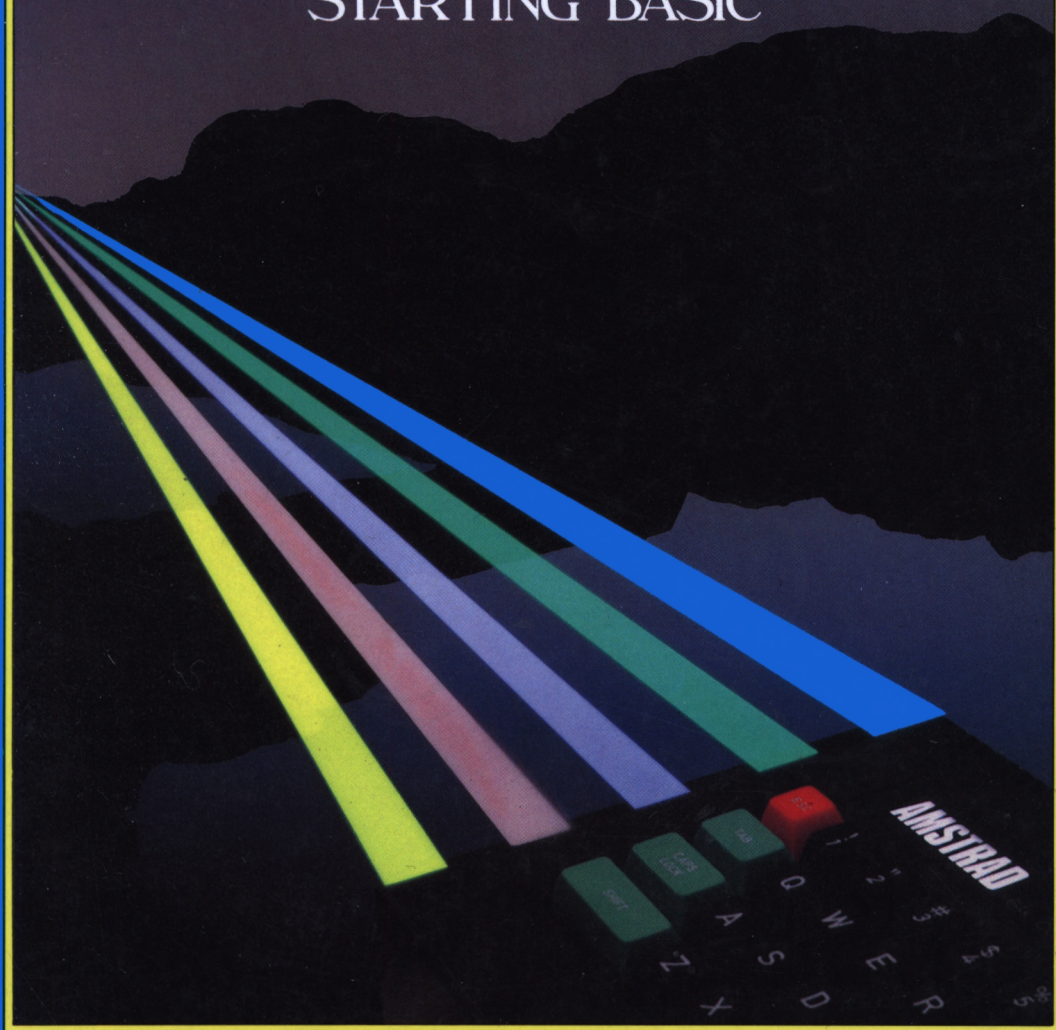




AMSTRAD BOOK 1

STARTING BASIC



Sean Gray & Eddy Maddix

AMSTRAD BOOK 1 STARTING BASIC

**STARTING
BASIC
FOR THE
AMSTRAD
BOOK 1**

**BY
SEAN GRAY & EDDY MADDIX**

Glentop Publishers Ltd.

FEBRUARY 1985

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publishers for any errors or omissions contained herein.

COPYRIGHT © Glentop Publishers Ltd 1985
World rights reserved.

No part of this publication may be copied, transmitted or stored in a retrieval system or reproduced in any way including but not limited to photography, photocopy, magnetic or other recording means, without the prior permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

Graphics: Samantha Borland
Wordprocessing: Jane Grant

ISBN 0 907792 39 1

Published by: Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 1ED
Tel: 01-441-4130

* Dr Watson is a Trademark of Glentop Publishers Ltd.

CONTENTS

Introduction

- Chapter 1 In the Beginning * The Keyboard * SHIFT * spaces * Mistakes * Getting Started in BASIC * PRINT * LET * Variables * Your First Program * INPUT * LIST * Separators * Prompts * EDIT * NEW * LOCATE * CLS
- Chapter 2 Guess The Number * RND * GOTO * IF...THEN * STOP * Flow Charts * FOR...NEXT * Storing A Program * LOAD * SAVE * IF...THEN * Comparing Numbers * Simple Logic * OR * AND * Mathematical Precedence
- Chapter 3 Graphics * MODE * BORDER * INK * Pen and Paper * Pixels and Points * Plotting a Point * Drawing Lines * Etcha Sketcha * INKEY\$ * Drawing Boxes * Drawing Circles
- Chapter 4 Structure * READ And DATA * RESTORE * LEFT\$, RIGHT\$ and MID\$ * LEN * GOSUB and RETURN * Arrays * True * False * Upper\$
- Chapter 5 More Structure * Skeleton Programming * Program Control Module * INSTR * Program Development * Flags
- Chapter 6 Solutions to Exercises
- Appendix One Binary and Hexadecimal
- Appendix Two Using The Amstrad Data recorder

I N T R O D U C T I O N

This is the first of a two-book series for the Amstrad. The object of this book is to teach the reader the fundamentals of Amstrad Basic i.e. the most common commands and how to use them. Also, the aim is to demonstrate various programming techniques including 'structured programming'. In order to achieve these aims each new command encountered is explained in careful steps and in most cases this includes a demonstration program showing the reader not only what the command does but also how it is used in a program.

The reader is first introduced to the simplest of Basic's commands in chapter one and by the time the reader has finished reading the book he or she will be fully capable of programming in Amstrad Basic.

Included in this book are three games. The first is a simple number guessing game, the second an anagram game and finally a hangman game using computer graphics. The object of each of these games is to illustrate specific points. For example the number game takes the reader through the world of random numbers and simple logic. By the end of chapter five the reader will have learnt different ways of storing information, how to draw pictures and the advantages of modular, structured programming.

It is stressed that as an introduction to the Amstrad's Basic commands this book stands on its own. Those who wish to delve further into the Amstrad should consider reading book two as well. We hope you will gain as much in the reading of this book as we did in the writing.

S. Gray

E. Maddix

January 1985

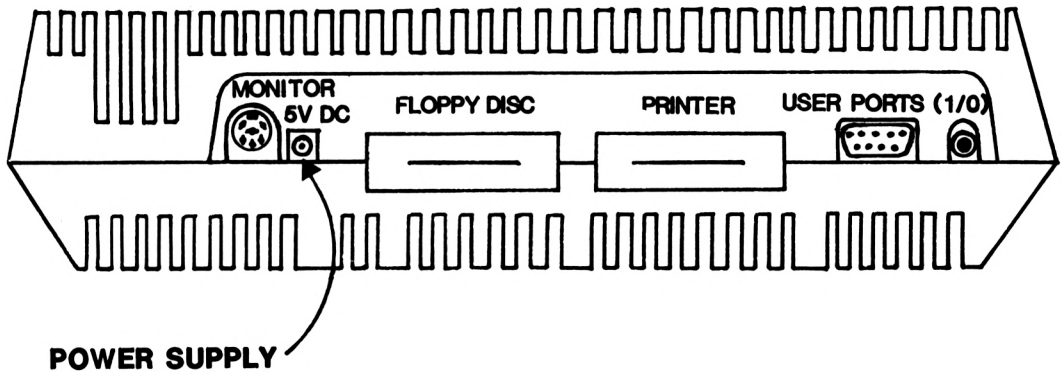
CHAPTER 1

PART ONE

Are you sitting comfortably? Then we will begin. This part of the chapter will deal with the setting up of your Amstrad. Unlike most computers the Amstrad comes with its own specially designed monitor. Because the monitor has been built to work with the Amstrad computer the picture (and colours) are much clearer than on an ordinary television.

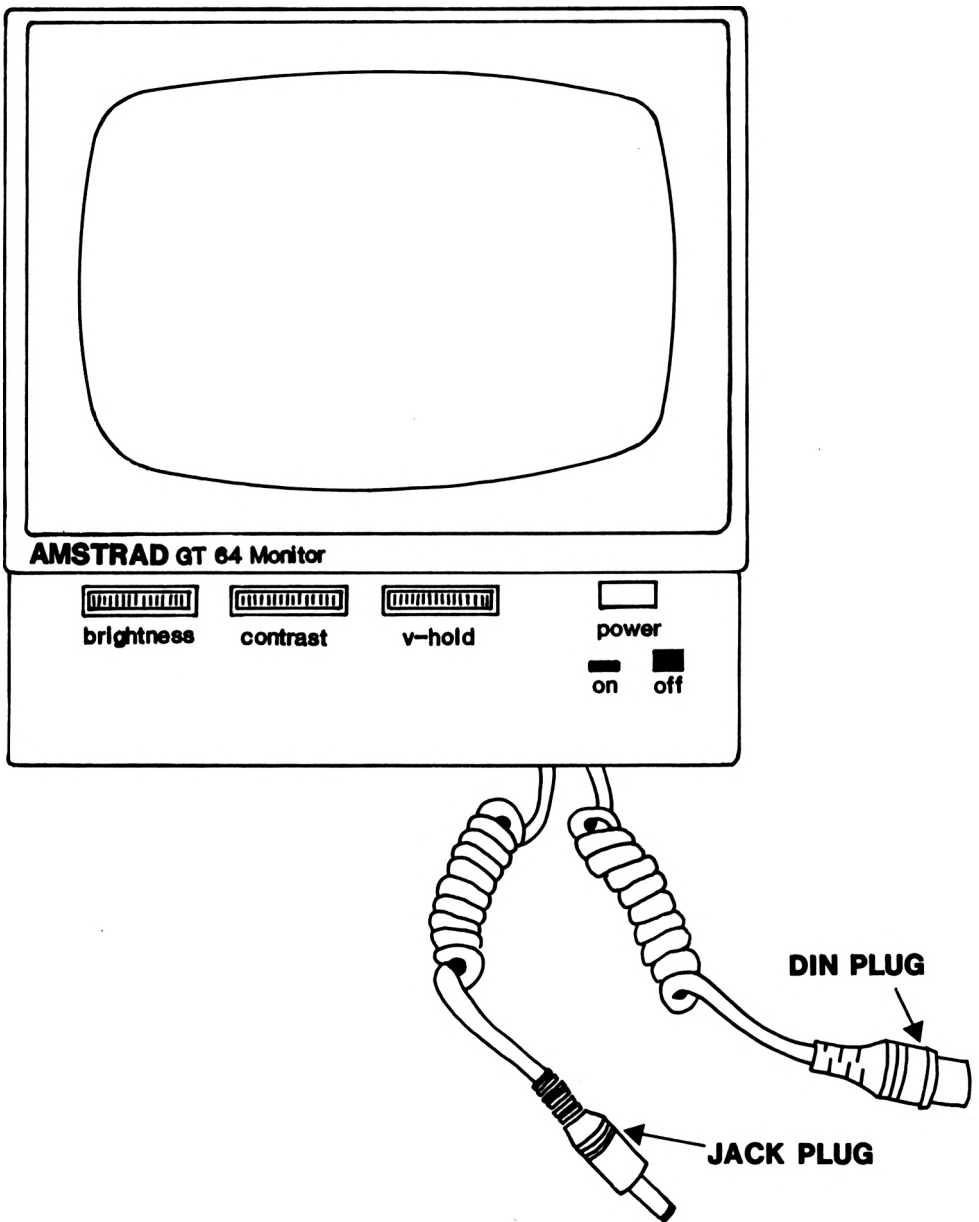
Connecting The Amstrad

The Amstrad monitor also houses the computer's power supply. This cuts down on the amount of wires one has lying around. At the front of the monitor there are two wires, one ending in a din plug, the larger of the two, the other ending in a jack-plug.



The Amstrad Keyboard

FIGURE 1.1(a)

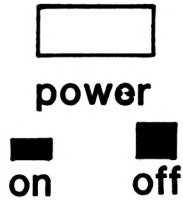


The Amstrad Monitor

FIGURE 1.1(b)

The din plug plugs into the socket in the back of the Amstrad labelled monitor (see Figure 1.1(a)). Next to the monitor socket there is another one labelled '5VDC'. This is where the jack plug is inserted. When both of these are connected properly then you are ready to begin.

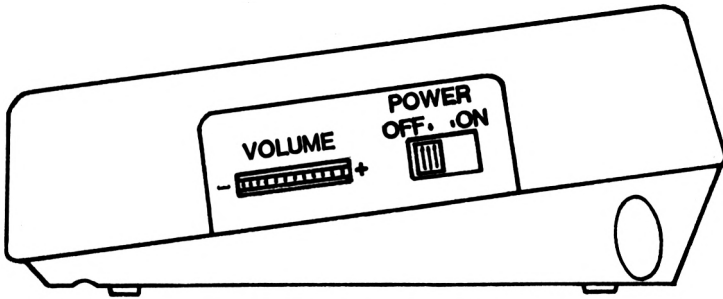
At the front of the monitor there is a switch marked POWER (see Figure 1.2). Pressing this button switches the monitor on. It also switches the Amstrad's power supply on but not the computer.



The Monitor Power Button

FIGURE 1.2

The Amstrad computer has its own ON/OFF switch and this is on the right side of the computer.



The Keyboard Power Switch

FIGURE 1.3

If all the wires are plugged in correctly and the monitor is switched on you should flick the keyboard switch to 'ON'. A red light will 'light' up on the keyboard. Your Amstrad personal computer is now all set up and raring to go.

Once all is switched on you will see the following message:

```
Amstrad 64K Microcomputer (v1)
© 1984 Amstrad Consumer Electronics plc
and Locomotive Software Ltd.
```

```
BASIC 1.0
```

```
READY
```



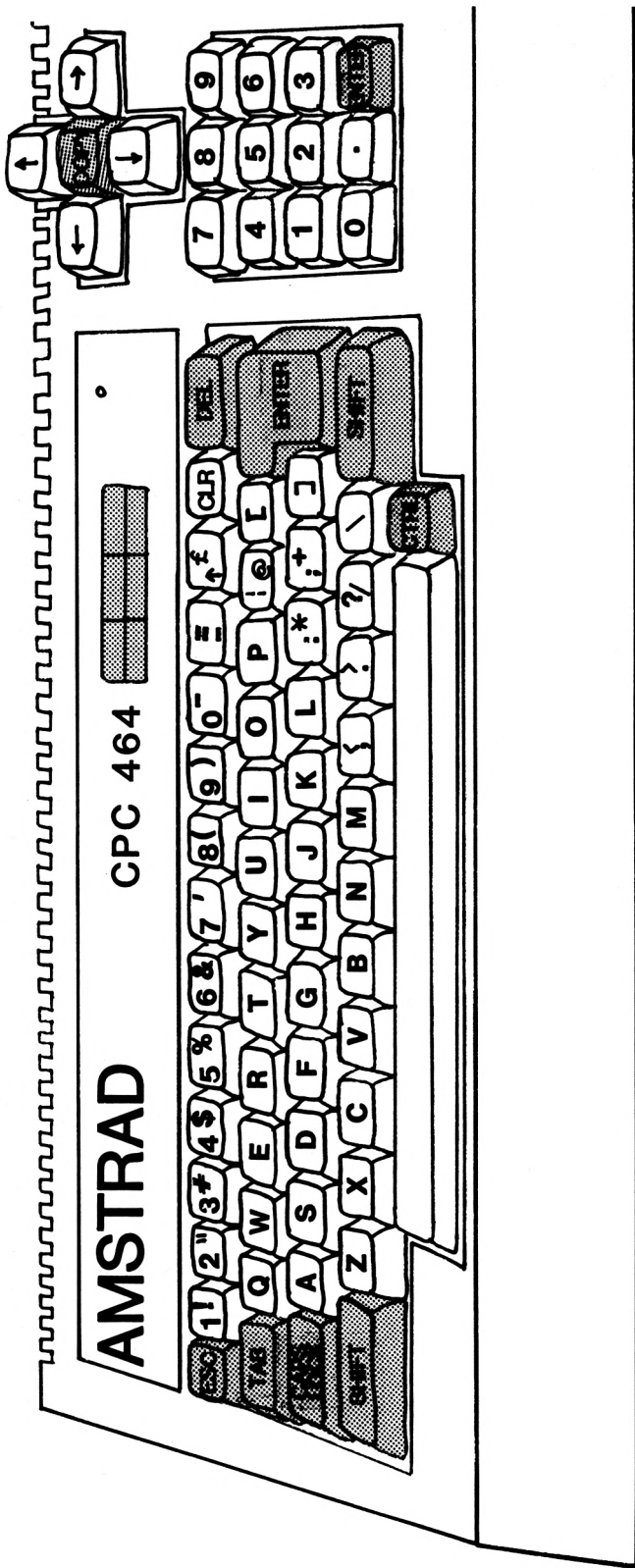
The 'Start' Display

FIGURE 1.4

The start display tells us a number of things. The first line tells us which computer we have got, in this case an Amstrad microcomputer with 64K. Your computer has 64K of memory, which is quite a lot. The second line tells us to whom the Amstrad computer design and software copyright belongs, Amstrad Consumer Electronics and Locomotive software.

Next we see the message 'BASIC 1.0'. This tells us that the language the Amstrad understands is BASIC version 1.0. There are many versions of BASIC, almost one for each new computer. The word BASIC is an acronym and it stands for Beginners All Purpose Symbolic Instruction Code.

The Ready message means the computer is ready to begin. Underneath the 'R' of 'Ready' there is a light coloured block, this is called the '**CURSOR**'. The cursor is the computer's way of telling us where the next character will appear. Press one of the letter keys on the keyboard and it will appear on the screen where the cursor was. The cursor will move one space to the right. If you press another letter then the same thing will happen. If you continue to keep pressing a key the cursor will move across until it reaches the righthand side of the screen. It will then appear on the left-hand side on the next line down the screen.



The Amstrad Keyboard

FIGURE 1.5

The Keyboard

The Amstrad allows you to use the keyboard as you would the keys of a typewriter. If you press any of the letter keys you will see that letter appear in lower case. To get upper case letters (capitals) you simply press the key marked Caps Lock. This is a big green key on the left hand side of the keyboard. Upon pressing the **Caps Lock** key you are in capital mode and every letter pressed henceforth will be upper case.

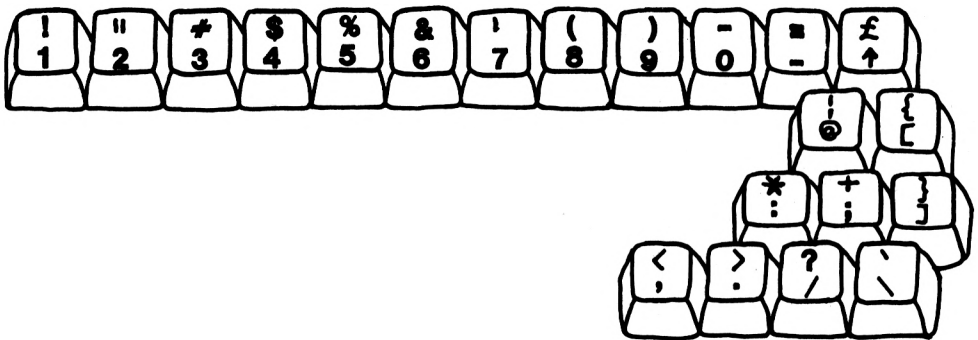
To return to lower case you simply press the Caps Lock key once again. You are now back in lower case mode. The Caps Lock key is known as a toggle switch. If you are in lower case and you press it you go into upper case mode, but if you were already in upper case pressing the caps lock switches you to lower case.

SHIFT

There are two SHIFT keys on the Amstrad. They are both green and there is one on either side of the keyboard. It should be understood that it does not matter which SHIFT key you use. They both do exactly the same thing. Having got that sorted out it is time to find out exactly what they do.

You may have noticed that some of the keys on the Amstrad keyboard have more than one symbol, for example the '4' key also has a '\$' dollar symbol on it. To get the 4 character you simply press the key. To get the \$ character you need to hold down SHIFT and then press the key. Holding down either of the SHIFT keys allows you to get the extra characters on some of the keys. The 'extra' or 'shifted' character is the top one shown on the key.

The following keys have extra SHIFT values.



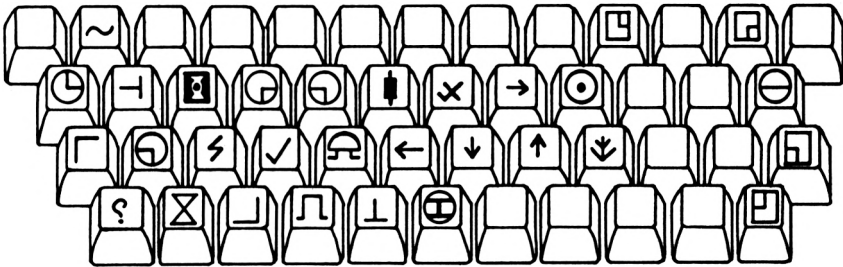
Shift Characters

FIGURE 1.6

All these 'shifted' characters are obtained by holding down either of the SHIFT keys and pressing the appropriate key. The letters don't have extra shift characters on them but when CAPS LOCK is off, holding down SHIFT and pressing a letter key will give the upper case letter.

CTRL

As well as having extra shift values some keys have special control values. These values are not displayed on the keys so you should consult Figure 1.7. To get these special characters you should hold down CTRL and press the appropriate key at the same time.



The Special CTRL Keys

FIGURE 1.7

There is one CTRL key not shown in Figure 1.7 and that is CTRL and P. Holding down CTRL and pressing the 'P' key does not display a character on the screen but makes a 'beep' noise. Very nice. If you cannot hear it there is a volume wheel next to the keyboard on/off switch. Turn this up and press CTRL and P again. This time (if you turned the volume wheel the right way) you will hear the short noise, this is the computer's equivalent of the bell on an ordinary typewriter.

Auto Repeat

Most of the keys on the main part of the keyboard have 'auto repeat'. This means if you hold the key down for more than half a second the character you are pressing will start to appear over and over again. To demonstrate this hold down the 'H' key. Keep the 'H' key pressed down until eventually after six and a half lines of 'H's' the Amstrad will emit a loud beep and cursor will stay in the same place. This noise is to tell you that it has gone as far as possible and it will continue beeping until you take your finger off the key.

DEL

Having reached a situation where we cannot press any of the character keys without getting a beep its time to find out about the DEL key. The DEL key is green and situated at the top left of your keyboard. When this is pressed the character directly behind the cursor is DELETED. By pressing the DEL key we can delete all the 'H's typed in. DEL also has auto repeat so just hold the key down.

When the last H is deleted you will hear the beeping again. This is the computer telling you that you cannot delete past the start of the line. When you take your finger off DEL the beeping will stop.

Spaces

Along the bottom of the keyboard is a long grey key. When you press this key the cursor will move to the right but nothing will appear on the screen. With this key, known as the 'space-bar', you will get a space character. Because you can not see this character throughout this book it will be represented by a small delta symbol 'Δ'. This is used to denote that a space exists in situations where a space is very important, so when you see that symbol remember to press the space-bar.

Alternatively, if a lot of spaces are required then you will see something like the following: <12 spaces>. That means you should press the space bar twelve times.

Getting Started In BASIC

Now that you know what most of the keys do it is about time to find out what your brand new Amstrad computer can do. This is where the learning really begins.

Whilst you were pressing the keys to see what they do, you might at some stage have seen the following message displayed on the screen:

Syntax error

This is a very important message from your computer to you. Your Amstrad is saying that it doesn't understand what you have typed in. A 'Syntax error' means the computer does not understand the way you are trying to tell it something.

The Amstrad, like most home computers, understands the computer language called BASIC. This is a specially developed language to allow easy dialogue between computers and computer users. To illustrate the difference between BASIC and English, in your best hand writing, type in the following:

SAY HELLO

Having typed that in the cursor should be positioned after the 'O' in HELLO. What to do now?

ENTER

The ENTER key is arguably the most important key on the Amstrad keyboard. What the ENTER key does is tell the computer that you have finished typing in your instruction and wish the computer to act upon it. When you have finished typing in your instruction(s) you must always remember to press the ENTER key.

In the example above you have just finished typing in an instruction so press the ENTER key. You will get the following message:

Syntax error

Although we have given the computer a clear instruction, 'say hello' it is not an instruction that the computer understands. To tell the computer to say 'hello' you must use the BASIC command PRINT.

PRINT

PRINT is the command to tell the computer that you want it to say something to you. To tell your Amstrad to say hello to you, in a way it can understand, you must type in the following command:

PRINT "HELLO"

Now if you press ENTER the computer will display:

HELLO

Bingo! You have made the computer understand you. The PRINT command tells the computer to display on the screen whatever it sees inside the quotation marks. In this example it sees the word 'HELLO'. This is promptly displayed on the screen. Underneath you will see the message:

Ready

This is the Ready 'prompt', it is the computer telling you that it has understood you and has obeyed your instruction with no problems. The cursor is patiently waiting for you to type something in.

The double quotation marks on either side of HELLO are very important as these tell the computer to PRINT exactly what is enclosed between them. Whatever is between the quotes will be printed even if it is rude, splet wrong or logically incorrect. For example:

```
PRINT "Amstrad is rubbish"
```

This is incorrect but you have told your Amstrad to print it and so it will.

STRINGS

The characters that are between the two quotes are called 'strings'. To your Amstrad anything enclosed in quotes is regarded as just a load of characters 'strung' together.

The print command can also be used to display numbers. To prove this point type in the following.

```
PRINT 8
```

Upon pressing the ENTER key you will see 8 displayed on the screen. The difference between printing numbers and strings is that numbers do not have quotation marks. If the eight was enclosed in quotes, e.g.

```
PRINT "8"
```

Then the 8 would be regarded as a string, although the actual display would be the same as printing the number 8. Notice that in the case of the number 8, the computer has put a space before it, but it hasn't done this with the string "8".

The PRINT command can do sums and display the answer. If you type in PRINT and then a calculation, upon pressing ENTER you will see the answer ('/' is divide, '*' is multiply) e.g.

```
PRINT 47/5
9.4
Ready
```

Using PRINT in this way makes it easy to do quick calculations.

LET

So far all that we have done is to tell the computer to display things on the screen. The next step is to tell the computer to remember something. This is done by using the LET command.

The LET command tells the computer to remember what we tell it to. When using this command we need to tell the computer two things. The first thing is a name to remember by and the second thing is what we actually want remembered.

Variables

When asking our Amstrad to remember something we have to specify a name to remember it by. This name is called a 'variable'. It is called a variable because the actual value remembered can change, i.e. it varies.

There are rules controlling the naming of variables but thankfully not many.

The first rule is that variables must begin with a letter e.g. 'A7'.

The second rule is that your variable name must not be more than forty characters in length.

The third rule is that the variable name should not contain a punctuation mark.

The fourth and final rule is that variables used to store strings must have a dollar sign (\$) as the last character of the name, variables used to store numeric data must not.

Variables that contain strings are called string variables. The computer can tell the difference between string and numeric (those used to store numbers) variables because string variables end with a '\$' whereas numeric variables don't.

So, to tell the computer to remember your name type in the following and then press ENTER.

```
LET NAME$="ALBERT"
```

(If your name is not ALBERT you might like to type in your own.)

Once you have pressed ENTER you will see the Ready prompt and the cursor appear. Not much seems to have happened, but deep inside your Amstrad an area of memory has been given the name 'NAME\$' and the string 'ALBERT' was put into it.

A complete translation of the LET command reads like this:

- i) Your Amstrad reads 'LET NAME\$' and translates this into 'take an area of memory and call it NAME\$'. See Figure 1.8



An Area of Memory Called NAME\$

FIGURE 1.8

- ii) Next it reads the '"ALBERT"' part and translates this into 'store the string ALBERT in the string variable NAME\$'. See Figure 1.9.



Storing ALBERT in NAME\$

FIGURE 1.9

The Amstrad also checks that the variable has a dollar sign at the end of the name AND that the expression (what it is to become) is enclosed in quotes. If the variable is a string variable but the expression is not in quotes then you will see the following message:

Type mismatch

This message will also be given if the variable is numeric but the expression is enclosed in quotes. Type mismatch is the computer's way of telling us that an attempt was made to put a string into a numeric location or numeric data into a string location.

Having read the last page you have probably forgotten what you put (assigned) in NAME\$. Just to refresh your memory type in the following:

```
PRINT NAME$
```

The computer interprets this as 'find the memory location called NAME\$ and PRINT it out, i.e. display whatever is in that location onto the screen'.

```
LET works in the same way with numbers.
```

```
LET AGE=2
```

Here the value of two is placed in a numeric variable called AGE.

The LET command can also be used to perform sums.

```
LET AGE=AGE+1
```

This tells the computer to add one to the value stored in variable AGE and to store the answer back into the variable AGE. This will replace what was previously in there. To test this

```
PRINT AGE
```

The answer you should get is 3.

Your First Program: Line numbers and RUN

So far, all the entries made have been carried out (or 'executed') immediately after the ENTER key was pressed. These are known as DIRECT ENTRY or IMMEDIATE mode COMMANDS. Once these have been executed they cannot be automatically re-executed, they're gone forever!

However, when programs are used, they are stored in memory and re-activated when required. What differentiates an immediate command from one in a program is that the program has line numbers. Whenever the ENTER key is pressed, the Amstrad looks at the entry made and, if it sees a number at the beginning of the line, stores the entry in memory as a line of program. Thus, if the earlier immediate entry commands are replaced by Program 1.1 below, they constitute an actual program, short as it is. Each line of a Basic program is referred to as a STATEMENT. The difference between the two is that commands are direct entry whereas statements form part of a program. When you typed 'PRINT NAME\$' in immediate mode it is a 'command'. When you type in the line 30 'PRINT NAME\$' that is known as a 'statement'.

Just what the value of a line number is doesn't really matter too much as long as it is a whole number with a positive value between 1 and 65536. What does matter is the order of the line numbers, as the Amstrad will run the program starting at the lowest line number and work through the increasing line numbers unless told to do otherwise. Type this program in (remember to press ENTER at the end of each new line!):

PROGRAM 1.1

```
20 LET NAME$="ALBERT"  
30 PRINT NAME$
```

This time, when ENTER was pressed the machine simply responded by moving the cursor down a line: it did not execute the program lines. In order to execute them, simply type RUN. The program will, when you've pressed ENTER, print ALBERT onto the screen. Once you have done this, you will have run your first real program!

INPUT

In Program 1.1, your name was stored directly in the program, which means of course that the program is only of use to you. To make it of more general use, it would be handy to be able to set the value of NAME\$ once the program is running. This can be done by using an 'INPUT' command which causes the program to stop and wait while the user enters the required information via the keyboard. The machine also needs to be told what variable name to assign to this information. Program 1.2 shows how an INPUT statement is utilised to assign a value to NAME\$ at the beginning of the program. In order to erase the old line 20 it is only necessary to type in the new one and then press ENTER. The machine will then write the new line over the old. To display the modified program on the screen, type LIST and press ENTER.

PROGRAM 1.2

```
20 INPUT NAME$
30 PRINT NAME$
```

When Program 1.2 is RUN a question mark will appear, indicating that the computer is awaiting an INPUT of information. On typing in your name and pressing ENTER, the computer will assign the INPUT to NAME\$, i.e. will put your name in NAME\$, and, at line 30, print out the value of NAME\$ - your name!

INPUT can also be used to input numbers. The only difference is that they must go to a numeric location, i.e.

```
2 INPUT AGE
```

Typing in a number into a string variable will mean the number will be stored as a string. However, you cannot type in a letter into a numeric variable. If you try, upon pressing ENTER you will get the following message:

```
?Redo from start
```

This is the computer's way of telling you that the input was not of the type expected. The question mark prompt will reappear and you should now type in a number.

Once part of a program is stored in memory, new lines will need to be added, and fortunately the Amstrad handles this very smoothly, displaying lines on the screen as they are ENTERed. However, suppose we wish to recap what the program is doing at line 10, when we are at line 5000?

LIST

There is a command in BASIC called 'LIST'. This command tells the computer to display all the lines of our program on the screen in numerical order. Try typing in LIST now. You should see your program come up on the screen when you've pressed ENTER. However, the Amstrad's LIST command is more versatile than this. If you type LIST 30 for example, you will see line 30 on the screen. The various forms of the LIST command are given below:

```
LIST          lists all of the program
LIST 30       lists line 30 only
LIST-30       lists the program up to and including line 30
LIST 30-      lists the program from line 30 onwards
LIST 30-60    lists lines 30 to 60 inclusive
```

Formatting Screen Displays - Separators

So far, all the PRINT statements you have used had something simple to print out. However, it's sometimes necessary to print several items onto the screen at once. This is handled in BASIC by means of features which tell the machine which 'format' is required on the screen. Thus, in order to print NAME\$ in Program 1.2 on the screen twice, it would be possible to write a line in a program that had a PRINT statement followed by two NAME\$'s. However, the Amstrad expects to see strings in PRINT statements separated. Not surprisingly, the things used to separate them are known as 'separators'!

