
2110 PRINT
2120 PEN 2
2130 PRINT"The file has been loaded."
2140 INPUT"Press ENTER to return to menu.",en
ter
2150 fileload=1
2160 RETURN
```

This is similar to the last section. Line 2030 is included so that if we want we can come back and use this routine again and load another file. When the routine has been carried out once, the Amstrad knows from line 2040 that the array name$( ) and phone$( ) contain 100 elements. If we use the routine again, the Amstrad objects to 2040, because we have already dimensioned the arrays and the computer knows how big they are. Line 2150 sets fileload to 1 to show the routine has been used, and if we come to load a new file, the Amstrad detects this at 2030 and wipes out entirely the arrays mentioned after the ERASE statement. The computer is then quite happy with 2040, because it no longer knows what size the arrays are.

Subroutine 3000 saves the file:

```
3000 CLS
3010 INPUT"What is the new file name";file$
3020 OPENOUT file$
3030 counter=1
3040 WHILE counter<count
3050 WRITE #9,name$(counter),phone$(counter)
3060 counter=counter+1
3070 WEND
3080 CLOSEOUT
3090 PRINT
3100 PEN 3
3110 PRINT"The file has been saved."
3120 INPUT"Press ENTER to return to menu.",en
ter
3130 RETURN
```

This again is similar to the earlier program.

Subroutine 4000 searches for a phone number, having been given the friend's name:

```
4000 CLS
4010 INPUT"Whose number do you want";friend$
4020 counter=1
4030 found=0
4040 WHILE counter<count AND found=0
4050 IF friend$=name$(counter) THEN found=1:P
RINT friend$;" has this phone number: ";phone
$(counter)
4060 counter=counter+1
4070 WEND
4080 IF found=0 THEN PRINT friend$;" is unkno
wn to me."
4090 PRINT
4100 INPUT"Press ENTER to return to menu.",en
ter
4110 RETURN
```

The computer searches through the elements of the array until either it's looked unsuccessfully at them all, or the phone number's been found. Line 4080 is a useful line to include, as it is easy to make a mistake with the friend's name, and at least in this case the computer lets you know what's happening.

Subroutine 5000 allows you to change phone numbers and uses a similar search loop to that in the previous routine (a good excuse for a further subroutine!).

```
5000 CLS
5010 INPUT"Whose number do you want to change
";friend$
5020 counter=1
5030 found=0
5040 WHILE counter<count AND found=0
5050 IF friend$=name$(counter) THEN found=1:I
NPUT"What's the new number";phone$(counter)
5060 counter=counter+1
5070 WEND
```

```
5080 IF found=0 THEN PRINT friend$;" is unkno
wn to me."
5090 PRINT
5100 INPUT"Press ENTER to return to menu.",en
ter
5110 RETURN
```

The whole program is a skeleton file-handling program, and now perhaps you'd like to put some flesh on it.

Exercises

1) The program only works with existing files. Add a routine so that a new file can be created from the keyboard. Make sure that you use either UPPER$ or LOWER$ on any strings being saved to the file, to prevent later problems with string matching when searches are carried out.
2) Add a routine to allow the user to browse through the records in the file, displaying them one by one on the screen at the depression of a key.
3) Extend the program so that records can be added to or deleted from the file.
4) Rewrite the program so that it enables you to create and maintain a file of your own data on some other subject.

Going further

Hopefully, this book has helped answer some of your questions about the Amstrad computer. It is impossible to ever know everything there is to know about a new micro — this is the reason for the galaxy of computer magazines on sale at every newspaper shop. Each month someone discovers a new way of solving an old problem, or uncovers an ability of the computer that few realised existed.

In the limited space available I have attempted to introduce most of the commands which you will use frequently on the Amstrad. Inevitably, there have been omissions, and for that I apologise. In some case there remains a great deal to be said: the graphics commands that have been described here are the basic

ones, but there are other facilities which it would be inappropriate to introduce in a book which is essentially an introductory one.

If you wish to go further, pay a visit to your local newsagents — there are already several computer magazines devoted entirely to computing on the Amstrad.

Lastly, do remember that the people who write so knowledgeably in the computer magazines were once beginners like yourself. You are just starting to learn about programming on the Amstrad and what at present seems difficult will one day seem ludicrously easy. Enjoy your investigations.

Index

BASIC PROGRAMMING ON THE AMSTRAD COMPUTERS 464, 664 and 6128

The CPC 6128 and 664 incorporates the CPC 464 with Amstrad's disk drive in one unit. This book, as the title suggests, covers BASIC programming on all three machines. It takes the reader from the very beginning covering the first principles of BASIC programming. The last few chapters cover topics of more specialised interest such as the use of sound and how to create the files and read data from them. Each chapter contains example programs and exercises.

The Author

Wynford James writes educational material (including software) for a major microcomputer company. Prior to that he was a technical author for ICL. He has also taught mathematics and was actively involved in the development of computer studies throughout his school.

ISBN 0-7447-0036-1

9 780744 700367

£8.50

BASIC PROGRAMMING ON THE AMSTRAD COMPUTERS 464, 664 and 6128 Wynford James

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/