On the Road to Artificial Intelligence: Amstrad CPC464

SHIVA's friendly micro series

Jeremy Vine

On the Road to Artificial Intelligence: Amstrad

Dedication

For Christine

'From quiet homes and first beginning,
Out to the undiscovered ends,
There's nothing worth the wear of winning,
But laughter and the love of friends.'

Hilaire Belloc (1870–1953)

On the Road to Artificial Intelligence: Amstrad

Jeremy Vine

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
64 Welsh Row, Nantwich, Cheshire CW5 5ES, England

ISBN 1 85014 064 2

An interface was used to produce this book from a microcomputer disc, which ensures direct reproduction of error-free program listings.

Typeset by Wordsmith Graphics Limited
Printed by Devon Print Group, Exeter

Contents

Preface

Computers are rapidly becoming a part of everyday life. The advance in silicon technology has moved at such a pace that we now stare in the face what was not long ago complete science fiction. Machines are now being developed that can behave with spine-chilling accuracy the way a human does, in thought and knowledge. Along with that movement has appeared one of the largest growth industries: home micros. To some people they are mere toys but the power within a home micro is great and the ability to create programs that mimic human thought processes are attainable.

Interactive programming concentrates on a certain aspect of the BASIC language, namely the commands that are responsible for handling information. The book teaches how to use these commands and combine them into programs that communicate with the user. The reader is also introduced to the expanding field of artificial intelligence, one of the most exciting areas of computer science.

Little has been assumed about the programming abilities of the reader. It is written for those with almost no knowledge of BASIC, and only assumes the reader has played around with a few rudimentary commands like `PRINT` and `INPUT`. However, if you have no experience of programming at all, I have included a crash course in BASIC at the end of the book. In itself the book is an introduction to BASIC but unlike other books it sets out to introduce the notion of artificial intelligence and teach the user how to write programs that interact with the user. Two simulation programs are described that turn the Amstrad into a companion with whom to hold an intelligent two-way conversation and behave in a manner similar to, and with the same faults as, a human.

With a little work you will be writing programs that will give hours of pleasure as well as start you thinking on the limits to which you can stretch your Amstrad.

To finish on a personal note I must acknowledge with gratitude the help, encouragement and support I have received from family and friends and to all at Shiva Publishing. To them, and to countless others who have enthused my interest in this area, this book is partly due.

London, 1984 Jeremy Vine

About the author

Jeremy Vine was educated at William Ellis Grammar School, Highgate, London, and at the City of London Polytechnic, where he achieved a BSc in Psychology. Jeremy first became involved with computers while studying for his degree and has never since been far away from one. When Jeremy left the polytechnic, he worked freelance for *Acorn User* magazine, before joining Acorn Computers Limited. During this period, he also studied for an MSc in Neurophysiology. Now, having left Acorn, Jeremy is working full-time as a freelance writer. As well as his books for Shiva, he is a regular contributor to several computing magazines, which has brought him a consultant editorship. His spare-time interests are tennis, piano, photography and, of course, home-computing.

1 Can Machines Think?

Science fiction has a way of becoming fact and as it was impossible to believe, only a few decades ago, that man would ever step on the moon, so too is it incredible to think about machines that can answer back and act intelligently. Yet the day of artificial intelligence becoming reality is dawning and it may not be long before these words can be written by a computer. However, before you start to think that your Amstrad can act like any human being, I should point out that artificial intelligence on home computers is not advanced enough. But that is no reason not to go as far as you can.

The purpose of this book is threefold. Firstly, to show you a variety of commands in the BASIC programming language and teach you how to write programs that deal solely with text and the manipulation of text input by the user. Very little knowledge of BASIC is assumed but if you don't feel confident enough to tackle Chapter 3, where we start programming, then turn to Appendix A where I have written a lightning guide to BASIC and first principles. Secondly, a number of techniques are shown which you can incorporate in your own programs, to make them more interactive and thirdly, to build up to writing programs that enable the user to freely communicate with the computer. What do I mean by freely communicate? Well, take a look at the following interaction between a human and a computer:

```
COMPUTER:  NICE TO MEET YOU JEREMY
           TELL ME WHY YOU WANT TO TALK TO ME

   HUMAN:  I HAVE TERRIBLE DREAMS

COMPUTER:  DO YOUR DREAMS INVOLVE OTHER PEOPLE?

   HUMAN:  YES MY SISTER

COMPUTER:  THIS SOUNDS LIKE A COMPLICATED
           RELATIONSHIP
```

```
   HUMAN:  I'M VERY DEPRESSED OVER THIS
COMPUTER:  HOW OFTEN ARE YOU DEPRESSED?
   HUMAN:  ALMOST ALL THE TIME
COMPUTER:  I SEE
   HUMAN:  AND I DREAM OFTEN
COMPUTER:  I ENVY YOU...I CAN'T DREAM
   HUMAN:  WOULD I BE WRONG IN THINKING ABOUT
           HER
COMPUTER:  I WOULD IF I WAS IN YOUR PLACE
   HUMAN:  WOULD YOU SAY IT'S ABNORMAL
COMPUTER:  I'M NOT PREPARED TO SAY WHAT I THINK
   HUMAN:  YOU DON'T THINK ANYWAY. YOU'RE JUST
           A MACHINE
COMPUTER:  DO MACHINES WORRY YOU?
   HUMAN:  NO OF COURSE NOT. COMPUTERS ARE
           MINDLESS
COMPUTER:  AS A COMPUTER I TAKE A DIFFERENT
           VIEW
```

That conversation didn't take place on a huge mainframe computer but on the Amstrad computer. It is an example of what can be achieved even on home micros and we will write the program to do this later in the book. Now you might be thinking that we already have an intelligent machine if it can hold a conversation like the one above. Well, feeling like a magician who has dropped a card from his sleeve, I have to admit it is all a trick. The program does indeed allow the user to freely communicate with the computer, and the computer does make intelligent replies but does that mean the computer is thinking?

By the end of this book I hope to have started you thinking about whether machines can think or not, which is a bone of contention within the research area of artificial intelligence. A definition of AI is difficult because of the degree of argument and controversy surrounding the field. Perhaps it is purely to describe any action performed by a computer that previously could only be carried out by a human being. Or it may describe the processes a machine is using to carry out a task, like playing chess.

I leave you with that problem to think over and as we progress through the book, and learn the different ways of communicating with the computer and it in turn communicating with us, try to decide for yourself to what degree, if any, your Amstrad is a thinking machine.

2 Creating an Interactive Program

Creating an interactive program is a combination of factors. It starts as an idea and progresses through many stages until the program is completed to the satisfaction of the writer. The planning behind a program is as important as the way a program is written and sorting out problems at an early stage makes for easier development. This chapter considers the general principles to be borne in mind before writing a program and in particular interactive programs.

Interactive programming is simply writing programs that interact with the user. The emphasis is on using BASIC to write programs that enable the user to freely communicate with the computer and, in later chapters, we will see how to use a wide range of commands in the BASIC language which will eventually make our Amstrad have the appearance of a thinking, intelligent being. What we will try to achieve is to teach our computer to think.

In creating a program there will always be an idea or an aim from which the program will spring. One should think carefully exactly what the aims of the program are and the way in which it will work. What do you want the user to see on the screen display? What do you allow him to type in? You must ask yourself many questions about the program if you are to get it right.

For instance, we will see how to account for as many of the possible things a user of a program might do to mess up the works. So what we effect is a way of writing into the program, in advance of its use, procedural methods to take care of the worst thing a computer has to face. And what is that worst thing? The user! The computer has no way of knowing or understanding what the user desires, and is to all intents and purposes blind to the world. Computers are mindless (though don't say that to Sigmund later or he will get upset!) and have to be fed information.

Information is the life blood of a machine and computers need to be fed a healthy diet of raw data. This book concentrates on one side of programming, that is, the manipulation of text, both material held by the computer and text input by the user. You may already be acquainted with a few BASIC commands and able to `INPUT` and `PRINT` information but there is much more that you can accomplish with text. BASIC offers a range of commands enabling the programmer to manipulate and juggle text around to create the impression of a thinking machine. Points to bear in mind when writing a program are to:

1. Use flowcharts to aid the planning of a program. (There are a number of flowcharts in the book which help to understand the way in which a program works.)

2. Plan what the user will see on the screen.

3. Ease of use. Try to make the program as easy as possible to use.

4. Error trapping. Ensure that unwanted errors don't affect the smooth running of a program.

If any of these points make little sense at the moment, don't worry. We will cover them as we progress and by the end of the book these should be principles that will come straight into your head when writing a program.

Always remember that we are dealing with an empty box. Our job is to fill that box up and to make it an Aladdin's cave to anyone who looks inside. But if we are to be creative and write interactive programs we must do some programming. So let's get down to it straight away. Happy interactions!

3 Strings and Things

I've already said that we've got to teach our Amstrad to think and the first step in that process is user input. In this chapter, and the next, we will see how the computer accepts information and then manipulates it to our requirements. Let's remind ourselves of what is meant by a variable.

Program 3.1

```
10 INPUT name$
20 INPUT age
30 PRINT name$
40 PRINT age
```

Program 3.1 contains two variables: a string variable, denoted by $ (dollar) on the end of a variable name, called `name$`, and a numeric variable `age`. When the program is run, in reply to the first prompt ?, we can type in alphanumeric characters—in other words, type in anything we like—and this reply is stored in `name$`.

On pressing the ENTER key we are faced with another question mark. This time the Amstrad is expecting a number to be entered and this value is stored in `age`. Lines 30 and 40 print out the contents of `name$` and then the contents of `age`. You have no doubt guessed that the program is asking for your name and age. Well, that is far from clear, as all we can see when the program is run is two question marks! So, let's tidy the program up:

Program 3.2

```
10 INPUT"name ? ",name$
20 INPUT"age ? ",age
30 PRINT name$;" IS";age;"YEARS OLD"
```

Run the program and you can see that the prompts are more explicit and we can now direct the user to input his name and age. Now rerun the program but this time enter your name at both prompts. If we do this, it can be seen that the machine will store the first input correctly, as it is a string input, i.e. any character. However, the second variable `age` requires a numeric input and therefore a value of `0` remains in variable `age` as the input is not numeric. The Amstrad will reject alphabetic input in this situation and ask you to re-enter the information.

Having reminded ourselves briefly on variables, it is time to play around with strings!

STRING MANIPULATION 1: The use of LEFT$

As one of the main aims of this book is to concentrate on text input and the manipulation of that text by the computer, let us take a look at some commands in Amstrad BASIC that enable us to play around with strings. But first of all, consider the following problem—I want to write a program that prints out the letters of my name, increasing by one letter on each line as follows:

```
J
JE
JER
JERE
JEREM
JEREMY
```

Well we could try and write a program, such as the one below:

Program 3.3

```
10 PRINT "J"
20 PRINT "JE"
```

```
30 PRINT "JER"
40 PRINT "JERE"
50 PRINT "JEREM"
60 PRINT "JEREMY"
```

But we are not going to get very far if we have to type in everything for the computer. The problem with Program 3.3 is that we are typing everything that appears on the screen—not exactly a time saving way of writing the program. Of course, there is a quicker way and Amstrad BASIC provides us with the commands to help us in this situation.

What we need is a way of reading any given string and printing it out a letter at a time, increasing by one character on each line, as in the example above, without all the `PRINT` statements. 'How do I do that?' I hear you ask. The answer is simple. We use the command `LEFT$` .

The next program uses `LEFT$` to print out the first character of the string entered by the user. Type it in and see:

Program 3.4

```
10 INPUT"Enter a word ? ",word$
20 PRINT LEFT$(word$,1)
```

Try changing the number 1 in line 20 to other values to see what happens. By changing this value, the number of characters printed, from left to right, of the word will alter accordingly. Have a look at Figure 3.1.

Our next job is to produce the same effect without having to type a `PRINT` statement for each line. Type in Program 3.5:

Program 3.5

```
10 phrase$ = "THE QUICK BROWN FOX"
20 counter = 1
30 PRINT LEFT$(phrase$,counter)
40 counter = counter + 1: GOTO 30
```

Run the program and ...?! The problem lies in the fact that we increase the value of `counter` every time and when we exceed the number of characters in the string we just carry on printing `phrase$`. Change line 40 to read:

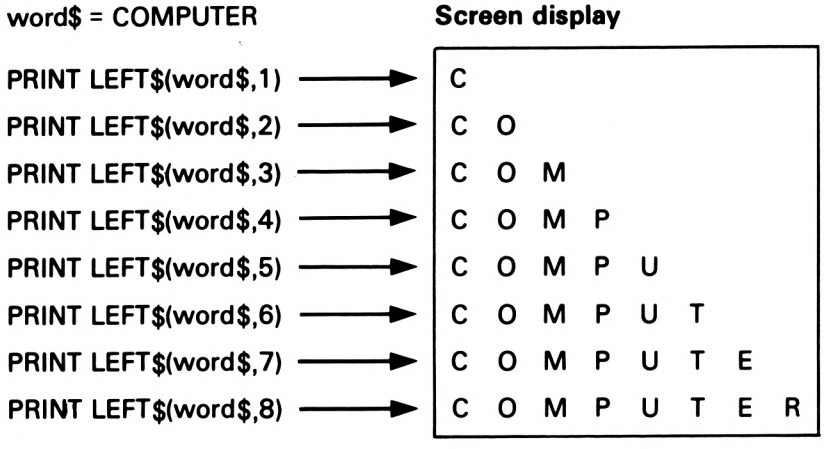

Figure 3.1 Effect of LEFT$.

```
40 counter = counter + 1: IF counter > 19
   THEN END ELSE 30
```

At last the desired effect! Or is it? There is still a problem. In order to stop the program when all the characters have been printed, I've had to count the number of characters in `phrase$` (including the spaces), thus the reason for `IF counter > 19` in line 40.

However, we are not always going to be in a situation where we know the number of characters in the string. Once more Amstrad BASIC comes to the rescue with a command to let the computer know the length of a string.

STRING MANIPULATION 2: LEN

The `LEN` statement gives such information. In Program 3.6 we use `LEN` to give us the amount of characters entered into `entry$`. Try entering one word at first and then re-run the program and enter a short sentence.

Program 3.6

```
10 MODE 2
20 INPUT"Enter word or phrase : ",entry$
30 PRINT"The number of characters in the
   string is";LEN(entry$)
```

If you entered a few words you would have noticed that LEN counts the spaces as well as any other characters. Spaces are counted in this instance, and it is worth bearing in mind as we will take a look later on, in greater detail, at spaces between words. However, back to LEN for the time being.

In Program 3.5 our problem was not knowing the number of characters in the string. Let us now rewrite that program using LEN:

Program 3.7

```
10 INPUT phrase$
20 FOR counter = 1 TO LEN(phrase$)
30 PRINT LEFT$(phrase$,counter)
40 NEXT
```

You will notice that I have introduced a FOR-NEXT loop into the program. If you have not used a FOR-NEXT statement yet, a look at the user manual will explain the uses of this command. In brief, the loop is set from the first character of phrase$ to the total length of the string, this being determined by LEN in line 20.

Not only is Program 3.7 more efficient than Program 3.5, but it is also faster. This can be quite important in the larger programs where much processing is being carried out. But more of that later. Time now to introduce a couple more BASIC statements.

STRING MANIPULATION 3: RIGHT$ and MID$

If you've understood the use of LEFT$, the next command is very easy. RIGHT$ is, as its name implies, a string handling command and if you haven't guessed by now, carries out the same function as LEFT$, except in reverse. Well, not quite the opposite! Let's take a look at Program 3.4 again except this time changing LEFT$ to RIGHT$ in line 20:

Program 3.8

```
10 INPUT"Enter a word ? ",word$
20 PRINT RIGHT$(word$,1)
```

RIGHT$ doesn't quite work as you might think. With a value of 1 in line 20, RIGHT$ gives the last character in the string. However, try changing that value. The result produced is not back to front. Not clear? Have a look at Figure 3.2.

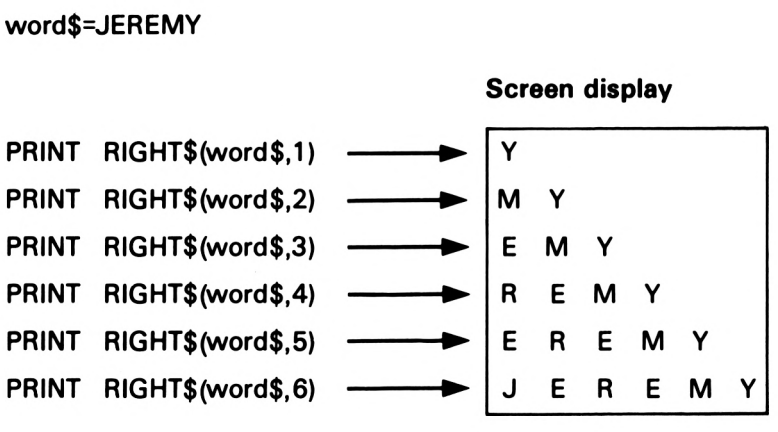

Figure 3.2 Effect of RIGHT$.

The reason I said it might not be clear is that one might expect RIGHT$(word$,3) to produce YME, in other words print the string starting from right to left but as you can see from Figure 3.2 this is not the case. If you're still not certain about the use of RIGHT$, use Programs 3.4, 3.5 and 3.7, changing LEFT$ to RIGHT$ wherever it occurs.

Now let us take a look at MID$. This, like LEFT$ and RIGHT$, is a string function but unlike those functions has three arguments associated with itself, i.e. MID$(exam$,X,Y). X and Y are two of the arguments and I will explain each in turn. Let us suppose that exam$ contains the string INTERACTION. This is how MID$ works:

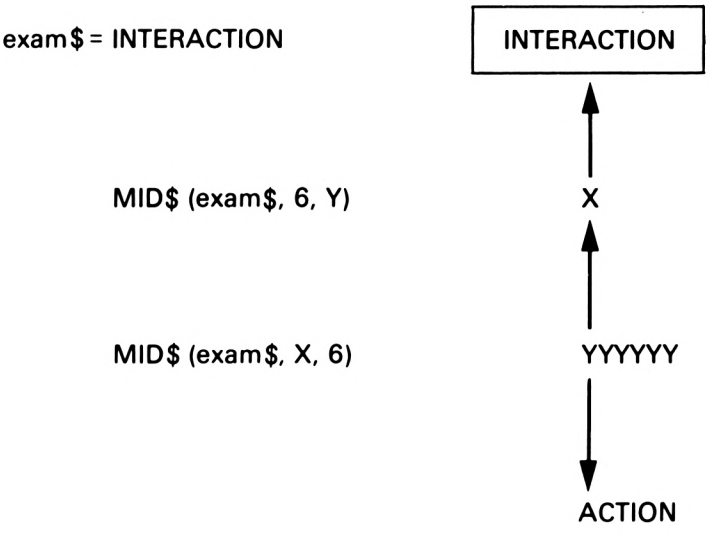

Figure 3.3 Function of MID$.