To test this out, Program 1.2 can be modified to print out NAME\$ four times, first of all using the ',' (comma) separator. Change line 30 to read:

```
30 PRINT NAME$,NAME$,NAME$,NAME$
```

Typing in a line number with nothing in it tells the computer to search until it finds that line in its memory. Once it has been found it removes whatever was in that line and promptly forgets the line number.

If you type in a line number that does not exist e.g. '12', and then press ENTER, the Amstrad will search through its memory until it realises that there is no line twelve for it to forget. The following error message is reported:

```
Line does not exist
```

telling you that you have made a mistake.

Now delete line 2 by typing 2 and pressing ENTER. You should now have Program 1.2(a)

```
PROGRAM 1.2(a)
```

```
20 INPUT NAME$  
30 PRINT NAME$,NAME$,NAME$,NAME$
```

When this is RUN with a NAME\$ INPUT of 'FRED' it yields a screen display of:

```
FRED          FRED          FRED  
FRED
```

Screen Display Using the Comma Separator

FIGURE 1.10

Each of the strings being PRINTed onto the screen is allocated a third of the screen. This is known as the 'print field' and it provides a useful means of spacing out a display of strings or numeric variables into columns.

One important thing to note about strings is that if they are over 12 characters in length, then they will fill the print field and overrun into the next one. In this case, the next character to be printed following a comma separator will be moved over into the next available field. See Figure 1.11

FRED FRED	FRED	FRED
--------------	------	------

Example of PRINT Fields

FIGURE 1.11

The second separator is the semi-colon, ';', which has the effect of causing one string to be PRINTed immediately after the previous one. This is demonstrated in Program 1.2(b) where line 30 is further extended to include a semi-colon separator.

PROGRAM 1.2(b)

```
20 INPUT NAME$  
30 PRINT NAME$,NAME$,NAME$,NAME$;NAME$
```

This illustrates the effect of the semi-colon when RUN and yields the display shown in Figure 1.12:

```
FRED          FRED          FRED  
FREFRED
```

Screen Display Using Commas and a Semi-Colon

FIGURE 1.12

In all the examples so far, separators have been used with string variables. It turns out that their use with numeric variables is almost identical. The only difference is that numbers are printed with a space on either side of them. Thus, statements can be printed which contain mixtures of string and numeric variables separated by commas and semi-colons.

Prompts

We have found that the INPUT statement can be used in a program to get a response from the user. Unfortunately, the question mark is not a very helpful way of asking for information and the addition of some brief message would greatly improve matters. Such a message, usually known as a 'prompt', can very readily be added using a PRINT statement. This is done in line 10 of Program 1.3. type in the new line and RUN the program.

PROGRAM 1.3

```
10 PRINT "PLEASE TYPE IN YOUR NAME"  
20 INPUT NAME$  
30 PRINT NAME$
```

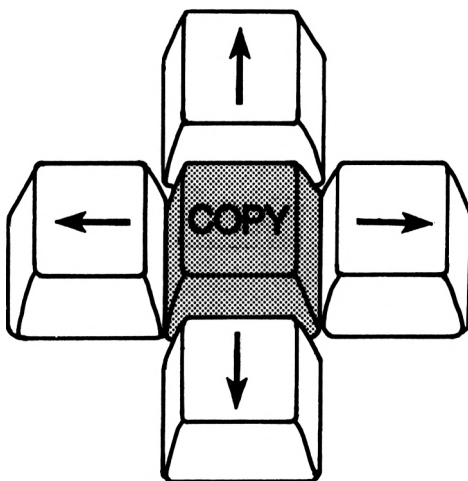
When Program 1.3 is run, things are a little more informative and you are actually asked for your name. However, there's an even neater way of doing this job in BASIC! The INPUT statement can itself be used to print a message by inserting the message between the 'INPUT' and the variable name. The line required is a mixture of lines 10 and 20.

What we will do is to add the statement in quotes in line 10 to the INPUT statement in line 20. The Amstrad computer is equipped with edit features to help you do this.

EDIT

To edit line 20 simply type EDIT 20 and press ENTER. A copy of line 20 will be displayed on the screen with the cursor placed on the '2' of '20'. You are now in EDIT mode.

Above the number pad to the right of the keyboard you will see the following keys.



The Cursor Keys

FIGURE 1.13

This set of keys are the edit keys. They move the cursor whilst in EDIT mode. Using the right-arrow key move the cursor until it is positioned between the INPUT statement and the variable name.

```
20 INPUT NAME$
```

When the cursor is in the correct position begin to type the following:

```
"PLEASE TYPE IN YOUR NAME";
```

As you type this in all that is in front of the cursor is moved forward. What you are typing in is being INSERTed between INPUT and NAME\$. When you type in the semi-colon NAME\$ is pushed forward so that the dollar symbol appears on a new line.

```
20 INPUT"PLEASE TYPE IN YOUR NAME"; NAME  
$
```

Don't worry about it. The Amstrad still understands that it is part of the program line. You should not press ENTER until you have finished typing in a line. The Amstrad allows you to keep typing in until six and a half lines of screen are full, then you will hear a beep telling you that you cannot type in any more.

If the line is now the same as above you have finished editing line 20, so press the ENTER key and you will have created a new line. You will also have left the EDIT mode.

That was a straightforward insert. A less straightforward task would be to change something like

```
100 PRINT "THIS MESSAGE IS WRONG"
```

to

```
100 PRINT "THAT MESSAGE WAS WRONG"
```

The Edit requires 'THIS' to be replaced with 'THAT' then the cursor needs to be moved to replace 'IS' with 'WAS'. The whole operation can be performed in easy steps.

- Type in : 100 PRINT "THIS MESSAGE IS WRONG"
- Press ENTER
- Type in: EDIT 100
- Use the right cursor key to move the cursor onto the 'I' of 'THIS'.
- Press the key marked CLR twice. Pressing CLR deletes whatever character the cursor is on.
- Type in: AT
- Press the right cursor key nine times so that the cursor is on the 'I' of 'IS'.
- Press CLR once
- Type in: WA
- Press ENTER
- You have now successfully edited line 100. Congratulations!

COPY

Another way to edit line 20 would have been to use the copy key. To see how this would have been done follow through the stages.

- Type in: LIST 10
- Press ENTER
- While holding down SHIFT use the cursor keys to move the cursor until it is on the first quotation mark on line 10. there are now two cursors on the screen. The one on the quotation mark is the EDIT cursor.
- Type in: 20 INPUT
This will appear where the normal cursor is.
- Press the COPY key. This will will move the edit cursor copying whatever is beneath it onto the position of the text cursor.
- Keep pressing the copy key until line 20 looks like this

20 INPUT "PLEASE TYPE IN YOUR NAME"
- Now type in: ;NAME\$
- Press ENTER
- You have successfully edited line 20 again. More congratulations!

Although it seems a long and difficult process, the EDIT features are fairly easy to use, especially with practice.

Anyway, going back to Program 1.3, having modified line 20 to include the message, the program now contains the same message twice and thus line 10 needs to be deleted. This is quite simply done by typing in the number 10 and then pressing ENTER.

Once line 10 has been removed, type LIST and Program 1.4 should appear as below.

PROGRAM 1.4

```
20 INPUT "PLEASE TYPE IN YOUR NAME";NAME$
30 PRINT NAME$
100 PRINT "THAT MESSAGE WAS WRONG"
```

When this is run a prompt will appear asking for your name and, following the entry, the computer will simply print your name back onto the screen! Now would be a good time to delete line 100 - so do so! Type in 100 and press ENTER.

Fortunately, the PRINT statement can also contain a variable in much the same way as the INPUT statement, so try to modify Program 1.4 as instructed in Exercise 1.1.

EXERCISE 1.1

Edit line 30 of Program 1.4 so that the program announces:

YOUR NAME IS FRED

(or whatever your name happens to be.)

An answer is given in the solutions chapter.

The INPUT statement can be used to INPUT more than one item at a time. To do this the comma (,) separator is used, e.g:

```
10 INPUT A$,B$,C$
```

When RUN the computer will expect three items of data to be entered. Suppose the data was 'ONE' , 'TWO' and 'THREE'. You would normally type in 'ONE' then press 'ENTER' (giving A\$ the value of ONE), then type TWO and press 'ENTER' and again with 'THREE'. With a multiple INPUT statement (like that above) after inputting the first value and pressing ENTER the Amstrad will respond with the message 'Redo from start!'. This is because the computer is expecting three items of data and it has only been given one.

The correct method of inputting the data is to type in 'ONE' then a comma then 'TWO' comma and then finally 'THREE' **NOT** followed by a comma but by pressing ENTER. The commas tell the computer that the next item of data is to be given to the next variable.

If the INPUT statement asks for three INPUTS, as above in line 10, and four pieces of data are entered, each separated by a comma, the user will receive the '?Redo from start' error message and you will be expected to type in the three items of data again.

Because a comma is used to separate strings during inputs it cannot be part of the INPUT string. The multiple INPUT statement can contain both messages and numbers. The same rules apply to numeric variables as to strings.

The messages or prompts you can now put into your programs are valuable in guiding the user through the problem of entering the right data. Try Exercise 1.2 using clear prompts.

EXERCISE 1.2

Modify the program developed in Exercise 1.1 so that it asks a person's name and age, and then reports back to them "YOUR NAME IS ..., YOUR AGE IS ... ". A possible answer is given in the solutions chapter.

LOCATE

As well as simply printing out your name on the screen, you can choose exactly where on the screen you wish your name to be printed. The Amstrad screen is divided into 'invisible positions' called 'cells', each character taking up one position. There are 40 'cells' across and 25 down, numbered 1-40 and 1-25 respectively. These cells are identified by referring to their horizontal and vertical positions known as 'coordinates'. In this book we will follow the usual convention and refer to the horizontal direction as the 'X coordinate' and the vertical direction as the 'Y coordinate'.

Using the LOCATE statement you have to specify the position of the 'cell' you want the computer to start printing at. The cells being referred to by their X and Y coordinates. For example:

```
LOCATE 10,2
```

will position the print cursor 10 characters along on the second line down. The next thing to be printed will begin there. The LOCATE statement can help 'tidy up' your screen display.

Before using LOCATE, it would be a good idea to get rid of the program which is currently stored in memory, namely Program 1.4 (as modified in the exercises). To do this use the command:

NEW

Typing in NEW and pressing ENTER makes the Amstrad forget whatever you have told it. It will 'forget' your program and the contents of any variables you have set up or used. To demonstrate this type NEW, then press ENTER, then type LIST and press ENTER again. There is no program there. Once you have entered NEW your program is gone for good so make sure you want to do that before entering NEW.

Now we can begin writing a new program using LOCATE.

PROGRAM 1.5

```
10 LOCATE 10,2
20 PRINT "HELLO"
```

Program 1.5 will print HELLO starting at the tenth cell across (X coordinate) and the second line down (Y coordinate). If there was anything on the screen around these locations then you might not be able to see the hello too clearly. It would be a good idea to clear the screen before running the program, the following command will do that for you.

CLS

The CLS statement is used to CLear the Screen either in a program or by direct entry. We will use it in our program to make sure that we always have a clean screen to start with. Normally after CLS whatever is to be printed will begin at the top left hand corner of the screen. However, line 10 moves the print cursor to a different start point, in this example ten characters across and two lines down.

With the addition of line 5 the program looks like this:

PROGRAM 1.6

```
5 CLS
10 LOCATE 10,2
20 PRINT "HELLO"
```

EXERCISE 1.3

Write a short program that will clear the screen and then ask your name, clear the screen again and print 'HELLO FRED' (or whatever) in the middle of the screen, using the LOCATE statement. A possible answer is given in the solutions chapter.

Now we have come to the end of the first chapter and you should understand some of the basic commands of Amstrad BASIC.

CHAPTER 2

PART ONE

Guess the number

This first mini project will be to develop a number guessing game and investigate various number manipulation techniques. In this game the computer will think of a number between 1 and 100 and the player will be asked to guess what the number is in less than six goes. The player will then be told whether the guess is too large, too small, or correct. After six goes, if it has not been guessed correctly the number will be displayed. At this stage, or when the number is guessed correctly, the player will be asked whether or not they wish to have another go.

The program will be built up in a modular fashion introducing various commands as and when necessary.

RND

In a game such as this the essential function is that of producing a random (ie. unpredictable) number for the player to guess. The Amstrad uses the RND command to generate a RaNDom number.

Try this:

```
PRINT RND
```

This will cause the Amstrad to print a random decimal number between 0 and 1. It will never be 0 or 1, always inbetween.

However, for our game this range is too small, we need a range of 1 to 100. This can be accomplished by simply multiplying the random number by 100, i.e.

```
PRINT 100*RND
```

Now the range is right, but the numbers being produced have digits after the decimal point, and what we need is just the whole number or 'integer' part. Amstrad Basic contains a command to remove the numbers after the decimal point, leaving just the integer value.

INT

The command to produce integer numbers from decimals is called, unsurprisingly, INT, where the brackets contain the number or expression to be INTegered. The Amstrad does this by rounding DOWN the decimal number to the nearest whole number,

e.g.

```
INT(6.0128)=6
INT(5.9)=5
INT(2.3*4)=9
```

Thus PRINT INT(RND*100) will print a random number between 0 and 99. The reason numbers up to 100 but not 100 itself are produced is because of the way the RND function provides numbers: never 100 but sometimes 99.99, which, when the integer value is taken, becomes 99. To produce numbers in the range 1 to 100 inclusive you simply need to add 1 to your random value, e.g.

```
PRINT INT(RND*100)+1
```

As this line will be used in a program it needs a line number. Also we need to assign this random number to a variable, which we shall call RANUM, meaning RANdom NUMbers.

PROGRAM 2.1(a)

```
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
```

GOTO

For lines 30 and 50 to be repeated a number of times, a command is needed which will send the program back to the beginning again. The command that does this is 'GOTO'. The GOTO command redirects a program to an indicated line number. As GOTO is self explanatory and should prove no difficulty in understanding. It can be added to Program 2.1(a) in line 80, to yield Program 2.1(b). Once this is done, Program 2.1(b) is said to LOOP back to line 30 from line 80.

PROGRAM 2.1(b)

```
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
80 GOTO 30
```

When this is RUN the program will enter an endless loop which PRINTs random numbers down the screen.

In order to terminate the proceedings, press the ESCape key twice. The first press halts the program and the second press exits from the program.

So far, the program can produce random numbers, but in a rather uncontrolled manner. What we need is some form of counting mechanism and some check on this count to say, for example, when 100 numbers have been delivered.

A counting mechanism is provided by the introduction of a variable called COUNT. This is set to zero at the begining of the program and is then increased (or 'incremented') by one each time a random value is PRINTed onto the screen. Thus, if we add lines 10 and 70 (below) to the program they will count the number of times we've gone round the loop. The program structure is as in Program 2.1(c):

PROGRAM 2.1(c)

Set counter to zero	10 COUNT=0
Generate a RaNDom number	30 RANUM=INT(RND*100)+1
PRINT the RaNDom number onto the screen	50 PRINT RANUM
Increment the counter	70 COUNT=COUNT+1
Go back for another random number	80 GOTO 30

However, counting how many random numbers have been printed is all that the program will do! So far it has not been told to respond in any other way to this number. As an experiment, RUN the program for a few minutes. When the fun(?) has worn off press the ESCape key twice to exit the program. Next, to check that the count routine has worked, type in PRINT COUNT and the machine will respond by telling you how many random numbers it has printed.

IF...THEN

So far so good - we can can count! The next job is to modify the program so that it can carry out a check on the state of the PRINTing and stop when enough numbers have been displayed. This is done by a checking or 'conditional' statement, which is in line 60 of Program 2.1(d).

PROGRAM 2.1(d)

```
60 IF COUNT=99 THEN GOTO 90
90 STOP
```

Line 60 checks the value of the variable COUNT and if - and ONLY if - it equals 99, the program goes to line 90 and STOPS.

STOP

The STOP in line 90 tells the program to STOP! The message 'Break in 90' is displayed. This is the Amstrad's way of telling us that the program has stopped at line 90.

The IF...THEN statement allows the user to place another BASIC instruction after an IF...THEN. This second BASIC instruction will only be executed when the IF condition has been met.

When using conditional statements, such as in line 60, care has to be taken over the number tested against. In this case the value which terminates the loop was 99, because the incrementing was done after the IF statement. Were this incrementing to have been done say, in a line 55, then the condition in line 60 would have been met when COUNT=100.

The combination of Programs 2.1(c) and 2.1(d) gives Program 2.1(e), which, when run will print out 100 random numbers.

PROGRAM 2.1(e)

```
10 COUNT=0
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
60 IF COUNT=99 THEN GOTO 90
70 COUNT=COUNT+1
80 GOTO 30
90 STOP
```

As programs become more and more complex, they become more and more difficult to follow and some means needs to be found for representing the flow of a program in a form that can be readily understood. Such a device is known as a:

FLOW CHART

A flow chart breaks the program down into simple elements which at the very simplest level:

1. STOP or END programs.
2. Process data: LET... statements.
3. Input and Output: statements such as PRINT and INPUT
4. Make decisions: IF...THEN... statements.

There are other program statements which don't quite fit into the above pattern. GOTO, for example, in effect changes the sequence of lines in a program while it is actually RUNNING.

It is often helpful to use flow charts to understand the logic of a program. Standard symbols are used for each of the four program elements mentioned above, as their use enables the diagrams or charts to be interpreted much more readily.

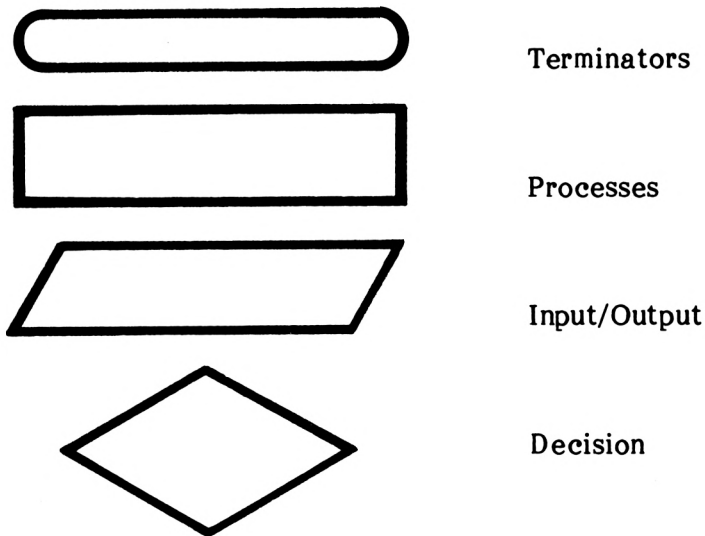
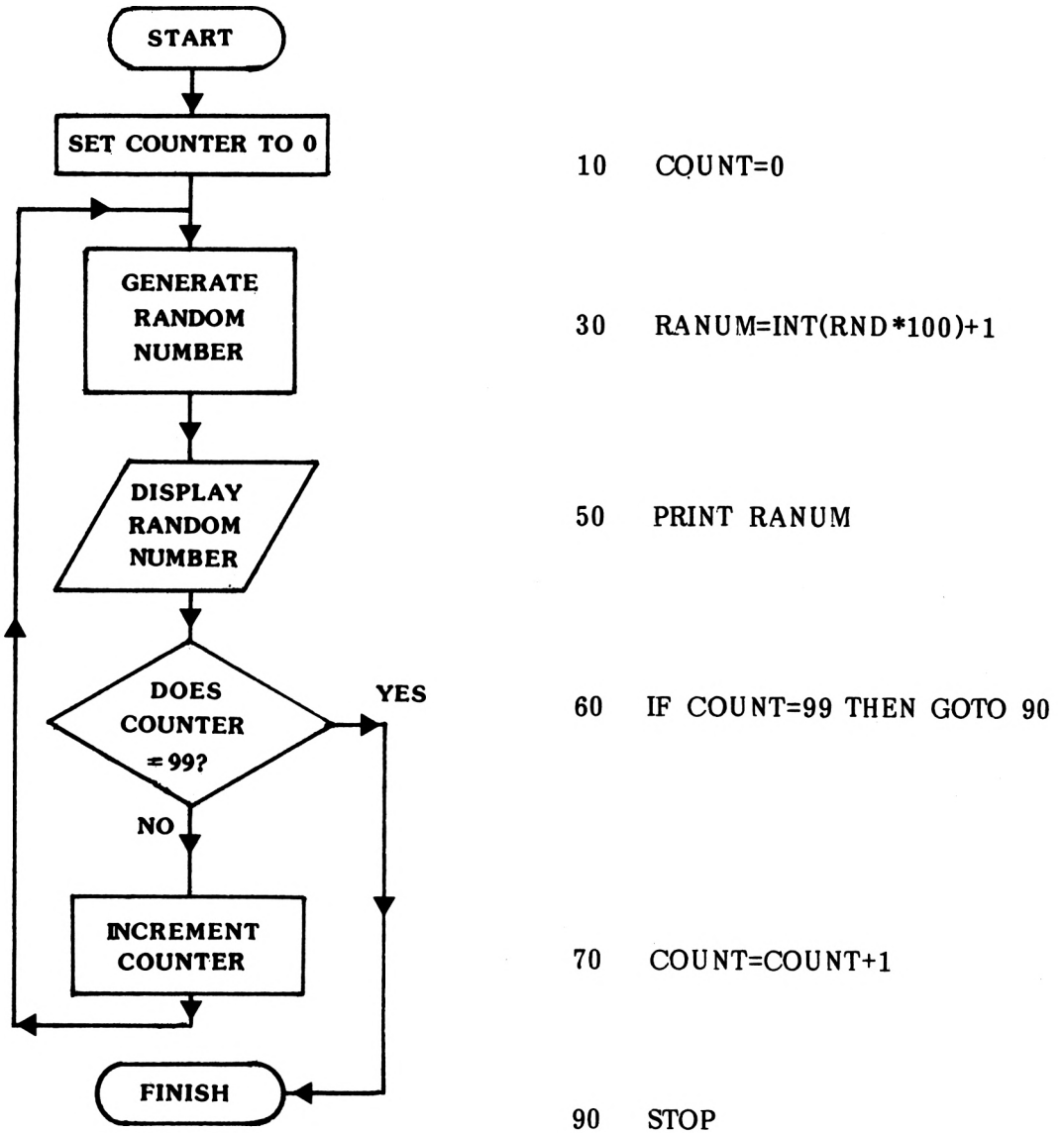


FIGURE 2.1

The rule for following a flow chart is really quite simple. You start at the top of the chart and follow the lines connecting the boxes, until you get to the end. Arrows on the connecting lines show the direction of flow.

Flow charts can be helpful when first designing a program. By convention, the explanations in the boxes should be written in plain English. It is a common mistake to write 'BASIC in boxes' and think that the end result is a proper flow chart. Always aim to make your flow chart's language and your computer's language independent.

Notice that the 'GOTO' in Program 2.1(e) is represented by a flow line on the flow chart, Figure 2.2. All the other equivalents to the program statements are contained in one of the four box types given above (Figure 2.1).



Flow Chart of Program 2.1(e)

FIGURE 2.2

Other conditional tests are available in BASIC and all the normal mathematical operators can be used to test values. For instance, Program 2.1(e) could be modified to use the 'greater than' sign, '>'. In this case delete line 60 and insert line 75.

PROGRAM 2.1(f)

```
75 IF COUNT>99 THEN GOTO 90
```

Another operator available for mathematical comparisons is '<' which means 'less than' and is used in a similar way to the 'greater than' symbol '>'.>

EXERCISE 2.1

Rewrite Program 2.1(e) to produce Program 2.2, which uses the line:

```
75 IF COUNT <(a number) THEN ....
```

The program should still print out 100 numbers.

Draw a flow chart to explain the operation of your program. A possible answer is given in the solutions chapter.

Using conditional tests in the programs has enabled loops to be produced, but BASIC contains its own built-in loop generator which makes life much easier - this is the:

FOR...NEXT Loop

When using this construction it is only necessary to define the beginning and end of a loop, as shown below:

```
FOR ..... Beginning of loop.  
Instructions within loop.  
NEXT..... End of loop.
```

As in Program 2.1(e), the number of passes through the loop needs to be defined and this is achieved by means of a variable that is incremented automatically on each pass of the loop. Thus the form shown above requires amendment, to become:

```

FOR COUNT=1 TO 100
loop
NEXT COUNT

```

In this statement the counting variable, in this case COUNT, is known as the 'loop variable' or the 'control variable', as it controls the number of times the loop is executed.

Incorporating this into Program 2.2 (produced from Exercise 2.1), the 'FOR' and 'NEXT' lines replace lines 10, 70 and 75 as indicated below.

PROGRAM 2.2(a)

```

10 COUNT=0 ← FOR COUNT=1 TO 100
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
70 COUNT=COUNT+1 ← NEXT COUNT
75 IF COUNT<100 THEN GOTO 30

```

Note that there is now no necessity for the STOP statement as the program will stop when 100 loops have been completed.

Using the FOR...NEXT statement for loops generally makes a program easier to understand. For instance, Program 2.2(a) can be simplified as shown in Program 2.2(b):

PROGRAM 2.2(b)

```

10 FOR COUNT=1 TO 100
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
60 NEXT COUNT

```

The FOR...NEXT loop above counts in 'steps' of '1'. We can however tell the computer to count in 'steps' other than one, using the 'STEP' command. The 'STEP' command is added to the end of the 'FOR...' statement like so:

```

FOR X=1 TO 100 STEP N

```

where N denotes the STEP size.

If we do not specify a STEP size then a STEP of one is assumed. To demonstrate the use of the 'FOR..NEXT...STEP' statement, enter and run Program 2.3(a). Type NEW first to get rid of the old program.

PROGRAM 2.3(a)

```
10 FOR X=1 TO 100 STEP 2
20 PRINT X
30 NEXT X
```

This particular loop starts at '1' and prints out every second number. So the display would be '1', '3', '5' up till the last value of 'X', 99.

EXERCISE 2.2

Change line 10 of program 2.3(a) so that the loop starts at 0 and increases in STEPS of three. A possible answer is in the solutions chapter.

The loop doesn't have to start at '0' or '1'. It can begin at any value less than (or equal to) the 'TO' value. If the first value is larger than the second (i.e. FOR X=100 TO 50) then what's needed is a countdown: 100, 99, 98, 97 etc. To do this we need a 'STEP' value of minus one (-1).

So to count down the instruction would read:

PROGRAM 2.3(b)

```
10 FOR X=100 TO 50 STEP -1
```

EXERCISE 2.3

Write a short program that will count down from '10' in 'STEPS' of '-1'. When the loop has been completed then the program will PRINT 'FIRE!'. A possible answer is in the solutions chapter.

If whilst using FOR...NEXT loops you made a mistake and typed in any of the following lines, the computer would quite sensibly ignore you (in the third case, it will do the loop once only):

```
FOR X=100 TO 10 STEP 1
FOR X=10 TO 100 STEP-1
FOR X=10 TO 20 STEP 30
```

As programs become more complex and include such features as FOR...NEXT loops, the danger of making mistakes increases. Fortunately, the Amstrad stays with you and when a bug creeps in error messages tell you what the error is. To demonstrate this, add line 30 of Program 2.3(c) to Program 2.3(b).

PROGRAM 2.3(c)

```
30 NEXT K
```

When this is RUN, the Amstrad will give an error message:

```
Unexpected NEXT in 30
```

This tells you that you have attempted to use a NEXT without a matching FOR at line 30 as the 'FOR' line used the variable X and the 'NEXT' line, the variable K.

Errors in Amstrad BASIC are readily picked up in this way as the computer has been taught its own logic. For instance, if you chose to say in English "The mat cat, on the sat", this would be incorrect in its 'syntax'. Thus, when similar errors occur in the Amstrad's language, the computer tells you that a 'Syntax error' has occurred. Just think of a Syntax error as the computer's way of saying "I DONT UNDERSTAND".

Having looked at a few essential BASIC statements, we can begin the development of this chapter's project: the number guessing game described at the beginning of the chapter.

First type in NEW and press ENTER to remove the earlier demonstration programs from the computer's memory.

The first thing the number game needs is a number for us to guess! This is quite easily done by setting up a random value RANUM which, for the time being, we will PRINT onto the screen:

PROGRAM 2.4(a)

```
30 RANUM=INT(RND*100)+1
35 PRINT RANUM
```

Next input a guess from the player. The guess will be stored in variable GUESS.

PROGRAM 2.4(b)

```
30 RANUM=INT(RND*100)+1
35 PRINT RANUM
50 INPUT GUESS
```

At this stage the guess can be compared with the random number using the IF...THEN statement. In the earlier example this was used only to end the program by means of a STOP command. However, the IF...THEN statement can be followed by any valid BASIC command, so in this case the statement could say: 'If the guess equals the random number then tell the player that the guess is correct.' Translating that into BASIC yields:

```
IF GUESS=RANUM THEN PRINT"WELL DONE-GUESS CORRECT."
```

One small tip before adding that line, though! During the development of this game you will probably RUN it hundreds of times. Fun as this may be for the first hundred or so times, it will probably get somewhat boring - eventually. To overcome this, line 35, the statement to PRINT out the value of the random number, is left in - it makes the game easier too! So far, then, the program reads:

PROGRAM 2.4(c)

```
30 RANUM=INT(RND*100)+1
35 PRINT RANUM
50 INPUT GUESS
60 IF GUESS=RANUM THEN PRINT"WELL DONE-G
UESS CORRECT."
```

At this stage the program should RUN and, when the answer is correct, give a message and then end. However, if an incorrect guess is entered, the program will simply end with no message. To handle this, two further conditional statements are added at lines 70 and 80 in Program 2.4(d).

PROGRAM 2.4(d)

```
70 IF GUESS>RANUM THEN PRINT"GUESS TOO L
ARGE-TRY AGAIN."
80 IF GUESS<RANUM THEN PRINT"GUESS TOO S
MALL-TRY AGAIN."
```

When Program 2.4(d) is run, it will handle both correct and incorrect answers but only for one INPUT. In order to give the player another chance, it clearly has to be re-routed back to the INPUT (line 50) if the answer was incorrect. This re-routing needs to be done conditionally based on the IF...THEN tests performed in lines 60, 70 and 80. Once again, BASIC comes to the rescue in that a second BASIC statement can be added to the end of an existing line provided that the two parts are separated by a colon(:). When this is done the line is referred to as a **multi-statement** line.

The second statement is executed immediately after the first, just as if it were on the next line, except that, as in this program, it comes after an IF...THEN statement. In this case, the extra statements will only be executed if the 'THEN' bit is executed, i.e. if the condition is met. Thus line 60 can be modified to read as in Program 2.4(e). The alteration to line 60 can be done most easily using an additional feature of the EDIT command. As in previous examples, you type in EDIT followed by the line number, which in this case is 60, and then you press ENTER. Move the cursor along to the end of the line using the cursor keys. Then all you have to do is type in the extra instruction (i.e. ':STOP'), and press ENTER. The new line 60 should look like this:

Program 2.4(e)

```
60 IF GUESS=RANUM THEN PRINT"WELL DONE-G  
UESS CORRECT.":STOP
```

This modification will STOP the program after a correct answer. Lines 70 and 80 can be similarly extended, in their particular cases to redirect the program, as in Program 2.4(f).

PROGRAM 2.4(f)

```
70 IF GUESS>RANUM THEN PRINT"GUESS TOO L  
ARGE-TRY AGAIN.": GOTO 50  
80 IF GUESS<RANUM THEN PRINT"GUESS TOO S  
MALL-TRY AGAIN.": GOTO 50
```

After the modifications in PROGRAM 2.4(f) the game will allow any number of incorrect guesses but comes to a STOP when the correct guess is made.

This ending is rather abrupt and the program would be improved considerably were the player to be given a choice after a correct guess - either to terminate the game or to play again. So a further routine is added at the end of the current program and offers the player the opportunity to continue. It takes the form of an INPUT with a message and a conditional test - see Program 2.4(g). In addition, the STOP will need to be removed from line 60 and the program redirected from here to the INPUT at line 110.

PROGRAM 2.4(g)

```
60 IF GUESS=RANUM THEN PRINT"WELL DONE-G  
UESS CORRECT":GOTO 110  
110 INPUT "DO YOU WANT ANOTHER GO(Y/N)";  
A$  
120 IF A$="Y" THEN XXX
```


In line 110, the INPUT is expecting a YES/NO type of answer and the bracketed '(Y/N)' is an additional prompt that gives the player a clear indication of what is expected in the way of inputted data. The use of such prompts makes it possible to test simply following the INPUT. In line 120, it is only necessary to test for the 'Y' - meaning 'Yes' - answer as this input is clearly expected. If the input is not a 'Y', then the program goes on to execute the next line or, if there isn't one, to end the execution. Note, the XXX in line 120 just refers to the line number without a 'GOTO'. This is because in Amstrad Basic when using an IF...THEN statement the GOTO is assumed if a line number follows a THEN. In fact you can, in the case of GOTO, make the statement an IF...GOTO instead of an IF...THEN, should that appear more clear to you.

