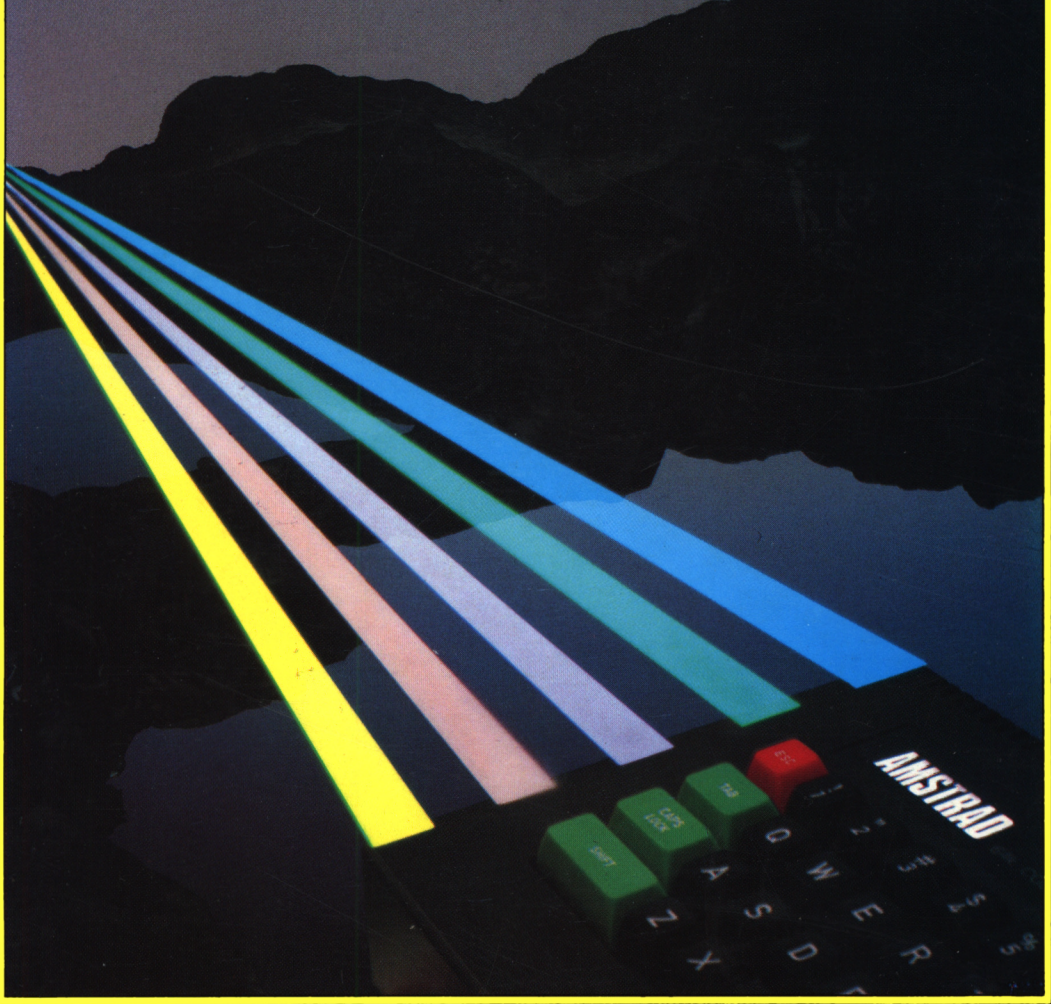




AMSTRAD BOOK 2

SOUND, GRAPHICS & DATA HANDLING



Sean Gray

AMSTRAD BOOK 2

Sound, Graphics & Data Handling

Sean Gray

AMSTRAD

BOOK
2

Sound, Graphics & Data Handling

by
Sean Gray



Glentop Publishers Ltd

JULY 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 prior permission from the publishers, with the exception of material entered and executed on a computer system for the reader's own use.

ISBN 0 907792 40 5

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

CONTENTS

	Page
Introduction	7
CHAPTER 1	9
LET ● PRINT ● INPUT ● LOCATE ● Loops ● GOTO ● IF... THEN ● FOR... NEXT ● WHILE... WEND ● READ... DATA ● RESTORE ● GOSUB... RETURN ● ON... GOSUB ● ON... GOTO ● LINE INPUT ● INKEY\$ ● Arrays ● String handling ● LEN ● RND ● RANDOMIZE	
CHAPTER 2	29
Modes ● BORDER ● INK ● PAPER ● PEN ● Graphics ● PLOT ● PLOTR ● DRAW ● DRAWR ● MOVE ● MOVER ● Triangles ● Circles ● TEST ● Windows ● Streams ● WINDOW SWAP ● CLG ● TAG ● TAGOFF	
CHAPTER 3	57
SOUND ● Channel ● Rendezvous ● Hold ● RELEASE ● Flush ● Tone ● Length ● Volume ● Volume Envelope ● ENV ● Tone Envelope ● ENT ● Noise Period	
CHAPTER 4	77
Video-Game Mechanics ● Moving Albert ● Firing ● The Enemy ● Multiple Enemies ● Random Start ● Simultaneous Movement ● Two-Dimensional Arrays ● TIME ● Improving The Display ● Sound ● Best Score	
CHAPTER 5	113
Data Handling ● Sequential Cassette Filing ● Fields, Records and Files ● Initialisation ● Create File ● Add, Delete or Alter Record ● Saving Records ● OPENIN, OPENOUT, CLOSEIN and CLOSEOUT ● PRINT#9 ● INPUT#9 ● Loading File ● Display Records ● Search File ● Sort File ● Exit	