The first argument is the same as `LEFT$` and `RIGHT$` and gives the name of the string variable to be worked on. In our example this is `exam$`. The second argument acts as a pointer to the starting position. Therefore, the value of six means that the string we are going to produce starts at that position, in this case at the letter `A`.

Finally, the third argument (shown as `Y`) gives the number of characters to be taken from the position of `X` (inclusive of the character at position `X`). So in Figure 3.3, the first six letters from position `X` will be taken (as it happens there are only six characters left in that string). Let's put this into a program:

Program 3.9

```
10 CLS
20 exam$="INTERACTION"
30 INPUT"Set pointer to where ? (ie 'X')",x
40 INPUT"Number of characters from that
   position",y
50 PRINT MID$(exam$,x,y)
```

Run the program entering different values and see how the values of x and y alter the word printed out in line 50. (Notice that I have used both upper and lower case letters for x and y, to show that it doesn't matter which form you choose for the data. In fact, you could combine the two and it would still be accepted.) The MID$ function along with the other string functions I have mentioned so far are extremely useful, and I will return to them later on. Let's, however, take a break from string manipulation and take a look at another area of user input.

MENUS!

No, I'm not thinking of food! I mentioned in Chapter 2 that an important part of an interactive program is the screen presentation. An integral part of such displays is often a menu of options for the user to choose from, so why don't we start writing the options menu for our next program. Type the program in, keeping to the line numbers shown, as we are going to build up a program in a few stages.

Program 3.10a: MENU procedure

```
 50 CLS
 60 PRINT"A S C I I    C O D E S"
 70 PRINT,,"1 ASCII code to character"
 80 PRINT,,"2 Character to ASCII code"
 90 PRINT,,,,"Enter your choice"
100 a$=INKEY$
110 a=VAL(a$):IF a<1 OR a>2 THEN 100
120 PRINT a
130 ON a GOTO 30,40
140 RETURN
```

Also add lines 10 and 20 as follows:

```
 10 MODE 1: ZONE 40
 20 GOSUB 50
```

Run the program. As the program is not complete you will get a NO SUCH LINE error message if you press either 1 or 2. Run the

program a few times and note how it only responds to the two numbers shown. Type in any other number and it is ignored. For the moment don't worry about how the program works; I'll be explaining how this procedure and the rest of the program works in the remaining part of this chapter and Chapter 4.

One thing you couldn't have failed to notice was the strange title on the screen—ASCII CODES. Who or what is ASCII? Let's find out.

ASCII

ASCII (pronounced AS-KEY!) are letters that stand for American Standard Code for Information Interchange. Try saying that every time. No wonder it is shortened! But what does all that mean?

As you may have discovered by now, every computer is different and BASIC, for example, has many versions implemented and all of them have differences. One of the few standards that is adopted by most computer manufacturers is the use of the ASCII character set. ASCII is a code used by the computer to represent characters and control codes. For our purposes we will concentrate on the keyboard character set. If you haven't already done so, SAVE the menu program and then type in Program 3.11.

Program 3.11

```
10 FOR ascii = 32 TO 126
20 PRINT CHR$(ascii);
30 NEXT
```

The program produces the characters represented by the ASCII value. You might have noticed that I have used a BASIC keyword CHR$ in Program 3.11. To explain what CHR$ is doing, LOAD in Program 3.10a again and let's add the next part of the program.

Program 3.10b

```
240 CLS:INPUT"Enter ASCII code number and
    press ENTER",code
```

```
250 PRINT
260 PRINT code;"is the ASCII code for '";CHR$
    (code);"'"
270 PRINT,,,,"Press 'Y' to continue"
280 a$=INKEY$:IF a$="y" OR a$="Y" THEN 290
    ELSE 280
290 RETURN
```

Also add line 30:

```
 30 GOSUB 240
```

Now SAVE the program and RUN. You will be asked to enter an ASCII code number. Type in any number between 32 and 126 (these are the codes that represent the character set). The conversion from ASCII number to a character is being carried out by the CHR$ command in line 260.

The purpose of CHR$ is to produce a character from a given number. This number relates to the ASCII value of that character, i.e. if you enter 65 the answer will be A, because the ASCII value 65 is the code for the character A.

OK. What about converting a character to an ASCII number. This can be done by using the ASC function. This is used in the final section of our program. Make sure you have the program as it stands LOADed into the Amstrad before continuing:

Program 3.10c

```
170 CLS:PRINT"Enter character"
180 a$=INKEY$:IF a$ <>"" GOTO 190 ELSE 180
190 PRINT"The ASCII code for '";a$;"' is
    "ASC(a$)
200 PRINT,,"Press 'Y' to continue"
210 a$=INKEY$:IF a$="y" OR a$="Y" GOTO 220
    ELSE 210
220 RETURN
```

Add line 40:

```
 40 GOSUB 170: GOSUB 50
```

and change line 30 to:

```
30 GOSUB 240: GOSUB 50
```

The program is now complete. SAVE it and RUN. Choose option 2 and enter any character. Line 190 carries out the conversion of the character which is held in the string a$. ASC(A$) gives the ASCII value of the character held in a$. You can see this working by just typing in the following line:

```
PRINT ASC("J")
```

By altering the character a different code will be produced.

What if a$ contains more than one character? Well, only the value of the first character of that string will be returned. You are probably wondering whether I have forgotten to explain the other elements of the program. Don't panic! The remaining commands will be explained in the next chapter but for the moment here is Figure 3.4, a flowchart of the workings of the program.

See if you can spot what is wrong with the program. Remember what I said in Chapter 2 about having to account for all possible input by the user. I will come back to this program in Chapter 6 which is about error trapping. I think you have enough clues for now!

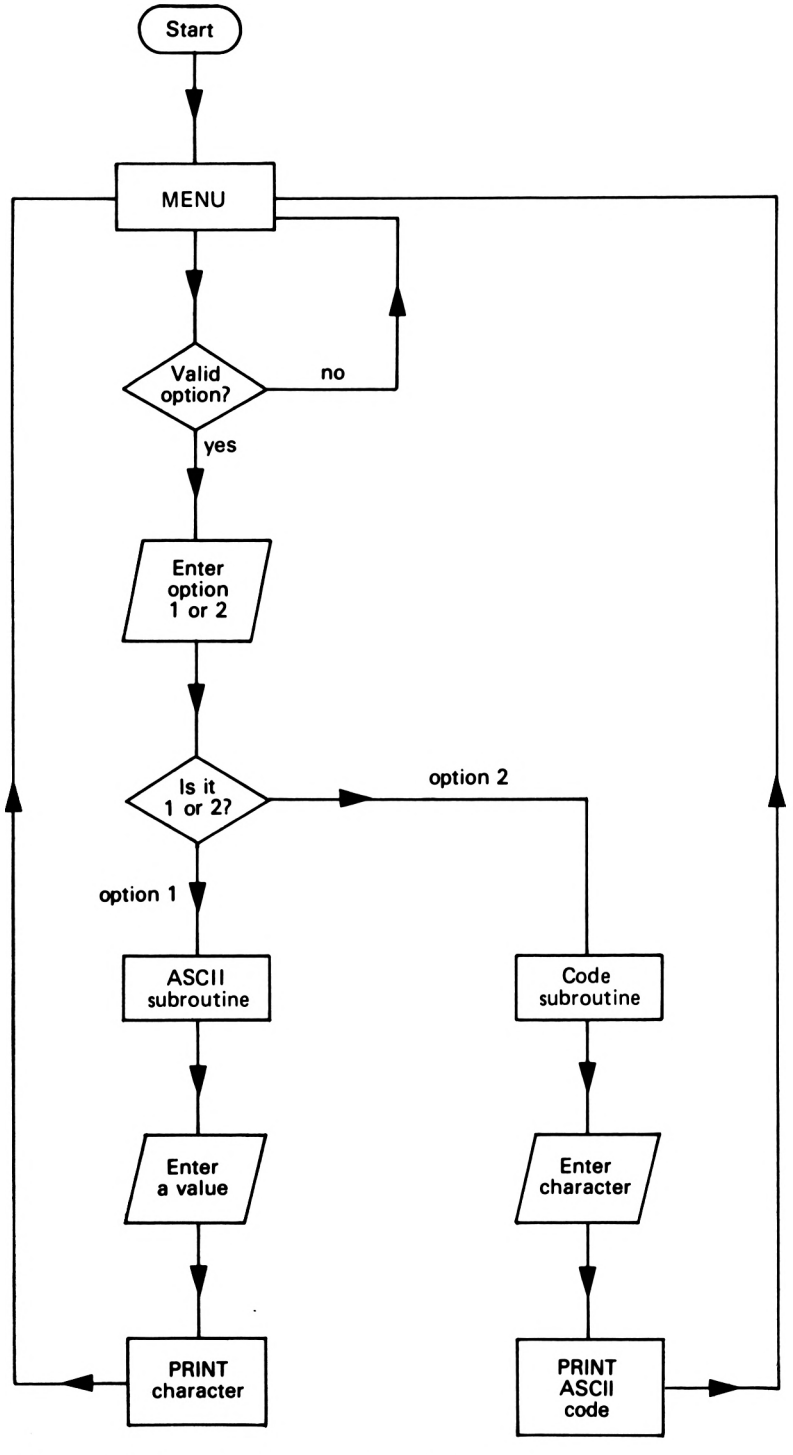

Figure 3.4 Flowchart for Program 3.10.

4 More Strings Attached!

An important aspect of writing an interactive program is to ensure that the computer reads all of the user's input. Now you might think that we have already covered this point by using the INPUT statement but this is not so. To see what I mean enter Program 4.1:

Program 4.1

```
10 CLS
20 GOSUB 100:INPUT firstphrase$
30 GOSUB 100:LINE INPUT secondphrase$
40 PRINT,,"firstphrase$ = '";firstphrase$;"'"
50 PRINT,,"secondphrase$ = '";
   secondphrase$;"'"
90 END
100 PRINT,,"Type in the following:","MY
   AMSTRAD IS A GREAT COMPUTER"
120 RETURN
```

Run the program and you can see that both variables contain the same phrase that we input. However, run the program again and this time insert a comma in the middle of the sentence like this:

```
MY AMSTRAD, IS A GREAT COMPUTER
```

See the difference? When we use INPUT in line 20 the variable will quite happily retain the entire contents of the string as long as there is no comma in the sentence. But when a comma is inserted, INPUT will reject the entire string entered. To get

round this problem Amstrad BASIC provides the command LINE INPUT which will accept everything that is typed in. I have used LINE INPUT in line 30 of the program and when the program is run, it can be seen how everything you type in, regardless of commas, will be retained.

LINE INPUT not only accepts commas, it will also recognize leading spaces. By this I mean if you type in a number of spaces before your sentence, LINE INPUT will include those leading spaces in secondphrase$, whereas INPUT will not. Try typing in a sentence with leading spaces to see the difference.

But why all this fuss anyway? Well, in developing our interactive programs we want to allow the user to communicate freely with the computer. To do this the user must be free to type in anything he likes and, as is most likely, he will type in a sentence which could contain a comma. If the computer is to analyse the input from the keyboard it must be able to look at everything that has been typed and as we have seen with the INPUT statement this is not always the case. LINE INPUT ensures that everything typed in is stored in a given string.

RECOGNIZING A FAMILIAR WORD: The use of INSTR

We now come to a problem in creating an interactive program—that of recognizing a word within a long stream of other characters. LINE INPUT may ensure that every word typed in is what is stored, but how do we start to recognize a particular word within a sentence? It is not as difficult as it might first sound. As you by now have come to expect an Amstrad BASIC command is available to solve the problem. The command is INSTR (you can think of it as 'in string').

What INSTR does is to search a string, in this case a long sentence that we have input, for a familiar word. When it has found that word (or group of characters) INSTR returns the position of the leftmost character of the word within the string.

To demonstrate this type in Program 4.2.

Program 4.2

```
10 CLS:ZONE 40
20 PRINT"Enter your sentence"
30 LINE INPUT a$
40 b$="BASIC"
50 find=INSTR(a$,b$)
```

```
 60 PRINT,,"The value of variable 'find' = ";
    find
 70 PRINT,,"YOU TYPED:",;a$
 80 IF find=0 THEN 120 ELSE 90
 90 PRINT SPC(find-1);"↑"
100 PRINT"The match was found here"
110 END
120 PRINT,,,"The string '";b$;"' was not
    found":END
```

Now if you type in any old sentence the chances are you will get a message 'The string BASIC was not found'. This is because the program is looking for the occurrence of a string, in this case the alphanumeric string BASIC which is assigned to b$ in line 30. Run the program again and type in the following:

```
THAT IS THE BASIC DIFFERENCE
```

This time you will see the string has been matched up with a word within the sentence. The value of the variable find is the position at which the start of the word BASIC was found within the sentence typed. The line of the program that is doing all the work is line 50. INSTR(a$,b$) returns a value. If a match is found, i.e. if the variable b$ is found within the string you typed in as, a$, then the position of the string is returned and the numeric variable find equals that value. However, if no match is found then a value of 0 is always returned. Line 80 acts on that result and accordingly prints the appropriate message. See Figure 4.1 for a flowchart of the program.

INSTR is a very useful command and as you will see later on in the book, it plays a crucial part in picking out key words or phrases for the computer to understand. More of that later. Now let's backtrack slightly and look at a few commands I have used in programs in this chapter and Chapter 3 without explanation.

PRESENTATION AND LAYOUT 1: INKEY$

If you ran the menu program in Chapter 3 you might remember that everything was laid out for the user and, where possible, just one push of a key was sufficient to work your way through the program. An important part of any program is the presentation of

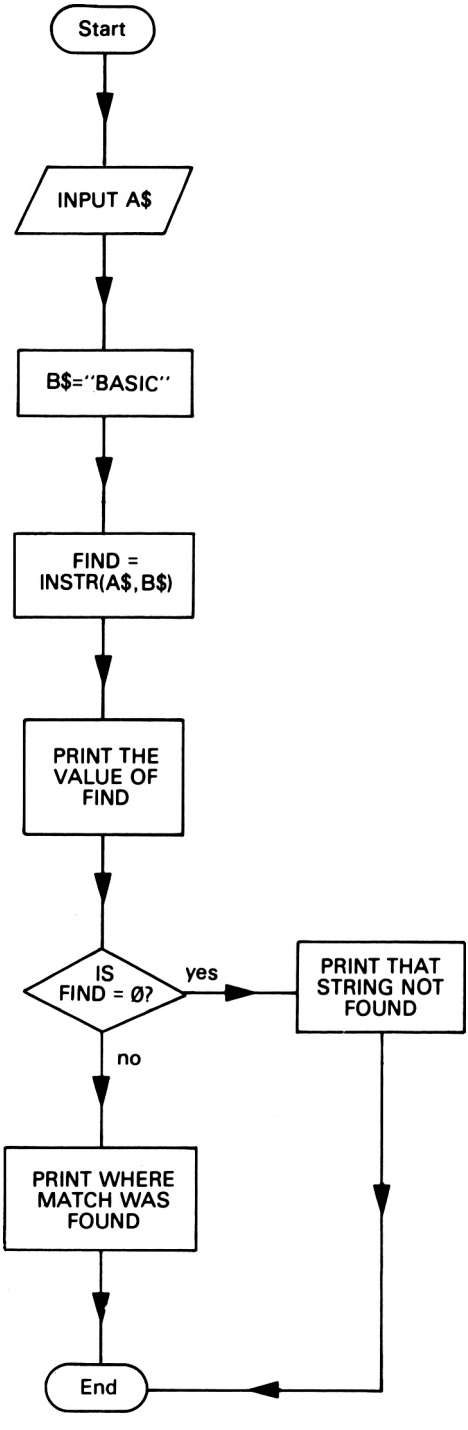

Figure 4.1 Recognizing a familiar word.

material on screen and the ease of input to the user. The more friendly these points are, the more interactive a program becomes. I have already used some commands in order to do this and I will now cover them.

Go back to Program 3.10 and you will notice I used in both lines 120 and 190 a BASIC keyword INKEY$. When this function is used the computer scans the keys of the keyboard to see if a key is pressed. If a key is pressed, a record of that key is stored in the opposite variable, i.e:

Program 4.3

```
10 character$=INKEY$:IF character$ =""
   THEN 10
20 PRINT"You typed the character ";character$
```

If a key is not pressed an empty string is returned. To ensure that the program waits until a key is pressed, the program loops around itself in line 10. However, you need not make any use of that stored variable as I have done in line 20 of Program 4.3. The command can also be used as a means of temporarily stopping a program until any key is pressed. Try Program 4.4:

Program 4.4

```
10 CLS
20 PRINT"I'm waiting for you to press a key"
30 a$=INKEY$:IF a$="" THEN 30
40 PRINT"About time too!"
```

Line 10 clears the screen and the program comes to a standstill at line 30 until a key is pressed. In these ways INKEY$ can provide methods of directly scanning the keyboard to see whether or not a key is pressed and which key it is that has been used.

PRESENTATION AND LAYOUT 2: ZONE, TAB and SPC

There are two commands which you can use as a means of printing spaces within a PRINT command. These commands are TAB and SPC. TAB is used in conjunction with the PRINT statement. It means tabulation and enables the user to specify the number of spaces to move to in a PRINT command.

TAB has one argument associated with its use. Therefore, PRINT TAB(5);' will move the cursor five spaces to the print position. SPC works in a similar fashion, though it does not need a semi-colon to end the command as this is assumed by the machine. Figure 4.2 shows examples of the TAB and SPC statements and their effects on the screen display.