As the INPUT expected is a string, i.e. alpha characters (letters), it was necessary to assign an appropriate string variable name - in this case A\$ is used. Line 120 is not complete and it is left for you to finish the loop. Just in case you make a mistake, or are not too sure, it is completed in later versions of the game.

As the game stands at the moment, the player can take any number of goes to guess the number. Just to add a bit more interest the number of attempts will be restricted to six. Ways have already been explored of getting programs to loop around a given number of times and as in Program 2.2(b), a FOR...NEXT loop can be used. This will be required to repeat the guessing part of the program, starting after the random number has been generated in line 40. The loop back - the 'NEXT' - will take place after the tests for the guess have been made and before the 'ANOTHER GO?' question is asked - at 90. These are shown in Program 2.4(h) where the variable 'COUNT' is used in the loop.

PROGRAM 2.4(h)

```
30 RANUM=INT(RND*100)+1
35 PRINT RANUM
40 FOR COUNT=1 TO 6
50 INPUT GUESS
60 IF GUESS=RANUM THEN PRINT"WELL DONE-G
UESS CORRECT.":GOTO 110
70 IF GUESS>RANUM THEN PRINT"GUESS TOO L
ARGE - TRY AGAIN.": GOTO 50
80 IF GUESS<RANUM THEN PRINT"GUESS TOO S
MALL - TRY AGAIN.": GOTO 50
90 NEXT COUNT
110 INPUT"DO YOU WANT ANOTHER GO (Y/N)";
A$
120 IF A$="Y" THEN 30
```

Just to prove this program for yourself, run it through a few times - firstly with a correct answer and then to check the loop with incorrect guesses. If you count the incorrect guesses you will find that the loop is not actually activated. To help you to see why, the flow chart for this program is given in Figure 2.3. You can use this to correct Program 2.4(h). Don't worry if you get stuck; the correction is explained below. Incidentally, there are no less than four faults or 'bugs' in the program at present. See if you can find them!

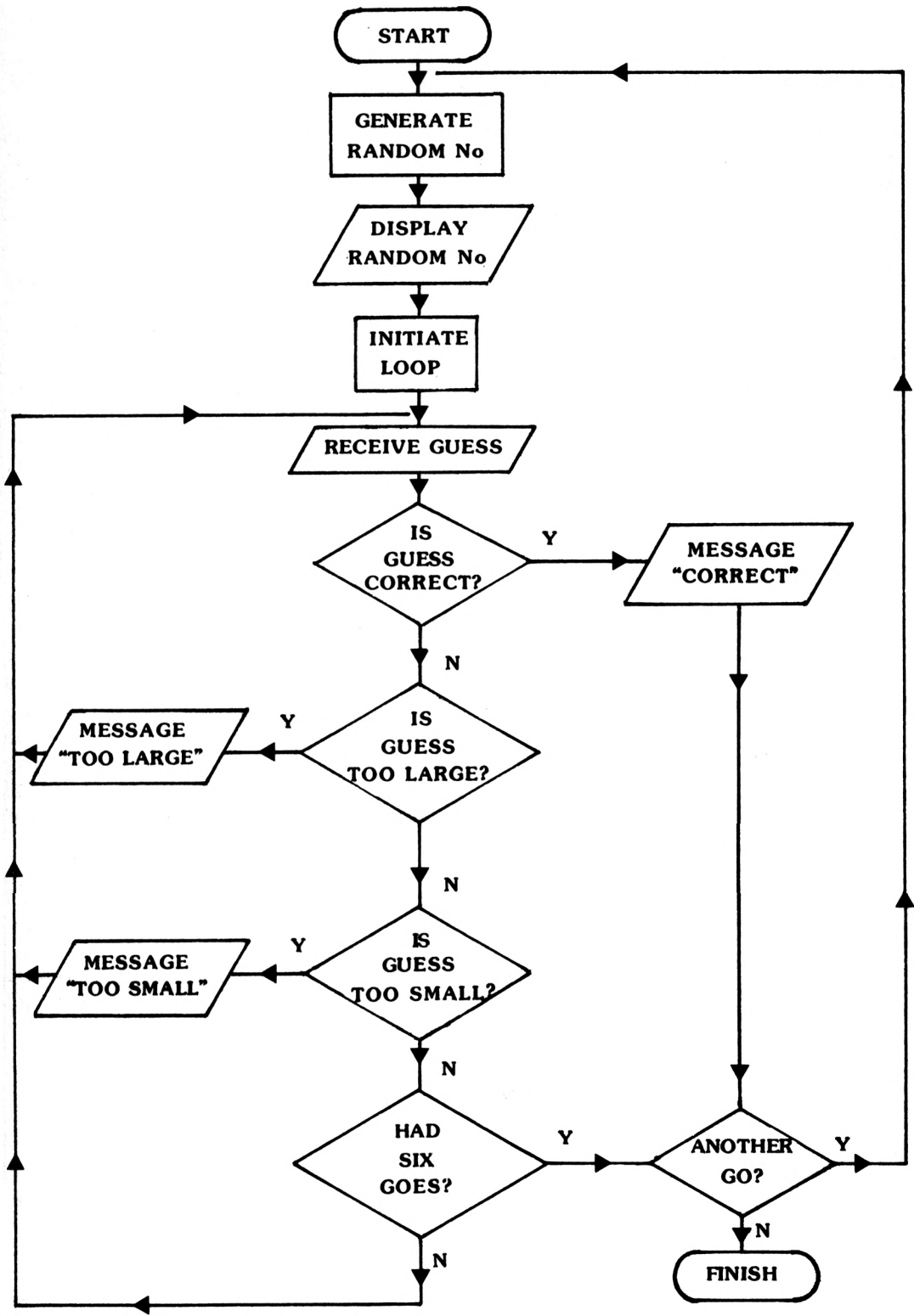


FIGURE 2.3

EXPLANATION - don't read this until you've had a go!

Following the program through for a correct answer yields no obvious problems. However, in line 60, a correct guess directs the program to line 110. While this is logically correct, it has the effect of jumping out of the FOR...NEXT loop which is BAD programming practice. The FOR is left waiting for a NEXT which will never arrive. It's a bit like arranging to be met from a train and then going by bus! Some computers will crash if you do this sort of thing but fortunately the Amstrad is smart enough to sort things out and the program will run as shown.

However, we will amend line 60 so that, instead, COUNT is set to some value, say 99, which will mean that when the NEXT is met in line 90 the computer will think that the loop has been fully completed. Also the loop fails to activate after the allowed six attempts. If a 'yes' results from the check 'is guess too high?' then a message is output. However, the program simply loops back from this point to allow another INPUT. Removal of the 'GOTO 50' in lines 70 and 80 will allow these lines to be followed by line 90, where 'NEXT COUNT' is met. The 'NEXT' function performs the necessary incrementing of 'COUNT' as well as checking whether it has yet reached 6. Once the loop is completed the NEXT command allows the program to run through to line 110. At this point the player will be asked: "DO YOU WANT ANOTHER GO?" Before this the player should be told: "YOU'VE HAD YOUR SIX GOES" but only if the guesses have all been wrong. To accomplish this a PRINT line could be added at line 100 if COUNT is less than 99, which it will be if none of the guesses were correct.

To summarise, the four modifications required to Program 2.4(h) are:

- (i) Remove the GOTO 110 in line 60 and insert COUNT=99
- (ii) Remove the GOTO 50 on line 70
- (iii) Remove the GOTO 50 on line 80
- (iv) Insert line 100:
100 IF COUNT<99 THEN PRINT"SORRY, YOU'VE HAD YOUR SIX
GOES."

This yields Program 2.4(i).

PROGRAM 2.4(i)

```
30 RANUM=INT(RND*100)+1
35 PRINT RANUM
40 FOR COUNT=1 TO 6
50 INPUT GUESS
60 IF GUESS=RANUM THEN PRINT"WELL DONE -
  GUESS CORRECT.":COUNT=99
70 IF GUESS>RANUM THEN PRINT"GUESS TOO L
  ARGE - TRY AGAIN."
80 IF GUESS<RANUM THEN PRINT"GUESS TOO S
  MALL - TRY AGAIN."
90 NEXT COUNT
100 IF COUNT<99 THEN PRINT"SORRY, YOU'VE
  HAD YOUR SIX GOES."
110 INPUT"DO YOU WANT ANOTHER GO (Y/N)";
  A$
120 IF A$="Y" THEN 30
```

IF...THEN...ELSE

The IF...THEN command has been used so far only to act if a condition is true, i.e. if A=5 THEN PRINT "Five". However IF...THEN has a built in option that allows the program to continue on the IF...THEN line only when the condition is false. For example, line 120 of Program 2.4(i) has the following test

```
IF A$="Y" THEN 30
```

This tests an input, if the key pressed was 'Y' then another go is required and the program loops back to line 30. However if the key pressed was not a 'Y' then the program simply ends. Using ELSE it is possible to tell the player that the program has finished on line 120. If the player decides that they do not want to play again the program will say goodbye and End. Thus line 120 needs to include the following:

```
IF A$="Y" THEN 30:ELSE PRINT"GOODBYE":EN
D
```

The ELSE section will only be preformed if A\$ is not "Y". If A\$ is "Y" then the program loops back to line thirty, as before.

PROGRAM 2.4(j)

```
120 IF A$="Y" THEN 30:ELSE PRINT "GOOD B
  YE!":END
```

All that really remains to be done now is for the program to report back on how many goes the player took to get the right answer.

EXERCISE 2.4

Add a reporting-back function to Program 2.4(i) such that it tells the player how many goes it took to get the correct answer. A possible answer is given in the next program but see if you can work it out yourself before looking.

Once Exercise 2.4 is completed, the result should be a functioning number guessing game. In many ways it is a bit simple, but from Program 2.5 onwards the rest is up to you. The major improvement that is needed is an introductory message to tell the player what the game is about and what the rules are and a polite 'goodbye' when the player finishes. It might also be an idea to stop PRINTing the random number at the beginning of the game!

In writing this program, provision has been made for additions at a later date in lines 10 and 20. In Program 2.5 below, both of these lines start with a **REM** command, which identifies each line as a REMark line. Once the Amstrad detects a **REM**, it then ignores anything that follows on this line. By means of REMs, comments can be inserted into programs to enable either the program's author or any other user to follow its logic more easily. A generous sprinkling of REMs is to be recommended to all.

PROGRAM 2.5

```
10 REM **NUMBER GUESSER**
20 REM *****GAME*****
30 LET RANUM=INT(RND*100)+1
40 FOR COUNT=1 TO 6
50 INPUT GUESS
60 IF GUESS=RANUM THEN PRINT "WELL DONE
- GUESS CORRECT.":PRINT"YOU TOOK ";COUNT
;" GOES":COUNT=99
70 IF GUESS>RANUM THEN PRINT"GUESS TOO L
ARGE - TRY AGAIN."
80 IF GUESS<RANUM THEN PRINT"GUESS TOO S
MALL - TRY AGAIN."
90 NEXT COUNT
100 IF COUNT<99 THEN PRINT "SORRY, YOU'V
E HAD
YOUR SIX GOES."
110 INPUT "DO YOU WANT ANOTHER GO (Y/N)"
;A$
120 IF A$="Y" THEN 30:ELSE PRINT"GOOD BY
E":END
```

There are many ways the program can be developed - for instance a 'GETTING WARMER/COLDER' function could be inserted instead of the higher/lower message. Another development could be to improve the display and messages. For example, telling an unsuccessful player what the number was!

EXERCISE 2.5

Alter line 100 of Program 2.5 to print out the number if it has not been guessed. A possible answer is given in the solutions chapter.

Storing a Program

Now to save that program onto tape!

Once a program of any length has been developed, it becomes a chore to keep typing it into the computer. It can, as you will see, be saved onto a storage device and then re-loaded back into the memory when you need it. Unlike most home computers on the market the Amstrad has a built-in cassette recorder. You cannot play music on it, however, because it is specially made to work with the Amstrad. There are two main advantages of having a built-in recording device. The first is that you don't have to buy one and the second is that saving programs is extremely reliable. Consequently there is no verify command on the Amstrad to check if your program has been saved properly.

Cassette storage is described as 'non-volatile' as it doesn't need any power to keep the program stored. The area of memory in the computer where your programs are stored is described as 'volatile' because, once the machine is turned off, all the contents of memory are lost. Amstrad Basic contains two commands for cassette storage. These commands form part of the machine's operating system, those built-in programs which make the whole computer work.

The BASIC commands for storage are: SAVE and LOAD.

SAVE and LOAD

The command used for saving a program onto tape is SAVE. It is not too difficult to use, but a few points need to be remembered:

- Press the 'PLAY' and 'RECORD' keys down before you enter the SAVE command.
- Remember that program names should be no more than 16 characters (letters and numbers) long.
- Remember not to try and record on the blank leader at the beginning of the tape.
- It is good practice to save important programs on two different tapes. Thus if you lose one you have a second copy. This is known as a 'backup' copy.
- Always label your tapes so that you know what's on them. This saves a lot of time when you are looking for a program that could be on any one of a number of tapes.

Now, let's suppose you have a program to save onto cassette, for example, the guessing game "GUESSER" which you have just written. To save that program, do the following:

- Insert a blank (unrecorded) cassette into the recorder, making sure that the tape is wound past the blank leader portion so that the actual brown recording section of the tape is visible.
- Press both the 'PLAY' and 'RECORD' buttons on the recorder.
- If you wish to call your program something other than GUESSER, you may substitute your own program name between the quotation marks below - but remember, the name must not be more than sixteen characters in length.
- Type SAVE "GUESSER" and press ENTER
- The Amstrad will report:

Press REC and PLAY then any key:

If you have not already done so then press record and play down on the tape.

- Press any key, other than a green key or the red escape button.
- Wait. The Amstrad is now, hopefully, saving the program onto the tape. You will see the following message:

Saving GUESSER block 1

- When it has finished, the Ready prompt will appear below the Saving message.

LOAD

Once the program is SAVED onto tape it can be transferred back into the computer by means of the LOAD command. Thus, to reLOAD the program called 'GUESSER' the command:

```
LOAD "GUESSER"
```

is used. Once this is ENTERed, you will be asked to:

Press PLAY then any key:

Upon pressing the play button and then any key on the Amstrad keyboard the program will begin to LOAD.

If all is well then the Amstrad will tell you that it has found the program by reporting:

Loading GUESSER block 1

A block can be looked upon as a lump of program. Depending on the size of the program the block number may be a small number (in this case 1) or a large number, the larger the program the higher the block number reached. The block number reflects how much memory the program fills.

Once the program has been safely loaded then the READY prompt will appear on the screen below the Loading message.

If whilst loading you should get the message:

Read error

on your screen do not panic! Most times this means that you have not wound the tape to the beginning of the program. If after doing this you still get an error, then you are unlucky enough to have got one of the extremely rare errors that do occur. If so, you must save your program again. One thing of paramount importance to remember is that when a load is successful any other program currently in the computer's memory will be erased.

A useful feature of your computer's BASIC is that it is not necessary to name the program that you are LOADING. If you have forgotten the name that you SAVED it under, then the command

LOAD""

will LOAD the first program found on the tape.

CAT

The last of the Amstrad cassette commands is CAT. CAT is short for CATALOGUE and is used to display the names of all files saved on the tape in the cassette drive.

After typing in CAT the Amstrad will respond with

Press PLAY then any key:

Once a key is pressed the Amstrad will begin searching through the tape. Every time a program is encountered the following will be displayed:

<filename> block <n> <filetype> OK

Where <file name> is the name of the file found. <n> is the block number and <file type> is one of the following file characters:

- \$ Basic program
- % Protected Basic file
- * ASCII text file
- & Binary file
- ' Protected Binary file

The OK message tells you that the program was saved okay. The CATalogue message for the tape that "GUESSER" was saved on would read:

The diagram illustrates the components of a catalogue display. It shows the text 'GUESSER' with an upward arrow from the label 'filename' below it. To the right, 'block 1' has an upward arrow from 'filesize' below it. Further right, '\$' has an upward arrow from 'basic program' below it. To the right of '\$' is 'OK', with an upward arrow from 'saved OK' below it. The labels 'filename', 'filesize', 'basic program', and 'saved OK' are positioned at the bottom, with arrows pointing to their respective parts in the display above.

The CATalogue Display

FIGURE 2.4

Amstrad Basic contains quite a few variations on LOAD and SAVE. These are all covered in Appendix Two.

PART TWO

Comparing Numbers

Various techniques are allowed in BASIC when comparing numbers; one very useful one allows two comparisons to be made in one statement. Using this, a program will be developed from Program 2.5 to produce a game which asks the player to guess two numbers. In order to simplify this, the equality check, the 'greater than' and the 'less than' lines should be removed, i.e. lines 60, 70 and 80.

Next, a second random number must be introduced. We will call the two numbers RANUM1 and RANUM2, to stand for Random NUMber1 and Random NUMber2. As the player is now to be asked to guess two numbers it would also be easier if the possibilities for each number were to be reduced to, say, the range 1 to 4.

PROGRAM 2.6(a)

```
30 RANUM1=INT(RND*4)+1:RANUM2=INT(RND*4)
+1
```

In this particular game, two guesses will be required and, as with the random's these can be called 'GUESS1' and 'GUESS2' as in line 50 of Program 2.6(b).

PROGRAM 2.6(b)

```
50 INPUT "GUESS1";GUESS1:INPUT "GUESS2";
GUESS2
```

AND

With two guesses and two random numbers, the testing process becomes much more complex than in the earlier game. However for BASIC command 'AND' eases things somewhat. It enables one, for instance to compare two guesses on one line of a program. Thus, using AND it is possible to say:

```
IF GUESS1=RANUM1 AND GUESS2=RANUM 2 THEN
PRINT "WELL DONE"
```

The "WELL DONE" will only be printed when your first guess is equal to the first random number AND the second guess equals the second randomly generated number. The above line checks for correct guesses and can be used in the program at line 60. If the guesses are correct then the loop counter is set to an extreme value and the program jumps to line 90, where the NEXT is encountered and the loop is terminated as COUNT is greater than 6.

PROGRAM 2.6(c)

```
60 IF GUESS1=RANUM1 AND GUESS2=RANUM2 TH
EN PRINT"WELL DONE - GUESS CORRECT":COUN
T=99:GOTO 90
```

It would be nice if the player could win after having typed in the correct numbers but in the wrong order. To allow for this line 65 of Program 2.6(d) compares GUESS1 with RANUM2 and GUESS2 with RANUM. If they are both equal then the player has guessed correctly.

PROGRAM 2.6(d)

```
65 IF GUESS1=RANUM2 AND GUESS2=RANUM1 TH  
EN PRINT"WELL DONE - GUESS CORRECT":COUN  
T=99:GOTO 90
```

Having tested for a correct answers it is now time to look at the possibilities of incorrect guesses. Logically speaking if the two were not correct then they are incorrect and the player should be told that she has got it wrong. This is the easy way out, a path that we will not be taking! Instead, further checks will be made on the user's guesses to see if one or both of them are wrong.

The test for two incorrect guesses will simply read:

```
If GUESS1 not equal to RANUM1 AND GUESS1  
not equal to RANUM2 AND GUESS2 not equal  
to RANUM1 AND GUESS2 not equal to RANUM2  
then PRINT "Both wrong"
```

In BASIC 'not equal to' is indicated by the less than '<' and greater than '>' signs placed together i.e.

<>

Program 2.6(e) contains the line which tests to see if the guesses are incorrect. Line 100 has also been amended to print out both numbers if they have not been guessed after six tries.

PROGRAM 2.6(e)

```
30 RANUM1=INT(RND*4)+1:RANUM2=INT(RND*4)
+1
40 FOR COUNT=1 TO 6
50 INPUT"GUESS1";GUESS1:INPUT"GUESS2";GU
ESS2
60 IF GUESS1=RANUM1 AND GUESS2=RANUM2 TH
EN PRINT"WELL DONE - GUESS CORRECT":COUN
T=99:GOTO 90
65 IF GUESS1=RANUM2 AND GUESS2=RANUM1 TH
EN PRINT"WELL DONE - GUESS CORRECT":COUN
T=99:GOTO 90
70 IF GUESS1<>RANUM1 AND GUESS1<>RANUM2
AND GUESS2<>RANUM1 AND GUESS2<>RANUM2 TH
EN PRINT "BOTH WRONG":GOTO 90
90 NEXT COUNT
100 IF COUNT<99 THEN PRINT "SORRY, YOU'V
E HAD YOUR SIX GOES":PRINT "THE NUMBERS
WERE"; RANUM1,RANUM2
110 INPUT"DO YOU WANT ANOTHER GO (Y/N)";
A$
120 IF A$="Y" THEN 30:ELSE PRINT"GOOD BY
E":END
```

So far the guesses have been tested to see whether they are both correct or both wrong. If neither of these conditions are true then the program is diverted to line 90 where the count is incremented. Then you are left with a pair of guesses one of which is correct and, logically enough, one incorrect. It is not necessary to conduct any further testing because if the guesses are not both correct or both incorrect then one of them must be right. Line 80 tells the player this.

PROGRAM 2.6(f)

```
80 PRINT"ONE RIGHT"
```

Once line 80 has been entered you have a working two-number guessing game, to delight and amuse your friends.

OR

As demonstrated above the AND command allows us to compare two variables and act upon the result. BASIC provides us with a second command to use when comparing numbers. The OR command works like this:

"If this is correct OR if that is correct
then do something"

In Basic that would be:

```
IF A=1 OR A=2 THEN PRINT A
```

The above condition would print the value of A, if the value was '1' or '2'. This is a more flexible command than AND.

The OR command can be used in the number guesser game to combine lines 60 and 65, the correct guess tests. This will produce the following line:

PROGRAM 2.7

```
60 IF (GUESS1=RANUM1 AND GUESS2=RANUM2)  
OR(GUESS1=RANUM2 AND GUESS2=RANUM1) THEN  
PRINT"WELL DONE GUESS CORRECT":COUNT=99  
:GOTO 90
```

The brackets on line 60 are used to separate each section of tests. The tests in the first brackets are those that were in the original line 60 and the second set of brackets contains the tests that were in line 65.

Line 120 tests to see if the user has typed 'Y' for another go. However, if a lower-case 'y' were entered there would be no match and the program would end. To cater for this as well we can use 'OR' and test for 'y'. This gives the following line:

PROGRAM 2.8

```
120 IF A$="Y" OR A$="y" THEN 30:ELSE PRI  
NT" GOODBYE":END
```

The program is now complete and should look like Program 2.9. Try improving it - increasing the range of the numbers to be guessed, better displays using the LOCATE statement, improved messages, etc.

PROGRAM 2.9

```
10 REM NUMBER GUESSER
20 CLS
30 RANUM1=INT(RND*4)+1:RANUM2=INT(RND*4)
+1
40 FOR COUNT=1 TO 6
50 INPUT"GUESS1";GUESS1:INPUT"GUESS2";GU
ESS2
60 IF (GUESS1=RANUM1 AND GUESS2=RANUM2)
OR(GUESS1=RANUM2 AND GUESS2=RANUM1) THEN
PRINT"WELL DONE GUESS CORRECT":COUNT=99:
GOTO 90
70 IF GUESS1<>RANUM1 AND GUESS1<>RANUM2
AND GUESS2<>RANUM1 AND GUESS2<>RANUM2 TH
EN PRINT "BOTH WRONG":GOTO 90
80 PRINT"ONE RIGHT"
90 NEXT COUNT
100 IF COUNT<99 THEN PRINT"SORRY YOU'VE
HAD YOUR SIX GOES. ":PRINT"THE NUMBERS WE
RE ";RANUM1;" AND ";RANUM2
110 INPUT"DO YOU WANT ANOTHER GO (Y/N)";
A$
120 IF A$="Y" OR A$="y" THEN 30:ELSE PRI
NT "GOODBYE"END
```

PART THREE

Mathematical Precedence

There are rules in mathematics that determine the order in which calculations are done. Multiplications and divisions are always performed before addition and subtraction, i.e. they have 'precedence'. For example in the calculations:

$$5+3*4$$

The calculation '3*4' would be performed **before** five is added. Similarly in the calculations:

$$10/2-3$$

The division (10/2) will be the first calculation. Supposing a calculation had multiplication and division, much like this one:

$$3/4*2$$

Then the sum is calculated starting from the left. The first calculation being $3/4$ and then the multiplication by two is done ($2 \times 3/4 = 1.5$).

Brackets

There is a way of breaking the mathematical order and that is the use of brackets. Calculations enclosed in brackets are **always** done first. For example, in the following sum:

$$2*3+(4-2)$$

$4-2=2$ will be calculated first. The next precedence is multiplication ($2*3=6$) and lastly the addition ($6+2=8$). In the event of brackets within brackets, or 'nested' brackets, the calculations within the innermost set of brackets are done first, the calculation then working outwards. For example:

$$4*2+(6+4*(3+2)+1)*2$$

The $(3+2)$ is done first, giving 5. Then $4*5$ is done, giving 20. Then 6 is added, then 1, giving 27. At this stage the calculation reads:

$$4*2+27*2$$

Thus the computer does $8+54$, giving 62.

If a calculation contains several sets of brackets (not necessarily nested), then it will do these from left to right - any nested inner brackets first of course.

Summarising this, the mathematical order of precedence is:

innermost brackets
brackets
* and /
+ and -

In cases where there is no competition for precedence (e.g. a calculation containing only + and -) then it is calculated from left to right.

CHAPTER 3

PART ONE

Graphics

In the previous two chapters we have seen how to print characters onto the screen. In this chapter you will be shown how to produce pictures using some of the Amstrad's high resolution graphic commands.

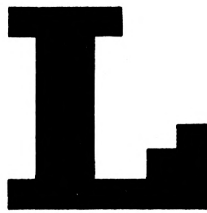
Modes

The Amstrad has three different modes, i.e. screen display formats, called MODE 0, MODE 1 and MODE 2. When the Amstrad is switched on it goes into MODE 1. This is the default mode and is probably the one you are using at the moment.

The most obvious difference between each mode is character size. Figure 3.1 illustrates the different sizes of characters in each mode. MODE 0 has very large characters, MODE 1 characters are medium sized and MODE 2 has very small characters.



MODE0



MODE1



MODE2

The Three Character Sizes

FIGURE 3.1

To change any of the MODEs you simply type in 'MODE' and then the required MODE number. For example:

MODE 0

will switch the computer to MODE 0. Notice the large letters. As well as changing the mode the screen was cleared and the cursor placed in the top left hand corner of the screen. This is referred to as the 'home' position.

Because Mode 0 characters are so big only twenty can fit onto one line on the screen, although there are twenty five lines. Another difference between the modes is the different number of colours available at any one time. Figure 3.2 shows the different number of colours and characters for each mode.

MODE	Colours Available	Text Display
0	16	25 lines x 20 characters
1	4	25 lines x 40 characters
2	2	25 lines x 80 characters

The Screen Display Available In Each MODE

FIGURE 3.2

So far you have only been using the colours the Amstrad starts up with. It is possible for the user to change the border, screen and character colours. The Amstrad has twenty seven colours for you to choose from so you should find something to your taste. Those of you without a colour monitor will see each different colour in varying shades of green.

The Amstrad comes complete with four colour commands, these being BORDER, INK, PAPER and PEN.

BORDER

This command is used to change the colour of the screen's border. The colour you wish to change it to is indicated by a number following the BORDER command. The Amstrad's twenty seven colours are numerically coded, ranging from 0-black to 26-bright white. The colour used in the border is separate from the colours available on the screen area inside it. A complete list of colours and their numeric codes are given in Figure 3.3

0	BLACK	14	PASTEL BLUE
1	BLUE	15	ORANGE
2	BRIGHT BLUE	16	PINK
3	RED	17	PASTEL MAGENTA
4	MAGENTA	18	BRIGHT GREEN
5	MAUVE	19	SEA GREEN
6	BRIGHT RED	20	BRIGHT CYAN
7	PURPLE	21	LIME GREEN
8	BRIGHT MAGENTA	22	PASTEL GREEN
9	GREEN	23	PASTEL CYAN
10	CYAN	24	BRIGHT YELLOW
11	SKY BLUE	25	PASTEL YELLOW
12	YELLOW	26	BRIGHT WHITE
13	WHITE		

The Amstrad Colours and Codes

FIGURE 3.3

Thus to change the border to pink you simply type:

```
BORDER 6
```

Upon pressing ENTER the BORDER will become pink. In this manner the BORDER colour can be set to any of the available colours.

One special feature of the Amstrad monitor is that you can change the BORDER to two colours. This is done by adding another colour value after the first.

```
BORDER 16,0
```

This causes the Amstrad to constantly flash the BORDER colour from pink to black. A very interesting (!) effect.

INK

The INK command is used to select which colours you wish to use for the screen and characters. This command takes the form of:

```
INK i,c
```

Where 'i' is the INK number and 'c' is the colour number. To understand this command you have to imagine a row of ink pots. Each ink pot has its own unique number (the INK number) and contains the colour 'c'.

So INK 0,6 means fill the ink pot numbered 0 with colour 6 (bright red). When the Amstrad is switched on the 'ink pots' are given default colour values. The colour in each ink pot differs depending upon which mode you are in. Figure 3.4 shows the default INK values for each of the three modes.

INK	MODE 0	MODE 1	MODE 2
0	1	1	1
1	24	24	24
2	20	20	1
3	6	6	24
4	26	1	1
5	0	24	24
6	2	20	1
7	8	6	24
8	10	1	1
9	12	24	24
10	14	20	1
11	16	6	24
12	18	1	1
13	22	24	24
14	Flashing 1,24	20	1
15	Flashing 16,11	6	24

The Default INK Values

FIGURE 3.4

From a look at the chart you will see that MODE 2 has only two colours, MODE 1 has four colours and MODE 0 has sixteen. These are the different colour limitations for each mode. It doesn't matter what the particular colours are but you are only allowed two in MODE 2, four in MODE 1 and sixteen in MODE 0.

PAPER

The PAPER command changes the background colour of the characters on the screen and takes the form:

PAPER i

Where 'i' is the INK number.

For example:

PAPER 8

will change the background to the colour that was put into ink pot eight. This colour depends on which mode you are in. The colours are cyan, blue and blue for modes 0,1 and 2 respectively.

If you now type in some characters you will find that they have a different background colour to those already on the screen. To change the background colour for the whole screen just type 'CLS'. The whole screen clears and each character cell has been filled with new background colour.

PEN

This changes the colour of the cursor and subsequently any characters that appear AFTER the PEN command has been used. Pen works in the same way as Paper. So:

PEN 3

will change the character colour to whatever ink pot 3 contains.

Now we can change the colour of the screen and characters to any combination we require. Suppose you required a lime green screen with a pen colour of bright cyan. The Amstrad uses INK 0 as the default value for the screen colour and INK 1 as the default value for the character colour. Thus to change the current colours to the required values we simply change the value of INK 0 and INK 1. Type in the following direct entry commands:

```
INK 0,21
```

```
INK 1,2
```

Twenty one and two are the colour codes for lime green and blue respectively.

EXERCISE 3.1

Write a short program that will switch the computer to MODE 0. The BORDER colour should be set to white and the background to sea green. The character colour should become pastel magenta. A possible answer is given in the solutions chapter.

Graphics

As promised earlier, we will now show you some of the Amstrad's graphic commands. As you found out earlier, the size of the characters on the screen varies with each mode, being largest in MODE 2 and smallest in MODE 0. A similar situation exists when using graphics. To produce drawings we need to 'light-up' the appropriate points on the screen. These points are known as 'PIXELS' (short for PICTURE ELEMENTS - well sort of!) and they also vary in size with each mode (largest in MODE 2, smallest in MODE 0). The difference in size is known as the 'resolution', i.e. the sharpness of the display, of each mode. If you look closely at a photograph in a newspaper you will see that it is made up of hundreds of little dots - it is the same with graphics on your Amstrad. The smaller the pixel, the sharper the display that you will get. The difference in pixel size for each mode is illustrated in Figure 3.5.




MODE	PIXEL SIZE (relative)	PIXELS PER SCREEN
0		160 x 200 pixels
1		320 x 200 pixels
2		640 x 400 pixels

FIGURE 3.5

PLOT

Each point has its own unique 'X' and 'Y' coordinates. These range from 0,0 (the bottom left hand corner) to 640,400 (the top right hand corner). To 'set', i.e. 'light-up' a pixel at any point on the screen you use the PLOT command, in the form:

PLOT X,Y

where X and Y are the coordinates of the pixel to be set. For example:

PLOT 100,100

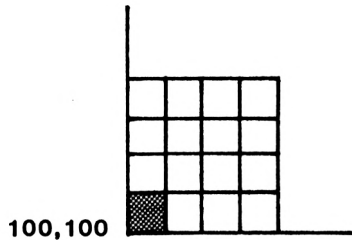
This will set a pixel at the point one hundred across and one hundred up from the origin. The origin is the position that has the coordinate value of 0,0, i.e. the bottom left hand corner of the screen. The colour is decided by the value of INK 1.