APPENDIX 1	157
Arithmetic Statements	
APPENDIX 2	163
Using the Data Recorder and Disc Unit ● SAVE ● CHAIN ● RUN ● MERGE ● CHAIN MERGE ● CAT	
APPENDIX 3	169
Printers ● LIST#8 ● PRINT#8	
APPENDIX 4	171
Character Codes	
APPENDIX 5	173
Special Control Codes	
APPENDIX 6	179
Defining Characters ● Function Keys	

INTRODUCTION

Welcome to the second book for the Amstrad CPC 464 in the Watson's Workbook series. The aim of this book is to demonstrate the Amstrad's musical features and the Amstrad's graphics commands. As well as this the book includes a section on Databases and sequential files using the Amstrad's data recorder.

This book is aimed mainly at two groups of people: those who have recently completed book one and wish to continue learning about their Amstrad, and those who already have a fair understanding of BASIC, perhaps through having used other home computers. Throughout this book a moderate level of competence in BASIC programming is assumed.

Chapter one is devoted to explaining some of the more general BASIC commands and any variations that the Amstrad has on them. Readers are advised to read through this, because the rest of the book assumes that the reader fully understands all the commands covered in Chapter one. Once the 'refresher course' is finished, the book moves on to Chapter two which explains the Amstrad's graphics commands. When each command is introduced to the reader it is accompanied with at least one example program demonstrating how the command works and how the reader can use it in programs.

Chapter three investigates the Amstrad's musical capabilities. Not only are the sound and envelope commands explained in depth, but the reader is also shown how to use sound queues and play harmonies. Having learnt about graphics and sound the reader is then shown how to utilise both of them together. Chapter four takes the reader through various game programming techniques and shows the reader how to develop a game program.

The last chapter, Chapter five teaches the reader the principles of databases and cassette filing systems. The program developed in this chapter is a database program that allows the user to create, edit and delete files and save and retrieve them to and from tape. The six Appendices cover the following range of subjects: the Amstrad's mathematical commands; tape and disc output; printers; the Amstrad's character set; the Amstrad's own special characters (\emptyset –31); and defining your characters. By the time readers have finished this book they will be able to write their own!

The completion of this book has taken a vast amount of effort from many corners. It has been left to me, the 'author', to take all the credit, but a special mention must go to all those involved in the editing stage. These people are Eddy Maddix, Jeremy Irwin, Paul Gilbert and Martin Thompson. I'm sure these people will be just as pleased as me to see the book in print.

Sean Gray

June 1985

Chapter 1

The object of this chapter is to familiarise the user with some of the more common Amstrad commands. If you are already familiar with BASIC to some extent, you might prefer to skim through this chapter fairly rapidly and go on to Chapter 2.

Nearly all home computers use a language called BASIC (Beginners All-purpose Symbolic Instruction Code). Because most home computers use the same language, many commands work in the same way on different machines. The three most common BASIC commands are LET, PRINT and INPUT. These three commands operate in the same way, on almost all home computers.

LET

The LET command is used to assign data to a variable, for example

```
LET A=8
```

This is interpreted by the computer as meaning:

‘Take an area of memory and call it A.
Store the value eight in this variable’

The Amstrad allows the user to dispense with the LET command and use an implied LET. Thus

```
A=8
```

produces the same effect. The Amstrad allows you to have variable names of up to forty characters.

PRINT

The PRINT command is used to display information on the screen, be it the value of a variable, or a string of characters. Typing:

```
PRINT "HELLO, GOOD EVENING"
```

causes the Amstrad to display on the screen:

```
HELLO, GOOD EVENING
```

i.e. exactly what was between the quotation marks (“”). The characters between the quotes are referred to as a ‘literal string’. The PRINT command can also be used to display the contents of a variable. For example:

```
PRINT A
```

is interpreted as:

```
‘Display the contents of variable A’
```

PRINT can also be used to display a mixture of literal strings and variables; e.g.

```
PRINT "THE VALUE OF A IS";A
```

INPUT

INPUT is used to communicate information from the user to the computer. The information is typed in by the user and stored in a specified variable, e.g.:

```
10 INPUT A
```

When this is run, a question mark appears on the screen, and the user is then expected to type something in. Because A is a numeric variable the user should type in a number. INPUT can also be used with string variables:

The Amstrad’s special form of BASIC allows multi-variable inputs and mixtures of PRINT and INPUT, called ‘prompts’.