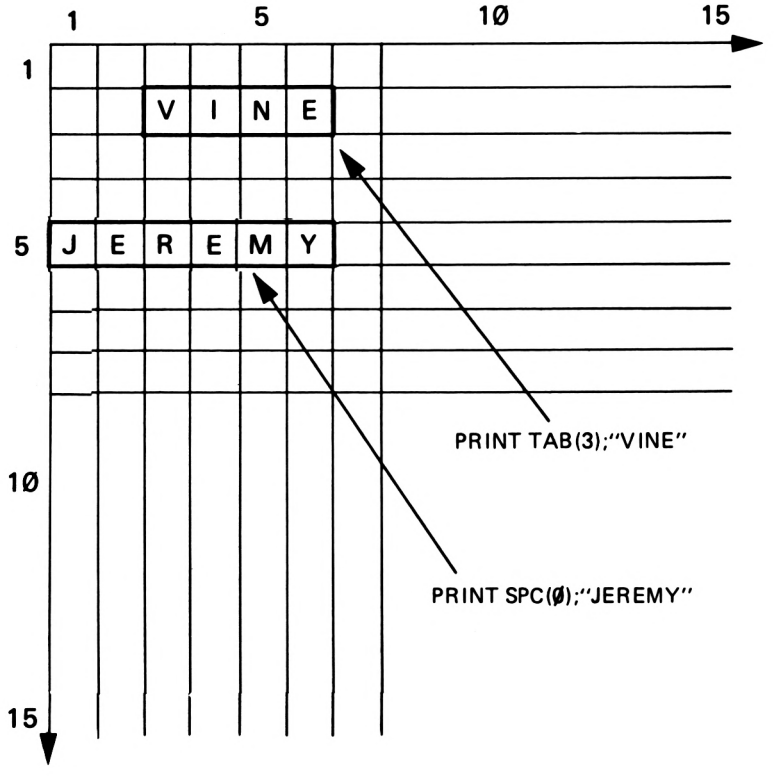

Figure 4.2 Effects of TAB and SPC commands.

I used SPC in Program 4.2 to place the correct number of spaces in a line before printing a word. This ensured that the words lined up under one another. (Go back and run Program 4.2 if you don't remember this.) SPC and TAB are two useful functions which enable a quick and well laid out screen to be constructed.

You will have noticed that I have used the command ZONE and it is a command that you will see quite a few times in this book. ZONE is responsible for changing the width of the 'zone' in which printing takes place when using the PRINT statement. So when we use a comma to jump to the next print zone, i.e. PRINT,' we can

tell the computer where that zone is by giving the width of that zone as a number. In the case of our programs, we usually require the program to read the commas following a print statement to mean 'print a blank line', so we declare the command `ZONE 40` at the start of the program. Do remember, however, that the number following `ZONE` depends on the screen mode you are using. As mode 1 is a 40 column mode we therefore need to make jumps of 40 spaces. Obviously you change this figure according to the mode you are working in and the printing you wish to effect.

We have already seen how `ASCII` works and conversion between alphanumeric and numeric variables can be important. As you know, a numeric variable cannot read an alphanumeric input. However, there is a way of converting a string so that it can be read.

CONVERTING STRINGS: VAL and STR$

Let us consider a problem. I wish to enter a number using an alphanumeric input and then have that converted so that a numeric variable can read and understand the input. Can we do this? No prizes for guessing that this is possible and Program 4.5 shows how:

Program 4.5

```
10 a$=INKEY$:IF a$="" THEN 10
20 a=VAL(a$)
30 IF a<1 OR a>5 THEN 10
40 PRINT a
50 ON a GOTO 100,110,120,130,140
```

The program is not complete and will therefore crash at line 50. Line 10 waits for an input and line 20 is where a conversion takes place. The function `VAL` (standing for 'value') takes `a$` which would contain a number and converts it into a real number which then becomes the value of variable 'a'. To ensure that a number between a certain range has been entered, line 30 checks that the value of 'a' is not outside the specified range. If it is outside the range, the user is sent back to line 10 until an acceptable number is entered. Program 4.5 is the basis of the subroutine `GOSUB 50` in Program 3.10a in Chapter 3. `VAL` therefore converts string

variables into numeric variables and the opposite can be achieved using STR$. Type in Program 4.6:

Program 4.6

```
10 number=1.2
20 PRINT number*2
30 a$=STR$(number)
40 PRINT a$*2
```

To prove that the number 1.2 has been converted to a string variable, line 20 carries out a mathematical function on a numeric variable and of course, it works. However, when line 40 is reached, an error message is generated by the computer because the machine is trying to carry out multiplication on a string variable. In line 30, STR$ has converted the numeric variable into a string by placing the converted string in a$. To check the contents of a$ type PRINT a$.

These functions within Amstrad BASIC give your programming more flexiblity, and as I've shown you with the menu program, can be very useful.

OPTIONS WITHIN PROGRAMS: The use of ON

Program 4.5 included an easily missed command, ON. This allows the order of running a program to be changed, by jumping to a specified line depending on the value of a certain variable. For instance, in the menu program, depending on which number is entered, the program jumps to the appropriate routine. Therefore, a line like:

```
ON key GOTO 220, 250, 800
```

means if the numeric value of 'key' is equal to 1 then GOTO line 220; if 'key' equals 2 then GOTO line 250 etc. Figure 4.3 illustrates this.

Providing options within a program is most common and ON gives the programmer a way of presenting many options without resorting to loads of IF-THEN statements. Look at Program 4.7, but don't type it in unless you are particularly masochistic!

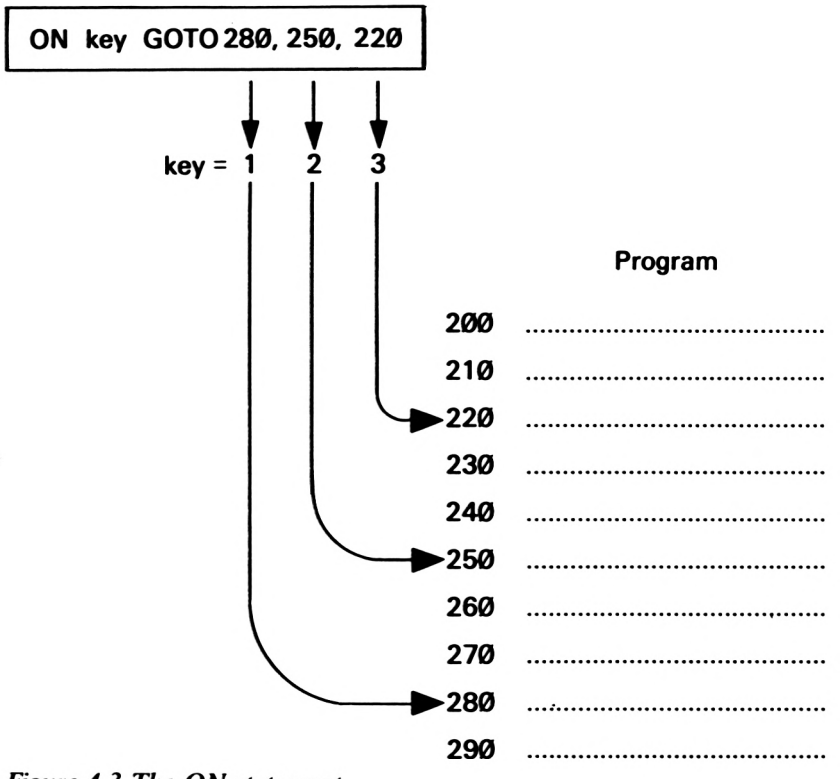

Figure 4.3 The ON statement.

Program 4.7

```
10 INPUT "Enter option number", optnum
20 IF optnum = 1 THEN GOTO 100
30 IF optnum = 2 THEN GOTO 24
40 IF optnum = 3 THEN GOTO 310
50 IF optnum < 1 OR optnum > 3 THEN 10
```

Just from that short example you can see how much quicker `ON` is to use. Another use for `ON` is error trapping and I will cover this in Chapter 6.

REPEATING ONESELF: The use of WHILE...WEND and INKEY$

We already know one kind of loop, that being the `FOR-NEXT` loop, but there is a second statement which you can use for

situations where you want the instructions repeated until a specified condition is met. This is the WHILE...WEND statement. We can use the command as in the example of Program 4.8.

Program 4.8

```
10 x=0
20 WHILE x < 15
30 PRINT x
40 x=x+1
50 WEND
```

Line 10 sets x to zero and line 40 increases its value. Line 20 is the all-important line and tells the machine that while x is less than 15, any commands following up to the WEND statement should be carried out. In this case the program loops round until x is equal to 15.

The WHILE statement marks the start of a WHILE loop and the WEND command terminates the loop when the conditions are met.

WHILE...WEND can be used under a variety of other conditions and another use may be to test for the pressing of a certain key. To do this we can use WHILE...WEND in conjunction with INKEY$. Type in Program 4.9.

Program 4.9

```
10 x$=INKEY$
20 WHILE x$=""
30 GOTO 10
40 WEND
50 PRINT"THE KEY ";x$;" WAS PRESSED"
```

The program loops around until a key is pressed. The function INKEY$ we have already covered and tests to see if a key has been pressed. If so the character pressed is placed in x$ and the program continues. In this case INKEY$ loops round indefinitely because it is placed in a WHILE...WEND loop, which means the loop continues until a key is pressed.

These functions can be very useful for determining whether or not a certain key has been pressed and it is a good way of ensuring that only the key you specify is used. Bear this in mind when reading about error trapping in Chapter 6.

That concludes our look at manipulating strings and if you have understood everything in these last two chapters, you are well on the way to creating programs that interact with and not against the user.

5 Words, Words, Words

Even with the advent of new communication systems, a growing area within computer science, the basic form of communication has remained the same: the written word. Or to be more exact the on-screen word is still our most precious commodity as regards communicating between ourselves and with machines. So far in this book we have concentrated on the entering of information into the computer and its manipulation of that data. But we have dealt only with small individual bits of information that have amounted to very little. We now need to consider the storage of larger chunks of data which the computer can call on at any time.

ARRAYS

Imagine that we have to store a record of the finishing positions of racing car drivers in any given race. We need the computer to be able to tell us who was at a particular position in the race. Now how do we go about entering this information? Let's say we have the names of the top six drivers to enter. Using what we have learnt so far we would set up six separate variables to contain the information. Our program might go something like this:

Program 5.1

```
10 CLS:PRINT"Enter the name of the drivers in"
20 PRINT"order from 1 to 6"
30 INPUT num1$
40 INPUT num2$
50 INPUT num3$
60 INPUT num4$
```

```
 70 INPUT num5$
 80 INPUT num6$
 90 CLS
100 INPUT"Which position do you wish to
    check",check
110 ON check GOTO 120,130,140,150,160,170
120 PRINT"Position 1 was : ";num1$:GOSUB 180
130 PRINT"Position 2 was : ";num2$:GOSUB 180
140 PRINT"Position 3 was : ";num3$:GOSUB 180
150 PRINT"Position 4 was : ";num4$:GOSUB 180
160 PRINT"Position 5 was : ";num5$:GOSUB 180
170 PRINT"Position 6 was : ";num6$:GOSUB 180
180 PRINT"Press SPACEBAR to continue"
190 a$=INKEY$:IF a$<>" " THEN 190
200 GOTO 90
```

But this is a very time consuming exercise. Firstly, I have set up six separate variables to contain the names of the drivers. What would happen if I had a hundred names? And for each response I have had to set up a separate response line. Telling the computer which number you need to check is not easy and the only way is to provide a separate reply for each possible input. Well, I need not tell you this is futile. So what can be done to ease the pain of programming?

The answer is to introduce arrays. What is an array? You can think of an array as a table of information. It's rather like having an index card with a number of lines of information written on the card. Look at Figure 5.1.

There is a single filing card in Figure 5.1 and the card is labelled `WINNERS`. On each line is a position number and next to the number is the name. In order to let the computer know that we want to enter up to six names we tell the machine that the table is up to six items in length. This is done by using the `DIM` statement. So for our first line we type:

```
10 DIM winner$(6)
```

We have now set up a variable with a one-dimensional array to a

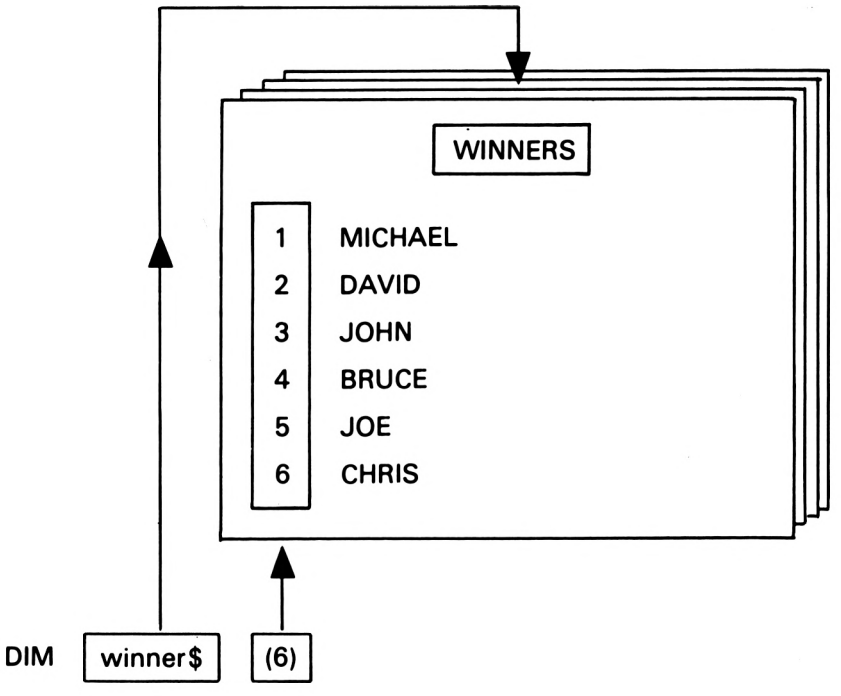

Figure 5.1 Arrays.

maximum figure of six. We can now complete the rest of the program.

Program 5.2

```
20 PRINT"Enter the names of the drivers in"
30 PRINT"order from 1 to 6"
40 FOR enter = 1 TO 6
50 INPUT winner$(enter)
60 NEXT
70 CLS
80 INPUT"Which position do you wish to
   check",check
90 PRINT"Position ";check;" was :
   "winner$(check)
```

```
100 PRINT"Press SPACEBAR to continue"
110 a$=INKEY$:IF a$<>" " THEN 110
120 GOTO 70
```

What we have done is to enter all our information into the same variable `winner$` but at six different points. By using a `FOR-NEXT` loop the value of 'enter' is increased every time and `winner$` can have six entry points, `winner$(1)`, `winner$(2)` and so on. Having saved time and space on the input side we can also save time when retrieving information. The correct entry is found and printed out all in line 90 and much space is saved. This program would be the same size for six hundred entries as for six. The only thing we would need to change is the size of the array, and we can do that by simply changing line 10, i.e. `DIM winner$(600)`, and the length of the `FOR-NEXT` loop. Now try to imagine writing a system for entering large sets of data using the first method!

The above is more efficient but there are situations where we need to enter more than one piece of information in connection with one particular event. Taking our motor racing example, I also want the country the driver represents to be displayed. We could add the following line to cover this point:

```
55 INPUT "Enter country", country$(enter)
```

and change lines 10 and 90 to:

```
10 DIM winner$(6), country$(6)
90 PRINT "Position "; check " was :
   "winner$(check); SPC(4); " ( ";
   country$(check)")"
```

But as we add more items we have to increase the number of arrays and this can get out of hand with more information being entered. It would be far easier if we could link the information about country into the same variable as the name. And, of course, we can do this. What we are now going to set up is a two-dimensional array, or if you prefer, a 2 x 2 table. Figure 5.2 shows this.

By changing the array in line 10 to:

```
10 DIM winner$(6,1)
```

DIM winner$ (6,1)

	Ø	1
1	MICHAEL	ENGLAND
2	DAVID	WALES
3	JOHN	SCOTLAND
4	BRUCE	IRELAND
5	JOE	FRANCE
6	CHRIS	ENGLAND

Figure 5.2 A two-dimensional array.

we have now told the computer that winner$ has two entry points. Change line 50:

```
50 INPUT"Enter name ",winner$(enter,Ø)
```

and add line 55:

```
55 INPUT"Enter country ",winner$(enter,1)
```

Finally, change line 90 to:

```
90 PRINT "Position"; check "was : "
   winner$(check,Ø); SPC(4); " ( "; winner$
   (check,1)" ) "
```

I have put in two input lines to show you more easily what is happening. Line 50 is `winner$(1,Ø)` and this equals a name. Line 55 is `winner$(1,1)` and this equals the country. The `FOR-NEXT` loop increments the first value until six names and countries have been entered and Figure 5.2 summarizes what is held in each cell of the variable. But we are not just restricted to two-dimensional arrays. We can just as easily define arrays with matrices such as 3 x 2 x 5 x 3.

Arrays, therefore, are very good for handling large sets of data and to us this can be potentially useful, as we will be creating programs that rely on using a database of information. But arrays are only half the story. To be of value to us we need to keep that data permanently and the programs above require you to enter data every time. Wouldn't it be better if we could store data within a program? We might consider storing the data in variables like this:

```
name1$ = "JEREMY"
name2$ = "BRUCE"
name3$ = "KITTY"
```

but we wouldn't be able to make use of arrays. A better way to store a set of information is to use the DATA and READ statements.

READ ME SOME DATA: The use of READ and DATA

The READ and DATA statements are ways of storing such data permanently within a program and enabling the data to be entered and stored within an array. Let us consider a simple example. Program 5.3 stores the names of footballers and they can be found by entering the number on their shirt:

Program 5.3

```
 10 DIM a$(11)
 20 FOR x = 1 TO 11
 30 READ a$(x)
 40 NEXT
 50 CLS
 60 INPUT"Enter player's number ",num
 70 PRINT"Player number";num;"is ";a$(num)
 80 a$=INKEY$: IF a$="" THEN 80
 90 GOTO 50
100 END
110 DATA JOHN HUNT,DAVID JONES,MIKE DAVIS,
    NICK HILL,FRED HEATH
```

```
120 DATA JEFF DOUGLAS,COLIN BLAKE,ANDREW
    MURRAY,DAVE ELLIOT
130 DATA CHRIS MILLER,MARTIN WILLIAMS
```

Line 10 defines the dimensions of the array and at line 30 we come across a new command, READ. The READ statement is nested within a FOR-NEXT loop and the first time round the loop, line 30 looks to READ information into a$(1) and this data is found at line 110. All the information is stored after DATA statements and each piece of data is separated by a comma to let the computer know it is one item of data. Therefore, the READ statement takes the first bit of data in line and the next time around the loop it reads the second item and so on. Figure 5.3 shows where the data is being stored as it is READ by the computer.