To PLOT a pixel at the centre of the screen type in the following program:

```
PROGRAM 3.1
```

```
10 PLOT 320,200
```

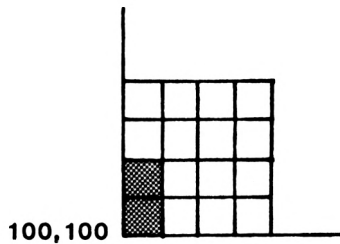
When plotting a point the Amstrad looks at two things. Firstly the X and Y coordinate values and secondly the MODE. In Mode 2 the pixel coordinates correspond exactly with the point coordinates, therefore in mode two the Amstrad will plot only the point specified by the coordinates. This is demonstrated below.



A Point Plotted In Mode 2

FIGURE 3.6(a)

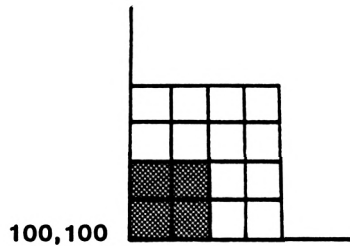
However, when Mode 1 is being used the computer plots two points because each pixel in Mode 1 is twice that of those in Mode 2. In Mode 1 the pixels are grouped together in twos. Consequently the plotting at one location will cause the second in that group of two to be plotted as well. Thus the plotting of location 100,100 will cause location 100,101 to be switched on as well.



A Point Plotted In Mode 1

FIGURE 3.6(b)

Mode 0 is treated in exactly the same way, but this time the pixel size is twice that of Mode 1 and four times that of Mode 2. Consequently one plot command in Mode 0 causes four points to be set.



A Point Plotted In Mode 0

FIGURE 3.6(c)

In other words, the sizes of the pixels may differ, but regardless of the mode being used the graphics screen always has the same number of graphics points, i.e. 640x400. This means that, when using modes 0 or 1 (where the resolution is less than 640x400), drawing at any particular graphics point on the screen will 'light-up' certain surrounding points.

To demonstrate this point (?) further type in and run Program 3.2.

PROGRAM 3.2

```

5 MODE 0
10 FOR X=0 TO 640 STEP 4
20 PLOT X,100
30 NEXT X

```

Because each pixel in Mode 0 takes up four points this program will produce a whole line across the screen even though it is plotting in steps of four (it's also much quicker!). When it is finished change line 5 to the following:

```

5 MODE 1

```

Now run the program. This time, because Mode 1 pixels are half the size of Mode 0 pixels, the program will produce a dotted line where every second point is set. When the program finishes again change line 5.

```

5 MODE 2

```

This time every fourth point is set producing an even 'dottier' line.

PART TWO

An Etcha-Sketcha Game

The main problem associated with any etcha-sketcha type of game is that the program must get the computer to draw lines of indeterminate length.

This problem can be overcome by arranging the program so that it slightly increases a line's length every time a key is pressed. This problem is tackled in Program 3.3. In order to prepare the Amstrad to receive a new program it is essential to type in the command NEW; do this now and then type in Program 3.3.

PROGRAM 3.3

```
10 MODE 1
70 INPUT D$
80 PLOT X,Y
90 X=X+1
100 GOTO 70
```

If you follow the program through, here is what it does.

- First the screen is cleared and set to MODE 1
- The computer asks for an input and waits for ENTER to be pressed
- The point (X,Y) is plotted on the screen. (You may have to look carefully for it!)
- The X coordinate is increased by 1.
- The program loops back to line 70 for another input. The question-mark prompt appears one line lower each time line twenty is reached.
- After a second input another pixel is set. You cannot see this pixel because the PLOT coordinates for this pixel have already been set by the first PLOT statement. In this method it would take two inputs to set each pixel.

To see a new pixel after every input the X coordinate will have to be increased in steps of two. This, however, is only true for Mode 1. Those of you who may wish to try the program out on Mode 1 or 2 should add 4 and 1 to X respectively.

This still leaves us with a problem: the object of the game is for the line to increase each time a key is pressed, but using INPUT we have to press ENTER as well. Not only that but the question-mark prompt appears on the screen. There is a command which allows you to input characters without pressing ENTER and without displaying any form of input prompt. This is the INKEY\$ command.

INKEY\$

What the INKEY\$ command does, as its name implies, is to take IN the value of which ever KEY is pressed. The command takes the form of:

```
A$=INKEY$
```

Where A\$ is the string variable in which you want the value of the key pressed to be stored.

The major differences between INKEY\$ and INPUT is that INKEY\$ accepts the keystroke without requiring ENTER to be pressed and also the character is not displayed on the screen. Once INKEY\$ is encountered, the computer scans the area of memory where the characters from keys that have been pressed are stored. This is known as the 'keyboard buffer'. The first character it finds is assigned to A\$ and if nothing is found then a null string is assigned to A\$ and the program simply passes on. This command can be used to halt a program until a key is pressed by simply testing the keyboard buffer to see whether a particular key has been pressed. If the test proves negative, i.e. if the key has not been pressed, then the program is returned to the beginning of the line and the check repeated. The following program shows this:

PROGRAM 3.4(a)

```
2 PRINT "PRESS 8 TO BEGIN"  
3 A$=INKEY$  
4 IF A$ <>"8" THEN 3
```

When the program is run now, nothing happens initially until the '8' key is pressed. Once this is done, the program continues to operate as it did before.

With a line such as 4 in Program 3.4(a), the program waits until a particular key is pressed. For this line of the program to work satisfactorily, the user needs to be told which key needs to be pressed. A program can be made more general by using the INKEY\$ command to look in the buffer to see whether NO key has been pressed, in which case the 'empty string' or 'null string' will be stored. In BASIC a null string is described by means of two quotation marks together, i.e. "". In a program it would be entered as in line 4 of Program 3.4(b) below. What this line is saying is that if the keyboard buffer contains the 'null string', i.e. nothing, then go back for another input. This process will be repeated forever until a key - except SHIFT or CAPS LOCK - is pressed. The ESC key is also picked up but that only halts the program, as usual.

PROGRAM 3.4(b)

```
4 IF A$="" THEN 3
```

Such a device is commonly used when one needs to pause a program whilst the user reads a message. In this case some sort of prompt is incorporated to tell the user what is expected. To do this, lines 2 and 6 of Program 3.4(c) need to be added. Line 2 prints the message onto the screen and once a key has been pressed, line 6 clears the screen using the CLS command.

PROGRAM 3.4(c)

```
2 PRINT"PRESS ANY KEY TO BEGIN"  
3 A$=INKEY$  
4 IF A$="" THEN 3  
6 CLS
```

Lines 2 to 6 were used only for demonstration purposes and are no longer needed so we have to get rid of them. This offers the opportunity to examine another of Amstrad BASIC's editing commands.

DELETE

To remove a single line you just type the relevant line number and then press ENTER. This process can be repeated until all the unwanted lines have been removed. Amstrad BASIC does, however, have an easier way of removing groups of successive program lines; this is the DELETE command.

To remove a whole block of lines you have to specify two things. First, the line number that you want to start deleting at and second, the line number that the deletion will finish at. Thus to delete lines 2 to 6 type in:

```
DELETE 2-6
```

upon pressing ENTER these lines will be deleted. They are no more! You must be careful when using the DELETE command. If you make a mistake you could end up removing your entire program.

Delete has variations similar to those used with LIST.

```
DELETE          : Delete all of a program
DELETE 10       : Delete only line 10
DELETE 10-100   : Delete all lines from 10 to 100 INCLUSIVE
DELETE 10-      : Delete all lines from line 10 onwards
DELETE -100     : Delete all lines up to and including line 100
```

Going back to the etcha-sketcha program, the INKEY\$ command can be incorporated via line 70. If no key has been pressed then the program will loop around 'inside' line 70 until one is.

PROGRAM 3.5

```
70 A$=INKEY$:IF A$="" THEN 70
```

The next task is to modify the program so that it draws a line in a chosen direction. This is achieved by testing the value of A\$. The Amstrad keyboard has four arrow keys, one pointing in each direction (left, right, up and down). It would be nice if we could use these keys to move the 'line' in the appropriate direction. To allow for this the program could contain the following sequence of instructions:

```
IF A$="↑" THEN Y=Y+1
IF A$="↓" THEN Y=Y-1
IF A$="←" THEN X=X-2
IF A$="→" THEN X=X+2
```

However, there is a slight problem. When you attempt to type in an arrow between the quotes you do not get an arrow, the cursor moves in the arrow's direction. This is quite a problem. If the cursor arrows will not appear between the quotation marks how can an input be tested for them? Help is at hand.

CHR\$() and ASC()

Each key on the Amstrad keyboard has a special numeric value referred to as its ASCII value. ASCII is an acronym from American Standard Code for Information Interchange. The way to find out the ASCII value of any character is to print its ASCII value using the ASCII function. For example

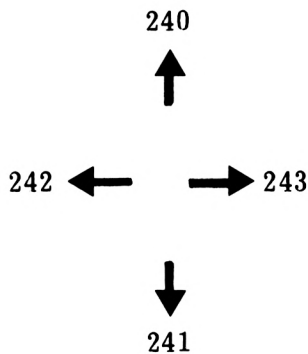
```
PRINT ASC("A")
```

Will result in a display of 65. Sixty five is the ASCII value of upper case 'A'. The letter 'A' can now be displayed on the screen by printing CHR\$(65), e.g.

```
PRINT CHR$(65)
```

The number enclosed in brackets is an ASCII value of a character, in this case an upper case 'A'.

Thus the arrow keys can be tested for by using their ASCII values which are



The test can be done in one of two ways:

```
IF ASC(241)=A$ THEN
```

or

```
IF A$=CHR$(241) THEN
```

The latter has been chosen but if you prefer to use the other then you may.

PROGRAM 3.5(a)

```
80 IF A$=CHR$(240) THEN Y=Y+1
90 IF A$=CHR$(241) THEN Y=Y-1
100 IF A$=CHR$(242) THEN X=X-2
110 IF A$=CHR$(243) THEN X=X+2
220 PLOT X,Y
230 GOTO 70
```

Once these lines are added the program should draw in all four directions. Experiment to see if you can draw rectangles using it.

There is still one flaw however: the program won't stop unless you press ESC twice. Therefore a further INKEY\$ test is added. This time if the test is positive, the command STOP is given.

PROGRAM 3.5(b)

```
120 IF A$="S" THEN STOP
```

This program certainly works now, but there are still 'bugs'. For example, it is still possible to draw over the edge of the screen. The Amstrad will allow you to keep plotting points off the screen, but this is not much fun as you cannot see anything. One way to avoid this is to use a further series of tests, as shown below in Program 3.5(c):

PROGRAM 3.5(c)

```
130 IF X>640 THEN X=640
140 IF X<0 THEN X=0
150 IF Y>400 THEN Y=400
160 IF Y<0 THEN Y=0
```

If we include these lines then it becomes impossible to plot points which are off the screen as, for instance, when X exceeds 640, line 100 resets it to this maximum value. Lines 110, 120 and 130 do a similar service for the minimum value of X and the maximum and minimum values of Y respectively. The complete program is listed below for your convenience. Lines 20 to 60 have been added to allow the user to choose the position at which to start drawing:

PROGRAM 3.5(d)

```
10 MODE 1
20 LOCATE 1,1
40 INPUT"X START";X
50 INPUT"Y START";Y
60 PLOT X,Y
70 A$=INKEY$
80 IF A$=CHR$(240) THEN Y=Y+1
90 IF A$=CHR$(241) THEN Y=Y-1
100 IF A$=CHR$(242) THEN X=X-2
110 IF A$=CHR$(243) THEN X=X+2
120 IF A$="S" THEN STOP
130 IF X>640 THEN X=640
140 IF X<0 THEN X=0
150 IF Y>400 THEN Y=400
160 IF Y<0 THEN Y=0
220 PLOT X,Y
230 GOTO 70
```

EXERCISE 3.2

Add an extra line so that if the 'R' key is pressed, the program allows drawing to begin again at a new start position. A possible answer is given in the solutions chapter.

PART THREE

DRAW

Not only does PLOT put a pixel onto the screen but it also moves the graphics cursor to the plot coordinates. The graphics cursor is used to keep track of the last point that was plotted by the computer. These coordinates are the start values for the DRAW statement.

PROGRAM 3.6

```
10 MODE 1
20 PLOT 50,25
```

This program will put the computer into MODE 1 and then plot a pixel at coordinates 50,25. You will be able to see this pixel if you look closely enough, it is in the bottom left hand part of the screen. This is where the DRAW statement will start to draw from.

PROGRAM 3.6(a)

```
30 DRAW 100,100
```

This will cause the Amstrad to draw a line from point 50,25 (as specified in the PLOT statement) to point 100,100. The DRAW statement allows us to draw lines starting from the coordinate chosen in the PLOT statement and ending at the coordinates specified in the DRAW statement itself.

The colour of the line drawn defaults to the value of INK 1. This can be changed by adding the ink number that contains the required colour to the DRAW statement.

PROGRAM 3.6(b)

```
30 DRAW 100,100,3
```

Looking at Figure 3.4 on page 3.4 we see that INK 3 has the colour value of red whilst in MODE 1.

EXERCISE 3.3

Draw a sea green line from the centre of the screen to location 64,93 in MODE 0. The answer is in the solutions chapter.

The DRAW statement itself moves the graphics cursor, so the next DRAW statement will continue where the last one left off.

PROGRAM 3.6(c)

```
10 MODE 1
20 PLOT 50,25
30 DRAW 100,100
40 DRAW 200,25
```

Line 20 positions the graphics cursor at coordinates 50,25, i.e. fifty points across and twenty five up. The DRAW statement on line 30 begins at this point and draws a line to the one hundredth point across and the one hundredth point up. This point is used by the next DRAW statement, line 40, as the starting point and a line is drawn from coordinates 100,100 to 200,25.

DRAWR

This variation of the DRAW command allows you to **DRAW** a line to a point **Relative** to the current position of the graphics cursor. For example,

```
PLOT 300,100: DRAW 50,50
```

This will draw a line starting at 300,100 and ending at point 50,50. DRAWR, however, will do something quite different.

```
PLOT 300,100: DRAWR 50,50
```

This will draw a line from point 300,100 to location 350,150, a position fifty points up and across from the start location.

Using PLOT and DRAWR statements it is possible to write a small program which allows the user to draw a variety of boxes onto the screen. Program 3.7 draws a square with sides of 50 points.

PROGRAM 3.7

```
10 MODE 1
20 PLOT 320,200
30 DRAWR 50,0
40 DRAWR 0,-50
50 DRAWR -50,0
60 DRAWR 0,50
```

Line thirty plots a point at the centre of the screen. Then, using DRAWR, line 30 draws a line fifty points to the right. This is the top line of our square. The next stage is to draw the right side (line 40). Having an X coordinate value of zero in the DRAWR statement means the line will be drawn using the same X coordinate. Having a minus value as the Y coordinate causes the computer to draw the line down the screen. The program up to line 40 produces the following picture.

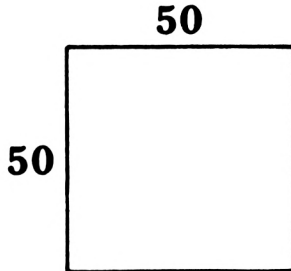


FIGURE 3.7

Lines 50 and 60 complete the square by first drawing the bottom line and then the left hand side.

This program, wonderful though it is, is boring. It always draws the same box. Really we would like to produce lots of different boxes, some of them being squares and some rectangles. To do this the plot coordinates will have to be variables. At the start of the program the user can be asked to input the coordinates for both X and Y. Add lines 2 and 4 of Program 3.7(a) to Program 3.7:

PROGRAM 3.7(a)

```
2 INPUT"TOP LEFT X:";X
4 INPUT"TOP LEFT Y:";Y
```

Now line twenty needs to adjusted accordingly.

PROGRAM 3.7(b)

```
20 PLOT X,Y
```

The program will now draw the same box anywhere on the screen. To make the program really general we would need to input the height and width values of a box. These modifications are included in the program by adding to it Program 3.7(c).

PROGRAM 3.7(c)

```
6 INPUT"HEIGHT OF BOX:";H
8 INPUT"WIDTH OF BOX :";W

30 DRAWR W,0
40 DRAWR 0,-H
50 DRAWR -W,0
60 DRAWR 0,H
```

The box-drawing routine is now 'universal' and we can draw a rectangular box anywhere on the screen.

EXERCISE 3.4

Add lines 3 and 5 to Program 3.7(a-c) so that the start locations for the box are tested to see if they are within the the allowed range. If they are not then the program should be looped back to recieve another input. A possible answer is given in the solutions chapter.

PART FOUR

Circles

The Amstrad does not have a circle command but it is possible to draw a circle using PLOT. If some way could be found of calculating the plot positions of a circle, then a circle could be 'plotted' onto the screen. This task is not quite as hopeless as it sounds for help is at hand.

SIN() and COS()

For a circle, the mathematical functions we need are our old friends from school, SIN and COS. These two useful functions will be used to the full when we draw a circle. Let's first see what they are and how they can help in this task.

Figure 3.8 shows the X and Y axes (horizontal and vertical directions) with a line 'CB' one unit long drawn from the centre 'C' of a circle, at an angle 'a' from the X axis.

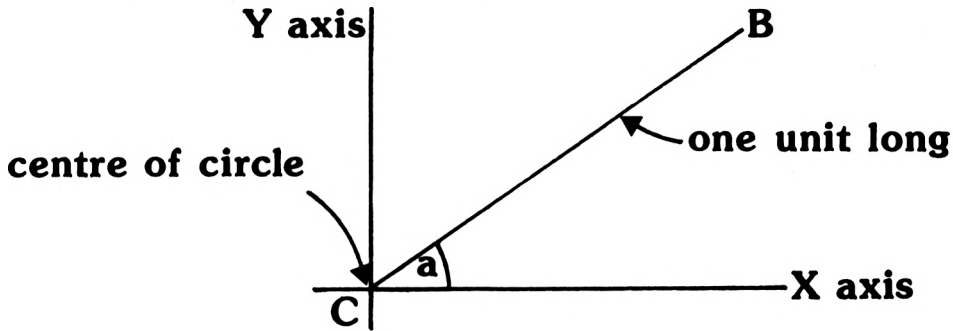


FIGURE 3.8

If the line CB were rotated about the centre C the end B would trace out a circle:

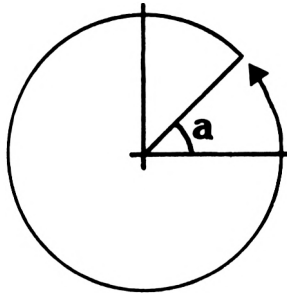


FIGURE 3.9

In order to use these ideas to draw a circle on the computer we will need to know the 'X' and 'Y' coordinates of the end point of CB. This, as you may have guessed, is where SIN and COS come in.

For a particular angle 'A' the 'X' and 'Y' coordinates are:

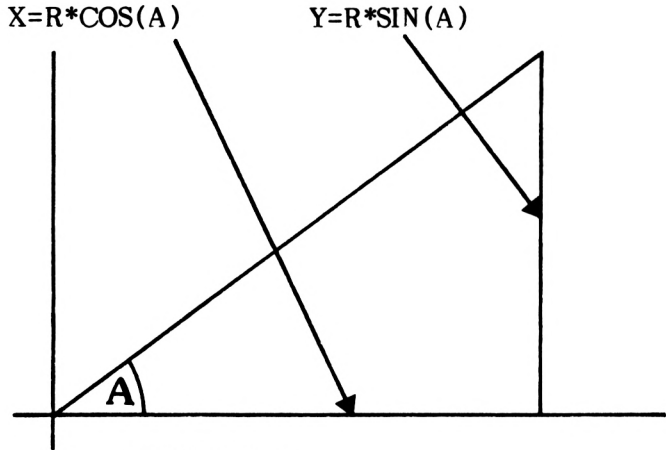


FIGURE 3.10

This value 'A' would be in radians, there being 2π radians in 360 degrees (a circle). Pi (pronounced 'pie' as in 'Apple pie') is a number which is the ratio of the radius of a circle to its circumference - the circumference of a circle is $2\pi R$ (R =radius). The actual value of pi is 3.14159... but you don't have to worry about it because the Amstrad has a special variable called PI. This has the value of 3.14159265. Check this by typing:

```
PRINT PI
```

Your Amstrad works in radians because the mathematical formulae that the computer uses to calculate COS and SIN work naturally in radians.

If we let 'A' be all the angles in a circle, i.e. 0 to 2π radians, then SIN(A) and COS(A) give all the 'X' and 'Y' coordinates of the points around the circle. Well that's all very well for a circle with a radius of '1' but what about a circle with a radius of '100'? If the radius is 100 times longer, then everything is 100 times bigger, so the coordinates of 'X' and 'Y' become:

$$X=100*\text{COS}(A) \quad Y=100*\text{SIN}(A)$$

For our circle we will use a radius value of 100.

To draw a circle we need to plot every point on the circumference and to do this a FOR..NEXT loop will be needed. It will loop from 0 to 2π radians, or if you prefer 0 to 360 degrees (a complete circle). The step size used allows the computer to draw a complete circle, a larger step size will produce a dotted circle.

PROGRAM 3.8

```
10 MODE 1
20 R=100
30 FOR A=0 TO 2*PI STEP 0.01
40 X=R*COS(A):Y=R*SIN(A)
50 PLOT 320+X,200+Y
60 NEXT A
```

How's that? A wonderful circle created with SIN and COS. There is however one problem: the circle takes a long time to be plotted, about twenty seconds. Do not fret, it is possible to produce a circle in a much shorter time. This can be done by one of two methods.

First, increase the STEP size. Care must be taken when doing this because too great a step value will produce a dotted circle. A quite reasonable circle can be plotted using a step value of 0.02. This will cut the circle plotting time in half, but that is still quite slow.

The second method is to replace PLOT with DRAW. This causes the program to draw a line to each new coordinate. Because DRAW produces a complete line there will not be any gaps in the circle. This allows us to increase the value quite significantly. Once again, if the STEP size is increased too far problems arise and the circle will appear more like a polygon. A suitable STEP size is that of 0.1, ten times the step size used in line 30 of Program 3.8. These modifications are incorporated in Program 3.8(a). Notice line 25; this moves the graphics cursor to the start position of the circle.

PROGRAM 3.8(a)

```
10 MODE 1
20 R=100
25 PLOT 420,200
30 FOR A=0 TO 2*PI STEP 0.1
40 X=R*COS(A):Y=R*SIN(A)
50 DRAW 320+X,200+Y
60 NEXT A
```

Having typed in Program 3.8(a) and RUN it you will be aware that we have a slight problem. The Amstrad has not drawn a complete circle. You may feel that the computer has done something wrong but computers do what you TELL them to do, not what you MEAN to tell them.

To understand this problem we must refer back to the earlier lessons about FOR...NEXT loops. In chapter 2 we learnt that the counter increases until it is equal to or greater than the count limit. In the case of Program 3.8(a) the counter A will have the values 5.9, 6.0, 6.1 and 6.2. The counter is then incremented to 6.3 but as this value is greater than the limit ($2*PI=6.28318530$), so the FOR...NEXT loop ends without a further line being drawn. This means that a small value ($6.28318530-6.2$) is left undealt with. This error of 0.08... is nearly equal to the STEP size of 0.1 and so the gap appears.

In general, to deal with this type of problem we must ensure that the step size divides exactly into the difference between the counter start and limit values. In this case a simple solution is to approximate the value of $2*PI$ as 6.3 in line 30. The step size of 0.1 then divides exactly into 6.3 and a complete circle is drawn.

EXERCISE 3.5

Draw a semi-circle using SIN and COS. The centre should be at location 120,200 and the circle radius 57. A possible answer is given in the solutions chapter.

CHAPTER 4

Strings and Structure: an Anagram Game

In chapter 2, a number guessing game was developed which utilised a random number generated by the Amstrad. In this chapter a similar game will be written, but this time using words, an anagram game. Thus, instead of asking the player to guess a number, the player will be required to guess a word. First of all though, this chapter will investigate ways of storing these words and then of delivering them one by one when required. This will be done using structured programming techniques to demonstrate how a complex program can be divided into simply understood sub-sections.

Each separate part of the Anagram game will be developed as a module. A module is a section of program that performs one major operation. By way of example, the Anagram game developed in this chapter has one module to choose a random word and another module to jumble up the letters.

When dealing with random numbers, the Amstrad can generate an endless supply to order. It has built into its ROM (Read Only Memory) a program which produces these as rapidly as they can be consumed. Of course, when dealing with words, the same thing is not possible. Computers don't know anything about words and so all the words to be used must be stored in the program somewhere, and thus must be defined by the programmer. A common way of storing such data is in strings, and the program could contain such statements as:

```
LET A$="AMSTRAD"  
LET B$="KEYBOARD"  
LET C$="SCREEN"
```

READ...DATA

This, however, would be an extremely tedious way of doing the job and BASIC provides an alternative method. It utilises two commands, READ and DATA, the first one telling the machine to READ one piece of data and the second telling it where to find the data. The DATA statement is the one piece of program that is footloose - it can go anywhere in the program. It is, however, usual to put it right at the end so that it is out of the way. Program 4.1 illustrates this, with line 100 reading one piece of DATA, which is PRINTed out in line 110, the DATA originally having been entered at line 9000.

PROGRAM 4.1

```
100 READ A$
110 PRINT A$
9000 DATA ONE, TWO, THREE
```

When run, this program will retrieve only one piece of DATA, e.g. 'ONE', and then PRINT this onto the screen. String data does not have to be enclosed in quotes in a DATA statement. If the string contains a comma, as in JONES,ED then it is wise to enclose it in quotes. For example, try 9000 DATA "JONES,ED",ONE,TWO in Program 4.1.

Program 4.2 shows a very similar program where numbers are stored rather than words.

PROGRAM 4.2

```
100 READ A
110 PRINT A
9000 DATA 1,2,3
```

In this program, the changes from Program 4.1 are really only what one would expect: the numeric variable name 'A' replaces the 'A\$'. Under no circumstances is numerical data enclosed in quotes.

READ statements may be as simple or as elaborate as the program demands and a number of variables could be READ in by one line of program; e.g. READ A\$,A,B\$. However, when doing this the greatest care must be taken to ensure that when the READ statement tries to READ a number it finds a number and not a string. If a mismatch in types occurs, the machine will report a 'syntax error' in the data line where the unexpected string data is found. As the READ statement is straightforward, try this little exercise!

EXERCISE 4.1

Write a program to READ the numbers 1 to 4 from DATA statements in both numbers and words, and to display them on the screen like so:

```
1 ONE
2 TWO
3 THREE
4 FOUR
```

A possible answer is given in the solutions chapter.

Having found a way of storing and retrieving data, some way has to be found of making a random selection from it. By using a FOR...NEXT loop to READ a particular number of items from the DATA, one particular piece may be retrieved, as shown in Program 4.3. In this program, the loop is executed three times, and thus the third piece of data is retrieved.

PROGRAM 4.3

```
120 FOR X=1 TO 3
130 READ A$
140 NEXT X
150 PRINT A$
9000 DATA ONE,TWO,THREE
```

In fact, three pieces of DATA will have been retrieved but only the string 'THREE' is printed. On the first pass through the loop, the value of A\$ would have been 'ONE' but during the second pass this would be overwritten by 'TWO' and then finally by 'THREE'. It was this string 'THREE' that was stored in A\$ at the time that line 150 displayed the string on the screen.

On each pass through the loop, line 130 READs the next item in the DATA statement. It knows which item is next as, each time a READ is performed, a pointer is moved along one item to point to the next one to be read. This can give rise to problems if an attempt is made to read more DATA than exists. For instance, if Program 4.3 is made to READ through the DATA more than once, then there won't be enough data and the computer will tell you. Try this out by adding Program 4.3(a) into Program 4.3.

PROGRAM 4.3(a)

160 GOTO 120

When Program 4.3(a) is RUN, the computer will report back:

DATA exhausted in 130

It is simply telling you that it has READ all the DATA and is looking for more but can't find any! To keep track of where it is in the data, the Amstrad's BASIC uses a 'pointer' and this it moves along to point to the next piece of DATA to be READ. For instance before line 130 of Program 4.3(a) is read the Amstrad's DATA structure is as in Figure 4.1.

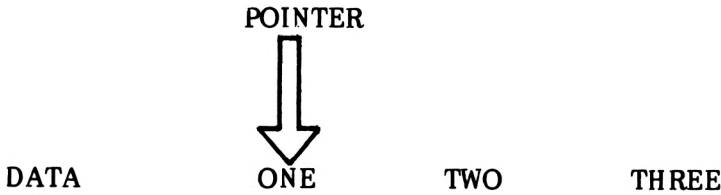


FIGURE 4.1

Thus in the first 'READ A\$', the string 'ONE' is assigned to A\$ and the pointer moved to point to 'TWO' as in Figure 4.2.

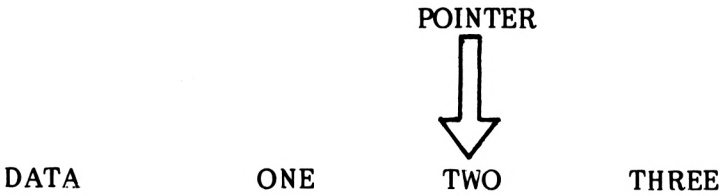


FIGURE 4.2

The next time around 'TWO' is stored in A\$ and the pointer moved again, Figure 4.3.

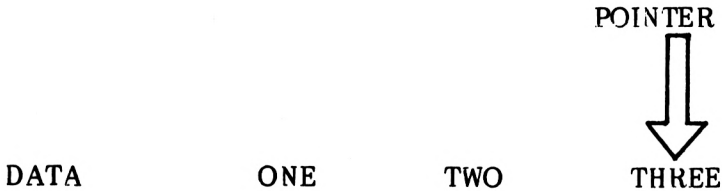


FIGURE 4.3

On the final pass through the loop, the string 'THREE' is stored in A\$ and, once again the pointer moved. Because there is no more data the pointer is moved to a position after three, pointing at nothing.



FIGURE 4.4

After this pass through the loop, the program passes the 'NEXT X' and line 150 PRINTS out the value of A\$ which is currently 'THREE'. Having done this, line 160 redirects the program to line 120 and the FOR...NEXT loop starts again.

When line 130 is encountered again the computer sees that the pointer is pointing at nothing thus indicating that there is no more data. It is at this stage that the computer gives the message.

DATA exhausted in 130

On many occasions, a program needs to use the same DATA over and over again and, for this to happen, the pointer needs to be moved back to the beginning of the DATA. This is brought about by means of the BASIC command:

RESTORE

This has the effect of moving the pointer back to the beginning of the DATA. It is demonstrated in Program 4.4 where a 'RESTORE' is performed before sending the program back to begin the loop again.

PROGRAM 4.4

```
120 FOR X=1 TO 3
130 READ A$
140 NEXT X
150 PRINT A$
160 RESTORE:GOTO 120
9000 DATA ONE, TWO, THREE
```

When Program 4.4 is run, it runs through the DATA endlessly. However, each time it runs through the loop it reaches the string 'THREE', this being defined by the loop value 3. For the Anagram game the word guessed needs to be chosen randomly. This will be done using a special form of variable called an ARRAY.

Arrays

An array is a series, or list, of related variables. They are related in that they have the same name. A street can be looked upon as a sort of array. The street has one name and all the houses have a number, Figure 4.5.

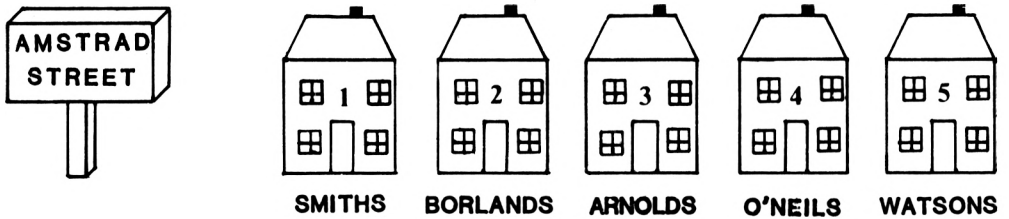


FIGURE 4.5

As the diagram shows, the Arnolds live at number three Amstrad Street. This could be written like so:

```
AMSTRAD STREET(3)=ARNOLDS
```

where 'Amstrad Street' is the array name and '3' is the array pointer and Arnolds is the value stored therein. If all of the words in the anagram game were to be stored under the collective name of WORD\$, the first could be WORD\$(1), the second WORD\$(2), etc. This is demonstrated in Program 4.4(a)

PROGRAM 4.4(a)

```
10 FOR X=1 TO 4
20 READ WORD$(X)
30 NEXT X
9000 DATA FLOWER, RAIN, AMSTRAD, COMPUTER
```

After running this program the array WORD\$ will contain the following values:

```
WORD$(1)="FLOWER"           WORD$(2)="RAIN"
WORD$(3)="AMSTRAD"          WORD$(4)="COMPUTER"
```

The random number can now be chosen from those in the array. Program 4.4(b) demonstrates this:

PROGRAM 4.4(b)

```
40 R=INT(RND*4)+1
50 A$=WORD$(R)
60 PRINT A$
```

Line 40 generates a random number which is then stored in R. Next the Rth value of the array WORD\$ is stored in A\$. This is then printed out. When we ran the program 'FLOWER' was displayed.