Multi-variable input:

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

INPUT with prompt:

```
INPUT"PLEASE TELL ME YOUR NAME ";NAME$
```

LOCATE

The three previous commands, LET, INPUT and PRINT, were all general BASIC commands that work on most computers. However, LOCATE is available on very few and the way in which it works on the Amstrad is unique. LOCATE is used to determine where the next character is to be printed. The Amstrad’s screen is divided into a number of small sections called ‘cells’. When you switch on, the screen has forty cells across and twenty-five down. Each cell is one character in size. The LOCATE command is used to point to individual cells.

```
LOCATE 6,4:PRINT"OVER HERE!"
```

This moves the cursor six characters across and four characters down. It is at this point that the Amstrad starts to print 'OVER HERE!', the O being in the sixth cell across and the fourth down.

Loops

A loop is part of a program that is repeated over and over again. The Amstrad has four main types of loops, GOTO, IF...THEN, FOR...NEXT and WHILE...WEND.

GOTO

GOTO is a BASIC command that directs a program's flow from one program line to another. If the line number in the GOTO command is less than the line number in which the GOTO appears, e.g. 80 GOTO 10, then a loop is created. Program 1.1 demonstrates this form of loop, known as a 'continuous' or 'unconditional' loop:

PROGRAM 1.1

```
10 CLS
20 count=count+1
30 LOCATE 6,12
40 PRINT"work day";count
50 GOTO 10
60 LOCATE 4,14
70 PRINT"You work too hard. Take a holiday"
```

The 'CLS' command CLears the Screen – i.e. it removes everything from the screen.

This form of loop is called 'continuous' because it will not stop without outside interference, i.e. the computer is switched off or the user breaks into the program by pressing the 'ESC'ape key twice. The loop involves line 10 (CLear Screen), 20 (add one to count variable), 30 (position the text cursor in the 6th column and 12th row), 40 (print message) and line 50 (go back to line 10). Lines 60 and 70 will never be executed, the program flow being directed away from them via the GOTO on line 50. All work and no play makes Jack a dull boy indeed!

IF...THEN

The IF...THEN statement can be used to direct a program's flow depending upon whether or not a stated condition is met. If the specified condition is met then another BASIC command (any of the available BASIC commands) can be executed. For example if Jack was allowed one day's holiday for every ten days at work the program would loop around like this:

PROGRAM 1.1(a)

```
10 CLS
20 count=count+1
30 LOCATE 6,12
40 PRINT"work day";count
50 IF count<10 THEN GOTO 10
60 LOCATE 4,14
70 PRINT"You work too hard. Take a holiday"
```

The '<' symbol on line 50 means 'is less than', so that line reads 'if count is less than 10 then go to line 10'. Now the program loops around lines 10, 20, 30, 40 and 50 until the value of COUNT is ten. Once COUNT reaches that value the condition on line 50 is no longer true (i.e. COUNT is not less than ten) and the loop finishes. Lines 60 and 70 are executed only when the loop is finished. Thus the loop is known as a 'conditional loop'.

An extension of the IF...THEN statement can be used to do something even if the condition is false. This uses the ELSE command. ELSE is placed at the end of an IF...THEN statement and can be followed by any BASIC command except another ELSE. For example, line 50 of Program 1.1(a) can be changed to that of Program 1.1(b) below.

PROGRAM 1.1(b)

```
50 IF count<10 THEN GOTO 10 ELSE CLS
```

The ELSE section is only performed when the condition preceding it is false. In this example the screen will be cleared before printing 'You work too hard....'. After ten runs through the loop the value of COUNT is ten, so the condition on line 50 is false and the ELSE section is executed. The ELSE part can be regarded as an independent program line, which is only reached when the preceding condition is false. In English, line 50 means:

'If the condition is true then goto line 10 else clear the screen'

FOR ...NEXT

A FOR...NEXT loop is a loop that is designed to repeat a section of program a specified number of times. The section of program repeated is that between the FOR and NEXT statements. Program 1.1(c) produces the same effect as Program 1.1(b) but using a FOR...NEXT loop instead of the IF...THEN. The indented lines are those which are repeated.

PROGRAM 1.1(c)

```
10 FOR count=1 TO 10
20 CLS
30 LOCATE 6,12
40 PRINT"work day";count
50 NEXT count
60 CLS
70 LOCATE 4,14
80 PRINT"You work too hard. Take a holiday."
```

The big advantage of using FOR...NEXT loops instead of IF...THEN is that they are much faster and easier to use.

The FOR...NEXT loop in Program 1.1(c) starts at one and counts up to ten in steps of one. The FOR...NEXT command can be used to count from any value to any other value with any increment, using the 'STEP' statement. For example:

```
FOR X=1 TO 30 STEP 3
```

will start the loop at one and count up to 30 in steps of three. Thus if line 10 of Program 1.1(c) was changed to that of Program 1.1(d) the loop would still be performed ten times but the numbers displayed would be 1,4,7,10 etc. up to 28.