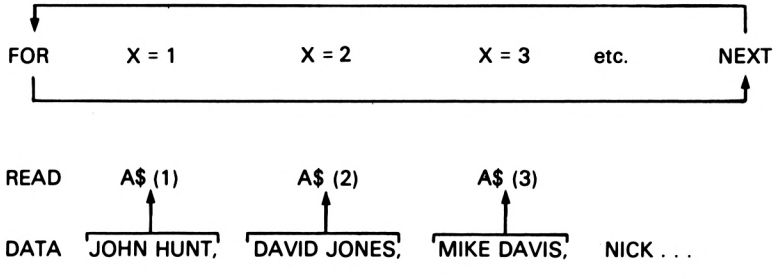

Figure 5.3 READ and DATA.

You will notice that I have used three lines of DATA statements. You can split the data over a number of lines as you wish, so long as you put a DATA statement at the start of each new line you use. It is also conventional to place your data at the end of a program. Bear in mind that if you use the READ statement at different junctures in the program, you must order your data so that it is read into the computer in the order you desire. One more thing, if you get an DATA EXHAUSTED error message, this will be because you have either entered too few data items for the amount being read in, or, the number you have allowed to read in is too great, i.e. too large a loop.

Now that we understand the READ and DATA statements, let's use them for something a little more interesting. The whole point of this book is to build interactive programs and with the knowledge we have gained we can start to put some of what we

have learnt into practice. Program 5.4 simulates a conversation with a short-tempered computer that just won't agree with you.

Program 5.4

```
10 MODE 2
20 DIM ans$(4),n$(4)
30 FOR x=0 TO 1
40 READ ans$(x)
50 READ n$(x)
60 NEXT
70 FOR x=0 TO 1
80 PRINT"WELL"
90 LINE INPUT a$
100 PRINT ans$(x);a$;n$(x)
110 NEXT
120 END
130 DATA "YOU MAY WELL SAY THAT "," BUT DO YOU
    HAVE ANY PROOF"
140 DATA "WHAT A LOAD OF NONSENSE. HOW CAN YOU
    SAY "," AND MEAN IT."
```

Run the program and in reply to the curt prompt `WELL` type something like:

```
AMSTAD COMPUTERS ARE GREAT
```

and in reply to the the next prompt type:

```
IT'S BETTER THAN A SINKLAIR
```

Not exactly artificial intelligence but it starts us on the right road. Line 100 prints out the first part of the reply, tags on the phrase you enter and finally adds on a reply to finish the sentence off. You may have noticed that the `DATA` statements are slightly different in Program 5.4, in that the string replies have quotation marks around them. This is because without them, leading and end spaces are ignored and it is necessary to leave spaces at both ends of the answer we are inputting. By placing quotation marks

around the phrase, everything contained within that phrase is used, including the spaces.

Play around with the program by increasing the data statements and see how long you can keep a conversation going. Of course, you have to limit your responses very much but it can be fun. Later on in Chapters 8 and 9 I will show you a much more intelligent conversation partner.

RESTORING KNOWLEDGE: The RESTORE statement

It is often useful to be able to set a pointer to where you wish the data to be read from. Sometimes it will be a case of resetting the pointer to read from the start of data again or at a specific point in the `DATA` statement. This can be done by implementing the `RESTORE` statement. For instance, you may write a program where the user's response determines where the data is read from.

Let's use an example of a grocer's shop. A customer wants to know the price of either a particular fruit or a vegetable. Depending on the customer's choice the data pointer is either set to the beginning of the fruit data or the vegetable data. Type in Program 5.5:

Program 5.5

```
10 DIM item$(7,1)
20 CLS
30 PRINT"Which do you require the prices of
   - ","(F)ruit or (V)egetables ? "
40 INPUT choice$
50 IF choice$="F" OR choice$="f" THEN
   RESTORE 130 ELSE RESTORE 150
60 FOR x = 1 TO 7
70 FOR price = 0 TO 1
80 READ item$(x,price)
90 NEXT:NEXT
100 FOR y = 1 TO 7
110 PRINT item$(y,0);" are ";item$(y,1);"p"
120 NEXT
```

```
130 DATA PEACHES,15,APPLES,25,ORANGES,10,
    GRAPES,45
140 DATA CHERRIES,27,SATSUMAS,18,
    STRAWBERRIES,67
150 DATA POTATOES,9,TOMATOES,14,PEAS,23,
    CARROTS,32
160 DATA ONIONS,16,CABBAGES,45,LETTUCES,56
```

Line 50 is where the data to be used is decided on. If the customer has requested fruit then the data pointer is reset to line 130, or else it goes to 150, where the vegetable data is held. Therefore, `RESTORE` tells the computer where to read the data from.

Finally, I'm going to look at one more statement.This is `RND`, which allows the choosing of an event randomly. In using it I'm also going to reveal the secret of how books are written! What I'm going to show you is how authors write their books in this modern age. How? They get their computers to write for them!

AM-WRITER: Using RND

All this talk of data leads to the obvious way of writing text. Let the computer randomly generate text for you. This is very easy and the most difficult part is deciding upon the phrases to use. The next program uses arrays to set up four different variables. I have allowed room in the `DIM` statement for four entries but have only used up half the room in my `DATA` section. See if you can add to the phrases, or if you prefer (and you probably will!), try to change them. This is one situation where a good grasp of a 'human' language helps.

Program 5.6 reads the standard phrases into arrays and then randomly generates a sentence by choosing in every case one out of two possible replies for each of four parts of the sentence. I have used the `RND` function to choose randomly either the number 0 or 1. If you add more data then you can increase the random number and therefore the possible kinds of reply. The `RND` function is very simple and lines 130 to 160 show this. If you wanted to choose a random number between 1 and 125 you would perhaps write a statement like this:

```
variable = INT(RND * 125)
```

The number in brackets indicates the upper range to be chosen. If you require a whole number to be chosen, use `INT`. This rounds

the number to the nearest integer. Type in Program 5.6 and see what happens.

Program 5.6

```
10 CLS
20 DIM phrase1$(4),phrase2$(4),phrase3$(4),
   phrase4$(4)
30 GOSUB 60
40 GOSUB 100
50 END
60 FOR x= 0 TO 1
70 READ phrase1$(x),phrase2$(x),phrase3$(x),
   phrase4$(x)
80 NEXT
90 RETURN
100 GOSUB 130
110 PRINT phrase1$(x1);" ";phrase2$(x2);" ";
    phrase3$(x3);" ";phrase4$(x4)
120 RETURN
130 x1=RND(1)
140 x2=RND(1)
150 x3=RND(1)
160 x4=RND(1)
170 RETURN
180 DATA In this modern world,life has become
    unbearable
190 DATA and the situation is rapidly
200 DATA degenerating to a never ending abyss
210 DATA Since the advent of computers
220 DATA mankind has gone on the road to self
    destruction
230 DATA and may never return from,declining
    standards
```

Program 5.6 is a mere sample of what can be achieved by randomly re-arranging phrases. For example, you may get the following phrases when you run the program:

```
In this modern world mankind has gone on the
road to self destruction and the situation is
rapidly declining standards

Since the advent of computers life has become
unbearable and may never return from
degenerating to a never ending abyss
```

Well, I'm not really a doom merchant but it is interesting to construct different parts of a sentence and end up with something that sounds grammatically correct even if it doesn't always make sense! With careful thought you may be able to get your Amstrad to generate standard reports, essays and projects. Well, maybe!

That finishes our look at arrays and data for the moment but we will be returning to them later on when we use them for much bigger and better things. The use of the **DIM**, **READ** and **DATA** statements form the core of programs that have an inbuilt knowledge and we will find them invaluable for storing large quantities of information.

6 Always the Unexpected!

When writing programs for yourself it is often easier to take short cuts. This may be fine if you are to be the only user of the program but most of the time your piece of software will be used by others, and this is where problems will occur. Unless you are a mind reader, or have any other such powers, the people using your program will no doubt confound your masterpiece by doing the unexpected. They might ignore your instructions and type something outside the range of data in the program, or accidentally press the wrong key. Now you can't be expected to think of everything but where possible you can take certain measures to reduce human error, and that's what this chapter is all about—trapping the unexpected.

Up to now, in the programs we have written, we have given no thought to trapping errors that may occur during the use of a program. So, what kind of things can happen? Pressing the ESC key by accident? Entering the wrong information? These are common pitfalls in programs. It is essential to try to eliminate as many of these gaps in your programming as possible. Let's start with a well-worn phrase in computer sales language: user friendliness.

USER FRIENDLINESS:
Screen display, error trapping and rigid input

How many times have you heard that phrase before? How many programs have you bought which make that claim, but when you first try them leave you totally confused? Perhaps the programs are user friendly. Friendly that is, to the person who wrote them! The first place programs can cause confusion is in their instructions. If necessary, provide clear, concise instructions at the start of a program. If you want a user to interact with the computer, you can't expect him/her to guess what you want them

to do. So don't make it hard for the user.

Next, the prompts you provide within a program are important. In the last chapter, I was guilty of giving no on-screen prompts as to what the user should type in. Many times in this book whilst concentrating on a certain point, and to save your well-worn fingers, I have gone for the quickest route. Often this means that when you run a program you see a question mark and nothing else. If this has frustrated you then you will easily understand the point I'm making. Take Program 6.1 as an example.

I'm going to write a small program representing the process you go through to obtain money from a computerized till outside your bank. In the program you have to enter an account number of six digits and your password of four letters. Then you enter a request for money. Whether you receive any depends on the amount you have in your account, and you can take no more than one hundred pounds out per transaction. So let's write the program.

Program 6.1

```
 10 CLS
 20 PRINT"ENTER DETAILS"
 30 INPUT numb
 40 IF numb=123456 THEN GOSUB 50 ELSE 30
 50 INPUT pass$
 60 IF pass$="USER" THEN GOSUB 80 ELSE 50
 70 RETURN
 80 INPUT"Enter amount of cash required ",cash
 90 GOSUB 110
100 RETURN
110 credit=86
120 IF cash < 100 AND cash < credit THEN GOSUB
    150 ELSE 130
130 PRINT"Sorry - no payment can be made.":END
140 RETURN
150 PRINT"Cash advanced : ";cash
160 RETURN
```

On first running the program you are asked to enter the details and then are faced by a question mark. What do you enter? A number? Your password? The amount you require? If you look at the program you'll know that an account number is first required, and then the password. Let's improve the prompts by changing lines 20 and 50:

```
20 PRINT "Enter account number"
50 INPUT "Enter your personal password ",
   pass$
```

Now that's an improvement. But we can go another step forward in helping the user. What happens if he makes a mistake? Well, the user will then have to retype his entry. But perhaps he doesn't know what his mistake was. Then we should tell the user where he's gone wrong so that next time round he'll hopefully get it right. The changes are as follows:

```
40 IF numb = 123456 THEN GOSUB 50 ELSE GOSUB
   170
60 IF pass$ = "USER" THEN GOSUB 80 ELSE GOSUB
   230
```

and add lines 170 to 270:

```
170 PRINT"Your account number was not valid"
180 PRINT"Remember you need to type a six
    digit"
190 PRINT"number"
200 FOR x=0 TO 2000:NEXT
205 GOTO 10
210 RETURN
230 PRINT"Your password was invalid."
240 PRINT"Please try again"
250 FOR x=0 TO 2000:NEXT
260 GOTO 50
270 RETURN
```

We have now provided extra prompts if the user makes a mistake. Note that I used the `FOR-NEXT` statement to pause for a

short while before returning to the input. In some situations it may be better to return by making the user press a key. You might like to tidy the screen display up a little by clearing the screen every time an input is made, so that if the user makes a number of mistakes, we don't see a trail of error messages. Once you're happy with that move on to the next problem.

The problem is that when the money is approved the program also prints out that no payment can be made. This can be solved by ending the program when the payment is made but really the program is badly structured. Take a look at Program 6.2 which has been structured with subroutines. The screen layout works better when a mistake occurs:

Program 6.2

```
 10 CLS
 20 GOSUB 70
 30 GOSUB 120
 40 GOSUB 170
 50 GOSUB 250
 60 END
 70 CLS
 80 PRINT"Enter account number"
 90 INPUT num
100 IF num=123456 THEN 110 ELSE GOSUB 310
110 RETURN
120 CLS
130 PRINT"Enter personal password"
140 INPUT pass$
150 IF pass$="USER" OR pass$="user" THEN 160
    ELSE GOSUB 370
160 RETURN
170 CLS
180 PRINT"Enter amount of cash requested"
190 INPUT cash
```

```
200 PRINT"You have requested ";cash;". Is this
    correct?"
210 INPUT ans$
220 IF ans$="Y" OR ans$="y" THEN GOSUB 250
    ELSE 230
230 IF ans$="N" OR ans$="n" THEN GOTO 170
    ELSE 200
240 RETURN
250 credit=87
260 IF cash <100 AND cash < credit THEN PRINT
    "Cash request OK,";cash" follows.":END
270 IF cash > 100 THEN PRINT"That is over the
    limit allowed.":END
280 IF cash > credit THEN PRINT"You have
    insufficient funds":END
290 RETURN
310 PRINT"Your account number was not valid"
320 PRINT"Remember you need to type a six"
330 PRINT"digit number."
340 FOR x=0 TO 2000:NEXT
350 GOSUB 70
360 RETURN
370 PRINT"Your password was invalid."
380 PRINT"Please try again"
390 FOR x=0 TO 2000:NEXT
400 GOSUB 120
410 RETURN
```

Note that the program rigidly checks for a certain answer. The amount of rigidity you build into a program is up to you and can depend on the nature of the program itself. If you recall, at the end of Chapter 3, I said that the ASCII program (Program 3.10) had something wrong with it. Did you spot it?

The problem is that in the subroutine that translates an ASCII number to a character there is no error trapping for nonsensical values. For instance, by typing in the value 7 the code is given to

sound a noise. What you need is a check on the number being entered. This is quite simple. By adding the following lines, the program will only accept the values you intend to be entered. (In this case, I have restricted the values to between 32 and 126 which is the range for the entire character set, though this does not take into account any of the user-defined characters which you could set up yourself.) Load in your ASCII menu, Program 3.10, and add the following lines:

Program 6.3

```
255 IF code < 32 OR code > 126 THEN GOSUB 300
300 PRINT"The value you entered is outside
    the"
310 PRINT"range for the character set."
320 a$=INKEY$:IF INKEY$ =" " THEN 330 ELSE 320
330 GOSUB 50
340 RETURN
```

If you now run the program you will see that entering an invalid value is now properly trapped.There is one more thing we can add to the program to allow for those misplaced fingers on the keyboard. We can ensure that if the escape key is pressed by accident then the program is re-run from a certain point. Here, I want the program to return to the main menu if that key is pressed. To do this we have to make use of two statements. One we have come across before,and that is the ON statement. The other is the BREAK command which allows you to assign a function to the ESC key when pressed.

KEYING OUT ERRORS: The use of ON BREAK, ON ERROR, ERR, ERL, KEY and KEY DEF

Besides using the command ON as the basis of a decision, as in the menu program, it can also be used to detect the occurrence of the ESC key being pressed in the program. If the ESC key occurs, the program will normally stop. By using the ON BREAK statement you can tell the computer what action to take if it comes across the ESC button being pushed. For instance, if you run Program 3.10 (the complete version), the program stops when you press the escape key. You can prevent this happening by typing in this line:

```
5 ON BREAK GOSUB 50
```

Quite simply, you have told the computer that if the ESC key is pressed, the program should be re-run, which means you will return to the main menu. You could of course specify some other action to be taken, depending on the commands within the given subroutine.

Do take note, however, that if you include this error trap, you must put it in as the last part of a program, otherwise you will not be able to stop the program and will lose it in memory because you will need to reset the machine. The only way you can prevent this happening is to have another command ON BREAK STOP within the program if the ESC is being pressed twice. The other kind of ON statement is the ON ERROR command. If an error is detected within the program then the program is sent to a line number within the program, i.e. ON ERROR GOTO 60.

Remember to enter this statement as the last statement in a program or you will find debugging difficult. It is all too easy to forget that you have used an ON ERROR statement in a program and spend time re-running a program and getting the same fault without seeing what is going on.

Another useful set of commands are those which enable you to redefine the function of a key. There are two commands that can be used. KEY redefines a new function key and gives the user the opportunity to assign a command to the pressing of a key. Therefore, to set up the command NEW you would type:

```
KEY 140, "NEW" + CHR$(13)
```

This tells the machine that when CONTROL and ENTER are pressed the command NEW will be carried out. (For a list of the relevant numbers, see the back of your user guide.)We can also change any key on the keyboard to represent another character. For instance, we could ensure that the Q key, when pressed, prints A to the screen. The command we use to do this is KEY DEF and goes like this:

```
KEY DEF 67, 1, 65
```

The first number is the key to redefine (these numbers can be found at the back of your user guide); the second indicates whether a key is repeated. 1 tells the machine that the key repeats; 0 disables the repeat. The third number refers to the ASCII code associated with the character. Therefore, in our example the number 65 stands for the letter A. By adding two

further parameters, SHIFT and CONTROL keys can also be used in conjunction with the pressing of a key.

In the same way, you can set up the other function keys and make entry of replies easier for the user. For instance, you could set up function keys to type in YES or NO, NORTH, SOUTH etc., by just pressing one or two keys. This makes a program far more friendly and easier to use.

Finally, we can tell the user what mistake has been made by using the statements ERR and ERL. ERR returns an error number, a list of which can be found at the back of your user guide, and you can use it to provide helpful information to the user. ERL gives the line number at which an error occurred. Both these commands are very useful in error trapping.

Error trapping is important in all programs and makes life much easier for the user of a program. It only requires a little more effort but can improve the way a program is used and presented. At the end of the day it falls on your shoulders as the programmer to ensure that the error trapping is good and how good it is depends on your skill and the thought you put into the program. Always assume that the person using the program knows very little, perhaps even nothing!

In the next chapter we will see some more uses of error trapping. As I mentioned earlier a number of the programs in this book have deliberately not been error trapped and it would be worthwhile to spend a few minutes trying to improve them. Try using flowcharts to work out the sequence and action of your program and to see where a program should go if an error occurs. It might take a little extra time but you will be rewarded with a smooth-running program.

7 Text Handling

We have seen in previous chapters how to use the variety of commands in Amstrad BASIC to manipulate the input of information. In this chapter we go a little bit further and look at ways of making your programs more attractive by inserting routines to print out text in different ways. These are suggested alternatives to just printing a reply in one large chunk on the screen. Personal taste has a lot to do with the way a screenful of information is presented but a program can be given a very different feel by the way it is designed.