Using this method all of the words for the Anagram game can be stored in array WORD\$. Program 4.4(c) uses WORD\$ to store twenty different words.

PROGRAM 4.4(c)

```
10 FOR X=1 TO 20
20 READ WORD$(X)
30 NEXT X
40 R=INT(RND*20)+1
50 A$=WORD$(R)
60 PRINT A$

9000 DATA FLOWER, RAIN, AMSTRAD, COMPUTER
9010 DATA ADAPT, CREATE, IMAGINE, FRUIT, WALL
9020 DATA CONFUSION, STRANGE, BEAUTIFUL
9030 DATA PLASTIC, ELASTIC, BOMBASTIC, GRAND
9040 DATA YESTERDAY, NEWSPAPER, POT, PEANUT
```

What this program should do is read twenty words into the array WORD\$ and then randomly print out one of the words. What actually happens is another matter! The program reports back:

Subscript out of range in 20

This is the Amstrad's way of telling you that you are trying to assign too many values to an array. If the computer is not told how many values the array WORD\$ can hold then a value of 10 is assumed. Any attempt to read in, or print out, the 11th value of WORD\$ will result in a subscript out of range error.

The command that tells the computer how many array values we need is called:

DIM

The DIM command (DIM is an abbreviation of DIMension) tells the computer to allocate a specified number of locations to a certain variable; e.g.

```
DIM WORD$(20)
```

will reserve twenty locations for the array WORD\$. Now if we add Program 4.4(c) to Program 4.4(d) it will work properly.

PROGRAM 4.4(d)

```
5 DIM WORD$(20)
```

One quirk of DIM is that you cannot DIMension the same array more than once in a program. For example, if line 7 of Program 4.4(c) was added and the whole thing run the following error message would appear:

PROGRAM 4.4(e)

```
7 DIM WORD$(20)
```

```
Array already dimensioned in 7
```

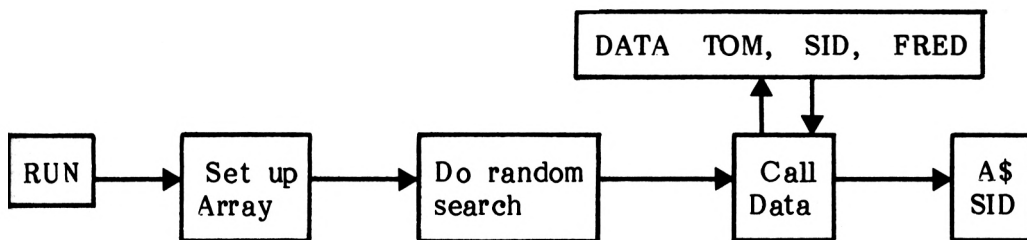
This error message tells the user, quite clearly, that the array dimensioned in line 7 has already been DIMed earlier in the program.

Arrays work in the same way with numeric variables. To demonstrate this type in this short program.

PROGRAM 4.4(f)

```
1 DIM T(50)
2 FOR X=1 TO 50:T(X)=X:NEXT X
3 FOR X=1 TO 50
4 PRINT T(X);:NEXT:STOP
```

Upon running this program the computer will display the numbers one to fifty which are the contents of the variables T(1) to T(50). Before continuing with the Anagram game delete lines 1-4.



Module 1

FIGURE 4.6

Figure 4.6 illustrates diagrammatically how this particular module works - run it and you'll find a randomly chosen item of the DATA in A\$.

As we will use this piece of program later on let's store it away out of harm's way by renumbering it from line number 1000 onwards.

RENUM

The Amstrad BASIC command RENUM is used to RENUMber lines. Renum has three arguments. The first is where you want the new line numbers to begin at. In this example it will be line 1000. The second value is the line number that you want to renumber from, in this example line 120. The third and final argument is the step size that you wish the lines to increase by. We will use 10.

RENUM	new numbers to start at line	,	begin to renumber at line	,	increase line numbers in steps of
-------	------------------------------------	---	---------------------------------	---	---

In our example the RENUM command will look like this:

```
RENUM 1000,5,10
```

All three values in the RENUM command are optional. If left out they will be replaced by 10. For example, typing in RENUM with no arguments will cause the whole program to be numbered starting at line 10 and increasing in steps of ten. Before doing this, delete line 60.

PROGRAM 4.5

```
1000 DIM WORD$(20)
1010 FOR X=1 TO 20
1020 READ WORD$(X)
1030 NEXT X
1040 R=INT(RND*20)+1
1050 A$=WORD$(R):AA$=A$
1060 DATA FLOWER, RAIN, AMSTRAD, COMPUTER
1070 DATA ADAPT, CREATE, IMAGINE, FRUIT, WALL
1080 DATA CONFUSION, STRANGE, BEAUTIFUL
1090 DATA PLASTIC, ELASTIC, BOMBASTIC, GRAND
1100 DATA YESTERDAY, NEWSPAPER, POT, PEANUT
```

This particular section of program needs to be split into two sections. The first section sets up the array and this will only be called up once at the beginning of the program. The second section is that which randomly chooses the word. Another Renum command can be used to separate the two sections. Type in:

```
RENUM 1100,1040,10
```

The program should now look like this:

```
1000 DIM WORD$(20)
1010 FOR X=1 TO 20
1020 READ WORD$(20)
1030 NEXT X

1100 R=INT(RND*20)+1
1110 A$=WORD$(R):AA$=A$
1120 DATA FLOWER, RAIN, AMSTRAD, COMPUTER
1130 DATA ADAPT, CREATE, IMAGINE, FRUIT, WELL
1140 DATA CONFUSION, STRANGE, BEAUTIFUL
1150 DATA PLASTIC, ELASTIC, BOMBASTIC, GRAND
1160 DATA YESTERDAY, NEWSPAPER, POT, PEANUT
```

Line 1110 has been altered to create a second copy of the randomly chosen word. The first copy, stored in array A\$, will be used to create the Anagram version.

Slicing Up Words

In the anagram game that we are developing, it will be necessary to 'slice up' the words to identify their individual letters. To do this we utilise the BASIC functions:

LEFT\$, RIGHT\$ and MID\$

Amstrad BASIC provides several ways to chop up strings. Two of these are LEFT\$ and RIGHT\$. These simply lop off the left and right ends of strings respectively. Let's try chopping up a few strings for practice! Firstly we'll set A\$ to "COMPUTER". To use the jargon, we will 'set the value of the variable A\$ to the literal value "COMPUTER"'. The first practice will be with 'LEFT\$'. LEFT\$ gives the specified number of left-most characters of a string and takes the form:

LEFT\$(X\$,N)

where 'X\$' is the string we wish to chop up and 'N' is the number of characters we wish to remove.

In our example the string is A\$, which is set to 'COMPUTER'. If we wished to remove the first 4 characters i.e. 'COMP' we would use the following method:

```
A$="COMPUTER"  
B$=LEFT$(A$,4)  
PRINT B$
```

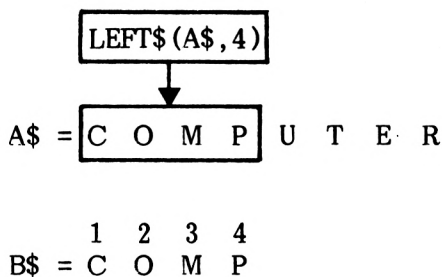


FIGURE 4.7

The command RIGHT\$() works in exactly the same way, except that it starts counting from the RIGHT-hand side of the string. RIGHT\$ takes the following form:

RIGHT\$(X\$,N)

where X\$ is the string, and N is the number of characters to be removed.

Thus, if A\$="COMPUTER", C\$=RIGHT\$(A\$,5) will set the string C\$ to "PUTER", the 5 right-most characters of "COMPUTER". Try it out to make sure!

In contrast, the MID\$ function can start anywhere in the string. It allows the programmer to chop away selectively small or large bits of the string being worked upon. MID\$ takes the form:

MID\$(X\$,S,N)

where X\$ is the string to be 'cut-up', S is the character from which the operation should start and N is the number of characters to be removed. It can cut away from the middle or either end of the string. Let's examine that in more detail, using the example

A\$ = "COMPUTER"
C\$ = MID\$(A\$,4,3)

- When the computer sees C\$=MID\$(...) it knows that a string is to be dissected and the result stored in the string C\$.
- It then carries on and sees MID\$(A\$...). It translates this into 'first find A\$ and get prepared to operate on it'.
- Next it sees the '4' in MID\$(A\$...) and this tells it to start at the fourth character of the string.
- Then it reads the '3' in MID\$(A\$,4,3) and, starting at the fourth character of the string, it strips off three characters. These it stores in C\$.

Thus, following the operation, C\$ would contain "PUT".

In general terms, the structure of the command is:

MID\$(A\$,START,LENGTH)

'START' and 'LENGTH' must both be whole numbers. MID\$ will cut out a part of the string A\$ starting at character number 'START' and of length 'LENGTH' characters. Diagrammatically, MID\$ appears as:

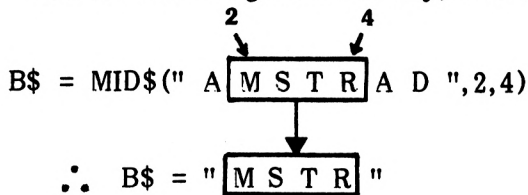


FIGURE 4.8

Using these string handling commands enables you to slice up a word on demand, taking individual letters as required. Thus, each time a letter is required from anywhere within a word it can be obtained by means of MID\$. Program 4.6 shows this in action as MID\$ reads step by step through a word, printing each letter as it goes.

PROGRAM 4.6

```
1500 LET A$="COMPUTER"  
1510 FOR X=1 TO 8  
1520 B$=MID$(A$,X,1)  
1530 PRINT B$  
1540 NEXT X
```

One of the problems that Program 4.6 would present, were we to try and use this as a module in a program, is that it works when the word has eight letters but for words with more or fewer letters, it would prove to be problematical. But, you've guessed it, BASIC has a fix for this with the function:

LEN()

This command is used to tell how many characters are in a particular string - in the jargon, LEN() 'returns' the length of a string. Test it out with:

```
PRINT LEN("COMPUTER")
```

and then with a few other strings; each time it should print out the length of the string involved. Program 4.6 can now be re-written so that the loop will run through the correct number of times, whatever the string length, as in Program 4.7.

PROGRAM 4.7

```
1500 LET A$="COMPUTER"  
1510 FOR X=1 TO LEN(A$)  
1520 B$=MID$(A$,X,1)  
1530 PRINT B$  
1540 NEXT X
```

Whatever the string that is assigned to A\$, the loop will now always handle it. However, the aim of this project is an anagram guessing game so the module that we are developing should rearrange the letters and not leave them in the same order! It is line 1520 that dissects the string and this it does in an orderly way, starting at 1 and progressing through to LEN(A\$). One way to make it less orderly would be to replace the variable X in line 1520 with a random number which lies between 1 and LEN(A\$). This can be done by setting a variable, say R, to the appropriate random number and replacing line 1520 by:

```
B$=MID$(A$,R,1)
```

And the random number - remember the drill from chapter 2 where we found we needed to add 1 to produce the correct range -

```
R=INT(LEN(A$)*RND)+1
```

This can be added in Program 4.7:

PROGRAM 4.7

```
1515 R=INT(LEN(A$)*RND)+1
1520 B$=MID$(A$,R,1)
```

So, when lines 1515 and 1520 are added to Program 4.7 it yields Program 4.8, below:

PROGRAM 4.8

```
1500 LET A$="COMPUTER"
1510 FOR X=1 TO LEN(A$)
1515 R=INT(LEN(A$)*RND)+1
1520 B$=MID$(A$,R,1)
1530 PRINT B$
1540 NEXT X
```

Now, when this is run it will print out the letters of A\$ in a random way. Just one problem though! If you look at the letters printed its almost certain that one or more has been repeated. This is because, once a letter has been chosen at random, it remains in A\$ to be chosen again! What we need to do is to remove each letter once it is guessed, a fairly tricky operation! However, it can be done without too much difficulty, thanks to the commands LEFT\$ and RIGHT\$.

Say for instance that the random letter chosen was the 'P' in 'COMPUTER', i.e.

B\$
↓
A\$ = COMPUTER

In order to 'remove' B\$ we must take the section of string to the left of P and that to the right of P and add them together, calling this the new A\$.

As B\$ is the letter defined by the random value R, i.e. the R'th letter (in this case the fourth), there are R-1 letters to the left of it (in this case 3). The string to the right of B\$ is slightly more complex as it consists of the whole word minus R characters, i.e. LEN(A\$)-R characters. In this case 8-4 = 4 characters. The new value of A\$ is then made up from the part to the left of the 'R' plus the part to the right i.e.

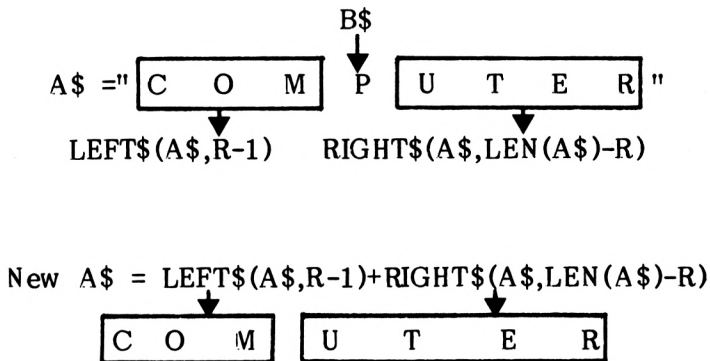


FIGURE 4.9

To form the new A\$ the left and right parts can be added together just as if they were numbers! When this is done to strings it is given the fancy term 'concatenation'! To try this out type in the direct line:

```
A$="FRED":B$="DY":C$=A$+B$:PRINT C$
```

This will print out 'FREDDY'.

Using this device, Program 4.8 can be modified so as to take out one letter at a time, close up the remainder of A\$ and then to add the letter removed to the new string. The result of this concatenation - or adding together - will be stored in a string called AN\$ (ANSwer) which will be gradually increased, one letter at a time, until it contains all the letters. However, to start off, this string must be emptied or set to an empty string, i.e: AN\$="". If you're not too sure of this process carry out the following little exercise. Type in Program 4.9

PROGRAM 4.9

```
1 A$="FRED":AN$="AND"  
2 PRINT A$;AN$;A$  
3 STOP
```

When you run this you should get a display of

FREDANDFRED

Now modify Program 4.9 as shown below in Program 4.9(a) in order to set AN\$ to an empty string. (Also known as a null string).

PROGRAM 4.9(a)

```
1 A$="FRED":AN$=""
```

When this is RUN the display should show:

FREDFRED

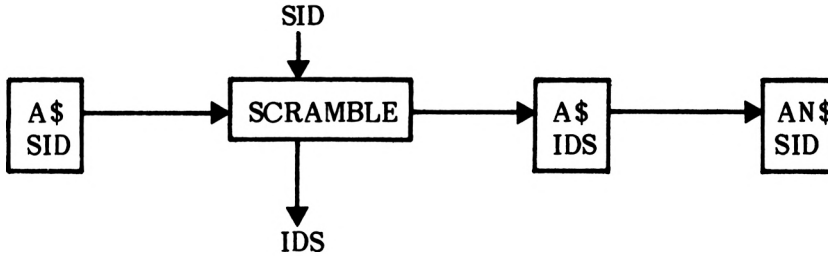
In other words, the string AN\$ is now empty. Having seen that, delete lines 1-3. Applying that idea to Program 4.8 yields Program 4.10:

PROGRAM 4.10

```
1500 LET A$="COMPUTER":AN$=""  
1510 FOR X=1 TO LEN(A$)  
1520 R=INT(LEN(A$)*RND)+1  
1530 B$=MID$(A$,R,1)  
1540 AN$=AN$+B$:PRINT AN$  
1550 A$=LEFT$(A$,R-1)+RIGHT$(A$,LEN(A$)-R)  
1560 NEXT X
```

When run, Program 4.10 will print out the anagram, stage by stage as it is built up. Also, as A\$ becomes progressively smaller, the random number that selects the letter (R) becomes progressively smaller too - very convenient eh?

Putting this into a diagram and calling the module number 2 gives Figure 4.10.



Module 2

FIGURE 4.10

Program 4.11 shows the module with the intermediate 'PRINTS' removed.

PROGRAM 4.11 - Module 2(Part)

```
1500 AN$=""
1510 FOR X=1 TO LEN(A$)
1520 R=INT(LEN(A$)*RND)+1
1530 B$=MID$(A$,R,1)
1540 AN$=AN$+B$
1550 A$=LEFT$(A$,R-1)+RIGHT$(A$,LEN(A$)-R)
1560 NEXT X
```

Now that two modules have been written, what is needed is a way of calling these up, as and when required. This is achieved by treating the modules as subroutines and 'calling' them when required. The 'calling' part is done by the BASIC command 'GOSUB' which, in effect means 'go to the subroutine starting at the given line number'. Test this in direct mode by typing in the direct command:

```
GOSUB 1000
```

What happened? Then computer should have executed module 1 and chosen a word from the DATA list. However, this did not happen. The Amstrad has displayed the following message:

```
Array already dimensioned in 1000
```

The Array was already dimensioned when we ran the program previously. The RUN command tells the computer to forget the value of all variables, including arrays, before starting to execute the program. This time however, the program was not RUN but executed by a

```
'GOSUB 1000'
```

This does not 'initialise' any variables and so the computer tells you that the array WORD\$ has already been dimensioned.

To get around this problem simply type in:

```
'GOSUB 1100'
```

this will execute the program with the exception of line 1000. Once you have typed in the above line you will see the ready message appear. You think that the Amstrad has run Module one and finished, but it will have done more than that; once it has run through module 1 it will have run directly into module 2 to carry out the randomization of AN\$. The running of module 2 was unplanned and, therefore, out of control. To regain control the computer needs to be told when the subroutine is ended by means of the construction:

GOSUB...RETURN

This construction directs a program to a subroutine by means of GOSUB and sends it back once the RETURN is encountered. The program returns to the statement immediately after the GOSUB call.

To accommodate the changes required, modify the two modules by adding the lines

```
1040 RETURN  
1170 RETURN  
1600 RETURN
```

The two modules can now be used in a really structured way, by being called from a program control module starting at 500, i.e:

PROGRAM 4.12

```
460 GOSUB 1000:REM SET UP THE ARRAY  
510 GOSUB 1100:REM PICK A WORD  
520 GOSUB 1500:REM SCRAMBLE IT  
640 END
```

Once the word has been scrambled and the anagram produced it can be printed onto the screen. This is done in line 530 of Program 4.12(a).

PROGRAM 4.12(a)

```
530 LOCATE 1,6:PRINT"THE ANAGRAM IS ";AN$
```

Module 3: Inputting the guess

This module should, apparently, present no great problems as it seems that a straight-forward 'INPUT' command could cope. However, it is necessary during the inputting of data to check whether detectable errors have been made. It's much easier to do this when the characters are accessible individually than when they are all recieved with a single 'INPUT' command. For this reason an input routine will be created using the INKEY\$ command. By this means, each character can be checked as it is typed in and the end of the input sequence will be detected when the user presses the RETURN key. The problem is how to test for a press of the ENTER key. Unlike most of the keys on the Amstrad keyboard the ENTER key does not place a character on the screen when pressed. To test for a press of the ENTER key we need to use a new command.

CHR\$

Every key on the Amstrad keyboard has a special character number which the computer uses to identify it. This number is referred to as its ASCII value. ASCII is an acronym from American Standard Code for Information Interchange. One of the commands used in association with the ASCII values is CHR\$. To demonstrate the CHR\$ command type in the following:

```
PRINT CHR$(47)
```

The Amstrad will display '!'. The CHR\$ command has told the computer to display on the screen the character with a ASCII value of forty seven. As mentioned above every key has its own ASCII value, even the ENTER key.

```
PRINT CHR$(13)
```

Thirteen is the CHR\$ value for the ENTER key. The Amstrad cannot display 'ENTER' without being told to PRINT"ENTER", which is NOT the same thing so it has done the next best thing - it has actually performed an ENTER. This makes the cursor appear two lines lower on the screen.

To test if ENTER has been pressed a line 2020 in Program 4.13 can be used.

PROGRAM 4.13

```
2010 Z$=INKEY$:IF Z$="" THEN 2010      Input a character
2020 IF Z$=CHR$(13) THEN END          check for RETURN
2060 GOTO 2010                        look for next input
```

Program 4.13 will accept a string of inputted characters but does not store them. In order to do this, the individual characters must be added together to form the guess, say G\$. Don't forget, though, that each time a guess is inputted it must be built up from scratch by setting G\$ equal to a null string.

PROGRAM 4.14

```
2000 G$=""
2040 G$=G$+Z$
```

If all the words that are to be guessed are to contain only letters, then a check can be made to ensure that this is so. These all lie between 'a' and 'z' and therefore can be checked by saying: 'If Z\$ does not lie between a and z then go back for another input' i.e.

```
2030 IF Z$<"a" OR Z$>"z" THEN GOTO 2010
```

Line 2030 tells the computer to check the CHR\$ value of the input character with 'a' and 'z'. If the input character is less than the character value of 'a' or it is greater than the character value of 'z' then it is not a letter and the input is ignored. Because the Amstrad has different CHR\$ values for upper and lower case letters you must make sure that CAPS LOCK is off before you run this program.

Putting this together to form an input module - module 3:

PROGRAM 4.15

```
2000 G$=""
2010 Z$=INKEY$:IF Z$="" THEN 2010
2020 IF Z$=CHR$(13) THEN RETURN
2030 IF Z$<"a" OR Z$>"z" THEN 2010
2040 G$=G$+Z$
2050 LOCATE 16,8:PRINT G$
2060 GOTO 2010
```

Line 2050 displays what you are inputting, letter by letter. Take great care whilst typing in your guess because you cannot correct a mistake.

So that's the inputting process! However, it still needs hooking up by means of the program control module, i.e. as Program 4.16.

PROGRAM 4.16

```
460 GOSUB 1000:REM SET UP THE ARRAY
510 GOSUB 1100:REM PICK A WORD
520 GOSUB 1500:REM SCRAMBLE IT
530 LOCATE 1,6:PRINT"THE ANAGRAM IS ";AN$
535 LOCATE 1,8:PRINT"YOUR GUESS IS"
540 GOSUB 2000:REM INPUT A GUESS
```

Once the guess has been inputted, it needs to be compared with the original word and the appropriate message given. Let's develop this as Module 4.

Module 4: Testing the guess

A simple comparison will serve to test whether or not the word is correct, i.e.

```
IF G$=AA$ THEN guess is correct
```

Tests of this type can be used where tests are made for each condition and the appropriate message given immediately as in Program 4.17.

PROGRAM 4.17

```
2510 IF G$=AA$ THEN PRINT"WELL DONE GUES
S CORRECT"
2520 IF G$<>AA$ THEN PRINT"SORRY THAT'S
NOT CORRECT! TRY AGAIN"
```

However one of the problems with this direct technique is that it is restricted to producing a message immediately after the IF...THEN. Even more of a problem is where more instructions are needed following the test. For instance, when the guess is correct it would be desirable to ask the player whether another go is required.

This problem is overcome by having one routine to check the guess and another two routines to report on it. One routine for a correct guess and the other for an incorrect guess. In order to incorporate this, a method is needed to transfer information from one routine to another. This is done using FLAGS. A flag is a special variable that is used to indicate whether a condition has been met. If the condition is met then the flag is set to '1' otherwise it is set to '0'. Using flags in this manner it is quite simple to develop a routine to check the player's guesses.

PROGRAM 4.18

```
2510 IF G$=AA$ THEN F1=1
```

The variable AA\$ contains the word that the anagram was created from. If G\$ (the player's guess) is equal to this then the flag F1 is set to one.

As well as testing to see whether the player has guessed correctly, this routine can be used to check how many guesses the player has had. This program will allow a maximum of six goes. If the word has not been guessed by the sixth attempt then the player has lost. Another flag is used when checking the COUNT value: this is called F2. If all the guesses have been used up then F2 is set to '1' otherwise it is zero.

PROGRAM 4.18(a)

```
2520 IF COUNT=6 THEN F2=1
```

Having completed the two tests the routine needs to be RETURNed to the control program, line 2530. Line 2500 sets the initial value of both flags to zero. This initialisation prevents the flags from passing the wrong information during a second run of the program.

PROGRAM 4.19

```
2500 F1=0:F2=0
2510 IF G$=AA$ THEN F1=1
2520 IF COUNT=6 THEN F2=1
2530 RETURN
```

The next step is to act upon the value of the flags. If F1 is 1 then the guess was correct and the player needs to be told that they have won. The control module tests the guess; if it was correct then the routine at 2600 is called up. If the guess was wrong then the routine at 2800 is called.

PROGRAM 4.20

```
550 GOSUB 2500:REM CHECK GUESS
560 IF F1=1 THEN GOSUB 2600:GOTO 590:REM WIN
580 IF F1=0 THEN GOSUB 2800:GOTO 535
```

The routine at 2600 needs to tell the player that they have won and how many guesses it took. Lines 2610, 2620 and 2630 use a new command called SPACE\$. The SPACE\$ command is used to print spaces! The number of spaces required is indicated by the number in brackets. In Program 4.20(a) forty spaces are printed. By printing spaces in the appropriate places all the unwanted messages are removed from the screen prior to telling the player that they have won.

PROGRAM 4.20(a)

```
2600 REM YOU HAVE WON
2610 LOCATE 1,8:PRINT SPACE$(40)
2620 LOCATE 1,12:PRINT SPACE$(40)
2630 LOCATE 1,14:PRINT SPACE$(40)
2640 LOCATE 1,8
2650 PRINT"THAT IS CORRECT!"
2660 LOCATE 1,10
2670 PRINT"THAT TOOK YOU";COUNT;"ATTEMPTS"
2680 RETURN
```

If the guess was incorrect then F1=0 and the program is directed to the incorrect guess subroutine. Once this routine has been returned from, the program is redirected to line 530 for another input.

PROGRAM 4.20(c)

```

2800 REM INCORRECT GUESS
2810 LOCATE 1,10
2820 PRINT"I'M SORRY THAT IS WRONG"
2830 FOR X=1 TO 1000:NEXT X
2840 LOCATE 1,12
2850 PRINT"YOU HAVE HAD";COUNT;"GOES"
2860 LOCATE 1,14
2870 PRINT"YOU HAVE";6-COUNT;"TRIES LEFT"
2880 FOR X=1 TO 1000:NEXT X
2890 LOCATE 1,10:PRINT SPACE$(40)
2900 LOCATE 1,8:PRINT SPACE$(40)
2910 COUNT=COUNT+1
2920 RETURN

```

Lines 2890 and 2900 use SPACE\$ to remove the player's guess and the 'I'M SORRY THAT IS WRONG' message.

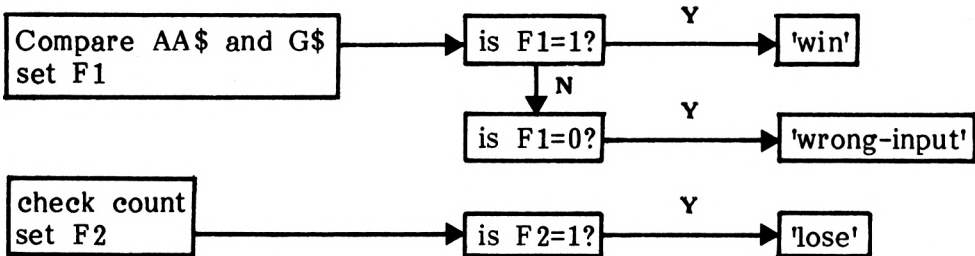
The final part of the fourth module checks to see whether all the goes have been used up (i.e. F2=1). If they have then the player needs to be told that they have lost and what the word was that they were trying frantically to guess.

PROGRAM 4.20(a)

```

570 IF F2=1 THEN GOSUB 3000:GOTO 590:REM LOST
3000 REM YOU HAVE LOST
3010 LOCATE 1,10:PRINT SPACE$(40)
3020 LOCATE 1,12:PRINT SPACE$(40)
3030 LOCATE 1,14:PRINT SPACE$(40)
3040 LOCATE 1,8
3050 PRINT"I'M SORRY YOU HAVE LOST"
3060 LOCATE 1,10
3070 PRINT"THE WORD WAS ";AA$
3090 RETURN

```



Module 4

FIGURE 4.11

The major part of the program is the control routine. So far it looks like this:

PROGRAM 4.21

```
460 GOSUB 1000:REM SET UP THE ARRAY
510 GOSUB 1100:REM PICK A WORD
520 GOSUB 1500:REM SCRAMBLE IT
530 LOCATE 1,6:PRINT"THE ANAGRAM IS ";AN$
535 LOCATE 1,8:PRINT"YOUR GUESS IS"
540 GOSUB 2000:REM INPUT A GUESS
550 GOSUB 2500:REM CHECK GUESS
560 IF F1=1 THEN GOSUB 2600:GOTO 590:REM WIN
570 IF F2=1 THEN GOSUB 3000:GOTO 590:REM LOSE
580 IF F1=0 THEN GOSUB 2800:GOTO 535
```

The lines 500 to 560 call up all the routines and test all the conditions. This is the core of the program. All that needs to be added is the rules and a test to see whether another go is required. Also the value of COUNT needs to be set to zero at the beginning of the program.

In structured programming the rules of the game should be included in their own little subroutine which the computer calls up before anything else. This is what Program 4.22 does.

PROGRAM 4.22

```
450 GOSUB 3500:REM THE RULES

3500 REM THE RULES
3510 MODE 1
3520 LOCATE 12,2
3530 PRINT"***ANAGRAM GAME!***"
3540 LOCATE 12,4
3550 PRINT"HERE ARE THE RULES"
3560 LOCATE 8,8
3570 PRINT"I WILL THINK OF A WORD THEN"
3580 LOCATE 8,10
3590 PRINT"I WILL JUMBLE UP THE LETTERS"
3600 LOCATE 7,12
3610 PRINT"YOU MUST TRY TO GUESS THE WORD"
3620 LOCATE 5,18
3630 PRINT"***PRESS THE SPACE BAR TO BEGIN***"
3640 IF INKEY$<>" " THEN 3640
3650 RETURN
```

Having displayed the rules we now need to set COUNT to its initial value of zero (line 470). Lines 480 and 490 display the game's title once again so that you know which game you are playing.

PROGRAM 4.23

```
470 CLS:COUNT=1
480 LOCATE 12,2
490 PRINT"**ANAGRAM GAME**"
```

The final part of this game is to ask the player whether another go is required. This is done only if they have guessed correctly or if all the guesses have been used up. Thus it is placed at lines 590 to 630.

PROGRAM 4.24

```
590 LOCATE 4,12
600 INPUT "DO YOU WANT ANOTHER GO (Y/N)";B$
610 IF B$="Y" THEN 470
620 CLS:LOCATE 14,12
630 PRINT"GOODBYE"
640 END
```

LOWER\$

At the moment the game works quite well but with one minor irritation, the user has to make sure that the computer is set to lower case before running the program. This is because line 2030 test each inputted character to see if its ASCII value is less than or greater than the ASCII value of lower-case 'a' and 'z' respectively. If the Amstrad is in capitals mode then the letter characters would be ignored (Amstrad has different ASCII codes for 'A' and 'a'). Help is at hand in the shape of the LOWER\$ command. This converts a string of characters into lower case. For example:

```
A$="SMALL LETTERS"

PRINT LOWER$(A$)
```

This will display 'small letters' in 'small letters'! If the string already contained lower-case letters then all very well, nothing would have been changed.

LOWER\$ can be incorporated into the program by adding line 2025. This converts each character input into lower case so now it doesn't matter if the computer is switched to upper or lower case.

PROGRAM 4.25

```
2025 Z$=LOWER$(Z$)
```

UPPER\$

This, if you haven't already guessed, converts strings into upper-case letters. For example:

```
A$="big letters"
```

```
PRINT UPPER$(A$)
```

This will display 'BIG LETTERS' in 'big letters'!

That completes this introduction to structured, 'modular' programming, leaving you, the user, with a completely structured program.

CHAPTER 5

Top- down programming : A Hangman Game

There are many different ways of structuring a program and each one has its adherents. Those who believe in one way tend to do so with an almost religious fervour - an unshakable belief. In this chapter, one particular approach will be followed but no claim is made for its total omnipotence. It's just one way among many! However, it is a technique that many now feel to be particularly valuable.

But first....

When writing any complex program, the author must commit certain things to paper right at the beginning. Certain questions have to be asked and these are:

- What does the program set out to achieve?
- How will it interact with the user i.e. what inputs will be needed from the user and what outputs will these produce?
- What strategy will be used to carry out the necessary processes?

In the case of the hangman game, these are not really terribly difficult questions and are probably much more pertinent when large complex systems are involved. However...