PROGRAM 1.1(d)

```
10 FOR count=1 TO 30 STEP 3
```

To count down with a FOR...NEXT loop, you can use a negative step size (e.g. minus one or whatever is appropriate). By replacing line ten with that of program 1.1(e) the loop will execute ten times, counting backwards.

PROGRAM 1.1(e)

```
10 FOR count=0 TO -9 STEP -1
```

As well as being able to have positive or negative step values, you can also use fractional values. If line 10 is again changed the loop will run ten times, this time starting at 0.1 and counting up to 0.2 in steps of 0.01:

PROGRAM 1.1(f)

```
10 FOR count=0 TO 0.2 STEP 0.015
```

WHILE...WEND

This loop structure is similar to IF...THEN and FOR...NEXT. The WHILE command initiates the loop if a specific condition is true and WEND specifies the end of the loop. However, when the condition is no longer true, it is the WHILE command that terminates the loop NOT the WEND, which is just a marker. The WHILE and WEND commands appear on separate program lines. All the lines between are executed WHILE the given condition is true. In the example of Program 1.1(g) below, lines 20 to 50 form the WHILE...WEND loop.

PROGRAM 1.1(g)

```
10 a=1
20 WHILE a<8
30 PRINT a
40 a=a+1
50 WEND
```

The program will display the numbers one to seven, lines 30 and 40 being repeated until A is no longer less than eight. If line 40 was removed and the program re-run only the number one would be displayed. In fact, one would be displayed ad infinitum, or at least until you press ESCape twice. The WHILE condition states that whilst A is less than eight, the value in A will be printed. The WHILE loop is ended when A is not less than eight. If line 40 was deleted the value of A would always be one and the condition would never be met. Using the WHILE...WEND commands the whole 'GOTO WORK' program can be rewritten as follows:

PROGRAM 1.1(h)

```
10 CLS
20 count=count+1
30 PRINT"work day";count
40   WHILE count=10
50     PRINT"holiday time!"
60     FOR x=1 TO 2000:NEXT x
70     PRINT"holidays over back to work"
80     FOR x=1 TO 500:NEXT x
90     count=0
100    WEND
110 GOTO 20
```

The indented program lines are those of the WHILE...WEND loop. When the program is run the program loops around lines 20,30 and 110 until the value of COUNT equals ten. Then lines 40 to 100 are executed, first telling the user that it is time for a holiday and then, after a delay, that it is time to go back to work. The whole section from line 40 to 100 is only executed when the value of COUNT is equal to ten.

A WHILE command depends on a WEND in the same way that FOR needs a NEXT. If a WHILE was without its WEND the following error message would be displayed:

```
WEND missing in XXX
```

Where XXX would be the line number of the WHILE command. If, on the other hand a WHILE was missing then the following error message is displayed:

```
Unexpected WEND in XXX
```

Where XXX is the line number where the WEND command was found.

READ...DATA

The READ...DATA structure is used to assign information to a variable or variables. The information to be assigned is specified in a DATA statement. For example, Program 1.2 uses READ to assign a value to the variable NAME\$. The DATA statement on line 900 contains the information that is to be stored in NAME\$.

PROGRAM 1.2

```
10 READ name$
20 PRINT name$
30 DATA amstrad
```

When this program is RUN the name AMSTRAD is displayed on the screen. Line 10 tells the computer to READ the DATA statement, take the first value there and store it in the variable NAME\$. In this case the first piece of data is 'AMSTRAD'. This is stored in NAME\$ and then displayed on the screen via line 20.

The advantage of assigning variables using READ rather than LET is that READ allows variables to be set to a non-specific number of values, for example:

PROGRAM 1.2(a)

```
10 FOR x=1 TO 20
20 READ age
30 PRINT age
40 NEXT x
900 DATA 64,12,17,33,29,103,27,39,20,93
910 DATA 34,43,24,21,18,11,57,40,10,88
```

Program 1.2(a) assigns twenty different values to the variable AGE. The twenty values are stored as data on lines 900 and 910 and assigned to AGE by the READ command in line twenty. READ can be used to assign more than one variable. Program 1.2(b) Reads in both a person's name and age. Care must be taken when using multi-variable READ statements such as that below to ensure that the right DATA goes to the right variable. If an attempt is made to READ a string into a numeric variable the following error message is displayed:

Syntax error in XXX

where XXX is the line with the DATA statement.

PROGRAM 1.2(b)

```
10 FOR x=1 TO 20
20 READ name$,age
30 PRINT name$,age
40 NEXT x
900 DATA sean,64,samantha,12,jamie,17,leon,33
910 DATA alayne,29,abraham,103,dennis,27
920 DATA diana,39,veronica,20,karen,93,cathy,34
930 DATA peter,43,john,24,paul,21,george,18
940 DATA richard,11,christine,56,sheila,40
950 DATA james,10,geraldine,68
```

RESTORE

In all the examples of READ and DATA shown so far the exact amount of data required is provided. Line 10 of Program 1.2(b) sets up a loop to read in twenty sets of data (twenty names and twenty ages) and the data statements from line 900 to 950 provide twenty sets of data. Suppose a line 50 was added to the program and this line tried to read in another set of data (i.e. another name and age). What would happen? Well type it in and find out.