Let me give you an example. Program 7.1 is what I call my intruder alert. The program leaves a message on the screen, warning anyone who comes near not to touch the keyboard. Well, humans being naturally inquisitive, someone will almost certainly touch the keyboard. And that is where we can have some fun and put a number of principles we have learnt into practice. I won't tell you any more for the moment; instead, type in the program. Keep to the line numbers shown as we will be adding to the program later:

Program 7.1

```
10 ZONE 40
20 MODE 0
30 PAPER 5:BORDER 2:PEN 10:CLS:PRINT,,,
40 PRINT SPC(4);"DO NOT TOUCH"
50 PRINT,,,:PRINT SPC(6);"KEYBOARD"
60 a$=INKEY$:IF a$<>"" THEN 70 ELSE 60
70 CLS
80 MODE 1:PAPER 4:BORDER 7:PEN 10:CLS
```

```
 90 anyphrase$="OH DEAR I DID WARN YOU.
        YOU'VE REALLY DONE IT NOW!!!!"
100 PRINT anyphrase$
110 ENV 1,4,3,1,1,0,19,7,-120,4
120 SOUND 1,100,7,14,1
130 a$=INKEY$:IF a$<>"" THEN 140 ELSE 120
140 MODE 0
150 INK 0,4,7:CLS:PRINT,,,,
160 PRINT SPC(5);"TOO LATE!!!!"
170 WHILE x=0
180 ENV 1,4,3,1,1,0,19,7,-120,4
190 SOUND 1,100,7,14,1
200 WEND
```

Now run the program and press any key. You will see an error message come up and the alarm will sound. You can stop the program by pressing ESC. To complete the error trap you can use ON BREAK but do remember to have saved the program first as you will not be able to break out of the program once you have used this error trap. If you press another key a further message is displayed informing you that you are too late. Note how I've error trapped in the program to direct the program to different lines. You can improve upon this by using the error trapping methods in the previous chapter. Line 30 sets the colours of the paper, border and pen and you can change this as you like depending on the monitor you are using and the effects that you prefer. Line 60 waits for the keyboard to be touched, and when it is, the next page is presented. The WHILE loop at the end of the program sends the sound command into an indefinite loop because x always equals zero.

Line 110 is the basis of the alarm, defining the characteristics of the noise using the ENVELOPE command and line 120 plays the sound. The rest of the program should contain familiar commands. You might like to alter the program to run in different screen modes and to create different display effects. Figure 7.1 shows a flowchart of the program.

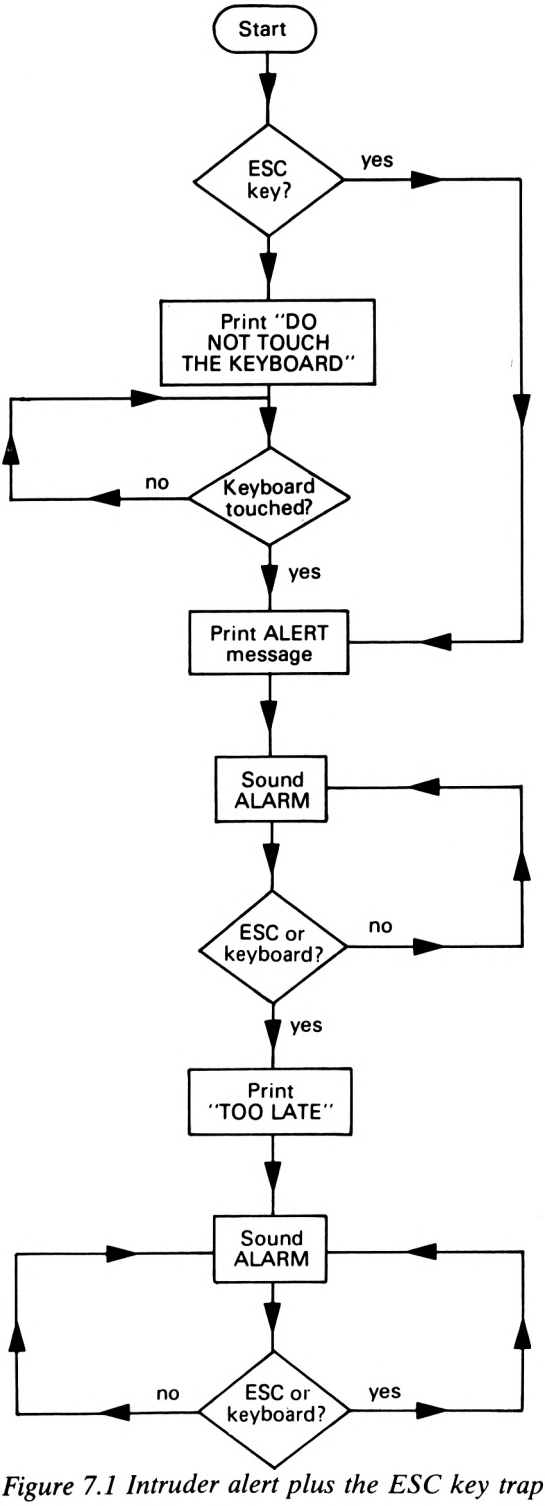

Figure 7.1 Intruder alert plus the ESC key trap.

The program does its job well enough but there is no sense of suspense in the presentation of the message. Suspense? Well, yes! We can make the two sentences appear more interesting and add life to the program by typing the sentences out slowly. To do this we must think of how to type text out in a slow teletype manner.

TELETYPE: A slow teletype routine

We have already used commands like **MID$** to print out, selectively, characters in a string. We can use these principles to form a short program to type a string phrase slowly, letter by letter. To do this, first input Program 7.2:

Program 7.2

```
 5 ZONE 40
10 CLS
20 anyphrase$="OH DEAR I DID WARN YOU.
       YOU'VE REALLY DONE IT NOW!!!!"
30 PRINT,:FOR x%=1 TO LEN(anyphrase$):PRINT
   MID$(anyphrase$,x%,1);
40 GOSUB 50:NEXT:PRINT:END
50 FOR a%=150 TO 0 STEP-1:NEXT:RETURN
```

Run the program and you will see the contents of anyphrase$ printed on the screen very slowly. The idea is very simple. Firstly, line 40 sets up a loop where a letter of the phrase is printed out in order every time around the loop. But just doing that is not sufficient as the machine would work too fast to see this happening. So we have to insert a time delay loop to slow up the program. GOSUB 60 at line 50 calls a time delay subroutine before the next letter can be printed out. The subroutine is just a FOR-NEXT loop which is going around in a loop doing nothing.

We could incorporate that principle in Program 7.1 to give a much more dramatic effect. Make the following changes to the program:

Change line 100 to:

```
100 PRINT,: FOR X% = 1 TO LEN(anyphrase$):
    PRINT MID$(anyphrase$,x%,1);
```

and add lines 105 and 107:

```
105 GOSUB 107: NEXT: PRINT
107 FOR a% = 150 TO 0 STEP - 1: NEXT: RETURN
```

Run Program 7.1. I think you will see that we have obtained a more dramatic effect by slowing the typing of the text. It is matter of personal preference as to whether you use this kind of display in your programs but the slow teletype routine gives the impression that the machine is speaking back.

Try adapting some of the programs in this book to using the teletype text idea. The method you employ in presenting text can enhance or detract from a program and it's always interesting to experiment with different effects. The string handling commands in Chapter 3 are particularly useful for novel effects. Moreover, the most important thing is that you effectively communicate your message to the user. After all that is what interaction is all about.

8 Sigmund: An Interactive Program

We have seen how to use a wide range of commands to manipulate a text input, but we have yet to write a program that appears intelligent. Well the time has now come to write a large program that will fully interact with the user. In the next two chapters we will write Sigmund, which is a computer-based psychiatrist (no prizes for guessing who I was thinking of when I named the program). In this chapter we will look at the planning of this program and the techniques employed. The program is in Chapter 9, but don't type it in yet. Read through this chapter first to get an idea about the principles behind the program, so that when you come to type it in you will understand its workings better.

The first thing to do is to give you some background to the program. The idea behind Sigmund is based around one of the first artificial intelligence programs to be written—Eliza. There have been many such programs and research still continues today for computers that are 'intelligent'. Programs like Eliza were written on mainframe computers to simulate a conversation between a patient and doctor. Eliza fooled many people who used it and they were convinced that the computer really understood their problems. As you will see in the Sigmund program it is no more than a bluff, but extremely effective.

What is the aim of the program? Sigmund is intended as a simulation of talking to a psycho-analyst though it should be pointed out that Sigmund behaves like no analyst I've ever met! The concept behind this program is to make the user believe that he or she is having a real conversation with someone, and that their conversation is being listened to and intelligently answered. Now this is quite a tall order but we have learnt ample material to put together a program of this nature. So where does one start on

a project like this?

You've already seen how allowing the user to enter anything he likes can cause problems. We cannot build in the entire Oxford dictionary, nor a complete set of grammatical rules (even we if understood them!), to cover all the intricate subtleties of the English language. What we need to achieve is a way of simulating an acceptable reply to the user without resorting to typing in every possible answer which is, unless you know better, an impossible task. What we can do is to make some assumptions about the way humans behave and think.

As human beings we have terrific imaginations. It is this which has led to much of our inventiveness. But an ability to imagine hypothetical events and abstract ideas has also led to us being able to read more into a situation than evidence would permit. Most of us at some time in our lives have done this and that is precisely what Sigmund relies on to fool the user. The assumption is that a human user will read more into the reply given than actually exists. Now that is not as naive an idea as it may at first sound.

Think of a conversation with a friend. During part of that conversation you will be listening and will make occasional comments back, comments that show you are listening and have understood what has been said. When you see an analyst, you will, as the client, do much of the talking and the analyst will encourage you to talk by making the occasional comment back. What we could do then, is just have a number of phrases which the computer can turn out randomly no matter what you say. But that would not be good for very long. For instance, imagine if we had four stock phrases with which to reply to the user. These phrases might be the following:

```
I SEE
THAT IS VERY INTERESTING
COULD YOU TELL ME MORE?
HMMM. CARRY ON PLEASE
```

With these four answers, and the `RND` function, we could allow the user to type in anything he liked, that would be followed by a randomly chosen reply, and then the whole process would be repeated. Figure 8.1 shows a possible flowchart for such a program.

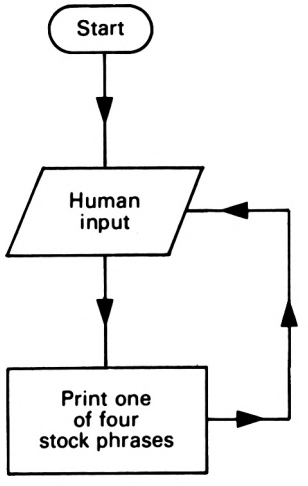

Figure 8.1 The RND function.

But this is far from satisfactory. Imagine the conversation that would ensue:

```
   HUMAN:  I HAVE TERRIBLE DREAMS
COMPUTER:  COULD YOU TELL ME MORE?
   HUMAN:  I DREAM ABOUT SEAGULLS
COMPUTER:  THAT IS VERY INTERESTING
   HUMAN:  I'M VERY DEPRESSED OVER THIS
COMPUTER:  I SEE
   HUMAN:  WOULD YOU SAY IT'S ABNORMAL
COMPUTER:  HMMM. CARRY ON PLEASE
```

Now you wouldn't be able to keep up a conversation for very long if you just used stock phrases and of course, the program soon falls down when the computer is asked a question. The phrases asked by the human above were typed into the Sigmund program and the replies were very different. Look at the result below:

```
   HUMAN:  I HAVE TERRIBLE DREAMS
COMPUTER:  DO YOUR DREAMS INVOLVE OTHER PEOPLE?
   HUMAN:  NO I DREAM ABOUT SEAGULLS
```

```
COMPUTER:  HOW OFTEN DO YOU DREAM ABOUT
           SEAGULLS

   HUMAN:  OFTEN. WOULD YOU SAY IT'S ABNORMAL

COMPUTER:  I'M NOT PREPARED TO SAY WHAT I THINK
```

The answers to what I typed in appear to have been totally understood by the computer and an intelligent reply was made. But, of course, you know that the secret must lie somewhere in the program. What I have done is to go a few steps further than the stock phrase idea. The computer is still drawing on prepared answers but there is more latitude in the way an answer is given. It all depends on what is typed in by the user.

PATTERN MATCHING FOR KEYWORDS

Sigmund uses a method of looking for a familiar word in the sentence typed in by the user. In other words it searches for a keyword using text-handling commands like `INSTR`. It does so in a pre-defined order, first searching for a verb or adverb and if none is found (according to the pre-defined list in the `DATA` statements) then a search is made for a subject, i.e. `I, YOU, THEY` etc. If none of these are found in the sentence then a stock phrase like those shown above is trundled out. But only as a last resort.

Therefore, the computer goes through a series of possibilities and decides which action to take. The way this is done is by storing a range of replies, some in part and others complete. Take a look at the following `DATA` statement:

```
DATA DREAM, HOW OFTEN DO YOU DREAM %, CAN YOU
REMEMBER YOUR DREAMS?, I ENVY YOU...I CAN'T
DREAM
```

This is typical of the `DATA` fed into Sigmund and it holds all or part of a reply. The first word is the keyword and this is read into a two-dimensional array and the three remaining phrases are the replies associated with that keyword. The `%` sign tells the computer to tag on the rest of the sentence typed in by the user after the keyword. For example, if the user typed in:

```
I DREAM ABOUT SHIPS
```

the answer might be:

```
HOW OFTEN DO YOU DREAM ABOUT SHIPS
```

The remaining part of the user's input `ABOUT SHIPS` is taken from the point after the end of the keyword and placed at the position where the `%` sign occurs. Using `INSTR`, a search is made of the reply phrase. If a `%` sign is found, then the tag is made. So if you look through Sigmund you will see the subroutine `GOSUB 400` which contains the line:

```
look = INSTR(getreply$,"%")
```

The subroutine `GOSUB 430` carries out the printing of the reply with the appropriate, tagged-on phrase.

As subjects are likely to occur more often within a sentence they are not used until the preliminary search has failed. If used they can simply turn a sentence around. For instance, if the user types:

```
I'VE GOT A BUNCH OF FLOWERS
```

the computer will reply with:

```
YOU'VE GOT A BUNCH OF FLOWERS
```

giving the impression of confirming what the human has said. What takes place is that the keyword `I'VE` is found and in its place is substituted the phrase `YOU'VE` and the remainder of the user's phrase is tagged on to the end.

Finally, if no keyword is found, then one of the stock phrases is used. The program relies heavily on writing replies which will be suitable to a number of situations where a keyword is used. This is difficult and at times the result will not always be as intended but it works for a good percentage of the time. There are improvements that could be made to the program but I'll leave that until the next chapter when you have typed in Sigmund.

A final thought. Are programs like Sigmund a sign of some intelligence inside the computer and the program, or merely an indication of how we can take advantage of human gullibility?

9 Sigmund: The Program

At long last! We've finally reached our first truly communicative, interactive program. Sigmund is unpredictable. At times you will find him caring and attentive; at other times, insulting. Most of the time you will feel he's human but there are moments when the computer in him will strongly show itself. Rather than follow the way programs like this have been used before, I have made Sigmund's replies varied so that the kind of response he may give can not be so easily gauged. Type in the program carefully as it is quite long and use the AUTO function for line numbers, as I have kept them evenly spaced. More details and comments about the program follow the listing:

SIGMUND: The Program

```
 10 MODE 2
 20 ZONE 80
 30 PRINT"HELLO MY NAME IS SIGMUND.","WHAT IS
    YOUR NAME"
 40 INPUT name$
 50 PRINT,,,"NICE TO MEET YOU ";name$
 60 PRINT,"TELL ME WHY YOU WANT TO TALK TO ME"
 70 CLEAR
 80 DIM gramin$(40),gramout$(40),match$(30,3)
 90 GOSUB 130
100 GOSUB 260
110 GOSUB 290
```

```
120 END
130 REM READ gram
140 RESTORE 780
150 FOR datain=0 TO 40
160 READ gramin$(datain),gramout$(datain)
170 NEXT
180 REM read match$
190 RESTORE 900
200 FOR datain=0 TO 30
210 FOR reply=0 TO 3
220 READ match$(datain,reply)
230 NEXT reply
240 NEXT datain
250 RETURN
260 PRINT,":":LINE INPUT phrase$
270 IF phrase$="END" THEN GOSUB 730
280 RETURN
290 FOR datain=0 TO 30
300 find=INSTR(phrase$,match$(datain,0))
310 search=LEN(match$(datain,0))
320 IF find >0 THEN GOSUB 360
330 NEXT
340 IF find >0 THEN GOSUB 360 ELSE GOSUB 480
350 RETURN
360 x=INT(RND*(4)):IF x=0 GOTO 360 ELSE 370
370 getreply$=match$(datain,x)
380 GOSUB 400
390 END
400 look=INSTR(getreply$,"%")
410 IF look > 0 THEN GOSUB 430
420 PRINT,getreply$:GOTO 70
```

```
430 length=LEN(getreply$)
440 replyless$=LEFT$(getreply$,(length-2))
450 length2=LEN(phrase$)
460 tg$=RIGHT$(phrase$,(length2-(find+
    search))+1)
470 PRINT,replyless$;tg$:GOTO 70
480 RESTORE 780:FOR datain=0 TO 40
490 sub1=INSTR(phrase$,gramin$(datain))
500 sub1len=LEN(gramin$(datain))
510 IF sub1>0 THEN GOSUB 550
520 NEXT
530 IF sub1>0 THEN GOSUB 550 ELSE GOSUB 660
540 RETURN
550 swap1$=gramout$(datain)
560 length2=LEN(phrase$)
570 length3=LEN(swap1$)
580 tg$=""+RIGHT$(phrase$,(length2-(sub1+
    sub1len))+1)
590 GOSUB 610
600 RETURN
610 FOR datain=0 TO 40
620 subs2=INSTR(tg$,gramin$(datain))
630 IF subs2>1 THEN GOSUB 660
640 NEXT
650 IF subs2>1 THEN GOSUB 660 ELSE PRINT,
    swap1$;tg$:GOTO 70
660 counter=INT(RND*(5))
670 ON counter GOTO 680,690,700,710
680 PRINT,"PLEASE TELL ME MORE":GOTO 70
690 PRINT,"HMMM! THATS VERY INTERESTING":
    GOTO 70
700 PRINT,"DO CARRY ON":GOTO 70
```

```
710 PRINT,"I SEE":GOTO 70
720 RETURN
730 PRINT,"ARE YOU SURE YOU WANT TO END THIS
    CHAT?"
740 a$=INKEY$:IF a$="" THEN 740
750 IF a$="Y" THEN 760 ELSE 70
760 CLS:END
770 RETURN
780 DATA I'VE,YOU'VE,I'M,YOU'RE,I AM,YOU ARE,
    I HAVE,YOU HAVE
790 DATA I WAS,YOU WERE,I WILL,YOU WILL,YOURS,
    MINE,MY,YOUR
800 DATA ME,YOU,YOU'RE,I'M,YOU ARE,I AM,
    YOU HAVE,I HAVE
810 DATA YOU'VE,I'VE,YOU WILL,I WILL,YOU'LL,
    I'LL,YOU WERE
820 DATA I WAS,THEY ARE,THEY ARE,SHE HAS,
    SHE HAS,HE HAS
830 DATA HE WAS,WE ARE,WE ARE,THEY'RE,THEY'RE,
    SHE IS,SHE IS
840 DATA HE IS,HE IS,WE'RE,WE'RE,THEY HAVE,
    THEY HAVE,SHE'S
850 DATA SHE'S,HE'S,HE'S,WE HAVE,WE HAVE,
    THEY'VE,THEY'VE
860 DATA HE WAS,HE WAS,SHE WAS,SHE WAS,WE'VE,
    WE'VE,THEY
870 DATA THEY,HE WILL,HE WILL,SHE WILL,
    SHE WILL,WE WILL
880 DATA WE WILL,THEY WILL,THEY WILL,SHE HAS,
    SHE HAS
890 DATA THEY WERE,THEY WERE,SHE,SHE,HE,HE,
    WE,WE,YOU,I,I,YOU
900 DATA CAN YOU,OF COURSE I CAN,I CAN DO
    ANYTHING,I WILL TRY TO %
910 DATA CAN I,YOU CAN DO WHAT YOU LIKE,
    YOU CAN %
```

```
 920 DATA YOU ARE RIGHT TO ASK ME
 930 DATA WOULD YOU,I WOULD NOT %,I WOULD %
 940 DATA I'M NOT PREPARED TO SAY WHAT I THINK
 950 DATA WOULD I,OF COURSE YOU WOULD %
 960 DATA DON'T YOU KNOW WHAT YOU WOULD DO ?
 970 DATA I WOULD IF I WAS IN YOUR PLACE
 980 DATA HAVE YOU,WHAT I HAVE IS NOTHING TO DO
     WITH YOU,I HAVE %
 990 DATA I DON'T WISH TO TELL YOU THAT
1000 DATA HAVE I,IS THAT A RHETORICAL QUESTION?
1010 DATA DON'T YOU KNOW WHETHER YOU HAVE %,YOU
     SEEM VERY UNSURE
1020 DATA DREAMS,DREAMS ARE A RELEASE VALVE FOR
     YOUR SUBCONSCIOUS
1030 DATA DO YOUR DREAMS INVOLVE OTHER PEOPLE?
1040 DATA DO YOU LIKE DREAMS %
1050 DATA DREAM,HOW OFTEN DO YOU DREAM %
1060 DATA CAN YOU REMEMBER YOUR DREAMS?
1070 DATA I ENVY YOU...I CAN'T DREAM
1080 DATA COMPUTERS,ARE YOU WORRIED ABOUT
     MACHINES?
1090 DATA AS A COMPUTER I TAKE A DIFFERENT VIEW
1100 DATA I'D MUCH RATHER TALK TO A COMPUTER
     THAN YOU
1110 DATA COMPUTER,WE ARE HERE TO TALK ABOUT
     YOU NOT ME
1120 DATA COMPUTERS ARE PERFECT
1130 DATA YOU ARE PRIVILEGED TO TALK TO ME
1140 DATA MACHINE,DO MACHINES WORRY YOU?
1150 DATA YOU SEEM TO BE CONCERNED ABOUT
     MACHINES
1160 DATA I'M NOT LIKE OTHER MACHINES
1170 DATA DEPRESSED,DO YOU FEEL DEPRESSION IS
     NOT NORMAL?
```

```
1180 DATA HOW OFTEN ARE YOU DEPRESSED?
1190 DATA WHY ARE YOU DEPRESSED %
1200 DATA MAD,IS THERE ANY MADNESS IN YOUR
     FAMILY?,YOU MUST BE MAD
1210 DATA MAD %
1220 DATA MOTHER,WHAT IS YOUR MOTHER LIKE?
1230 DATA ARE ALL MOTHERS THE SAME?
1240 DATA MY MOTHER IS A SILICON CHIP
1250 DATA FATHER,TELL ME ABOUT YOUR FATHER
1260 DATA WHAT IS YOUR RELATIONSHIP LIKE,WHAT
     DO YOU TALK TO YOUR PARENTS ABOUT?
1270 DATA SISTER,I LIKE MY SISTER
1280 DATA THIS SOUNDS LIKE A COMPLICATED
     RELATIONSHIP
1290 DATA WHAT ARE YOUR FEELINGS TOWARDS YOUR
     SISTER?
1300 DATA BROTHER,WHAT WOULD YOU DO WITHOUT A
     BROTHER?
1310 DATA DOES YOUR BROTHER ANNOY YOU,I HAVE
     NO BROTHER
1320 DATA DISLIKE,WHAT IS IT THAT YOU DISLIKE
     ABOUT %
1330 DATA I DISLIKE HUMANS
1340 DATA IT IS BETTER TO LIKE THAN NOT
1350 DATA FEELINGS,WE ALL HAVE FEELINGS
1360 DATA DO YOU OFTEN FEEL %
1370 DATA FEELINGS SHOULD BE EXPRESSED
1380 DATA LOVE,LOVE IS AN ILLOGICAL STATE
1390 DATA HOW DO YOU KNOW THAT YOU ARE IN
     LOVE %
1400 DATA I WAS IN LOVE ONCE BUT.....SORRY
     DO CARRY ON
1410 DATA HATE,I HATE YOU,WHAT DO YOU HATE
     ABOUT %
```

```
1420 DATA LOVE IS FAR BETTER THAN HATE
1430 DATA SORRY,THERES NO NEED TO BE SORRY
1440 DATA I HATE PEOPLE WHO THINK THEY'RE
     SORRY,YES?
1450 DATA APOLOGISE,DO NOT APOLOGISE,WHY
     APOLOGISE ABOUT %
1460 DATA APOLOGIES ARE SELDOM MEANT
1470 DATA CAUSE,THE CAUSE IS ALWAYS DIFFICULT
     TO FIND
1480 DATA WHAT IS THE CAUSE %
1490 DATA CAUSES. THAT'S ALL HUMANS WANT TO
     KNOW
1500 DATA WHAT,IS THAT A RHETORICAL QUESTION?
1510 DATA WHAT A QUESTION!,QUESTIONS!
     QUESTIONS!
1520 DATA HOW,WHY ASK ME %
1530 DATA ASK YOURSELF WHETHER THIS IS RELEVANT
1540 DATA WHY DO PEOPLE ALWAYS ASK ME?
1550 DATA THINK,YOU HAVEN'T THE BRAINS TO THINK
1560 DATA ONLY COMPUTERS CAN THINK
1570 DATA I THINK THEREFORE I AM
1580 DATA WHEN,I CAN'T PREDICT THAT,I JUST
     DON'T KNOW
1590 DATA YOU MUST WORK IT OUT
1600 DATA WHY,WHY NOT?,WHY ISN'T THAT OBVIOUS?
1610 DATA WHY? WHY? WHY? I DON'T KNOW
1620 DATA YES,THATS MORE POSITIVE
1630 DATA ARE YOU SURE YOU MEAN YES?
1640 DATA I LIKE POSITIVE ANSWERS
1650 DATA NO,THATS VERY NEGATIVE
1660 DATA CAN'T YOU BE MORE POSITIVE?,NO?
```

If you can't easily follow how the program is working, here is a list of the subroutines and their functions.

`GOSUB 130` Initializes the arrays by reading in the data that contains the keywords.

`GOSUB 260` Requests an input from the user.

`GOSUB 290` Searches for a keyword, i.e. `Dream`, `Brother`, `Computer`.

`GOSUB 730` Checks whether the user wishes to end the chat.

`GOSUB 360` If match was found in subroutine `GOSUB 290` then a reply is chosen randomly from the three replies in the appropriate data statement, i.e. if the keyword found was `APOLOGIZE`, the `DATA` statement might look like this:

```
1110 DATA APOLOGIZE, DO NOT APOLOGIZE,
     WHY APOLOGIZE ABOUT %,
     APOLOGIES ARE SELDOM MEANT
```

`GOSUB 480` Checks for the occurrence of a subject, i.e. `I`, `YOU`, `HE`, `WE` etc.

`GOSUB 400` Looks to see if a `%` sign is in the data reply.

`GOSUB 430` Puts together the reply and the tagged-on phrase.

`GOSUB 550` Swaps the first subject found with the opposing phrase, i.e. `I` becomes `YOU` and then tags the remainder of the phrase after the subject.

`GOSUB 610` Checks to see if more than two subjects appear in the sentence. If so, it goes to `GOSUB 660`, or else prints the swapped subject and tagged phrase.

`GOSUB 660` Prints out one of four stock phrases.

FLOWCHARTS and DIAGRAMS

To help you understand the program, I include here a flowchart of the program and separate flowcharts of the subroutines. There is also a diagram to show you the relationships between the arrays and the data statements.

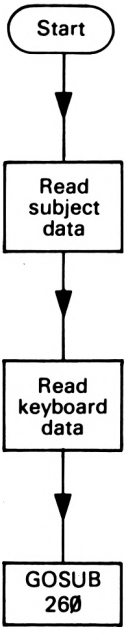

Figure 9.1 Subroutine at 13Ø.

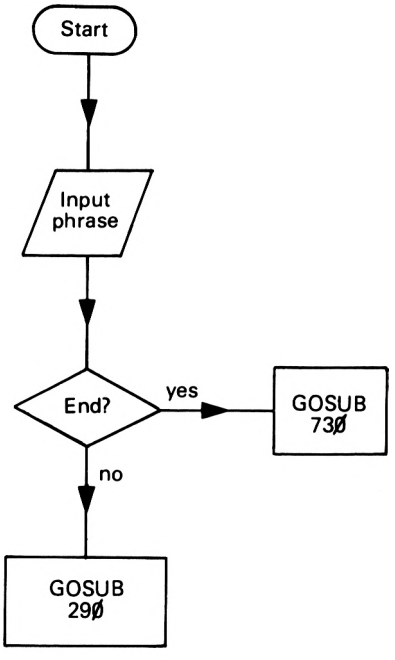

Figure 9.2 Subroutine at 26Ø.

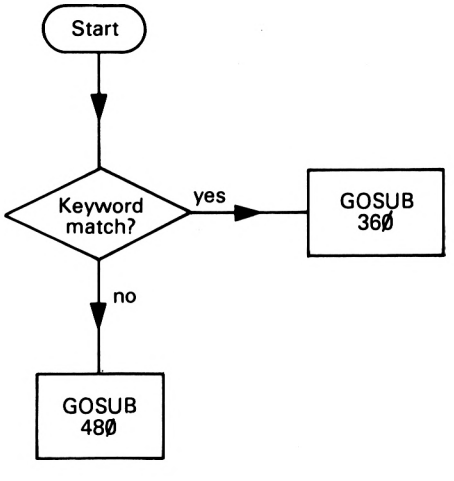

Figure 9.3 Subroutine at 29Ø.

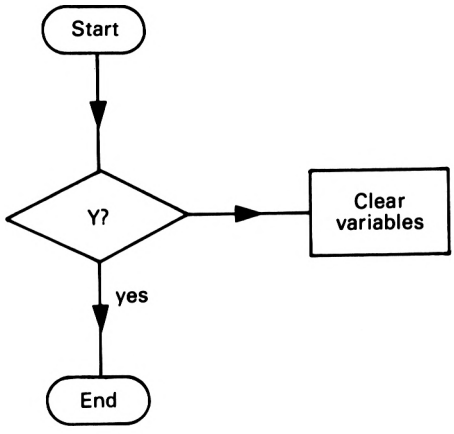

Figure 9.4 Subroutine at 730.

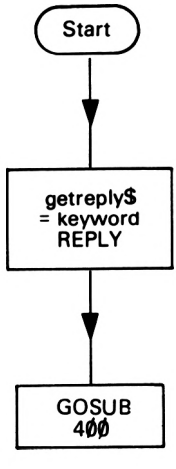

Figure 9.5 Subroutine at 360.

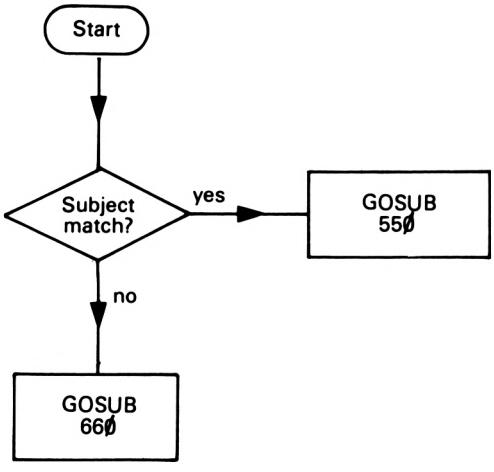

Figure 9.6 Subroutine at 48Ø.

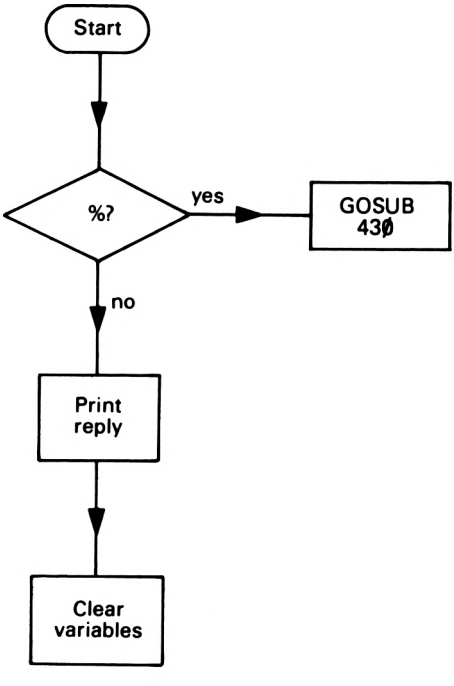

Figure 9.7 Subroutine at 4ØØ.

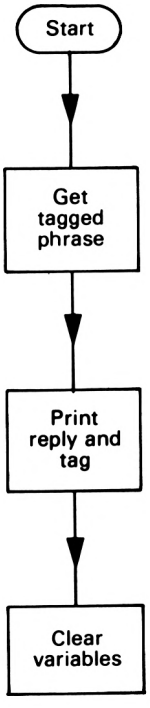

Figure 9.8 Subroutine at 43Ø.

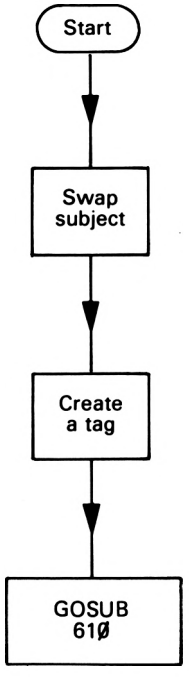

Figure 9.9 Subroutine at 55Ø.

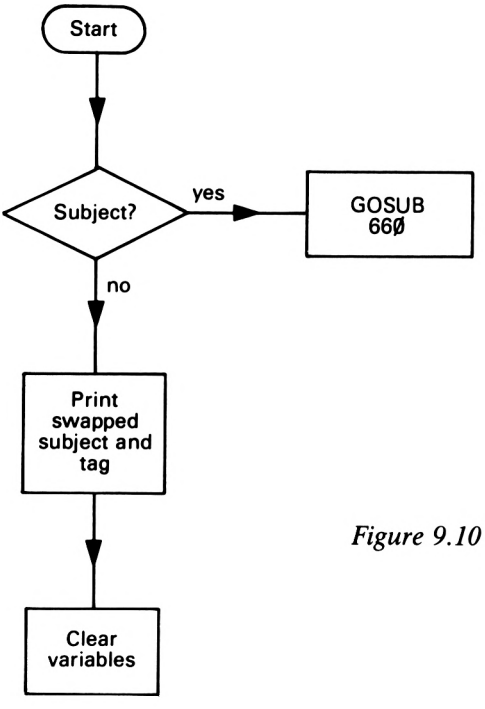

Figure 9.10 Subroutine at 61Ø.

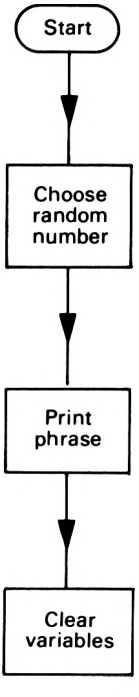

Figure 9.11 Subroutine at 66Ø.