- The program will set out to achieve an interactive dialogue between the computer which generates a random word and the user who has to guess what it is.
- Interaction between the user and computer will be via the keyboard and the monitor's screen.
- The program can be divided into two major sections. The first section is the control routine (part of a program that calls up subroutines) and the second part of the program will be the various subroutines.
- Develop a random word to be guessed.
- Compare the guessed character with the word.

When a program is developed using 'top-down' programming, the general structure is defined first and then the program proper is written in progressively greater detail. Thus, decisions made early on influence later parts of the program. If this procedure is reversed and the detail done first, then many changes will be needed in the detailed parts of the program as the structure is defined. However, before proceeding to this structure, let's have a look in very broad terms at the program as a whole. What we will do is to write it out in a sort of 'pseudo code'; that means in a way that is a bit like English and a bit like a computer language.

What the program should do:

- Display a title page and rules. (Module 1)
- Choose a word randomly by reading through DATA statements: call the word A\$. Set up an array called WORD\$() to have a dot for each letter in A\$ i.e. at the beginning:

If A\$="COMPUTER"then WORD\$()="....."

When an 'M' is guessed, WORD\$() would become "..M.....", i.e. the 'M' would be placed in the correct position (Modules 2 and 8).

- Tell the player how many letters in the word to be guessed (Module 3)
- Input a guess from the player: call the inputted character GUESS\$ (Module 4)
- Check if the guessed character (GUESS\$) is in the word to be guessed (A\$). If it is then replace the appropriate dot in WORD\$() with the guessed letter (Modules 7 and 8)
- Store a list of all the characters that have been guessed so far. Call this string X\$ (Module 5)
- Check if the currently guessed character has been guessed before, i.e. is it in the string X\$? (Module 5)
- Tell user if character has been guessed before (Module 6)
- Display the current state of the guessed word (Module 9)
- Check to see if the word has been guessed. This is done by seeing if any dots are left in the string WORD\$. If no dots are left then all the letters have been guessed (Module 10)

- If the player has won then they are to be congratulated and asked if they require another go (Modules 11 and 12)
- When an incorrect guess has been made the incorrect guess variable 'E' needs to be incremented by one. (Module 13)
- Once an incorrect guess is made the next section of the hangman needs to be drawn. (Module 14)
- A check is made on the incorrect guess count. If E is equal to ten then the player has lost and the man is hung. (Modules 15 and 16)

Program structure

Firstly the program will be developed in a skeleton structure so that it runs in very broad terms but without all the detail. Once this is done the structure can be checked and the detail filled in only when the structure is correct.

This program is made up of 16 modules 'glued' together by a general Program Control Module - so now to work through each module one by one.

The various program modules will be located in the program as follows:

Module	lines
Program control	0-
Ancilliary functions	900-
Module 1	1000-
" 2	2000-
" 3	3000-
" 4	4000-
" 5	5000-
" 6	6000-
" 7	7000-
" 8	8000-
" 9	9000-
" 10	10000-
" 11	11000-
" 12	12000-
" 13	13000-
" 14	14000-
" 15	15000-
" 16	16000-
Data statements	17000-

FIGURE 5.1

These subroutines will be written as dummy routines first, in order to test the logical flow of the overall program. By doing this, all the different routes through the program can be investigated and any problems ironed out.

Module 1: Initialisation

Several pieces of housekeeping need to be done. When a program starts, general user information needs to be given, variables to be set etc. However, for now, module 1 will simply clear the screen and then wait for the user to press a key. As each of these dummy modules will display a screen and then wait for a character to be inputted, the 'INKEY\$' routine will be written just once at 900 and called by a 'GOSUB' whenever it is required, i.e. Program 5

PROGRAM 5

```
900 A$=INKEY$:IF A$="" THEN 900
910 RETURN
```

PROGRAM 5.1

```
1000 CLS
1010 LOCATE 10,10
1020 PRINT"INITIALISATION"
1030 GOSUB 900
1040 RETURN
```

The subroutine is called by means of the Program Control Module (PCM) shown in Program 5(a)

PROGRAM 5(a)

```
500 GOSUB 1000:REM INITIALISATION
```

When it is RUN, the screen will clear and then the word 'INITIALISATION' will appear on the screen. The program will then sit and wait until a key is pressed and then.....Well try it! Run the program and press the space bar twice; the screen should now be showing:

Unexpected RETURN in 910

Can you see what has happened and prevent it from happening again? The problem is that once the PCM had been RUN and returned from Module 1, it ran back into the 'A\$=INKEY\$' routine at line 900. The second press of the space bar brought the program to line 910 and came across the 'RETURN'. This time, however, the program had nowhere to RETURN to because the routine was not called by a GOSUB, therefore, an error was reported. The problem can be prevented by terminating the PCM neatly by means of an END i.e. as in Program 5(a1)

PROGRAM 5(a1)

```
500 GOSUB 1000: REM INITIALISATION
899 END
```

Now, when this is RUN, the word 'INITIALISATION' will appear once and upon pressing a key the program will stop.

So far so good! However, the program is not really very user-friendly as, while it waits for the input the user cannot be sure what is happening. It would be much clearer, were the user to be told to press a key. Thus, the input routine could be improved by incorporating a message. As the subroutine is to be used on many occasions it is better if its screen position is always the same. Using LOCATE it poses no problem to print the message in the same place everytime.

PROGRAM 5(a2)

```
900 LOCATE 4,20:PRINT"PRESS ANY KEY
TO CONTINUE"
910 A$=INKEY$:IF A$="" THEN 910:ELSE RETURN
```

Now, when the program is RUN 'INITIALISATION' is printed and then the 'PRESS ANY KEY' message is given.

From here, the program follows only one route, that to:

Module 2:Choose word.

In this element a word will be selected from those made available.

In skeleton from this is

PROGRAM 5.2

```
2000 CLS:LOCATE 10,10
2010 PRINT"CHOOSE WORD"
2020 GOSUB 900
2030 RETURN
```

The module is called by line 510 in the PCM, shown in Program 5(b)

PROGRAM 5(b)

```
510 GOSUB 2000:REM CHOOSE WORD
```

Module 3:Input Guess

This module will set up the screen format that will be used for the rest of the program. Skeletally speaking it looks like this:

PROGRAM 5.3

```
3000 CLS:LOCATE 10,10
3010 PRINT"DISPLAY SCREEN"
3020 GOSUB 900
3030 RETURN
```

The module is called by the PMC, this being shown in line 520 of Program 5(c).

PROGRAM 5(c)

```
520 GOSUB 3000: REM DISPLAY SCREEN
```

Module 4: Display Screen

At this point a guess is made by the player and this is checked to trap various errors.

Program 5.4

```
4000 CLS:LOCATE 10,10
4010 PRINT"INPUT GUESS"
4020 GOSUB 900
4030 RETURN
```

This module is called by line 530 of Program 5(d)

PROGRAM 5(d)

```
530 GOSUB 4000: REM INPUT GUESS
```

Module 5: Was character previously tried?

At this stage of development, there is really nothing to compare, so a simple (Y/N) input is requested. As in the anagram game, flags will be used to pass information from modules to the PCM. Thus, if the guess is correct, a flag is set to -1 and, if incorrect, set to zero. If these numbers appear a little strange, don't worry, all will be revealed later in the chapter. In Module 5, the flag F1 is set to -1 for a 'Y' entry and to zero for a 'N' entry.

PROGRAM 5.5

```
5000 CLS:LOCATE 4,10
5010 PRINT"CHARACTER PREVIOUSLY TRIED (Y/N)?"
5020 GOSUB 900
5030 IF A$="Y" THEN F1=-1
5040 IF A$="N" THEN F1=0
5050 IF A$<>"Y" AND A$<>"N" THEN 5020:ELSE RETURN
```

The routine is called by line 540 in the PCM as in Program 5(f) below:

PROGRAM 5(f)

```
540 GOSUB 5000:REM CHARACTER PREVIOUSLY TRIED?
```

Module 6: Message "character previously tried"

The purpose of this element is simply to tell the player that the latest character input had previously been used.

PROGRAM 5.6

```
6000 CLS:LOCATE 4,10
6010 PRINT"CHARACTER PREVIOUSLY TRIED"
6020 GOSUB 900
6030 RETURN
```

This routine is called only when the character has been previously guessed, i.e. when F1=-1, line 550 of the PCM. Once this sub-routine has been called the program is directed back to line 530 for another input.

PROGRAM 5(g)

```
550 IF F1=-1 THEN GOSUB 6000:GOTO 530:
REM GO BACK FOR ANOTHER INPUT
```

Module 7: Is guess in word?

Once it has been established that the guess has not been made before, a check must be made for a correct guess and the program routed accordingly. As with the other testing module, Module 5, the skeleton program will allow the Flag to be set from the (Y/N) input.

PROGRAM 5.7

```
7000 CLS:LOCATE 4,10
7010 PRINT"IS THE GUESS IN THE WORD (Y/N)?"
7020 GOSUB 900
7030 IF A$="Y" THEN F2=-1
7040 IF A$="N" THEN F2=0
7050 IF A$<>"Y" AND A$<>"N" THEN 7020:ELSE RETURN
```

And the PCM that calls this, line 560 of Program 5(h) is shown below:

PROGRAM 5(h)

```
560 GOSUB 7000:REM GUESS IN WORD?
```

Once this module has been run, it is known whether or not the guess is correct and the program needs to be re-routed accordingly. To make that a little clearer, let's have a look at the program's flow on the flowchart in Figure 5.2

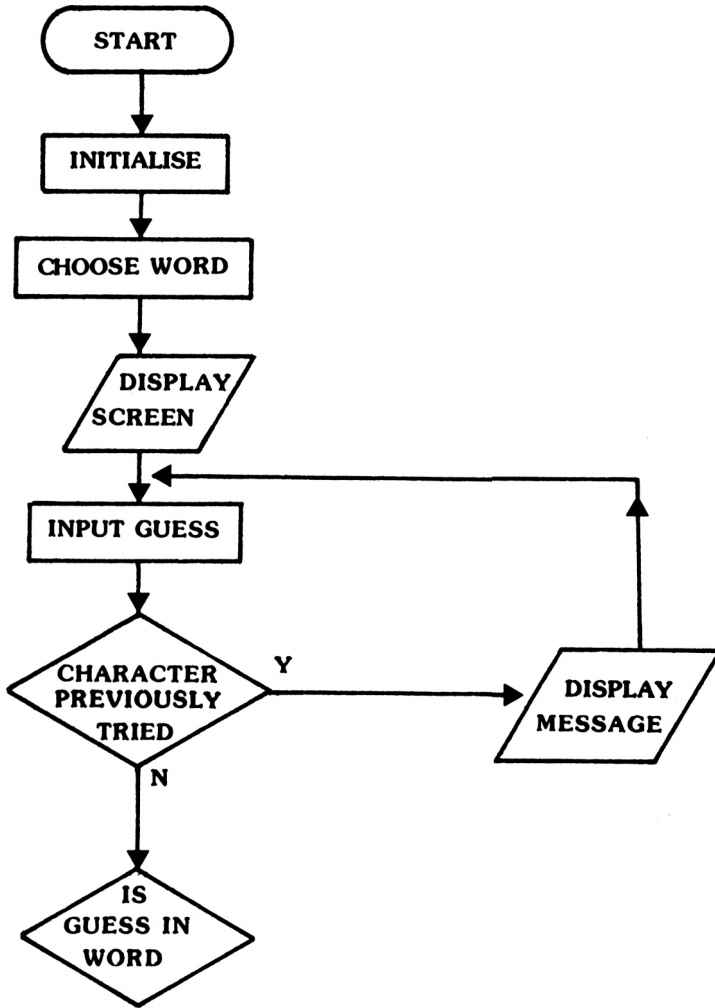


FIGURE 5.2

Spelling out the whole PCM to date, then, gives Program 5(i). In this a test is made for F1 which returns the 'character previously tried' condition. When the character was previously tried, i.e. F1=-1, the subroutine at 6000 is called to give that message i.e.

```
IF F1=-1 THEN GOSUB 6000
```

Once the message has been given, the program needs to return to the point in the program where another guess is inputted. Proponents of structured programming would require the program to work its way through all the tests prior to returning to line 530 in order to input another screen. However, proponents of pragmatic programming would add in a judicious 'GOTO' at this point! i.e.

```
550 IF F1=-1 THEN GOSUB 6000:GOTO 530
```

PROGRAM 5(i)

```
500 GOSUB 1000:REM INITIALISATION
510 GOSUB 2000:REM CHOOSE WORD
520 GOSUB 3000:REM DISPLAY SCREEN
530 GOSUB 4000:REM INPUT GUESS
540 GOSUB 5000:REM CHARACTER PREVIOUSLY TRIED?
550 IF F1=-1 THEN GOSUB 6000:GOTO 530
:REM GO BACK FOR ANOTHER INPUT
560 GOSUB 7000:REM GUESS IN WORD?
570 IF F2=0 THEN XXX:REM GUESS NOT IN WORD
```

When the test in line 550 fails, the program next tests to establish whether or not the guess was correct. As with all tests the program branches following it, the route depending on the result of the test. Figure 5.3 demonstrates this is in action:

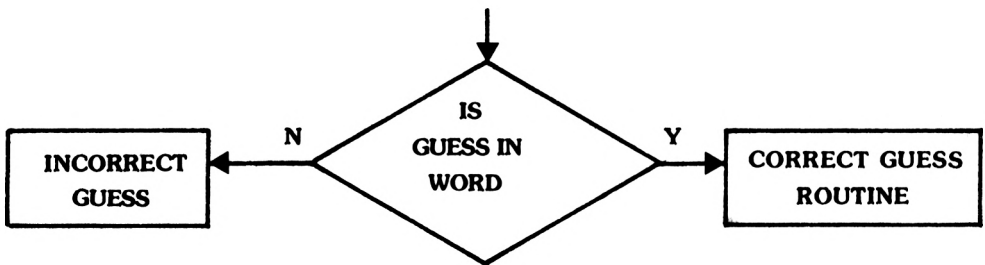


FIGURE 5.3

For the time being the 'GOTO' on line 570 will be left unfinished because at this stage the line number is not known.

So far the program has progressed as far as handling the conditions; where the letter guessed is in the word and the character was previously tried. Now our attention is turned to the situation where the guess was correct and had not previously been tried. The next three modules to consider are:

- 8 Store the newly guessed letter in the 'word guessed so far' array(WORD\$).
- 9 Modify the screen display to tell the user what is happening.
- 10 Test if there are any more letters left to guess, i.e. is the game over.

Putting this into a section of flowchart yields:

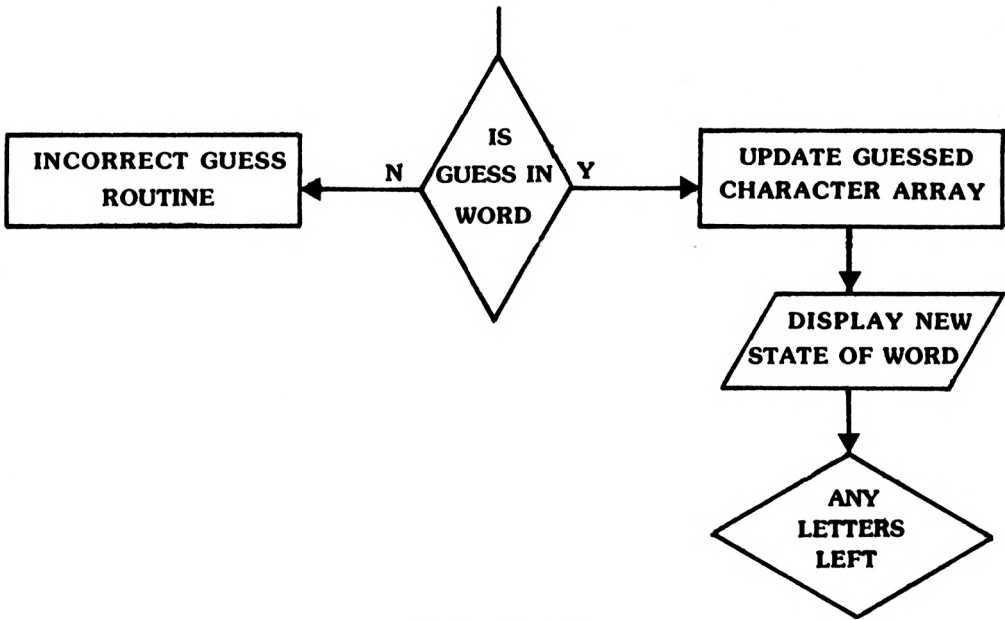


FIGURE 5.4

Firstly the Modules...

Module 8: Update array of guessed characters.

In this routine, the array WORD\$ needs to be updated by replacing the dot that is stored where the letter should go with the actual letter.

PROGRAM 5.8

```

8000 CLS:LOCATE 10,10
8010 PRINT"UPDATE ARRAY"
8020 GOSUB 900
8030 RETURN
  
```

Module 9: Display new screen

This module, shown in skeleton form in Program 5.9, simply reports back on the current state of the guessed word.

PROGRAM 5.9

```

9000 CLS:LOCATE 4,10
9010 PRINT"THE WORD SO FAR IS"
9020 GOSUB 900
9030 RETURN
  
```

Module 10: test for all characters guessed

At this point a test is carried out to determine whether or not the player has guessed all the characters in the word.

PROGRAM 5.10

```
10000 CLS:LOCATE 4,10
10010 PRINT"ANY CHARACTERS LEFT TO GUESS (Y/N)?"
10020 GOSUB 900
10030 IF A$="Y" THEN F3=-1
10040 IF A$="N" THEN F3=0
10050 IF A$<>"Y" AND A$<>"N" THEN 10020:ELSE RETURN
```

The PCM up to this point is fairly straightforward as it just flows directly through from Module 8 to Module 10 as in Program 5(j)

PROGRAM 5(j)

```
580 GOSUB 8000:REM UPDATE ARRAY
590 GOSUB 9000:REM DISPLAY SCREEN
600 GOSUB 10000:REM ANY CHARACTERS LEFT TO GUESS?
```

Once the test has been made to see if all the characters have been guessed, then the program diverges once more. If more characters remain to be guessed i.e. F3=-1, then the program returns for another input (to Module 4) and when all the characters have been guessed, the game is over, and the player should be told, Figure 5.5.

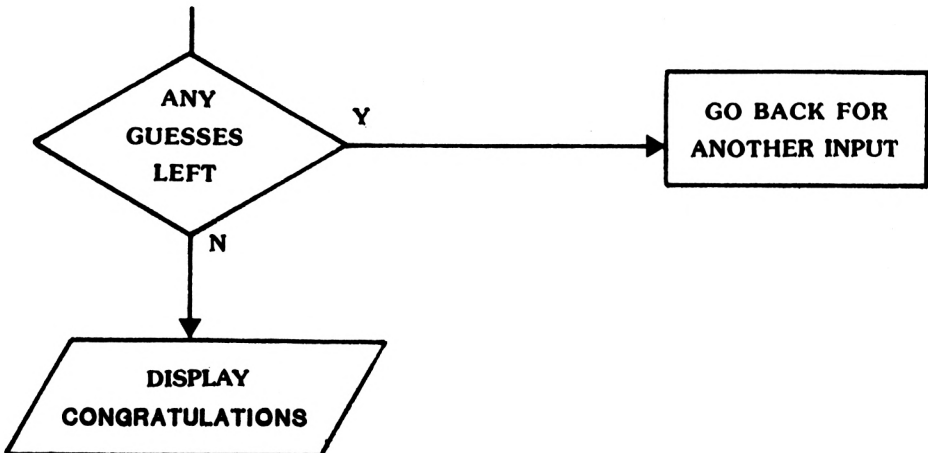


FIGURE 5.5

The easiest of these routes to tackle is the one going back for another input. All that is required is a re-directing of the program to the input routine when F3 is set to -1 i.e.

```
IF F3=-1 THEN 530
```

When this flag is not set then the program will call Module 11.

```
PROGRAM 5(k)
```

```
610 IF F3=-1 THEN 530
620 GOSUB 11000:REM WELL DONE
```

Once the player has been told that they have won (Module 11), Module 12 will ask the player if another go is required. Thus, Modules 11 and 12 are....

Module 11: Tell the player "well done"

```
PROGRAM 5.11
```

```
11000 CLS:LOCATE 4,10
11010 PRINT"WELL DONE YOU HAVE GUESSED
THE WORD"
11020 GOSUB 900
11030 RETURN
```

Module 12: Ask "do you want another go?"

```
PROGRAM 5.12
```

```
12000 CLS:LOCATE 4,10
12010 PRINT"DO YOU WANT ANOTHER GO (Y/N)?"
12020 GOSUB 900
12030 IF A$="Y" THEN F5=-1
12040 IF A$="N" THEN F5=0
12050 IF A$<>"Y" AND A$<>"N" THEN 12020:ELSE RETURN
```

Module 12 is called by a simple 'GOSUB' but then the Flag returned must be decoded. When 'another go' is required the program is re-routed right back to Module 1 and when no further goes are wanted, the whole program ends. The PCM section that calls this is:

PROGRAM 5(1)

```
630 GOTO 700:REM ANOTHER GO?  
.  
.  
.  
700 GOSUB 12000:REM ANOTHER GO?  
710 IF F5=-THEN 500:REM PLAY IT AGAIN SAM!  
720 CLS:LOCATE 10,10  
730 PRINT "GOODBYE!"  
740 END
```

If another go is required the program is directed to line 500 which calls up the initialisation routine, so that the program begins again.

Figure 5.6 shows the flowchart so far:

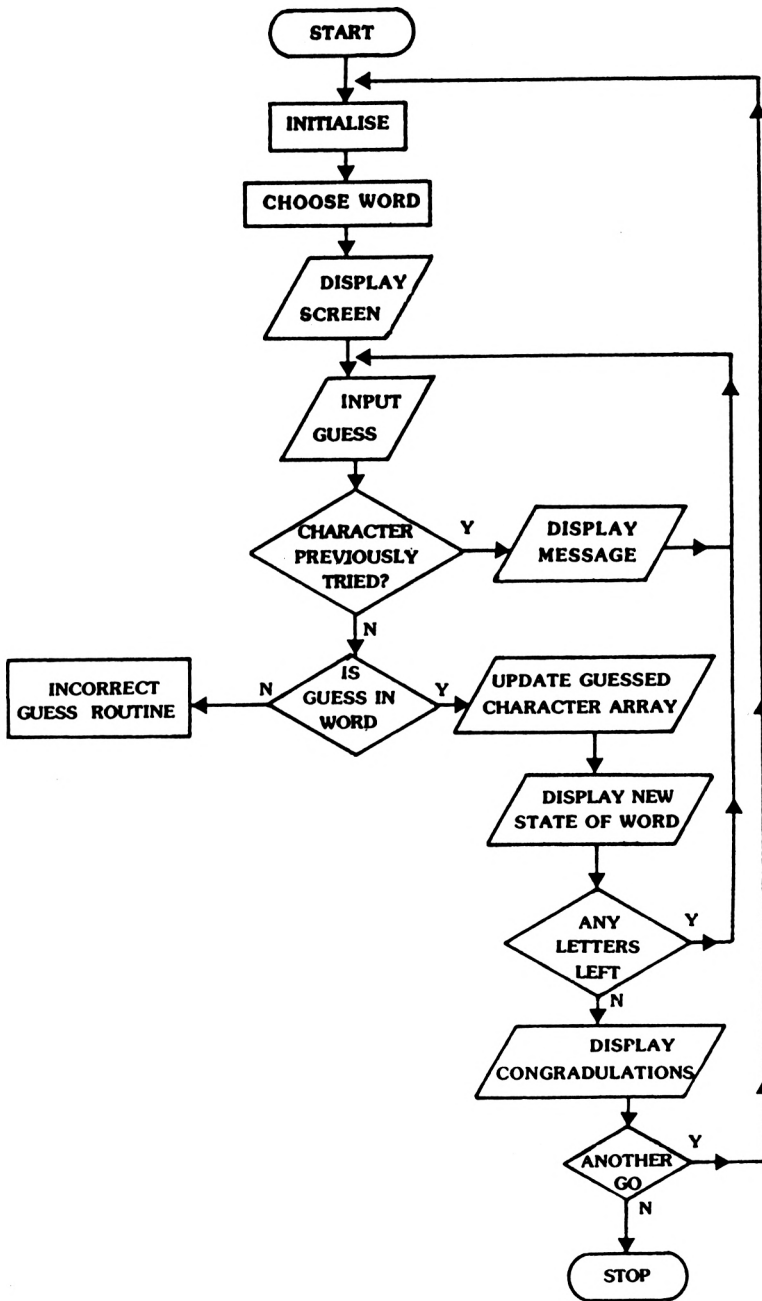


FIGURE 5.6

From Figure 5.6 it can be seen that all that remains to be done are the routines that handle the route from Module 6 where F2=0, i.e. the guess was not in the word. Line 570 can now be completed to read:

```
570 IF F2=0 THEN 640:REM GUESS NOT IN WORD
```

Along this route, the first stages are: to increment the wrong guess count (Module 13), display the new screen (Module 14) and then check if all the goes have been used (Module 15) Figure 5.7



FIGURE 5.7

Now the Modules.....

Module 13: Increment wrong guess count

PROGRAM 5.13

```
13000 CLS:LOCATE 4,10
13010 PRINT"INCREMENT WRONG GUESS COUNT"
13020 GOSUB 900
13030 RETURN
```

Module 14: display new screen

PROGRAM 5.14

```
14000 CLS:LOCATE 4,10
14010 PRINT"DISPLAY NEW SCREEN"
14020 GOSUB 900
14030 RETURN
```

Module 15: Are all goes used?

PROGRAM 5.15

```
15000 CLS:LOCATE 4,10
15010 PRINT"ARE ALL GOES USED (Y/N)?"
15020 GOSUB 900
15030 IF A$="Y" THEN F4=-1
15050 IF A$="N" THEN F4=0
15050 IF A$<>"Y" AND A$<>"N" THEN 15020:ELSE RETURN
```

PROGRAM 5(m)

```
640 GOSUB 13000:REM INCREMENT WRONG GUESS COUNT
650 GOSUB 14000:REM DISPLAY NEW SCREEN
660 GOSUB 15000:REM ARE ALL GOES USED?
```

Once Module 15 has been run a check must be made on the condition of F4. If it is not set (F4=0) then all the goes have not yet been used and the program loops back to line 530 for another input. However, if all the goes have been used, then the player is informed and asked if another go is required. Figure 5.8 sums up the general situation.

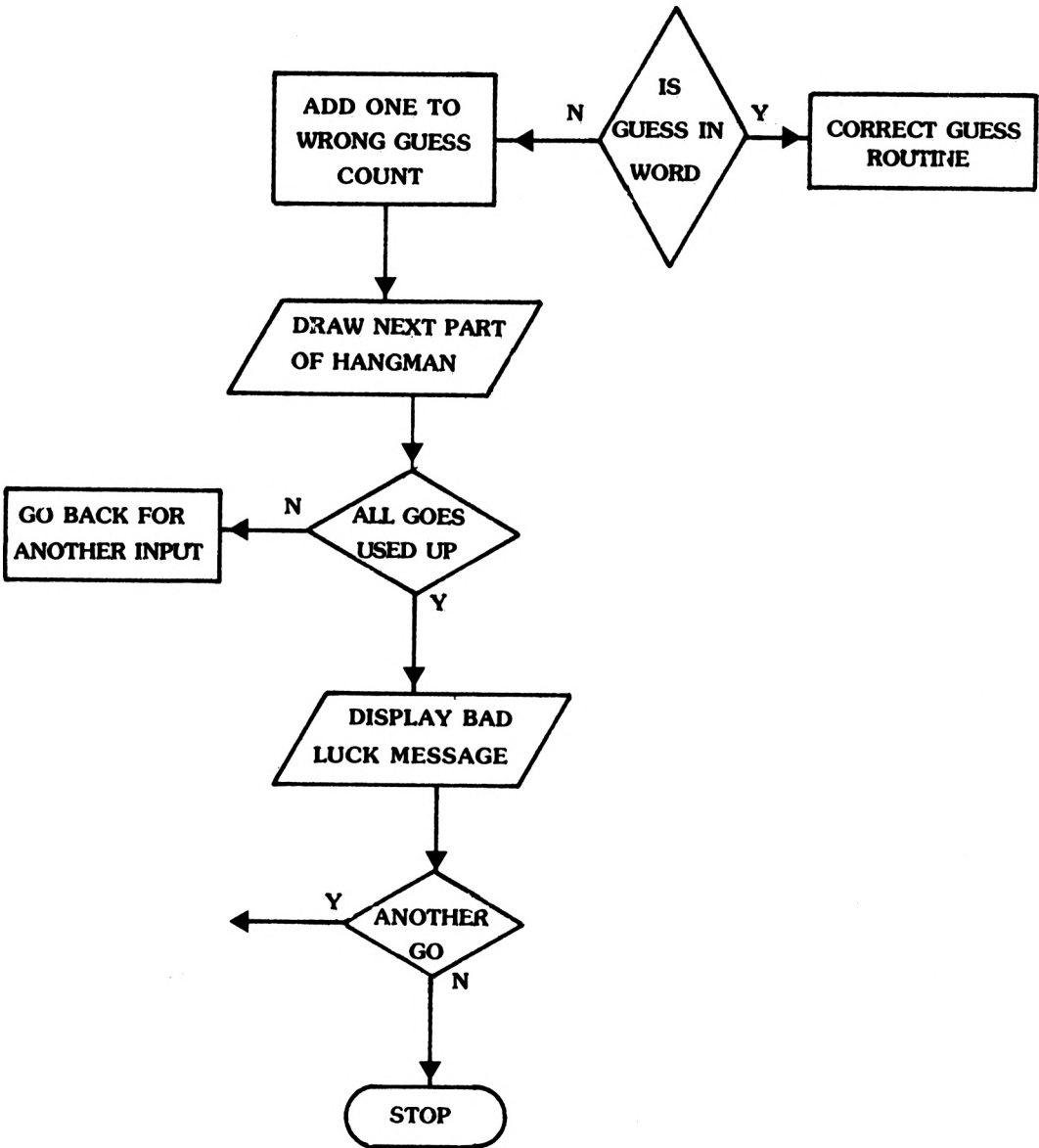


FIGURE 5.8

If all the goes have been used up then the player is hung and asked if another go is required. This program simply runs into line 700 where the player is asked if they want another go.

```
670 IF F4=0 THEN 530:REM MORE GUESSES
680 GOSUB 16000:REM YOU HAVE LOST
700 GOSUB 12000:REM ANOTHER GO?
710 IF F5=-1 THEN 500:REM PLAY IT AGAIN SAM
720 CLS:LOCATE 10,10
730 PRINT"GOODBYE!"
740 END
```

Module 16: You have lost

Only Module 16 now remains to be specified, that being the display to say that all the goes have been used up i.e.

PROGRAM 5.16

```
16000 CLS:LOCATE 4,10
16010 PRINT "ALL GOES USED UP"
16020 GOSUB 900
16030 RETURN
```

At this stage of the proceedings, the entire skeleton of the program has been written and the flow-chart can be drawn. Figure 5.9 shows it in all its glory!

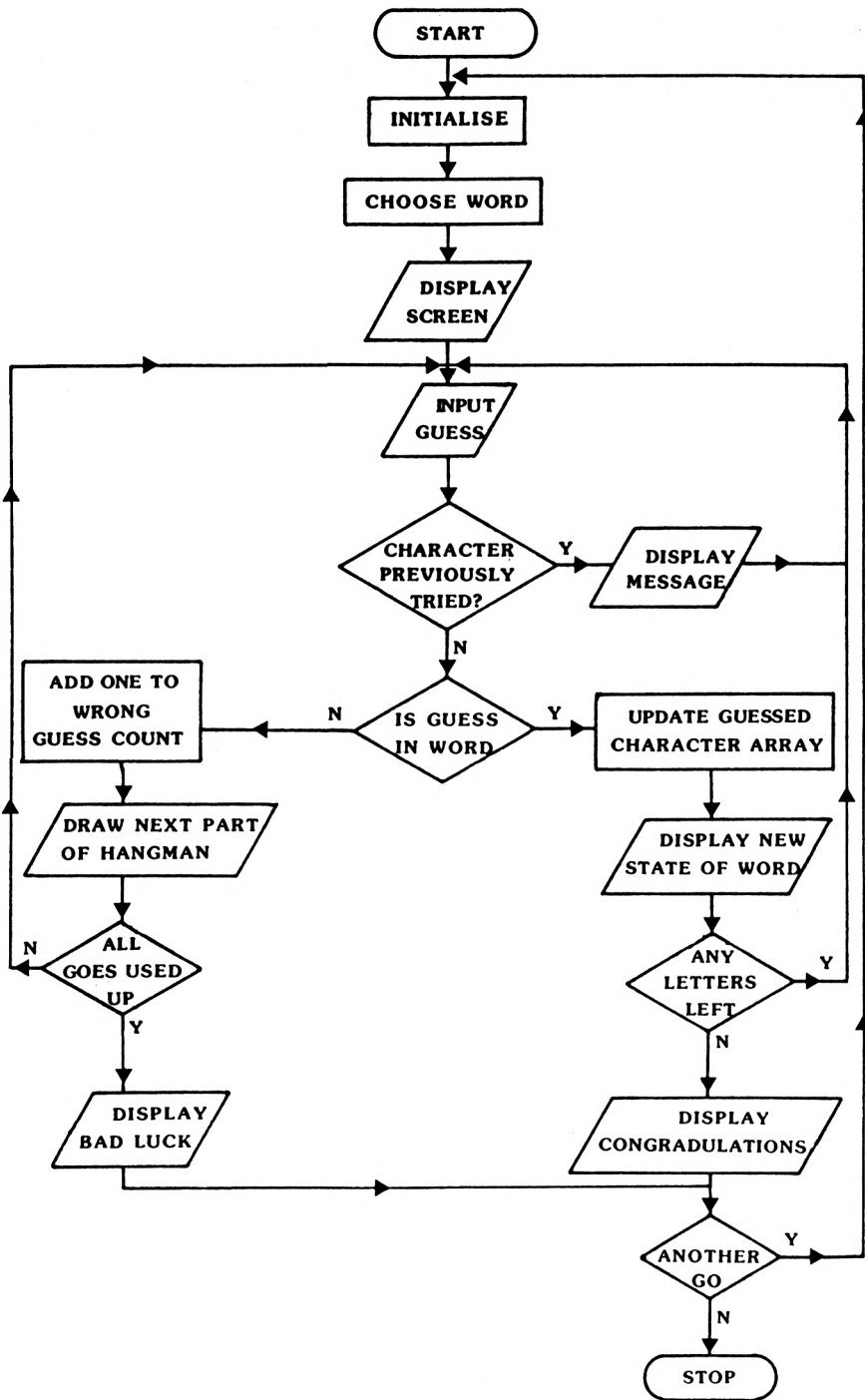


FIGURE 5.9

The next task is to test that all the conditions that can occur during a run of the game have been provided for. One way to tackle this is to list all the points at which branches occur and then to run the program, selecting each branch in turn.