PROGRAM 1.2(c)

```
50 READ name$,age
```

When run, as well as the names and ages, the following message is displayed by your Amstrad:

DATA exhausted in 50

All the data had been read in by the FOR...NEXT loop, so when an attempt was made to read even more data the computer objected by saying that there was no more data left. The RESTORE command tells the computer to start READING data from the very first DATA statement. By adding a RESTORE command, Program 1.2(d) will print out the twenty names and ages and then the first name and age will be printed again.

PROGRAM 1.2(d)

```
45 RESTORE
50 READ name$,age
60 PRINT name$,age
```

Using RESTORE in its simplest form, as in line 45 of Program 1.2(d), sets the data pointer back to the beginning of the first DATA statement. There is a variation on the RESTORE command that allows the user to specify which data line to start READING from. By way of example, add line 15 of Program 1.2(e) to the program currently in memory and run it.

PROGRAM 1.2(e)

```
15 RESTORE 910
```

Each time through, the data pointer is set to point to the first item of data on line 910. This has the effect of displaying 'ALAYNE' and '29' twenty times. When the loop is completed, the RESTORE on line 45 sets the data pointer back to the very first item of data (which is on line 900) and SEAN,64 is displayed. When using RESTORE with a line number, make sure that the line exists or a 'Line does not exist' error will be displayed.

GOSUB...RETURN

GOSUB is a command that temporarily directs the flow of a program to another section of program, known as a subroutine. The subroutine is executed until a RETURN statement is encountered. When this happens the program's flow is returned to the BASIC statement AFTER the GOSUB.

PROGRAM 1.3

```
10 PRINT"Hello everybody"
20 GOSUB 100
30 PRINT"I cannot hang around"
40 PRINT"talking all day. Goodbye!"
50 STOP

110 PRINT"How are you today?"
120 RETURN
```

When run the program will execute line ten and then GO to the SUBroutine at line 100, PRINT 'How are you today?' and RETURN to line 30. The program will then execute lines 30,40 and 50. Line fifty STOPS the program; if this was removed then the computer would run into the subroutine after executing line 50. When the RETURN on line 120 was encountered again the following error message would be displayed

Unexpected RETURN in 120

This is because the subroutine was not entered via a GOSUB command the second time. Thus when the RETURN was encountered the program had nowhere to return to and consequently an error was reported.

ON...GOSUB

The ON...GOSUB command is a type of conditional GOSUB, depending upon the value of a variable. For example:

```
ON A GOSUB 100,200,300
```

If the value of A is one then the subroutine at line 100 is called but if A is two then the subroutine at line 200 is called. If the value of A is decimal (e.g. 1.4) then the number will be rounded to the nearest whole number (in the example 1.4 rounded equals 1, as would 1.499 but not 1.5 or 1.6). The program is directed to the Ath routine, (in this example the first line number in the ON GOSUB command).

ON...GOTO

This works in a similar way to ON...GOSUB with the exception that the program does not GOTO a subroutine. This means that the program does not expect to see a RETURN statement.

```
ON A GOTO 100,200,300
```

Just like the ON...GOSUB command, the program will only be directed to line 100 if the value of A is calculated (e.g. 1.42::1, 1.7::2 etc) as one. If A is two or three then the program is directed to lines two hundred and three hundred respectively. Any value of A greater than 3.499999999 will result in the ON...GOTO being ignored. To illustrate the ON...GOSUB and ON...GOTO commands type in the following program:

PROGRAM 1.4

```
10 CLS
20 FOR x=1 TO 4
30 ON x GOSUB 100,200,300
40 PRINT"loop";x
50 NEXT x
60 PRINT"finished"
70 END

100 PRINT"This is subroutine 1"
110 RETURN
200 PRINT"This is subroutine 2"
210 RETURN
300 PRINT"This is subroutine 3"
310 RETURN
```

When run, Program 1.4 produces the following display:

```
This is subroutine 1
loop 1
This is subroutine 2
loop 2
This is subroutine 3
loop 3
loop 4
finished
```

FIGURE 1.1

The program chugs merrily away GOSUBbing on the value of X, but when X equals four there is nowhere to GOSUB to, so only 'LOOP 4' is printed. Remember that ON...GOSUB and ON...GOTO work in a similar way. Program 1.5 shows a similar example using ON...GOTO. In this case each GOTO is terminated with a STOP command.