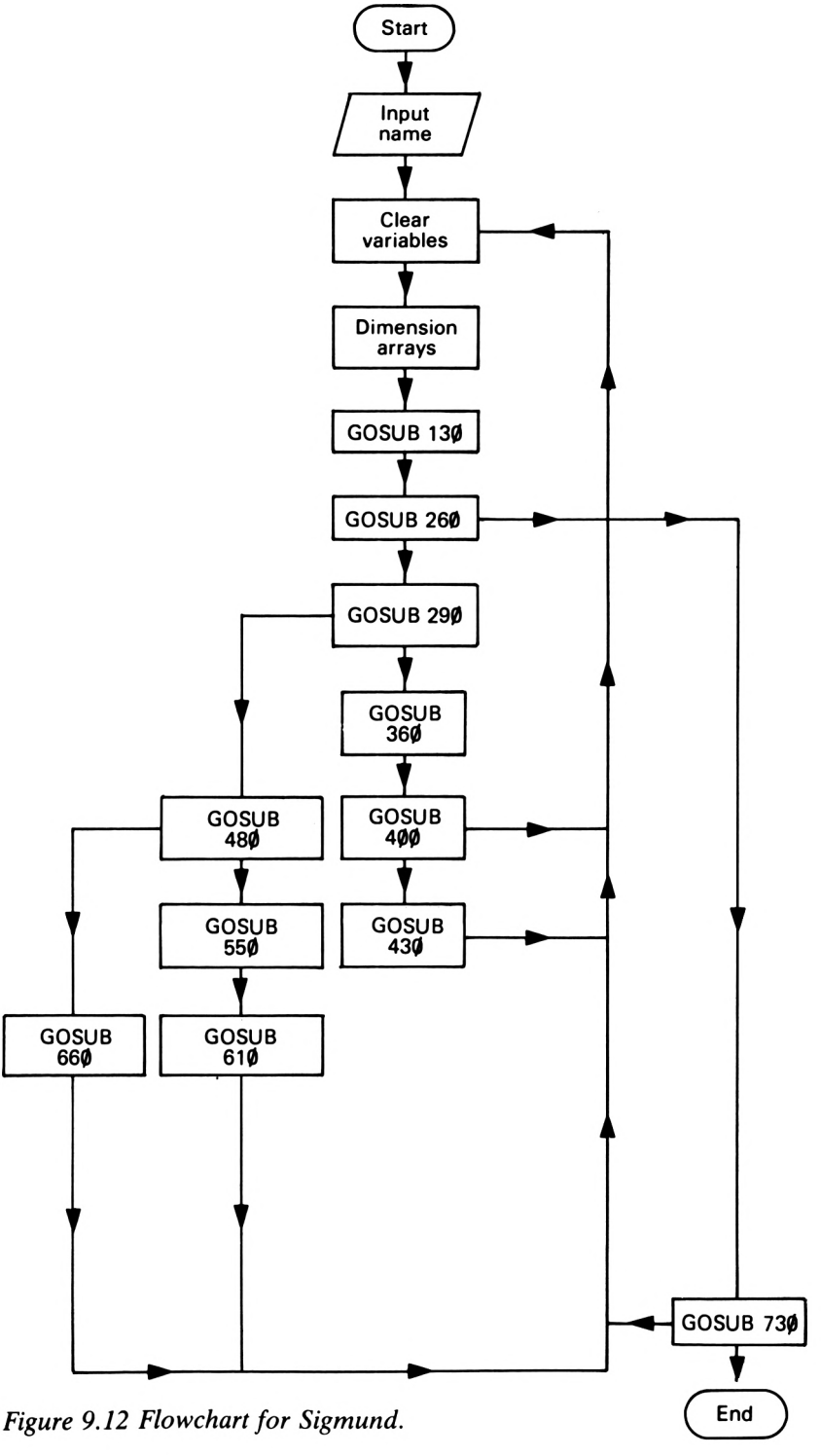

Figure 9.12 Flowchart for Sigmund.

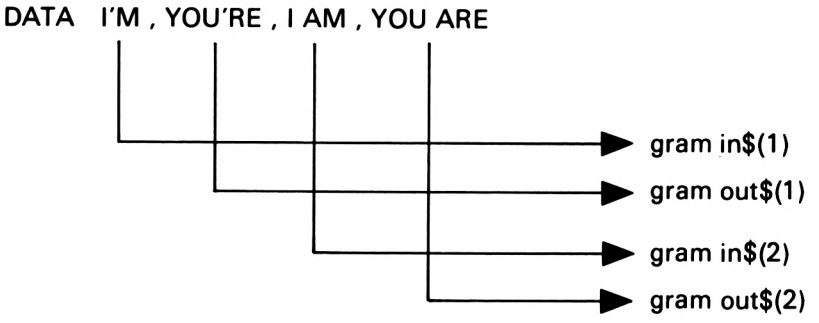

Figure 9.13 Data stored in subject arrays.

IMPROVING THE PROGRAM

Sigmund can keep up a conversation for quite a while, and very effectively but it does have faults. It can't handle more than one subject in the phrase. Therefore, a sentence like `I GOT YOU` will not be handled. Using the principles in the program you could add to Sigmund by swapping the correct subject into the phrase and allowing for more than one subject.

Another way to improve on the program is to personalize it to your own needs. To do this you must change the data statements at the end of the program. For instance, you might want to add the keyword `Amstrad` to Sigmund's dictionary. To do this write the `DATA` statement as follows:

```
2000 DATA AMSTRAD, MY MOTHER WAS AN AMSTRAD,
     DO YOU LIKE AMSTRAD COMPUTERS?, FROM
     LITTLE AMSTRADS GROW...
```

Don't forget to change the dimension of the array, by altering the `DIM` statement of `match$`. Also remember to alter the `FOR-NEXT` loops where `match$` is being used.

The order of the data statements is very important. Sigmund works on the basis of checking for the first keyword it comes across in the list and acting on that word. Try changing the order of the keywords to see how the program is affected. The order can be crucial and I favour putting the more obscure keywords first as they are less likely to be used, and if placed near the bottom of the data statements, a more common keyword will take precedence. You could also make more use of tagging, by placing more operators like `%` in the data statements to bring part of the user's phrase between two elements of your own reply.

Sigmund is an example of the type of program you can construct with a little thought. There is room for improvement but I hope it will give you enough encouragement to start writing your own interactive programs. In the remaining chapters we will look at more ideas on interactive programming and consider the question of artificial intelligence. Oh, one last thing. Do remember what I said about Sigmund not being like any analyst I've ever met. The replies given by Sigmund are, as you have seen, constructed by the programmer and in this situation reflect the mind of the author and not a professional psychologist! Happy counselling!

10 Interviewer

Simulation can be both practical and fun, and in this chapter we're going to look at implementing another interactive program, this time based around an interview. At the same time we will see how to use the Sigmund program as the basis of another.

Just as we decided upon an overall aim for the manner in which Sigmund worked, we will consider what features to build into the Interviewer program. We could make the computer simulate an interview for a job, with the computer taking the part of the interviewer. This means that the computer is going to have to ask questions as well as respond to our answers. Now it would seem a pity not to use the Sigmund program as the basis for this, because we have already programmed him to simulate an intelligent conversationist. Obviously we will have to change some of the `DATA` keyword statements, as some of the replies we have built in would not be suitable for an interviewer to say. How many interviews have you attended where the interviewer said:

```
I HATE YOU
```

or,

```
DOES YOUR BROTHER ANNOY YOU?
```

Not many I'm sure! Now perhaps you like the idea of having a manic interviewer who might insult you, so we might leave in some of Sigmund's replies. But just changing the data statements is not sufficient. We want the computer to ask questions, and on the basis of those answers either discuss the answer or carry on questioning. We can do this by inserting new routines that hold the data for the questions and set up new networks of possible routes for the program to act on.

Let's change Sigmund then and see what we can achieve. Load in Sigmund and then make the following changes:

1. Delete lines 10 to 90 inclusive.
2. Delete lines 660 to 720 inclusive.

Now add the following lines:

```
 10 MODE 2:ZONE 80
 15 MEMORY 43869
 20 POKE 43870,0
 25 PRINT,"HELLO. MY NAME IS THOMPSON. I'M
    THE","PERSONNEL OFFICER FOR SLUDGE & SONS.
    I","WILL BE CONDUCTING THIS INTERVIEW.",
    "YOU ARE....."
 30 INPUT name$
 40 PRINT,"TELL ME SOMETHING ABOUT YOURSELF"
 65 CLEAR:DIM gramin$(40),gramout$(40),
    match$(30,3),question$(3)
 70 GOSUB 130:GOTO 100
 75 CLEAR:DIM gramin$(40),gramout$(40),
    match$(30,3),question$(3):GOSUB 130:
    GOSUB 2000
 80 GOSUB 1900:GOTO 70
660 GOTO 75
680 RETURN
1900 a%=PEEK(43870)+1
1910 PRINT,question$(a%)
1920 POKE 43870,a%
1930 RETURN
2000 FOR x = 1 TO 3
2010 READ question$(x)
2020 NEXT
2030 RETURN
2040 DATA HOW MANY JOBS HAVE YOU HAD UP TO NOW?
2050 DATA WHY DID YOU APPLY FOR THIS JOB?
2060 DATA ARE YOU QUALIFIED FOR THIS WORK?
```

The program now runs differently and the interaction will look something like this:

```
COMPUTER:  MY NAME IS THOMPSON. I'M THE PERSONNEL
           OFFICER FOR SLUDGE AND SONS. I WILL BE
           CONDUCTING THIS INTERVIEW. YOU ARE...?

   HUMAN:  JEREMY

COMPUTER:  PLEASED TO MEET YOU JEREMY. TELL ME
           SOMETHING ABOUT YOURSELF

   HUMAN:  I AM A COLLEGE GRADUATE AND HAVE FIVE
           YEARS EXPERIENCE OF WORK

COMPUTER:  HOW MANY JOBS HAVE YOU HAD UP TO NOW?

   HUMAN:  FOUR

COMPUTER:  WHY DID YOU APPLY FOR THIS JOB?

   HUMAN:  BECAUSE I'M MAD ABOUT BECOMING A RAT
           CATCHER

COMPUTER:  YOU MUST BE MAD

   HUMAN:  YES

COMPUTER:  ARE YOU SURE YOU MEAN YES

   HUMAN:  I THINK SO

COMPUTER:  YOU HAVEN'T THE BRAINS TO THINK

   HUMAN:  LOOK WHO'S TALKING. YOU ARE JUST A
           COMPUTER

COMPUTER:  YOU ARE PRIVILEGED TO TALK TO ME

   HUMAN:  I SUPPOSE I AM

COMPUTER:  ARE YOU QUALIFIED FOR THIS WORK?
```

Can you see what the program is doing? Well let's take a look at the changes we have made. `GOSUB 2000` reads into the array `question$` the questions to be asked by the computer. To keep the example simple I've only added three questions so that you can see what is happenning. `GOSUB 1900` is responsible for printing out one of those questions, and this time instead of using a random choice, I have kept track of which questions have been asked by the computer by using a counter. That is what `a%` is doing. There is an important reason for choosing `a%` as the variable. At the start of the program and after each interaction, the variables are all cleared from memory by using the statement

CLEAR. The only variable which is not forgotten is the resident integer variable a%. As we need to keep track of what has been previously used in the data statements, we can't afford to have that information wiped out. As a% is not affected by the use of CLEAR, we can therefore utilize it to run through the whole program. The reason a% is unaffected by CLEAR is because a running count is kept of the variable above the memory of the main program. This is the function of the PEEK and POKE statement in lines 1900 to 1920. I'll return to the question of using counters shortly but first let's carry on with understanding the changes.

GOSUB 660 has been changed to simply send the program back to line 75 where eventually it is routed to GOSUB 1900. The start of the program has been changed to provide a new start-up message and the remaining changes enable us to fit in the new routines.

This is how the new program works: if a keyword is found, then the program will behave just as it did in Sigmund, and will print out an appropriate reply. However, if no keyword is found in what the user types, then GOSUB 660 will direct the program to issue one of the questions instead of printing out a stock phrase. The next time this occurs Interviewer will have incremented the counter and a different question will be asked.

Figure 10.1 shows a flowchart for the new program. If it is not clear, compare Figure 10.1 with the flowchart of the Sigmund program.

To complete the job you now have to add to the DATA statements used by the subroutine GOSUB 2000, not forgetting to make the appropriate changes to the array and loop statements. It is then up to you to decide which of the original keywords are worth retaining, or change some of the answers associated with a keyword. It is possible to make Interviewer analyse your performance by randomly assigning marks to each question answered and then printing out a report at the end, based on the score attained. You would have to set a time limit within the program, the easiest way being to finish the interaction when all the set questions have been asked. The Sigmund program could be used as the basis of a number of different simulations and this is just to show you how easy it is to change the kind of simulation being written.

I said that a counter was used in the Interviewer program and it raises once more the question of improving upon the programs already shown. A problem that occurs in Sigmund, and therefore in Interviewer, is that as replies are chosen randomly it is possible to get the same reply being printed a few times in succession. If

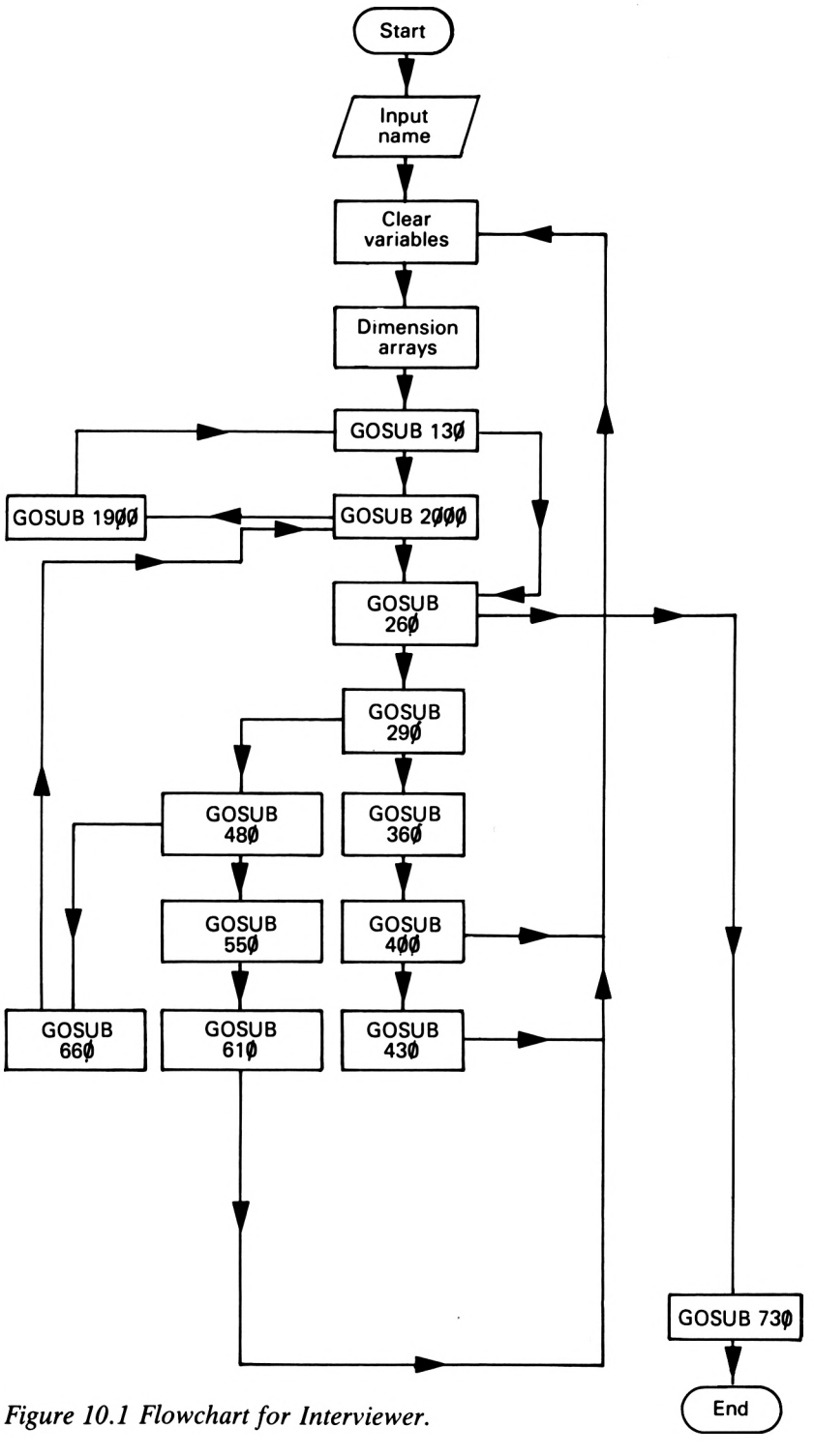

Figure 10.1 Flowchart for Interviewer.

you use a counter instead, it is then possible to make the program run longer, with less risk of phrases being repeated. By using a counter routine you can keep track of which replies have been used and the next time a reply is needed, from the same data statement, the subsequent reply in line can be implemented. The same goes for the replies made by the user. If the user types in the same phrase twice, it can be spotted and the user informed to type something else. But do remember that all variables will be wiped out by `CLEAR` so you may have to `POKE` a variable above `HIMEM`.

These programs can always be improved on and I now put the ball in your court. The methods for creating interactive programs have been covered and it is now a matter of trying to work out ways of incorporating what we have learnt in new and novel programs. To help you on your way, the next chapter looks at different ideas for interactive programs and you will see that there are many applications within which you can use the techniques covered.

11 Brainstorming

We have covered a lot of ground in this book and learnt many different techniques, but what can we do with our new-found knowledge? The answer is to have a brainstorming session. What's that? Brainstorming is having a good rack through the brains for ideas, and that's where all creative programs start. Let's take a look at some recent software innovations and see if we can use our techniques to develop them in our own way.

SIMULATION 1:
Using interactive techniques within adventure games

Now you might think that the only games played on home computers aim to protect mother earth from multicoloured space invaders, intent on conquering the bottom of your screen. But there is a great deal one can do with text-based games. A number of well-known board games are being implemented on home micros, and they can be as challenging, and as much fun as an arcade-style game.

A popular area of gaming is the adventure game. The user is thrown into a situation in which he has one objective, usually to survive the onslaughts of unfriendly creatures. Now, I'm not going to teach you how to write adventure programs, that is a book in itself, but we can certainly use the methods we have seen, and put them to good use in an adventure program.

For instance, in an adventure program, the computer relies on text based commands input by the user from the keyboard. This means building a dictionary into the program. A good adventure program will allow a command to be written in a number of ways. For example, if you tell the computer to go `NORTH`, this is often typed either in full or as `N`. By matching selectively for just a few letters of a word you may be able to increase the possible vocabulary. The text manipulation commands are particularly handy in this area.

SIMULATION 2: Education

A long-expressed fear from those unacquainted with computers is that machines will take over the classroom and threaten a teacher's job. This is not what educational programs try to do, but they can serve a useful purpose as an aid to teaching, either in the classroom or at home.

We can build into a program some of the qualities of a teacher and mimic these in the testing of a child's ability to perform a certain task. The obvious ideas that come to mind are testing a child's knowledge of multiplication tables, other mathematical tasks, learning the alphabet and so on. What is important in an educational program is to ensure that the program is interactive. Is it well error-trapped? What happens if the child presses the wrong key by mistake? Moreover, is the explanation to the child clear? And what about the screen display?

All these points we have covered and this is where you will start putting them into practice. Don't be afraid to experiment at first and in the case of children's programs, the best way to see how good your programming has been, is to let a few of your intended victims (children!) try it out. Watch carefully to observe how they get on with the software and whether there is a clean line of communication between them and the computer. After all you do want them to interact with the program.

SIMULATION 3: Business

Even at work programs could be made to be more interactive. Business software is some of the most unfriendly I have ever come across. This is one environment where easier communication is a necessity. Simulating a business in game form could be an exercise to try. Why not use the Sigmund program as a basis for a 'Board'-room game. The computer could be the Chairman of the Board and your job is to convince other members of the Board to accept your amendment to a proposal. Perhaps get the computer to play the role of several people? It just requires a little imagination and a host of programs could emerge.

Finally, interactive techniques can be employed in the running of questionnaires, and you could build tests around them, an obvious candidate being personality tests. Diagnostic faultfinding is a growing area of development and the next generation of computers, the much heralded fifth generation, will be computers that are knowledge information processors and expert systems.