All the flags in this program have two conditions, 'YES' and 'NO'. Figure 5.10 lists all the flags and their possible states.

Flags	States	
	YES	NO
F1	YES	NO
F2	YES	NO
F3	YES	NO
F4	YES	NO
F5	YES	NO

FIGURE 5.10

Now let's RUN the program and step through all the possible stages. Below is given the screen message and in darker type, the entry that you should make. Remember that the 'Y' should be in upper case.

Stage		
1	Initialisation	SPACE
2	Choose word	SPACE
3	Display screen	SPACE
4	Input guess	SPACE
5	Character previously tried?	Y
6	Display 'previously tried' message	SPACE
7	Input guess	SPACE

FIGURE 5.11

Figure 5.11 shows the stages that occur when the program is first run. The first is that which checks a previously guessed character. When the 'character previously tried?' message is displayed press 'Y'. Then the computer will display the following:

CHARACTER PREVIOUSLY TRIED

Obviously that section of the program control module is working. Moving on...

Flag	States	
F1	YES✓	NO
F2	YES	NO
F3	YES	NO
F4	YES	NO
F5	YES	NO

FIGURE 5.12

Now to carry on checking a few more routes: Press SPACE to clear stage 7

Stage		
8	Was character previously tried?	N
9	is guess in word?	Y
10	Update array	SPACE
11	Display new screen (the word so far)	SPACE
12	Any characters left to guess?	Y
13	Input guess	SPACE

FIGURE 5.13

That's another loop completed. Let's mark off what's been tested so far.

Flag	States	
F1	YES✓	NO✓
F2	YES✓	NO
F3	YES✓	NO
F4	YES	NO
F5	YES	NO

FIGURE 5.14

That's just half the routes tested so far. Have a go yourself at planning the rest of the tests. Just in case you have problems, one possible way of doing it is given below:

Stage

14	Was character previously tried?	N
15	Is guess in word?	Y
16	Update array	SPACE
17	Display new screen (the word so far)	SPACE
18	Are characters left to guess?	N
19	Tell player well done	SPACE
20	Want another go?	Y
21	Initialisation	SPACE
22	Choose word	SPACE
23	Display screen	SPACE
24	Input guess.	SPACE

FIGURE 5.15

Another loop and some more tests to be marked off.

Flag	States	
F1	YES ✓	NO ✓
F2	YES ✓	NO
F3	YES ✓	NO ✓
F4	YES	NO
F5	YES ✓	NO

FIGURE 5.16

That just leaves testing to be done along the 'guess incorrect' route. So off we go again with a SPACE to clear the Input.

Stage

25	Was character previously tried?	N
26	Is guess in word?	N
27	Increment wrong guess count	SPACE
28	Display new screen	SPACE
29	All goes used?	N
30	Input guess	SPACE
31	Character previously tried?	N
32	Is guess in word?	N
33	Increment wrong guess count	SPACE
34	Display new screen	SPACE
35	All goes used?	Y
36	Display screen	SPACE
37	Want another go?	N

FIGURE 5.17

The program should now have ended. If the program ended at that point then all is well, all routes having been tested successfully! If it seems a lot of bother then that's only because it is. Even an apparently simple program like this has a fairly complex structure and needs to be tested thoroughly before the subroutines are added. That is the next stage:

Developing the modules

Once the overall program structure is developed and working the individual modules can be designed, tested and then slotted into place one at a time. In this way the overall problem can be broken down into manageable parts. The various subroutines are developed into the program structure to facilitate testing.

Firstly then....

Program control module

```
500 GOSUB 1000:REM INITIALISATION
510 GOSUB 2000:REM CHOOSE WORD
520 GOSUB 3000:REM DISPLAY SCREEN
530 GOSUB 4000:REM INPUT GUESS
540 GOSUB 5000:REM CHARACTER PREVIOUSLY
    TRIED?
550 IF F1=-1 THEN GOSUB 6000:GOTO 530:
    REM REPORT CHARACTER PREVIOUSLY TRIED
560 GOSUB 7000:REM GUESS IN WORD?
570 IF F2=0 THEN 640:REM GUESS NOT IN W
    ORD
580 GOSUB 8000:REM UPDATE ARRAY
590 GOSUB 9000:REM DISPLAY SCREEN
600 GOSUB 10000:REM ANY CHARACTERS LEFT
    TO GUESS?
610 IF F3=-1 THEN 530:REM MORE CHARACTE
    RS TO GUESS
620 GOSUB 11000:REM WELL DONE
630 GOTO 700:REM ANOTHER GO?
640 GOSUB 13000:REM INCREMENT WRONG GUE
    SS COUNT
650 GOSUB 14000:REM DISPLAY NEW SCREEN
660 GOSUB 15000:REM REM ARE ALL GOES US
    ED?
670 IF F4=0 THEN 530:REM MORE GUESSES
680 GOSUB 16000:REM YOU HAVE LOST
```

```

700 GOSUB 12000:REM ANOTHER GO?
710 IF F5=-1 THEN 500:REM PLAY IT AGAIN
    SAM!
720 CLS:LOCATE 10,10
730 PRINT"GOODBYE"
740 END

```

You might like to get rid of the now superfluous END statement on line 899 at this stage, as line 740 is now the 'official' end of the program.

Module 1: Initialisation

The initialisation routine first clears the screen and sets up the colours the game will use in 'ink pots' 0 to 3. After this, lines 1040-1190 are used to give the rules of the game, these having been left for you, the reader, to do your literary best. Real care must be taken over this documentation phase as, if the game is to stand on its own without you to explain it, the rules must be absolutely clear.

The initialisation routine can now be written; what it will do is

```

Clear the screen and set colours - Lines 1010-1030
Announce the game - line 1040
Give the rules - Lines 1050-1190
Stop the program until the rules have been read and a key
pressed - Lines 1200-1205

```

Putting that into a Program:

PROGRAM 5.1(a)

```

1000 REM initialisation
1010 MODE 1
1020 INK 0,13:INK 1,26:INK 3,17:INK 4,3
1030 BORDER 17:PEN 1:PAPER 0:CLS
1040 LOCATE 14,2:PRINT"***HANGMAN***"
1050 LOCATE 4,6
1060 PRINT"Here are the rules!"
1070 LOCATE 4,8
1080 PRINT"There are no rules!"
1090 LOCATE 4,20
1200 PRINT"Press space bar to continue"
1205 IF INKEY$<>" " THEN 1190

```

Each time the game is played, several variables need to be reset and strings cleared, for instance Z\$. On the first run through the game this is set to ' FIRST ' i.e. the screen then says what is your Z\$ (first) guess. Immediately after use, Z\$ is reset to "NEXT" so that the player is then asked 'what is your Z\$ (next) guess?' Of course, once the game has been played Z\$=" NEXT " and needs re-setting or re-initialising. Other variables need re-setting too, such as E (the number of incorrect guesses so far) and X\$ (a string comprising all the guesses made to date).

PROGRAM 5.1(b)

```
1220 Z$=" FIRST "  
1230 E=0  
1240 X$=""  
1250 RETURN
```

Once this has been entered the program may be run. However, apart from the 'title' page it will appear little different from when run previously. No enhancements to the screen display have been made at this stage.

Module 2: Choose word

This routine, shown in Program 5.2(a), generates a random number and then searches that number of times through the DATA statements.

PROGRAM 5.2(a)

```
2000 REM CHOOSE WORD  
2010 R=INT(RND*10)+1  
2011 R=4:REM development only  
2020 RESTORE  
2030 FOR X=1 TO R  
2040 READ A$  
2050 NEXT X  
2060 L=LEN(A$)
```

Line 2011 sets 'R' to the value of '4', so that when testing this program we know what word to guess. This will save a lot of time when debugging the program.

Note the FOR....NEXT loop in 2030 runs from 1 to R, i.e. it changes randomly as different random numbers are generated in line 2010.

In addition to choosing the word, the subroutine also calculates its length, L, as this is needed in other subroutines. The variable 'L' is then used to make up the string WORD\$. At the beginning of the game this simply contains the requisite number of dots - i.e. one for each letter. As correct guesses are made the appropriate letters are inserted in the correct place in the array and then displayed on the screen. Thus if the word selected (A\$) is originally 'COMPUTER', then L=8 and each value in the array the WORD\$ is set to a dot.

PROGRAM 5.2(b)

```
2500 WORD$=""
2510 FOR X=1 TO L
2510 WORD$(X)="." : WORD$=WORD$+"."
2520 NEXT X
2530 RETURN
```

The DATA to be READ is stored at lines 18000 onwards as in Program 5.2(c)

PROGRAM 5.2(c)

```
18000 DATA CAT,TURBINE,PLATE,COMPUTER
18010 DATA
18020 DATA
18030 DATA
```

The rest of the data is left for you to fill in. No difference will be detectable when Module 2 is typed in and the program RUN as none of its handiwork is displayed on to the screen at this stage.

Module 3: Display Screen

At this point in the game, all that has to be displayed is the message telling the player how many letters the word has, and the string W\$.

PROGRAM 5.3(a)

```
3000 REM display the screen
3005 CLS
3010 LOCATE 14,1:PRINT"***HANGMAN***"
3020 LOCATE 4,3
3030 PRINT"The word is"
3040 LOCATE 16,3:PEN 2
3050 PRINT word$
3060 LOCATE 4,5:PEN 1
3070 PRINT"The word has";
3080 PEN 2:PRINT 1;:PEN 1
3090 PRINT"letters"
3100 RETURN
```

Now, when the program to date is RUN, the beginnings of the game will begin to emerge. However, the display will disappear right away because the 'Press any key' prompt has been removed and the routine immediately RETURNS to line 520. Again the exact nature of the finished screen display is up to you.

Module 4: Input a guess

All that is required here is a simple message to tell the user to input a guess. Sometimes, however, it will be the 'FIRST' guess and sometimes the 'NEXT' guess. This can be accommodated by assigning the word 'FIRST' to a string during the initialisation and then, once the program has run, changing the contents of the variable to 'NEXT'. The two elements that do this are shown in Program 5.4(a). The use of an INKEY\$ rather than INPUT allows a single character - presumably a letter! - to be input without the need to press ENTER.

PROGRAM 5.4(a)

```
1210 Z$=" FIRST "

4000 REM input a guess
4010 LOCATE 4,7
4020 PRINT"What is your";z$;"guess?      "
4025 LOCATE 29,7:PEN 3:PRINT CHR$(143):PEN 1
4030 g$=INKEY$:IF g$="" THEN 4030
4040 g$=UPPER$(g$)
4050 IF g$<"A" OR g$>"Z" THEN 4030
4060 LOCATE 29,7:PEN 2:PRINT g$
4070 z$=" next ":PEN 1
4080 RETURN
```

The inputted character is converted to capitals using the UPPER\$ command.

Now, when RUN, the program will get as far as inputting a guess which is assigned to the variable G\$ and printed onto the screen.

Module 5: Was character previously tried?

When a player puts in a guess that is a repeat of a previous entry, this program treats him kindly. It would be possible to charge this guess against his number of allowed attempts but the option chosen here is to report that that particular letter has been guessed before and then loop back for another input. In Program 5.5(a), F1 is initially set to zero, and is only set to one if the inputted character, G\$, is found in X\$.

The value, X\$, was set to "", i.e. an empty string on the line the initialisation procedure, Program 5.1(b). As a guess is made, it is added to the string (Module 8) and so, at this stage it is only necessary to read through the string to check whether any of its letters equal GUESS\$, the latest guess. One slight complication exists in that the string gets one letter longer each time the letter has not been guessed before so it is always necessary to recalculate its length (L2), as in line 5010.

PROGRAM 5.5(a)

```
5000 F1=0
5010 L2=LEN(X$)
5020 FOR X=1 TO L2
5030 IF G$=MID$(X$,X,1) THEN F1=-1
5040 NEXT X
5050 IF F1=0 THEN X$=X$+G$
5060 RETURN
```

Again, RUNning the program will yield no new display. An alternative way of checking X\$ is to use the INSTR function.

INSTR

This function automatically searches one string for the occurrence of a second. Thus, lines 5010 to 5050 inclusive could be replaced with

```
5010 IF INSTR(X$,G$)=0 THEN X$=X$+G$:ELSE F1=-1
```

INSTR returns a zero if it doesn't find the string G\$ within X\$. If G\$ is somewhere in X\$, INSTR returns its position within X\$.

For instance, with

```
PRINT INSTR("FRED","R")
```

you would get a 2 printed, because R is the second letter of "FRED". As a bit of an aside, you can also do this like

```
PRINT INSTR("FRED","ED")
```

which will return 3, because "ED" starts at position 3 in "FRED".

Module 6: Message: "Character previously tried"

The aim of this message is to inform the player clearly that the letter just guessed has already been tried, and then to clear the screen back to its previous state. The message is PRINTed onto a line that is currently empty. Once on the screen, the message is held there for a time while the player takes it in and then it is cleared. What is needed here is a technique for causing the program to wait for a specific time period i.e. a delay.

Once the message in Module 6 has sunk in, it needs to be removed by printing blank spaces over it. This is achieved by means of the SPACE\$ command.

PROGRAM 5.6(a)

```
6000 REM character previously tried!  
6010 LOCATE 4,24:PEN 2  
6020 PRINT"You have already tried that letter!"  
6030 FOR x=1 TO 1000:NEXT x  
6040 LOCATE 4,24:PEN 1  
6050 PRINT SPACE$(36)  
6060 RETURN
```

At this stage, when the program is RUN (in capitals mode) and a character is guessed for the second time, the 'already tried' message will appear.

Module 7: Is guess in word?

Once a guess has been made the subroutine in Program 5.7(a) needs to read through the word looking for a match with the inputted letter. Should it find such a match, then F2 will be set to -1. Note that, at the start of this subroutine, the flag is reset to zero and remains at zero unless the test at line 7020 proves positive.

PROGRAM 5.7(a)

```
7000 REM IS GUESS IN WORD?
7010 F2=0
7020 IF INSTR(A$,G$)<>0 THEN F2=-1
7030 RETURN
```

Remember that WORD\$() holds the status of the word being guessed, starting off with all dots. As correct guesses are made, the correct letters are inserted into this array at the appropriate place so that the word to be guessed is built up gradually.

As no more screen displays have been added, RUNNING the program at this stage will display nothing new on the screen.

Module 8: Update array of guessed characters

This Module handles the case where the character guessed has not previously been guessed; it is inserted into WORD\$ at the appropriate place.

PROGRAM 5.8(a)

```
8000 REM update array
8010 FOR x=1 TO 1
8020 IF g$=MID$(a$,x,1) THEN GOSUB 8100
8030 NEXT x
8040 RETURN
8100 word$(x)=g$
8110 RETURN
```

Figure 5.18 demonstrates the process for the INPUT of an 'S' (i.e. GUESS\$="S") where A\$="AMSTRAD" and the 'S' has not previously been guessed.

LOOP NUMBER	A\$	WORD\$() before	WORD\$() after
1	A	.	.
2	M	.	.
3	S	.	S
4	T	.	.
5	R	.	.
6	A	.	.
7	D	.	.

FIGURE 5.18

Module 9: The word so far

At this point the string WORD\$ needs to be made up out of each of the values of WORD\$() and then the program will print out the current state of the guessed word, this being done in line 9060 of Program 5.9(a).

PROGRAM 5.9(a)

```
9000 REM the word so far
9010 word$=""
9020 FOR x=1 TO 1
9030 word$=word$+word$(x)
9040 NEXT x
9050 LOCATE 16,3:PEN 2
9060 PRINT word$:PEN 1
9070 RETURN
```

Module 10: Are all characters guessed?

To check this, array WORD\$() needs to be read through to see if any character position remains unfilled. If this is so then F3 is set to a -1, in line 10030 of Program 5.10(a).

PROGRAM 5.10(a)

```
10000 REM ARE ALL CHARACTERS GUESSED?
10010 F3=0
10020 FOR X=1 TO L
10030 IF WORD$(X)="" THEN F3=-1
10040 NEXT X
10050 RETURN
```

Module 11: Message "Well done"

This Module tells the player that the word has been guessed correctly. Once the message is on the screen, the game is over and there is, therefore, no need to display it for a fixed time. As the next stage of the program is to ask the player if another go is wanted, the message can be left on the screen until a key is pressed.

PROGRAM 5.11(a)

```
11000 REM WELL DONE YOU HAVE WON
11010 LOCATE 28,10:PEN 2
11020 PRINT"WELL DONE!"
11030 LOCATE 28,12:PEN 3
11040 PRINT "YOU HAVE WON"
11050 RETURN
```

Module 12: Do you want another go?

This is a simple test to see if another go is required. If yes then F5 is set to -1 (line 12050), if not then F5=0 (line 12060).

PROGRAM 5.12(a)

```
12000 REM DO YOU WANT ANOTHER GO?
12010 LOCATE 4,24:PEN 2
12020 PRINT"DO YOU WANT ANOTHER GO (Y/N)?"
12030 Z$=INKEY$:IF Z$="" THEN 12030
12040 Z$=UPPER$(Z$)
12050 IF Z$="Y" THEN F5=-1
12060 IF Z$="N" THEN F5=0
12070 IF Z$<>"Y" AND Z$<>"N" THEN 12030
12080 RETURN
```

Module 13: Increment wrong guess count

The variable 'E' records the number of wrong guesses and is simply incremented at the appropriate time whenever F2 is set to '0'. It is done in line 13010 of Program 5.13(a)

When the guess is incorrect, the player is told "SORRY THAT LETTER IS NOT IN THE WORD."

PROGRAM 5.13(a)

```
13000 REM wrong guess!
13010 e=e+1
13020 LOCATE 4,23:PEN 2
13030 PRINT"I'm sorry that letter is not
in the word"
13040 FOR x=1 TO 1000:NEXT x
13050 LOCATE 4,23:PEN 1
13060 PRINT SPACE$(40)
13070 RETURN
```

As in a previous subroutine, the message is maintained on the screen by the FOR... NEXT loop on line 13040.

Module 14: Draw Hangman

Once an incorrect guess has been made another piece of the hangman will be drawn. The hangman we will draw is created by using Amstrad graphics commands PLOT and LINE and a new Basic command:

ON...GOTO

The individual pieces of the hangman will be drawn in separate program sections all within the one subroutine. One section will be for drawing the base, one for drawing the head etc. These individual sections will be accessed depending on the value of 'E' (the incorrect guess count), using a special version of the IF...THEN command, called ON...GOTO. This works a bit like lots of 'IF...THEN...' commands would. Taking the example:

```
ON X GOTO 100,200,300
```

The computer understands this as:

```
" ON the value of X, GOTO 100, 200, 300"
```

Thus if X = 1 then the program is directed to line 100, if X is 2 then the program goes to line 200 and so on. If X is '0', or greater than the number of line names listed the program will continue onto the next line - in the example below, to line 30.

Just to see this in action try the following:

PROGRAM 5.14(b)

```
10 INPUT X
20 ON X GOTO 40,50,60
30 END
40 PRINT"40"
50 PRINT"50"
60 PRINT"60"
```

Now RUN this and test it out with various entries.

As well as using 'GOTOS', the ON... command can just as well use GOSUBs. In this case, it directs the program to the subroutine in just the same way as would any GOSUB.

For drawing the hangman graphics we will use an 'ON GOTO' command, and each 'drawing section' will end with RETURN. Thus, on the first value of E (E=1) the program will GOTO line 14040 and draw part of the Hangman frame. Line 14090 RETURNS the program back to the PCM. In this way all ten sections of the Hangman can be drawn in sequence, i.e. the third guess will draw the third 'item'.

PROGRAM 5.6(s)

```
14000 REM draw hangman on the value of E
14010 REM
14020 ON e GOTO 14040,14100,14150,14210,
14260,14320,14410,14490,14620,14700
14030 REM the hangman drawing sections
14040 REM-----the frame#1-----
14050 PLOT 240,50:DRAW 240,280
14060 PLOT 242,50:DRAW 242,280
14070 PLOT 244,50:DRAW 244,280
14080 PLOT 246,50:DRAW 246,280
14090 RETURN
14100 REM-----the frame#2-----
14110 PLOT 246,280:DRAW 376,280
14120 PLOT 246,278:DRAW 376,278
14130 PLOT 246,276:DRAW 376,276
14140 RETURN
14150 REM-----the frame#3-----
14160 PLOT 246,200:DRAW 316,274
14170 PLOT 246,204:DRAW 314,274
14180 PLOT 246,208:DRAW 312,274
14190 PLOT 246,210:DRAW 310,274
14200 RETURN
14210 REM-----the frame#4-----
14220 PLOT 150,50:DRAW 350,50
14230 PLOT 150,48:DRAW 350,48
14240 PLOT 150,46:DRAW 350,46
14250 RETURN
14260 REM-----the rope-----
14270 PLOT 373,274:PLOT 375,276
14280 FOR x=1 TO 20 STEP 4
```

PROGRAM 5.6(s) continued

```
14290 PLOT 371,276-x:DRAW 376,271-x
14300 NEXT x
14310 RETURN
14320 REM-----the head-----
14330 r=30
14340 PLOT 406,224
14350 py=PI/180
14360 FOR a=0 TO 360 STEP 6
14370 x=r*COS(a*py):y=r*SIN(a*py)
14380 DRAW 376+x,224+y
14390 NEXT a
14400 RETURN
14410 REM-----the face-----
14420 LOCATE 23,11:PRINT"o o"
14430 PLOT 366,209:DRAW 386,209
14440 PLOT 368,209:DRAW 360,216
14450 PLOT 366,209:DRAW 358,216
14460 PLOT 386,209:DRAW 394,216
14470 PLOT 388,209:DRAW 396,216
14480 RETURN
14490 REM-----the body-----
14500 PLOT 348,194:DRAW 408,194
14510 DRAW 420,160:DRAW 400,130
14520 PLOT 348,194
14530 DRAW 336,160:DRAW 354,130
14540 PLOT 356,180
14550 DRAW 349,160:DRAW 363,135
14570 PLOT 398,180
14580 DRAW 406,160:DRAW 392,135
14590 DRAW 400,130
14600 PLOT 363,135:DRAW 350,130
14610 RETURN
14620 REM-----the legs-----
14630 PLOT 355,135
14640 DRAW 363,80:DRAW 375,80
14650 PLOT 410,143:DRAW 402,80
```


PROGRAM 5.6(s) finished

```
14660 DRAW 390,80:DRAW 386,145
14670 PLOT 375,80:DRAW 378,145
14680 PLOT 355,145:DRAW 395,145
14690 RETURN
14700 REM-----the shoes-----
14710 PLOT 363,80:DRAW 345,70
14720 DRAW 375,70:DRAW 375,80
14730 PLOT 404,80:DRAW 422,70
14740 DRAW 390,70:DRAW 390,80
14750 RETURN
```

Module 15: Are all goes used?

This module is a simple test of the variable 'E'. If all the allowable goes have been used then E is equal to 10, and F4 is set to -1.

PROGRAM 5.15(a)

```
15000 REM ALL GOES USED UP?
15010 F4=0
15020 IF E=10 THEN F4=-1
15030 RETURN
```

Module 16: Display losing message

By this point in the game, it's all over for the player and that's what the message says in lines 16000 and 16080. The player is given the correct solution to the game as some compensation!

PROGRAM 5.16(a)

```
16000 REM LOSER MESSAGE
16010 LOCATE 2,10:PEN 2
16020 PRINT"BAD LUCK!"
16030 LOCATE 2,12
16040 PRINT"YOU ARE HUNG"
16050 LOCATE 2,14:PEN 3
16060 PRINT"THE WORD WAS"
16070 LOCATE 2,16:PRINT A$
16080 RETURN
```

Now that the routines have been written and the game working you might be asking yourself why the flags were set to '-1' instead of the usual '1'. When set like this the flags can be 'logically' tested to see if they are true or false.

True/False

The Amstrad has built in special logic routines. A variable with the value of -1 is deemed to be TRUE. This means that the condition the flag indicates has been encountered. Instead of saying:

```
IF F1=-1 THEN GOSUB 6000:GOTO 530
```

Using the Amstrad's logic handling the line can be rewritten.

```
IF F1 THEN GOSUB 6000:GOTO 530
```

The Amstrad interprets this as meaning:

'if the variable F1 indicates a True value i.e. it is set to minus one, then perform the rest of the line'

By omitting the '=-1' the computer performs the testing operation much faster, almost twice as fast. We are only dealing with millionths of a second but as the saying goes a 'million half pennies is a lot of half pennies'.

The **NOT** command is used by the Amstrad to determine whether a variable has the value of zero i.e. a condition of Not True! Line 670 of the program control module can be changed from:

```
670 IF F4 =0 THEN 530
```

to:

```
670 IF NOT F4 THEN 530
```

The Not operation, like True, is quicker than a normal 'IF...THEN' operation. Using NOT and TRUE whenever possible to test values can speed up programs, particularly long programs with many flags.

PROGRAM 5

```
500 GOSUB 1000:REM initialisation
510 GOSUB 2000:REM choose word
520 GOSUB 3000:REM display screen
530 GOSUB 4000:REM input guess
540 GOSUB 5000:REM character previously
    tried?
550 IF f1 THEN GOSUB 6000:GOTO 530:REM g
    o back for another input
560 GOSUB 7000:REM guess in word?
570 IF NOT f2 THEN 640:REM guess not in
    word
580 GOSUB 8000:REM update array
590 GOSUB 9000:REM display screen
600 GOSUB 10000:REM any characters left
    to guess?
610 IF f3 THEN 530
620 GOSUB 11000:REM well done
630 GOTO 700:REM another go?
640 GOSUB 13000:REM increment wrong gues
    s count
650 GOSUB 14000:REM display new screen
660 GOSUB 15000:REM are all goes used?
670 IF NOT f4 THEN 530:REM more guesses
680 GOSUB 16000:REM you have lost
700 GOSUB 12000:REM another go?
710 IF f5 THEN 500:REM play it again Sam
    !
720 CLS:LOCATE 10,10
730 PRINT"Goodbye!"
740 END
899 END
900 LOCATE 4,20:PRINT"Press any key to c
    ontinue"
910 a$=INKEY$:IF a$="" THEN 910:ELSE RET
    URN
```

```
1000 REM initialisation
1010 MODE 1
1020 INK 0,13:INK 1,26:INK 3,17:INK 4,3
1030 BORDER 17:PEN 1:PAPER 0:CLS
1040 LOCATE 14,2:PRINT"***HANGMAN***"
1050 LOCATE 4,6
1060 PRINT"Here are the rules!"
1070 LOCATE 4,8
1080 PRINT"There are no rules!"
1190 LOCATE 4,20
1200 PRINT"Press space bar to continue"
```

```

1205 IF INKEY$<>" " THEN 1190
1210 z$=" first "
1220 e=0
1230 x$=""
1240 RETURN
2000 REM choose word
2010 r=INT(RND*10)+1
2011 r=4:REM development only
2020 RESTORE
2030 FOR x=1 TO r
2040 READ a$
2050 NEXT x
2060 l=LEN(a$)
2500 word$=""
2510 FOR x=1 TO l
2520 word$(x)="." :word$=word$+"."
2530 NEXT x
2540 RETURN
3000 REM display the screen
3005 CLS
3010 LOCATE 14,1:PRINT"***HANGMAN***"
3020 LOCATE 4,3
3030 PRINT"The word is"
3040 LOCATE 16,3:PEN 2
3050 PRINT word$
3060 LOCATE 4,5:PEN 1
3070 PRINT"The word has";
3080 PEN 2:PRINT 1;:PEN 1
3090 PRINT"letters"
3100 RETURN
4000 REM input a guess
4010 LOCATE 4,7
4020 PRINT"What is your";z$;"guess?      "
4025 LOCATE 29,7:PEN 3:PRINT CHR$(143):P
EN 1
4030 g$=INKEY$:IF g$="" THEN 4030
4040 g$=UPPER$(g$)
4050 IF g$<"A" OR g$>"Z" THEN 4030
4060 LOCATE 29,7:PEN 2:PRINT g$
4070 z$=" next ":PEN 1
4080 RETURN
5000 f1=0
5010 IF INSTR(x$,g$)=0 THEN x$=x$+g$:ELS
E f1=-1
5060 RETURN
6000 REM character previously tried!
6010 LOCATE 4,24:PEN 2
6020 PRINT"You have already tried that 1
etter!"
6030 FOR x=1 TO 1000:NEXT x

```

```

6040 LOCATE 4,24:PEN 1
6050 PRINT SPACE$(36)
6060 RETURN
7000 REM is guess in word?
7010 f2=0
7020 IF INSTR(a$,g$)<>0 THEN f2=-1
7030 RETURN
8000 REM update array
8010 FOR x=1 TO 1
8020 IF g$=MID$(a$,x,1) THEN GOSUB 8100
8030 NEXT x
8040 RETURN
8100 word$(x)=g$
8110 RETURN
9000 REM the word so far
9010 word$=""
9020 FOR x=1 TO 1
9030 word$=word$+word$(x)
9040 NEXT x
9050 LOCATE 16,3:PEN 2
9060 PRINT word$:PEN 1
9070 RETURN
10000 REM are all characters guessed?
10010 f3=0
10020 FOR x=1 TO 1
10030 IF word$(x)="." THEN f3=-1
10040 NEXT x
10050 RETURN
11000 REM well done you have won
11010 LOCATE 28,10:PEN 2
11020 PRINT"Well done!"
11030 LOCATE 28,12:PEN 3
11040 PRINT"You have won!"
11050 RETURN
12000 REM do you want another go?
12010 LOCATE 4,24:PEN 2
12020 PRINT"Do you want another go (Y/N)
?"
12030 z$=INKEY$:IF z$="" THEN 12030
12040 z$=UPPER$(z$)
12050 IF z$="Y" THEN f5=-1
12060 IF z$="N" THEN f5=0
12070 IF z$<>"Y" AND z$<>"N" THEN 12030
12080 RETURN
13000 REM wrong guess!
13010 e=e+1
13020 LOCATE 4,23:PEN 2
13030 PRINT"I'm sorry that letter is not
in the word"

```

```

13040 FOR x=1 TO 1000:NEXT x
13050 LOCATE 4,23:PEN 1
13060 PRINT SPACE$(40)
13070 RETURN
14000 REM draw hangman on the value of E
14010 REM
14020 ON e GOTO 14040,14100,14150,14210,
14260,14320,14410,14490,14620,14700
14030 REM the hangman drawing sections
14040 REM-----the frame#1-----
14050 PLOT 240,50:DRAW 240,280
14060 PLOT 242,50:DRAW 242,280
14070 PLOT 244,50:DRAW 244,280
14080 PLOT 246,50:DRAW 246,280
14090 RETURN
14100 REM-----the frame#2-----
14110 PLOT 246,280:DRAW 376,280
14120 PLOT 246,278:DRAW 376,278
14130 PLOT 246,276:DRAW 376,276
14140 RETURN
14150 REM-----the frame#3-----
14160 PLOT 246,200:DRAW 316,274
14170 PLOT 246,204:DRAW 314,274
14180 PLOT 246,208:DRAW 312,274
14190 PLOT 246,210:DRAW 310,274
14200 RETURN
14210 REM-----the frame#4-----
14220 PLOT 150,50:DRAW 350,50
14230 PLOT 150,48:DRAW 350,48
14240 PLOT 150,46:DRAW 350,46
14250 RETURN
14260 REM-----the rope-----
14270 PLOT 373,274:PLOT 375,276
14280 FOR x=1 TO 20 STEP 4
14290 PLOT 371,276-x:DRAW 376,271-x
14300 NEXT x
14310 RETURN
14320 REM-----the head-----
14330 r=30
14340 PLOT 406,224
14350 py=PI/180
14360 FOR a=0 TO 360 STEP 6
14370 x=r*COS(a*py):y=r*SIN(a*py)
14380 DRAW 376+x,224+y
14390 NEXT a
14400 RETURN
14410 REM-----the face-----
14420 LOCATE 23,11:PRINT"o o"
14430 PLOT 366,209:DRAW 386,209
14440 PLOT 368,209:DRAW 360,216

```

```

14450 PLOT 366,209:DRAW 358,216
14460 PLOT 386,209:DRAW 394,216
14470 PLOT 388,209:DRAW 396,216
14480 RETURN
14490 REM-----the body-----
14500 PLOT 348,194:DRAW 408,194
14510 DRAW 420,160:DRAW 400,130
14520 PLOT 348,194
14530 DRAW 336,160:DRAW 354,130
14540 PLOT 356,180
14550 DRAW 349,160:DRAW 363,135
14570 PLOT 398,180
14580 DRAW 406,160:DRAW 392,135
14590 DRAW 400,130
14600 PLOT 363,135:DRAW 350,130
14610 RETURN
14620 REM-----the legs-----
14630 PLOT 355,135
14640 DRAW 363,80:DRAW 375,80
14650 PLOT 410,143:DRAW 402,80
14660 DRAW 390,80:DRAW 386,145
14670 PLOT 375,80:DRAW 378,145
14680 PLOT 355,145:DRAW 395,145
14690 RETURN
14700 REM-----the shoes-----
14710 PLOT 363,80:DRAW 345,70
14720 DRAW 375,70:DRAW 375,80
14730 PLOT 404,80:DRAW 422,70
14740 DRAW 390,70:DRAW 390,80
14750 RETURN
15000 REM all goes used up?
15010 f4=0
15020 IF e=10 THEN f4=-1
15030 RETURN
15040 IF a$="N" THEN f4=0
15050 IF a$<>"Y" AND a$<>"N" THEN 15020:
ELSE RETURN
16000 REM loser message
16010 LOCATE 2,10:PEN 2
16020 PRINT"Bad luck!"
16030 LOCATE 2,12
16040 PRINT"You are hung!"
16050 LOCATE 2,14:PEN 3
16060 PRINT"The word was"
16070 LOCATE 2,16:PRINT a$
16080 RETURN
18000 DATA CAT,TURBINE,PLATE,COMPUTER
18010 DATA
18020 DATA
18030 DATA

```

Program Improvements

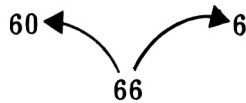
One of the unspoken rules in a computer company is that a programmer never finishes a program. The program can be developed to a stage where it is playable (the hangman game is at this stage now) but improvements are always possible. When a program has been written in a structured way it is normally quite simple to insert a few 'extra' features to make an otherwise great game even better. Below are a few suggestions that you might like to work on. Most of the suggestions are a matter of taste and have been left for you to implement.