PROGRAM 1.5

```
10 CLS
20 INPUT "A number (1-4)";a
30 ON a GOTO 100,200,300,400
40 PRINT "The number you typed in was out
  of range"
50 PRINT "Please try again"
60 GOTO 20

100 PRINT "Number one"
110 STOP

200 PRINT "Number two"
210 STOP

300 PRINT "Number three"
310 STOP

400 PRINT "Number four"
410 STOP
```

When running this program, try typing in an input value of 1.3 or something similar. What happens is that the value is rounded up or down to the nearest integer. Thus inputting a value of 1.9 would result in the second GOTO being executed.

Inputs: Variations on a Theme

The INPUT command is a standard BASIC command and all of the major home computers use INPUT in the same way. There is another thing that all home computers have in common: they all have other input commands that are specific variations of the INPUT command.

LINE INPUT

When using the standard INPUT command, if you wanted to input a comma then the comma would have to be enclosed in quotes, as in the following example:

```
INPUT A$:PRINT A$
? ", "
'
Ready
```

Although quotes were typed in, they were ignored by the computer and A\$ was assigned the value between quotes, in this example a comma. However, suppose you wished to input a quotation mark ("). You could not enclose it in quotes because the computer would take the second quote as being the end of the input. Upon encountering a third quote the computer would report an error.

This can create problems with some forms of input. Suppose we wished to input:

and then she said, "this is what she said."

The problem of a comma in the middle of a string cannot be resolved by enclosing the string in quotes because the string itself includes quotes. In circumstances like this the LINE INPUT command should be used. This command performs a literal input, exactly what you type in is assigned to the variable as long as the overall length of the string does not exceed 255 (two hundred and fifty five) characters. Like INPUT, the LINE INPUT command can contain a prompt, but, unlike INPUT, you cannot have a multi-variable LINE INPUT command.

You can have

```
10 LINE INPUT A$
```

or

```
10 LINE INPUT"TYPE IN 20 COMMAS";A$
```

but not

```
10 LINE INPUT A$,B$
```

Another difference between LINE INPUT and INPUT is that no question mark prompt appears when LINE INPUT is used: only the cursor appears.

INKEY\$

The second variation of INPUT is the INKEY\$ command. Whereas LINE INPUT allowed you to type in a string with a length of up to two hundred and fifty five characters, INKEY\$ allows only single characters to be input. The INKEY\$ command looks at the 'keyboard buffer', that section of the computer's memory that records which key is being pressed, and reports on what it finds there. To demonstrate this, type in and run Program 1.6.

PROGRAM 1.6

```
10 CLS
20 a$=INKEY$
30 PRINT a$
40 GOTO 20
```

This program continually scans the keyboard buffer and reports back on what it finds. When first run, nothing is displayed because the keyboard buffer is empty. Then as you begin to press keys the characters instantly appear on the screen, scrolling upwards rapidly. Note that this implies two things: firstly, when using INKEY\$ you do not have to press the ENTER key. Secondly it implies that the INKEY\$ statement on line 20 does not wait for a letter to be pressed, unlike INPUT. It looks at the keyboard buffer and if it finds nothing (i.e. you haven't pressed a key) then the value of A\$ is nothing (i.e. a null string ""). The program then progresses onto line 30 where nothing (the value of A\$) is printed, on a new line on the screen. The only time you see lines 20, 30 and 40 working is when a key is pressed but they are continually executed even when no key is pressed. To illustrate this point add line 25 (program 1.6(a)), this checks to see if the value of A\$ is nothing (i.e. "") if so it is changed to equal a hash character (i.e. "#").

PROGRAM 1.6(a)

```
25 IF a$="" THEN a$="#"
```

When run, the program will display a continuous stream of hashes broken only by the pressing of a key. Press any key as fast as you can over and over again. Notice how many #'s are displayed between each of your characters? This demonstrates how fast the keyboard buffer is checked.

The INKEY\$ command is especially nice to use when a program needs to be halted until the user is ready to continue. For example, a program might need to wait until a player has read the rules or adjusted a joystick according to a game's requirements. The program below also uses the 'REM' command, short for 'REMark' or 'REMinder'. This is used simply to add comments or remarks to programs. The computer ignores anything on a line following a REM statement.

PROGRAM 1.6(b)

```
10 REM An interesting use for INKEY$
20 CLS
30 PRINT"BLAH, BLAH, BLAH"
40 PRINT"BLAH, BLAH, BLAH"
50 PRINT"Press any key to continue"
60 IF INKEY$="" THEN 60
70 CLS
```

Notice that the INKEY\$ command does not require a variable but can be tested directly. Line 60 reads:

‘If the value of the keyboard buffer is
nothing then go and check it again’

Arrays

Often in computing, large amounts of data need to be stored in memory and be readily accessible at any time. Using an array allows large amounts of data to be stored by setting up a variable which is split into separate elements, one element per data item.