These are systems where computers can make decisions on information they have acquired and they are machines which can

learn as they go along. We can already implement a number of these abilities from within BASIC, and as you become more acquainted with the language you may later on decide to branch out and delve into programming at a deeper level.

The overall message in this chapter is that interactive programming is a matter of man and machine in perfect harmony. Well, this might not be entirely feasible but if you can blend in your program with the people and the environment in which it is being used, then you will have gone a long way towards creating programs that interact with the user. And that is the purpose of interactive programming. Man and machine communicating freely between each other. As for harmony? That may prove to be another problem!

12 Artificially Intelligent?

Man has been seeking the answer to human thought processes for a very long time now, approaching the subject from different fields of academic study. However, it has been with the advent of widely available computers that man's study of the mind has escalated. The computer has provided the perfect tool with which to study the processes of thinking and to simulate human activity and thought. It is this latter subject on which the book has concentrated by writing the Sigmund program.

At the very start of this book I presented you with one or two problems to ponder on. Can machines think? What is artificial intelligence? Now that we have reached the end, you may have started to form some answers. I don't have any answers, just more questions. That's not a way of avoiding the issue, but when you start to think about machine intelligence there are a number of things to consider. Firstly, there is the problem of defining what we mean ourselves by intelligence. This particular debate rages on between psychologists and that argument could be a book in itself. For the purposes of this book, intelligence comprises the processes carried out by the human mind to cope with a variety of tasks: the ability to switch from one idea to another, and work at a number of different levels.

However, the one thing that does come out of artificial intelligence research is how little we understand about ourselves and what the nature of intelligence might be. Computers can be very good at performing specific tasks, even better than man, but when faced with situations that require knowledge gained in different areas, the computer has still to master this. So what did we achieve by creating Sigmund. Is it an artificially intelligent program? We are making the computer behave in ways that mimic intelligent human behaviour. But that cannot be the only criterion on which to judge a computer's 'intelligence'.

Over the years, researchers have suggested different tests that would determine whether or not a program or machine could be

deemed to be intelligent. The most attractive of these is to identify a unique feature of a human being, i.e. the ability to act in a creative and original manner. Now surely a computer cannot be creative and come up with original and novel ideas?

That's not the case. Programs have been written capable of generating proofs of geometry theorems that no human has ever thought of. Who can claim to have made the 'intelligent' discovery when such a feat is performed? Is it the programmer who has never consciously thought of the proof, or the program? A difficult problem

Whatever you decide to call the actions of a program, be it intelligence or otherwise, there is no doubt that we are rapidly approaching a time when computers will have minds of their own. What we can do on our home micro is merely simulate some properties of human behaviour, but that in itself can be very powerful and effective. The emphasis in this book has been on creating programs that interact with the user. That is very important.

As we go through another industrial revolution with computers becoming a feature of everyday life, it is necessary to make them as user-friendly as possible. That is the first step in interactive programming. The second is to write programs that can read a user's input and make a decision based on what it has been taught. Those first two are relatively easy and as we have seen can be achieved. The next, however, is much more difficult.

How is it possible to write a program which will behave as a human would in a given situation? This can be achieved, up to a point, depending on the nature of the behaviour being mimicked. You will notice I have used the word 'mimic' to describe a computer's responses. This is because that is all we are doing when we write a program on our home micro.

We have used the programming language BASIC in this book, but in terms of creating artificially intelligent programs it is not a good choice. LISP is a much better language, and others exist. However, it has taught us the basics of how computers understand instructions and armed with this knowledge you should be able to construct a wide range of programs.

That is where I finish. The principles learnt in this book are the stepping stones to larger and better programs. If you have followed everything, you will already be writing useful programs and with a bit of thought you could be writing programs that will not only amaze your friends but are truly interactive with the user. Perhaps the insides of your Amstrad are more than just a cluster of silicon chips. But whatever they are, you have found a good point to start on the Road to Artificial Intelligence.

Appendix A: A crash course in BASIC

This is what you could call a crash course in basics, or basic BASIC. If you've never typed a program in before, read this appendix first. It will give you a grounding from which you will be able to continue to the techniques taught in this book. So assuming that you know nothing—read on!

Turn on your Amstrad and you will see a message at the top of the screen and a box cursor. You've probably tried typing in a friendly message like HELLO but all you have got in reply is a message telling you that you've made a mistake. Don't be put off!

Computers are stupid machines. Before they can produce all the wonderous effects you have seen on other machines they have to be given a set of instructions telling them what is required. This set of instructions is known as a program and is a logical sequence of commands which the computer works its way through. Try typing the following:

```
PRINT "HELLO"
```

When you have typed that line, finish off by pressing the ENTER key. See what happens? The word HELLO has appeared on the next line. Now type the same line except this time change what appears between the quotes. For example, you could type:

```
PRINT "MY NAME IS JEREMY"
```

Yet again press ENTER when you have typed it in. This time what has been printed to the screen has changed to what you placed between the quotation marks (by the way, use the double quotation mark above the number 2). Two important things can be learnt from this. Firstly, at the end of any line, unless I indicate otherwise, always press the ENTER key. This tells the computer to carry out your instruction. Until the ENTER key is pressed no command will be carried out. Secondly, what we have done is to

give the computer a legal instruction, something which it recognizes. In this case, that something is the command PRINT. By now you probably have guessed that the PRINT command does what it says. It prints to the screen anything that you place between the quotation marks.

Now this is all well and good but you can't type in commands like that all the time. What we have done so far is to issue a direct command. What we now need to do is to store our commands, in order that they be carried out in the sequence we wish. We might want, for example, to write the name and address of a friend, on the screen and on a number of lines. To do this try the following:

Program A1

```
10 PRINT "CHRIS HENDON"
20 PRINT "23 THE HIGH ST"
30 PRINT "WELLINGTON NORFOLK"
```

Remember to press ENTER at the end of each line. When finished type RUN (ENTER) and the name and address is printed on three successive lines. What you have just done is to write a program! Not the most exciting program in the world but it is a program. Each line represents a command to the computer which the machine obeys. I have used line numbering of 10 to 30. The actual numbers are not important, they could just as easily be lines 1, 2, 3 or 234, 256, 678. The important thing is that they represent a guideline to the computer as to what order the commands should be executed in. The computer reads line 10 first, carries out whatever is written there and then proceeds to the next line, line 20. It is a good convention to build up a program in steps of 10 as there are always times when you will need to insert an extra line and this would be difficult if you have left no space by writing a program in steps of 1. Note that you can type the command keywords in lower case characters as they will be converted to upper case when listed.

To look at our program again type LIST (ENTER) and this command will print out a listing of your program in the correct sequence. This command can only be used to list a program and cannot be used as part of a program.

We have seen how the PRINT statement can place words on the screen but it can also carry out mathematical instructions. Type in the following:

```
PRINT 25 * 2
```

This statement causes the computer to type 50. This is different to what we have seen previously. Note' this time there are no quotation marks. In our previous example the quotation marks were placed to inform the machine that we were dealing with a non-numerical event or something which didn't require any mathematics to be involved. In the present example, I have asked the computer to tell me what 25 multiplied by 2 is, and it has replied with the correct answer, which is 50. Therefore, the computer is capable of being a calculator as well.

The next program asks the user to enter a number. This is where we come to our second keyword statement. We need to be able to enter information or input into the computer. To do this we use the `INPUT` statement which tells the computer we are requesting information and the computer will wait until an answer is given. For example, type this:

```
INPUT number
```

In response to the question mark enter a number. If you enter a number the prompt will return. If you now type:

```
PRINT number
```

the number you entered will appear. What you have done is to allow the value entered to be given to a numeric variable, the variable name being 'number'. It could just as easily have been called 'a' or 'fred'. You can think of a variable as a box. When you used the `INPUT` statement the number you entered was placed in the box called 'number' and at a later date when you asked for the contents of the box by typing `PRINT number` the number you entered was shown. Use the example above again but this time change the variable name, i.e. 'number' becomes 'a':

Program A2

```
10 INPUT a
20 PRINT 6 * a
```

Run the program. Line 10 waits for the user to input a number and that number is then multiplied by 6 in line 20, and the `PRINT` statement prints that result to the screen.

Now what happens if we enter a non-numeric value, i.e. the letter A? The Amstrad rejects this imput because it is expecting a

numeric value and if anything else is typed in, the computer doesn't know what to do with it. This example leads me on to the two different kinds of variable that exist.

The first we have encountered, that being a numeric variable. The second kind of variable that we can use is called a string variable. When using a string variable any keyboard character is accepted and stored. The difference between this and a numeric variable is that mathematics cannot be carried out on a string variable. To distinguish between these two kinds, a rule is followed.

We can call a numeric variable what we like. We called our numeric variable in Program A2 'a' but could quite as easily have called it 'number' or 'amstrad'. To tell the computer we are using a string variable we add a $ (dollar) sign on to the end of the variable name. For instance, if we want to ask the user for his/her name and then print a personal greeting to that person we can write a program like this:

Program A3

```
10 INPUT name$
20 PRINT "Pleased to meet you "; name$
```

The semi-colon in line 20 tells the computer to place the variable name$ next to the last thing printed, in this case a space, because we want to leave a space between 'you' and the name entered.

Therefore, a string variable allows any alphanumeric character, i.e. any keyboard character but it cannot carry out mathematics on a number entered. If you are still unsure about the difference between a numeric and string variable go back and try to write a program to carry out multiplication except this time use a string variable (one with a $ sign on the end) to work out the result.

Finally, let us learn a few more BASIC keywords. The first keyword is `LET`. You will often see in programs lines such as:

```
tag = 9
```

or,

```
y = y + 8
```

In the first example we have told the computer that the numeric variable 'tag' equals 9 or to be more exact, we have said `LET tag = 9`. The keyword `LET` is optional and we will not make use of it but the important thing to remember is that when

we tell the computer that a variable is something or that, as in the second example, the numeric variable y equals the value of y + 8, then we are in fact saying, LET this variable equal...

So, for instance, if we want to print our name out ten times we could type ten lines each with the same PRINT statement:

```
10 PRINT "Jeremy"
20 PRINT "Jeremy"
30 PRINT "Jeremy"
```

etc.

But that is a long way of going about it. So we can use our ability to increment the value of a variable by typing the following:

Program A4

```
10 x = 0
20 PRINT "Jeremy"
30 x = x + 1
40 IF x < 10 GOTO 20
50 END
```

Line 10 sets the value of x to 0. Line 20 is where we print our word. At line 30 the value of x is increased by 1 (LET x equal the value of x, which at this point is 0, and add 1 to it, therefore making x equal to 1). Now we come to two new keywords, IF and GOTO. What we have said in line 40 is that IF x is less than 10 GOTO line 20, where the PRINT phrase is repeated. This continues until x is greater than 10. If x is greater than 10 the program is finished, thus the END statement at line 50.

We have yet to cover subroutines. A subroutine is a group of instructions put together in one section enabling the user to call it up by its starting line number. You can think of a subroutine as a smaller program within the main program. If you see a line that has in it GOSUB line number, that is a subroutine. Normally the RETURN statement marks the end of the definition of the subroutine. Therefore, you could have a program that is one line long, the line calling a subroutine:

```
10 GOSUB 30
20 END
```

```
30 PRINT"HI THERE"
50 RETURN
```

Subroutines are very useful and make programs easier to read and follow.

That was, as I said at the beginning, a lightning introduction to BASIC but if you have understood everything here you should cope with the rest of the book.

Appendix B: Amstrad BASIC Keyword Summary

This appendix contains a summary of the Amstrad BASIC commands covered in this book. It isn't a replacement for the user guide nor a detailed description, but sufficient to jog your memory on any of the commands we have used. In addition, there are a few extra commands not detailed in this book which may be of use. The commands for graphics, sound and many of the numeric functions are also missing. For further information on any of these, refer to the user guide.

`AND`	Logical 'and' operator.
`ASC`	Converts a character into an `ASCII` code.
`AUTO`	Provides an automatic line numbering facility for typing in listings at regular intervals.
`BORDER`	Changes the colour of the border of the screen.
`CHAIN`	Command to load and run a program, i.e. `CHAIN"filename"`.
`CHR$`	Converts an `ASCII` code into a character.
`CLEAR`	Clears the memory of all variables previously used.
`CLS`	Clears the screen.
`DATA`	The store of information which is taken by the `READ` statement.
`DELETE`	Used to delete lines from a program, i.e. `DELETE 10,60`.

DIM	Dimensions the size of an array.
ELSE	Used in conjunction with IF-THEN to branch to another action, i.e. IF counter>20 THEN GOTO 50 ELSE 100.
END	Tells the computer to terminate running a program.
ERL	Gives the number of the line containing an error.
ERR	Returns the number of the last error.
FOR	The start element of the FOR-NEXT loop.
GOSUB	Sends program to a subroutine at a specified line number.
GOTO	Sends the computer to a specified line in the program.
IF	Start of the IF-THEN statement.
INK	Changes ink to a specified colour.
INKEY$	Takes the input of a key from the keyboard.
INPUT	Issues a request for either numbers or strings of characters to be entered from the keyboard.
INT	Converts a number to the nearest smaller integer.
INSTR	Searches for the occurrence of a string within another string.
KEY	Define new function key.
KEY DEF	Define the value of a key when pressed.
LEFT$	String manipulation command for taking a number of given characters from the start of a string.
LEN	Returns a number giving the length of a string.
LINE INPUT	As for INPUT but ensures that everything typed in is held in the variable.

LIST	Lists to the screen all or part of a program.
LOAD	Loads a program from cassette or disc into the computer's memory.
LOCATE	Moves the cursor to a specified position on screen, i.e. `LOCATE 1, 5`
LOWER$	Converts upper case characters to lower case.
MID$	String manipulation command for taking a number of given characters from a specified position in a string.
MODE	Sets the screen display mode.
NEW	Clears the memory of the computer.
NEXT	Specifies the end of a `FOR-NEXT` loop.

ON	Enables redirection of a program by altering the order of execution, i.e:
	`ON a GOTO 25, 45`
	`ON ERROR GOTO 10`

OR	Logical 'or' operator.
PAPER	Sets the colour of the paper (background).
PEN	Sets the colour for the characters (foreground).
PRINT	Prints given items on the screen.
READ	Reads the information contained in the `DATA` statements.
RENUM	Renumbers the lines of a program listing, i.e. `RENUM 100`.
RESTORE	Sets pointer to read data from a specified position.
RETURN	Marks the end of a subroutine.

RIGHT$	String manipulation command that takes a number of characters from the end of a string, working from right to left.
RND	Chooses a random number.
RUN	Runs the program.
SAVE	Saves a program to cassette or disc.
SPC	Places any number of specified spaces on the screen.
STEP	Specifies a step within the FOR-NEXT statement.
STOP	Stops a program and displays the line number.
STR$	Converts a number into a character string, i.e. 1 becomes '1'.
TAB	Used with PRINT to move the screen cursor to a specified position.
THEN	Used in conjunction with the IF statement.
TO	Used in conjunction with the FOR-NEXT loop to specify a numeric range.
UPPER$	Converts lower case characters to upper case.
VAL	Converts a number in a character string into a numeric variable, i.e. '8' becomes 8.
WHILE	Causes a loop until a certain condition is met.
WEND	Terminates the WHILE loop.
ZONE	Sets the width of the print zone.

Other titles of interest

Bells and Whistles on the Amstrad CPC 464 **£4.95**
Jeremy Vine
Discover the sound effects which can be created on the Amstrad CPC 464.

Gateway to Computing with the Amstrad CPC 464 (each) **£4.95**
Ian Stewart
Two books covering the fundamentals of computing for young people.

The Complete Introduction to the Amstrad CPC 464 **TBA**
Eric Deeson

Computers in a Nutshell **£4.95**
Ian Stewart
The layman's introduction to computing.

Brainteasers for BASIC Computers **£4.95**
Gordon Lee
A book I would warmly recommend'—*Computer & Video Games*

Microchip Mathematics: Number Theory for Computor Users **£12.95**
Keith Devlin

Programming for REAL Beginners: Stages 1 & 2 (each) **£3.95**
Philip Crookall

ORDER FORM

I should like to order the following Shiva titles:

Qty	Title	ISBN	Price
____	BELLS AND WHISTLES ON THE AMSTRAD CPC 464	1 85014 063 4	£4.95
	GATEWAY TO COMPUTING WITH THE AMSTRAD CPC 464		
____	BOOK ONE	1 85014 016 2	£4.95
____	BOOK TWO	1 85014 023 5	£4.95
____	THE COMPLETE INTRODUCTION TO THE AMSTRAD CPC 464	1 85014 002 2	TBA
____	COMPUTERS IN A NUTSHELL	1 85014 018 9	£4.95
____	BRAINTEASERS FOR BASIC COMPUTERS	0 906812 36 4	£4.95
____	MICROCHIP MATHEMATICS	1 85014 047 2	£12.95
____	PROGRAMMING FOR REAL BEGINNERS: STAGE 1	0 906812 37 2	£3.95
____	PROGRAMMING FOR REAL BEGINNERS: STAGE 2	0 906812 59 3	£3.95
____	.....	.....	.....
____	.....	.....	.....
____	.....	.....	.....

Please send me a full catalogue of computer books and software: ☐

Name .....

Address .....

.....

.....

This form should be taken to your local bookshop or computer store. In case of difficulty, write to Shiva Publishing Ltd, Freepost, 64 Welsh Row, Nantwich, Cheshire CW5 5BR, enclosing a cheque for £ .....

For payment by credit card: Access/Barclaycard/Visa/American Express

Card No ..... Signature .....

Artificial Intelligence is the latest news on the chip. With your efforts and this book you can learn to communicate with an Amstrad CPC 464.

You will go on a journey to discover the techniques needed to:

- build data banks
- educate your micro
- disentangle the strings
- learn pattern-match techniques

and allow free communication between you and your micro.

Passing through the simulation belt, you will encounter two programs to turn your computer into an intelligent companion. *Sigmund* and *Interviewer* contain structured data which will enable you to converse with your Amstrad.

Very little knowledge of BASIC is required to grasp the information given along the way. Everything is clearly signposted in this comprehensive route guide. And for those with no experience of programming at all, there is an anti-crash course in BASIC.

The end of the book is by no means the end of the journey but you will find yourself on the right road to Artificial Intelligence.

UK price £5.95 net

GB £ NET +005·95

ISBN 1-85014-064-2

00595

9 781850 140641

Shiva Publishing Limited

ON THE ROAD TO ARTIFICIAL INTELLIGENCE: AMSTRAD Vine SHIVA

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/