- Better screen displays
- Difficulty level of words
- The better the player the less goes they are allowed
- Two player game

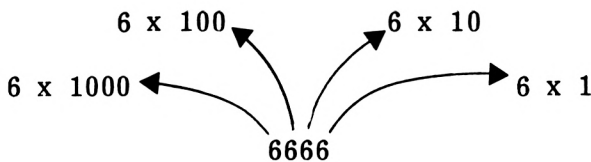
APPENDIX 1

Binary, Binary-Coded Decimal and Hexidecimal Notation

Counting systems in general use throughout the world the decimal system and this has been developed to count on past 10 and also below 1. In this standard, the digits to the left of a number are of greater value than those to the right for instance, in the number 66, the first 6 has a value 10 times the second, i.e.

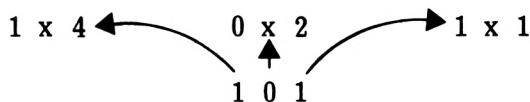


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit in a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater being multiplied by the base.

The computer, being basically electrical in operation, can only recognise two states, on or off ('0' and '1' respectively) and, thus uses the Binary system - base 2. Thus, any number in binary consists simply of 0's and 1's, or electrically speaking, offs and ons (or electronically, zero volts -OFF- and some volts -ON-). To count past one, the binary system must resort to place-value notation and, as with other cases, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e. $4+0+1=5$. Clearly the plethora of bases presents a problem when representing numbers as in base 10, '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers in that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

$$101_{10} = \text{One hundred and one in base ten.}$$

$$101_2 = \text{Five in base two.}$$

The present-day generation of home computers (1985-style) uses eight bit registers or memories and can, thus, represent numbers up to 11111111, i.e. in base 10:

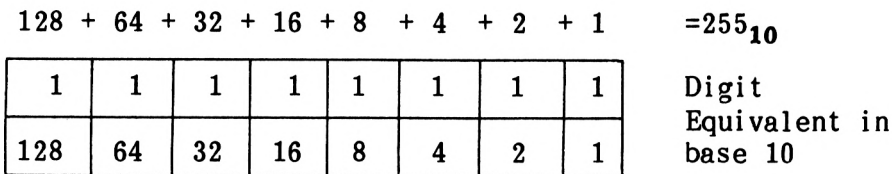
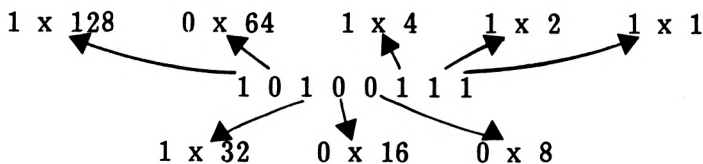


FIGURE A1.1

By way of example, let's take one more conversion - say, 10100111_2



$$\begin{aligned}
 \text{Thus } 10100111 &= 1x128 + 0x64 + 1x32 + 0x16 + 0x8 + 1x4 + 1x2 + 1x1 \\
 &= 128+32+4+2+1 \\
 &= 167_{10}
 \end{aligned}$$

Just to check your understanding, have a go at the following:

EXERCISE A1.1

Calculate the value of the following in base 10:-

- i) 0000011₂
- ii) 0000100₂
- iii) 1000000₂
- iv) 1000011₂
- v) 1011011₂
- vi) 01110011₂

Answers are given in the solutions chapter.

Hexadecimal

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1₁₀ and 1₂ there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255₁₀) and use a base of 256! what would you see as the objection to this? That's apart from the idea itself being a bit mind-bending! Time to think... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed so base 256 would need 256 digits!

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111₂ or 15₁₀ and this requires along with the 0, sixteen different symbols. The ones adopted for this job are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Decimal number
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Symbol

FIGURE A1.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NYBBLES. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) - see Figure A1.3.

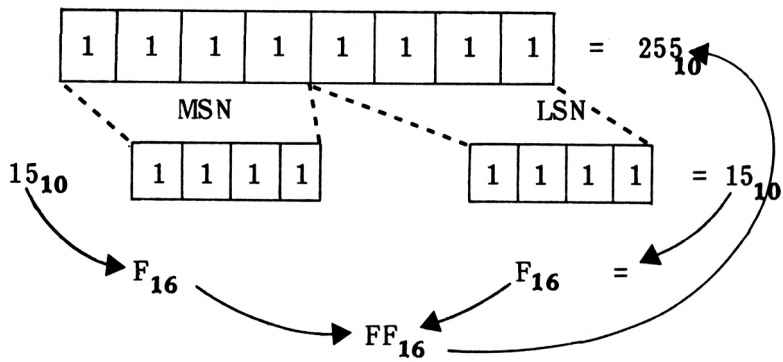


FIGURE A1.3

The system described, which uses sixteen symbols is, of course, given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

To gain an understanding of hex try the following:

EXERCISE A1.2

Calculate the value in decimal of the following:-

i) 0009 ₁₆	v) 000E ₁₆
ii) 0013 ₁₆	vi) 011A ₁₆
iii) 00A5 ₁₆	vii) 00EA ₁₆
iv) 0AAE ₁₆	

Answers are in the solutions chapter.

If you need more exercises in hex the Bin/Hex exercises program will supply them, select '1' for decimal to Hex practice and '4' for Hex to decimal. Remember <DELETE> to delete last entry and <RETURN> for MENU.

Binary-Coded Decimal

As well as decimal, binary and hexadecimal notations, one other system is used in computing - Binary-Coded Decimal. As its name suggests, it is a hybrid form with elements from both binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:

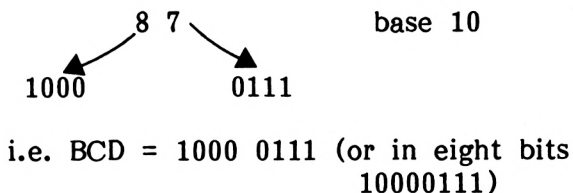


FIGURE A1.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10} = 1001_2$ thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A1.6 shows this, where 87_{10} is represented in BCD as 10000111_2 . This can give rise to ambiguity in that 10000111_2 in binary is 135_{10} . To overcome this, BCD representations will be given the notation 10000111 (BCD).

Using four bits of binary, it is possible to count up to 15_{10} (i.e. $1111_2 = 15_{10}$) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_2 and carries the 1 to the next nybble, i.e.

8_{10}	=	0000 1000	(base 2 BCD)
9_{10}	=	0000 1001	" " "
10_{10}	=	0001 0000	" " "
11_{10}	=	0001 0001	" " "

FIGURE A1.7

As you know all about BCD now! try the following:-

EXERCISE A1.3

Convert the following decimal numbers into BCD:

i) 4	v) 53
ii) 10	vi) 102
iii) 77	vii) 953
iv) 97	viii) 2579

Answers are given in the solutions chapter.

EXERCISE A1.4

Convert the following BCD numbers into decimal:-

- i) 0000 0001
- ii) 0000 1001
- iii) 0001 0101
- iv) 0010 0000
- v) 0100 1001
- vi) 1010 0011
- vii) 1001 0111
- viii) 1000 1000

Answers are given in the solutions chapter.

In the explanations given of the value of places in place-value notation a simplification was adopted in order to make these explanations clearer for the less mathematically inclined brethren! However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX 1.

With binary numbers, it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact, the multiplying factor is the base, raised to the power of its place starting with zero at the left. i.e. in binary:

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

Place
Previously stated
multiplication factor

Mathematically more
precise factor.

Thus the least significant bit is multiplied by 2^0 or 1. (If you are not sure of this try the direct program PRINT 2^0 .) The next bit is multiplied by 2^1 , and so on.

This rule holds for ANY base; let's apply it for hex, i.e. base 16:

$$\text{Least significant bit factor} = 16^0 = 1$$

$$\text{2nd most significant bit factor} = 16^1 = 16$$

$$\text{3rd most significant bit factor} = 16^2 = 256$$

$$\text{Most significant bit factor} = 16^3 = 4096$$

APPENDIX 2

Using The Data Recorder

Chapter two included a brief account of the **SAVE** and **LOAD** commands showing the user how to save a program onto tape and then load it back. The commands used were the simplest form of **SAVE** and **LOAD**. This Appendix will demonstrate extra features of the Amstrad's cassette data system.

Save

The **SAVE** command can be used to save three types of files, each type selected by a code.

Save "file name",A

When using **SAVE,A** a BASIC program is saved as an ASCII file. The **GUESSER** program in chapter two was saved as an ASCII file. If no file type is chosen then 'A' is the default value. The Amstrad (like most computers) saves BASIC programs in numeric form. Each character of a program is converted into its ASCII value. This ASCII value is, in turn, converted into an electronic sound which is stored onto the tape.

Save "file name",P

The **P** in this case stands for **Protect**. A program saved in this way cannot be loaded in the normal manner. To illustrate this point do the following:

- Type in: 10 REM PROTECTED PROGRAM
20 PRINT "I AM SAFE"
- SAVE "PROTECT",P
- The Amstrad will respond with
PRESS REC and PLAY then any key:■
- Press down the RECord and PLAY keys on the tape deck
and then hit the space bar
- The computer will display the following message:
Saving PROTECT block 1
- When the Ready message appears rewind the cassette and
type: LOAD "PROTECT"
- The Amstrad will report back
Press PLAY then any key:■
- Press the PLAY key on the tape deck and then hit the
space bar.
- The computer will begin looking for the program called
PROTECT. When it has found it you will see the
following message:
Loading PROTECT block 1
- When the Ready message appears the computer has
finished loading. All appears to be normal.
- Type: LIST
- Nothing is displayed. Your program has not been
loaded. Not only that but whatever was in the
computer's memory prior to loading has been deleted.

A program that is saved using protect can only be loaded back in one of two ways. The first of these is the CHAIN command.

CHAIN

The CHAIN command is a very special variation of the LOAD command. When CHAIN is used the loaded program automatically RUNS from a specified line number.

```
CHAIN "GUESSER",100
```

This will load the Basic program GUESSER and begin running the program from line 100. If no program line is specified then the program will start from the first program line. If the number used in the CHAIN command does not exist then the computer will report:

Line does not exist

and the attempted load will be unsuccessful.

The CHAIN command is used to load programs saved with Protection.

RUN

Another command that is used to load programs saved with Protection is the RUN command. When used to load RUN is followed by a file name. e.g.

RUN "PROTECT"

This will load the program PROTECT and RUN the program starting from the first line number.

Holding down CTRL and pressing the small ENTER key on the number pad causes the Amstrad to display the following:

RUN"
Press PLAY then any key:■

This will load and RUN the next program encountered on the tape.

Both CHAIN and RUN can be used in exactly the same way with protected and unprotected programs.

SAVE "file name",B

This form of SAVE is used to save the contents of specified memory locations. The Data is stored on tape as Binary data. The user has to type the location that saving is to begin at and how many locations to save. For example:

SAVE"MEMORY",B,55000,100

This will save the contents of all the memory locations from fifty five thousand to fifty five thousand one hundred.

There is one other argument that can be used with SAVE,B and this is the **entry point**. If this value is included then the file will begin execution from that point. For example:

```
SAVE"MEMORY",B,55000,100,55020
```

This will save the same block of memory and automatically RUN the data stored in that area of memory following a LOAD.

This form of save is used almost exclusively with machine-code programs. To find out about machine-code and assembly language, recommended reading is:

Beginners' Assembly Language Programming Course for the Amstrad.

This is part of the Dr. Watson series and is available from all good computer bookshops, or by mail order direct from the publishers, Honeyfold Software Ltd.

MERGE

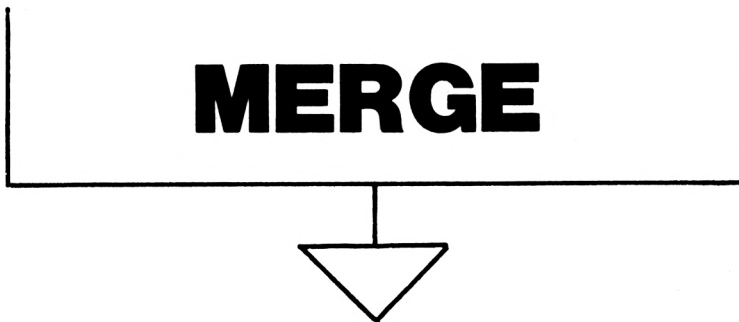
As well as simply loading a program it is possible to merge two programs together. One of the programs must be stored in memory and the other on tape. The MERGE command loads the program on tape line by line. Suppose the first line loaded is ten. The Amstrad checks to see whether the program in memory has a line 10. If it has then it is replaced by the line 10 loaded from tape, if no line 10 is present then the line from tape is used. Figure A2.1 demonstrates the principles of MERGE.

The program in the computer

```
10 REM COMPUTER
15 LET A=8
20 LET A=A+7
30 PRINT A
35 PRINT "THANK YOU"
```

The program on Tape

```
10 REM TAPE
20 LET A=A*A
30 PRINT "THANK YOU";A
40 END
```



The Final Program

```
10 REM TAPE
15 LET A=8
20 LET A=A*A
30 PRINT "THANK YOU";A
35 PRINT "THANK YOU"
40 END
```

A Diagrammatic Representation of the Merge Command

FIGURE A2.1

The new program is the program from tape and any lines of the program that was in the computer which do not have a corresponding line number in the tape program.

CHAIN MERGE

The CHAIN MERGE command is an interesting combination of both CHAIN and MERGE. The merge section works in the way outlined in Figure A2.1 and the CHAIN section works exactly the same way as the CHAIN command by itself.

```
CHAIN MERGE "ALBERT"
```

This will merge the cassette program ALBERT with the program currently in memory. Once merge is completed the program will automatically RUN starting at the first line of the program.

CHAIN MERGE "ALBERT",100

This will Merge the cassette program ALBERT with the program currently in memory. Program execution will start at line 100. Make sure that the line used in the CHAIN MERGE command exists. If it does not the Amstrad will report back:

Line does not exist.

and the program in memory will have been deleted.

The combination of CHAIN and MERGE can be used with a third argument. This is DELETE.

CHAIN MERGE "ALBERT",100,DELETE 300-

When entered this will cause the computer to Delete lines 300 onwards from the program currently in memory **BEFORE** the program on cassette is merged.

CAT

The last of the Amstrad cassette commands is CAT. CAT is short for CATalogue and is used to display the names of all files saved on the tape in the cassette drive.

After typing in CAT the Amstrad will respond with:

Press PLAY then any key: ■

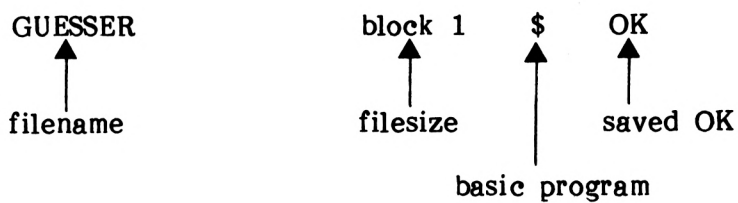
Once a key is pressed the Amstrad will begin searching through the tape. Every time a program is encountered the following will be displayed:

<filename> block <n> <filetype> OK

Where <file name> is the name of the file found. <n> is the block number and <file type> is one of the following file characters.

- \$ BASIC program
- % Protected BASIC file
- * ASCII text file
- & Binary file
- ' Protected Binary file

The OK message tells you that the program was saved okay. The CATalogue message for the tape on which "GUESSER" was saved would read:



The CATalogue Display

FIGURE A2.2

SOLUTIONS

EXERCISE 1.1

- Type EDIT 30
- Move the cursor until it is between PRINT and NAME\$
- Type in "Your name is ";
- Press ENTER
- You have successfully edited line 30

EXERCISE 1.2

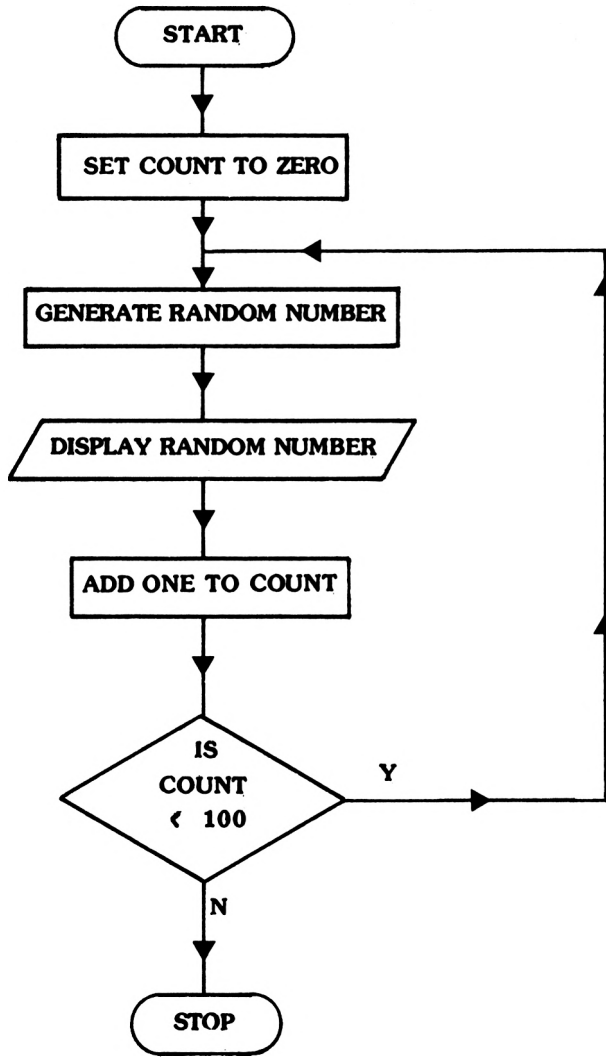
```
10 INPUT "What is your name "; NAME$
20 INPUT "How old are you "; AGE
30 PRINT "Your name is "; NAME$
40 PRINT "Your age is "; AGE
```

EXERCISE 1.3

```
10 CLS
20 INPUT "WHAT IS YOUR NAME";NAME$
30 CLS
40 LOCATE 15,10
50 PRINT "HELLO ";NAME$
```

EXERCISE 2.1

```
10 COUNT=0
30 RANUM=INT(RND*100)+1
50 PRINT RANUM
70 COUNT=COUNT+1
75 IF COUNT < 100 THEN 30
90 STOP
```



EXERCISE 2.2

```
10 FOR X=0 TO 100 STEP 3
```

EXERCISE 2.3

```
10 FOR X=10 TO 1 STEP -1
20 PRINT X
30 NEXT X
40 PRINT"FIRE!"
```

EXERCISE 2.4

One way is to change line 60.

```
60 IF GUESS=RANDOM THEN PRINT"WELL DONE-  
GUESS CORRECT ":PRINT"YOU TOOK";C;"GOES":  
GOTO 110
```

EXERCISE 2.5

```
100 IF COUNT <99 THEN PRINT"SORRY YOU'VE  
HAD YOUR SIX GOES ":PRINT"THE NUMBER WAS";  
RANUM
```

EXERCISE 3.1

```
10 MODE 0  
20 BORDER 13  
30 INK 0,19  
40 INK 1,17  
50 PAPER 0  
60 PEN 1
```

EXERCISE 3.2

```
10 MODE 0:INK 1,19  
20 PLOT 320,100  
30 DRAW 64,93,4
```

EXERCISE 3.3

```
75 IF A$="R" then 30
```

EXERCISE 3.4

```
3 IF X>640 OR X<0 THEN PRINT"NUMBER OUT  
OF RANGE":GOTO 2  
  
5 IF Y>400 OR Y<0 THEN PRINT"NUMBER OUT  
OF RANGE":GOTO 4
```

EXERCISE 3.5

```
10 MODE 1
20 R=57
25 PLOT 177,200
30 FOR A=0 TO 3.15
40 X= R*COS(A):Y=R*SIN(A)
50 DRAW 120+X,200+Y
60 NEXT A
```

EXERCISE 4.1

```
10 FOR X=1 TO 4
20 READ A,A$
30 PRINT A;A$
40 NEXT X
50 DATA 1,ONE,2,TWO,3,THREE,4,FOUR
```

APPENDIX ONE

EXERCISE A1.1

- i) 0000 0011 = 0+0+0+0+0+0+2+1
 = 3
- ii) 0000 0100 = 0+0+0+0+0+4+0+0
 = 4
- iii) 1000 0000 = 128+0+0+0+0+0+0+0
 = 128
- iv) 1000 0011 = 128+0+0+0+0+0+2+1
 = 131
- v) 1011 0111 = 128+0+32+16+0+4+2+1
 = 183
- vi) 0111 0011 = 0+64+32+16+0+0+2+1
 = 115

EXERCISE A1.2

- i) $0009_{16} = 0 \times 409 + 0 \times 256 + 0 \times 16 + 9 \times 1$
 $= 0 + 0 + 0 + 9$
 $= 9_{10}$
- ii) $0013_{16} = 0 \times 4096 + 0 \times 256 + 1 \times 16 + 3 \times 1$
 $= 0 + 0 + 16 + 3$
 $= 19_{10}$
- iii) $00A5_{16} = 0 + 0 + 10 \times 16 + 5 \times 1$
 $= 160 + 5$
 $= 165_{10}$
- iv) $0AAE_{16} = 0 + 10 \times 256 + 10 \times 16 + 14 \times 1$
 $= 2560 + 160 + 14$
 $= 2734_{10}$
- v) $000E_{16} = 0 + 0 + 0 + 14$
 $= 14_{10}$
- vi) $011A_{16} = 0 + 256 + 16 + 10$
 $= 282_{10}$
- vii) $00EA_{16} = 0 + 0 + 14 \times 16 + 10$
 $= 224 + 10$
 $= 234_{10}$

EXERCISE A1.3

- i) $4_{10} = 0100_2$ (BCD)
- ii) $10_{10} = 1 \times 10 + 0$
- iii) $77_{10} = 7 \times 10 + 7$
 $= 0111 \ 0111_2$ (BCD)
- iv) $97_{10} = 9 \times 10 + 7$
 $= 1001 \ 0111_2$ (BCD)
- v) $53_{10} = 5 \times 10 + 3$
 $= 0101 \ 0011_2$ (BCD)
- vi) $102_{10} = 1 \times 100 + 0 \times 10 + 2 \times 1$
 $= 0001 \ 0000 \ 0010_2$ (BCD)
- vii) $953_{10} = 9 \times 100 + 5 \times 10 + 3 \times 1$
 $= 1001 \ 0101 \ 0011_2$ (BCD)

$$\begin{aligned}
 \text{viii)} \quad 2579 &= 2 \times 1000 + 5 \times 100 + 7 \times 10 + 9 \times 1 \\
 &= 0010 \ 0101 \ 0111 \ 1001_2 \text{ (BCD)}
 \end{aligned}$$

EXERCISE A1.4

$$\begin{aligned}
 \text{i)} \quad 0000 \ 0001_2 \text{ (BCD)} &= 0 \times 10 + 1 \times 1 \\
 &= 1_{10}
 \end{aligned}$$

$$\begin{aligned}
 \text{ii)} \quad 0000 \ 1001_2 \text{ (BCD)} &= 0 \times 10 + 9 \times 1 \\
 &= 9_{10}
 \end{aligned}$$

$$\begin{aligned}
 \text{iii)} \quad 0001 \ 0101_2 \text{ (BCD)} &= 1 \times 10 + 5 \times 1 \\
 &= 15_{10}
 \end{aligned}$$

$$\begin{aligned}
 \text{iv)} \quad 0010 \ 0000_2 &= 2 \times 10 + 0 \times 1 \\
 &= 20_{10}
 \end{aligned}$$

$$\begin{aligned}
 \text{v)} \quad 0100 \ 1001_2 \text{ (BCD)} &= 4 \times 10 + 9 \times 1 \\
 &= 49_{10}
 \end{aligned}$$

$$\text{vi)} \quad 1010 \ 0011_2 \text{ (BCD)}$$

*** This is not a valid BCD number as the first nybble, $1010_2 = 10_{10}$, i.e. is greater than allowed in BCD.

$$\begin{aligned}
 \text{vii)} \quad 1001 \ 0111_2 \text{ (BCD)} &= 0 \times 10 + 7 \times 1 \\
 &= 7_{10}
 \end{aligned}$$

$$\begin{aligned}
 \text{viii)} \quad 1000 \ 1000_2 \text{ (BCD)} &= 8 \times 10 + 8 \times 1 \\
 &= 88_{10}
 \end{aligned}$$

INDEX

A

AND 2-24
Angles 3-19
Arithmetic 1-11
Array already dimensioned 4-8
Arrays 4-6
ASC 3-13
ASCII 3-13
Auto repeat 1-7

B

Binary A1-1 et seq
BORDER 3-2
Brackets 2-9
Break in 2-3

C

CAT 2-22, A2-7
CHAIN A2-2
CHAIN MERGE A2-5
Character sizes 3-1
CHR\$ 3-13, 4-19
Circles 3-19
CLS 1-24
Colour 3-3
Comma separator 1-16
Comparisons 2-23
Conditional loop 2-4
Connecting the Amstrad 1-1 et seq
Continuous loop
Copy 1-21
COPY key 1-19
COS 3-19
CTRL key 1-7
Cursor 1-4
Cursor keys 1-19, 3-12

D

DATA 4-2
Data exhausted 4-4
Data recorder 2-20, A2
Decision 2-5
Default INK values 3-5
DEL key 1-8
DELETE 3-11
DIM 4-8
Dimensioning arrays 4-8
Division (/) 1-11
DRAW 3-15
DRAWR 3-17

E

EDIT 1-18, 1-20
Edit mode 1-18
ELSE 2-17
Empty string ("") 3-11
ENTER key 1-9
Etcha sketcha 3-9 et seq

F

False 5-38
File types 2-23
Flags 4-9, 5-1 et seq
Flow charts 2-5
Formatting Screen displays 1-16
FOR...NEXT 2-7
FOR...NEXT...STEP 2-8

G

GOSUB 4-17, 4-18
GOTO 2-2 et seq
Graphics 3-1 et seq
Graphics grid 3-4, 3-5
Greater than (>) 2-6
Guess the number game 2-1 et seq

H

Hangman 5-1 et seq
Hangman picture 5-35, 5-36, 5-37
Hexadecimal A1-1 et seq

I

IF...THEN 2-3
IF...THEN...ELSE 2-17
INK 3-3
INKEY\$ 3-10
INPUT 1-14
Input/Output 2-5
INSTR 5-29
INT 2-2

K

Keyboard 1-5
Keyboard buffer 3-10

L

LEFT\$ 4-11
LEN 4-13
LET 1-11
Line numbers 1-13
LIST 1-15
LOAD 2-20, A2-1 et seq
LOCATE 1-23
Loops 2-1 et seq
Loop variable 2-3
LOWER\$ 4-26

M

Mathematical precedence 2-28
Memory locations 1-5 et seq
MERGE A2-4, A2-5
MID\$ 4-11
MODE 3-1 et seq
Modular programming 4-1 et seq
Multiple **INPUT** 1-22
Multistatement lines 2-11

N

NEW 1-24
NEXT 2-7
NOT 5-38
Not equal to (<>) 2-25
Null string ("") 3-11
Numerical Variable names 1-11

O

ON...GOTO 5-34
ON...GOSUB 5-34

P

PAPER 3-4
PI (3.14159265) 3-20
Pixel 3-7
PLOT 3-6
Point 3-7
Pointer (data) 4-4
Power up 1-4
PRINT 1-9
Print fields 1-17
Processes 2-5
Prompts 1-18

R

READ 4-2
Ready 1-10
REM 2-8
RENUM 4-9
RESTORE 4-5
RETURN 4-18
RIGHT\$ 4-11
RND 2-1
RUN 1-13
RUN" A2-3

S

SAVE 2-20, A2-1 et seq
Saving programs 2-20 et seq, A2-1 et seq
Semicolon separator 1-17
Separators 1-16, 1-17
SHIFT key 1-6
SIN 3-19
Slicing strings 4-10
Solutions 6-1 et seq
Spaces 1-8
Start up display 1-4
STEP 2-8
STOP 2-4
String slicing 4-11
String variable names 1-11
Strings 1-10
Structure 4-1 et seq
Subroutines 4-18 et seq
Subscripts out of range 4-7
Switching on 1-1
Syntax error 1-9

T

Terminators 2-5
The Amstrad keyboard 1-5
THEN 2-3
TO 2-7
Top down programming 5-1
True 5-38

U

Unexpected next 2-10
UPPER\$ 4-27
Using the data recorder A2-1 et seq

V

Variable names 1-11, 1-12
Variables 1-11 et seq

OTHER BOOKS FOR THE AMSTRAD

Watson's Workbook

Amstrad BASIC Book 2 - Continuing BASIC

ISBN 0 907792 40 5

Dr Watson Series

Amstrad Assembly Language Programming

ISBN 0 907792 41 3



AMSTRAD CPC 464 Starting Basic

This book is designed for the complete newcomer to computers and the programming language BASIC.

The book takes the reader step-by-step through Amstrad BASIC. Commands are thoroughly examined as they are introduced and are incorporated in numerous example programs. Readers will already know enough to write and run their own short programs by the end of chapter one!

The programs in this book have been written to illustrate the use of structured programming, making programs easier to write and understand.

This book is a complete course in itself, but when you wish to know more about the Amstrad, book 2, 'Continuing Basic', is also available.



Glentop Publishers
Standfast House, Bath Place,
High Street Barnet, London

ISBN 0-907792-39-1



STANDARD GRAMMAR

BOOK I

STANDARD GRAMMAR

BOOK II

STANDARD GRAMMAR

BOOK III

STANDARD GRAMMAR

BOOK IV

STANDARD GRAMMAR

BOOK V

STANDARD GRAMMAR

BOOK VI

STANDARD GRAMMAR

BOOK VII

STANDARD GRAMMAR

BOOK VIII

STANDARD GRAMMAR

BOOK IX

STANDARD GRAMMAR

BOOK X

AMSTRAD

CPC



MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA



<https://acpc.me/>

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.

[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.

[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.