Setting Up Arrays

Before attempting to use arrays, you must know what type of data you have and the amount of data to be stored. For example, if you want to store the ages of six children so that you can find their average age, you will need room for six items of numeric data. As the data is numeric the following statement would provide the appropriate space:

```
DIM AGE(6)
```

This will DIMension the array ‘age’ to hold six values. The ages of the children must now be put into the array:

```
AGE(1)=1st child's age
AGE(2)=2nd child's age
etc.
```

Notice that the ‘subscript’, the value in the brackets used to access elements of the array, has an increment of one, so the data can be put in the array using a FOR...NEXT loop, like this:

PROGRAM 1.7

```
10 DIM age(6)
20 FOR i=1 TO 6
30 READ b
40 age(i)=b
50 NEXT i
900 DATA 11,10,13,12,14,17
```

The data is now stored in the array and can be accessed whenever required. Thus to calculate the average age of the children, add the following lines.

PROGRAM 1.7(a)

```
60 av=0
70 FOR i=1 TO 6
80 av=av+age(i)
90 NEXT i
100 av=av/6
110 PRINT "Average age=";av
```

There are two important things to remember about arrays. If an array has not been DIMensioned then a default DIM value is used. The default DIM value is ten; if you go beyond this the computer will report a 'Bad subscript' error. The second point is that upon DIMensioning an array each array element is assigned either zero or the null string "" depending upon whether the array is a numeric or a string array respectively.

String Handling Commands

LEFT\$, RIGHT\$ and MID\$

Amstrad's BASIC provides several ways to chop up strings. Two of these are LEFT\$ and RIGHT\$. These simply chop 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 to be chopped up and 'N' is the number of characters to remove.

In the above 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$
```

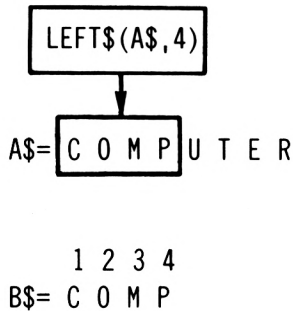


FIGURE 1.1

The **RIGHT\$** statement works in much 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 selectively chop away 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. **MID\$** 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)

knows that a string is to be dissected and the result stored in the variable C\$.

- When the computer sees 'C\$::MID\$(...)' it knows that a string is to be dissected and the result stored in the variable C\$.
- It then carries on and sees MID\$(A\$...). It translates this into 'find A\$ and prepare to operate on it'.
- Next it sees the '4' in MID\$(A\$,4,) 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 out 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' should be whole numbers: any decimal value will be rounded to the nearest whole number. MID\$ will cut out a part of the string A\$ starting at character number 'START' and of length 'LENGTH' characters. Diagrammatically, MID\$ appears as:

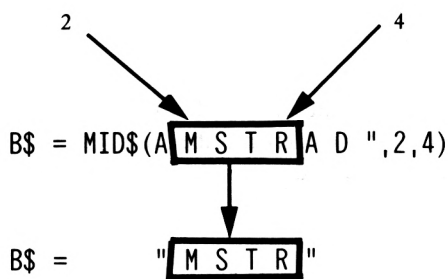


FIGURE 1.2

Using these 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 1.8 shows this in action as MID\$ reads step by step through a word, printing each letter as it goes.

PROGRAM 1.8

```
10 LET a$="computer"  
20 FOR x=1 TO 8  
30 b$=MID$(a$,x,1)  
40 PRINT b$  
50 NEXT x
```

One of the problems with Program 1.8 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, Amstrad 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 1.8 can now be rewritten so that the loop will run through the correct number of times whatever the string length, as in Program 1.8(a).

PROGRAM 1.8(a)

```
20 FOR x=1 TO LEN(a$)
```

Whatever the string that is assigned to A\$, the loop will now always handle it.

Randomly Amstrad

The Amstrad has two commands that deal with random numbers: RND and RANDOMIZE. The first produces the random number and the second determines which 'random' numbers RND actually produces.

RND

RND is a special variable which is set by the Amstrad (this type of variable is called a 'system' variable). To see what your Amstrad has set RND to type in:

```
PRINT RND
```

The author's machine produced 0.657773343'; no doubt your own Amstrad has reported something completely different. No matter what value your computer displayed, it will be a number with nine decimal places. Now for a startling prediction: if you switch your computer off and on and then 'PRINT RND' the random number displayed will be '0.271940658'. Try it and see! When producing random numbers, the Amstrad looks into a special part of its memory in which is stored a value called a 'seed'. This seed is used by the Amstrad in some impenetrable calculation to produce a random number. The value of the random number depends upon the value of the seed. Random numbers which aren't truly random, such as computer-generated random numbers, are known as 'pseudo-random' numbers.

RANDOMIZE

This command allows you to reset the seed to any value of your choice. To illustrate the use of RANDOMIZE, type in the following program (switch your computer off and on before typing in Program 1.9)

PROGRAM 1.9

```
10 CLS
20 FOR x=1 TO 5
30 a=RND
40 PRINT a
50 NEXT x
```

If you switched off your computer before typing that in, you will see this display the first time the program is run:

```
0.271940658
0.528612386
0.021330127
0.175138616
0.657773343
```

Now run the program several times and compare the numbers with those above. Notice how different they are? The chances of repeating the same sequence are infinitesimal, but it is not impossible. In fact with the RANDOMIZE command it is very simple to repeat any sequence of random numbers. The random seed is set to a default value of 2 when the Amstrad is switched on, so the numbers produced by Program 1.9 were all calculated with a seed value of two. To repeat the sequence of numbers produced by the first run of Program 1.9 the seed needs to be set to two.

PROGRAM 1.9(a)

```
15 RANDOMIZE 2
```

Every time the program is run, the same sequence of numbers will be produced.

There is one other quirk to the random number generator: if you print 'RND(0)', the number displayed will be the same as the previous number.

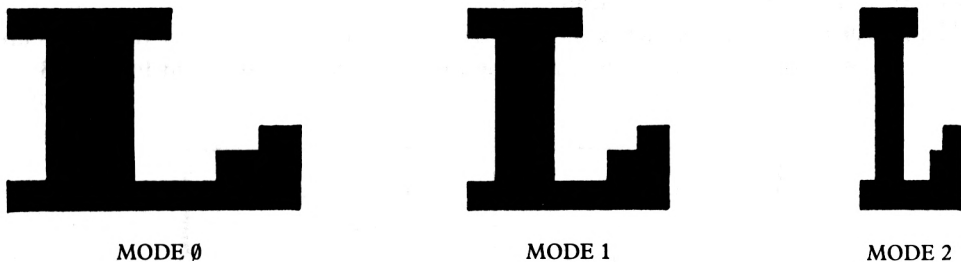
Chapter 2

This object of this chapter is to introduce the Amstrad's graphics commands. Part one of the chapter will look at the various types of screen display modes available, and at how to draw geometric shapes. Part two looks at giant-size text, and Part three investigates 'windows', which allow the screen to be easily divided into independent sections.

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 should be the one you are using at the moment.

The most obvious difference between each mode is the character size. Figure 2.1 illustrates the different sizes of characters in each mode. MODE 0 has very fat characters, MODE 1 characters are medium sized and MODE 2 has very thin characters.



The Three Character Sizes

FIGURE 2.1

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 big letters. As well as changing 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 large, only twenty can fit onto one line on the screen. There are still twenty five lines available, however. Another difference between the modes is the different number of colours available at any one time. Figure 2.2 shows the different number of colours and characters for each mode.

MODE	No. of Colours	Text Display
Ø	16	25 lines x 2Ø characters
1	4	25 lines x 4Ø characters
2	2	25 lines x 8Ø characters

The Screen Display Available In Each MODE

FIGURE 2.2

So far this book has only used the colours the Amstrad starts up with. It is possible for the user to change the colours used in screen displays. 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 has four colour commands: 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 Ø (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 is given in Figure 2.3

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

The Amstrad's Colours and Codes

FIGURE 2.3

Thus to change the border to pink you simply type:

BORDER 16

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:

INK i,c

where 'i' is the INK number and 'c' is the colour number. To understand this command, imagine a row of inkpots. Each inkpot has its own unique number (the INK number) which contains the colour 'c'.

So, 'INK 0,6' means 'fill inkpot number 0 with colour 6 (bright red)'. When the Amstrad is switched on, the 'inkpots' are given default colour values. The colour in each inkpot differs depending upon which mode you are in. Figure 2.4 shows the default INK values for each of the three modes ('F' stands for Flashing).

INKPOT	MODE 0	MODE 1	MODE 2
C O L O U R S			
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	F1,24	20	1
15	F16,11	6	24

The Default INK Values

FIGURE 2.4

You can change the colours in each inkpot to any that you like. 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 required. For example:

PAPER 10

will change the text background to the colour that is in inkpot ten. This colour depends on which mode you are in. The colours are pastel blue, bright cyan 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 the new background colour.

The computer defaults to a PAPER value of zero, i.e. the colour that is in inkpot zero. When you switch on this is blue, colour code 1.

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 inkpot 3 contains.

The computer defaults to a pen value of one, i.e. the colour that is in inkpot one. When switched on this is yellow, colour code 24.

Now we can change the colour of the screen and characters to any combination required. Suppose you required a lime green screen with a pen colour of bright blue. 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:




```
MODE 0  
INK 0,21  
INK 1,2
```

The screen should now have bright blue writing on a lime green background.

Changing the mode will set the PAPER and PEN values back to zero and one respectively. So, whenever you change mode, the paper colour becomes that of inkpot zero and the pen colour becomes that of inkpot one.

Graphics

As mentioned earlier, the size of the characters on the screen varies with each mode, the largest in MODE 0 and smallest in MODE 2. 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') and they also vary in size with each mode (largest in MODE 0, smallest in MODE 2 – i.e. the opposite way around to the text). The size of the pixels determines the 'resolution' of the modes, i.e. the sharpness of the display. The smaller the pixel, the sharper the display that you will get. The difference in pixel size for each mode is illustrated in Figure 2.5.

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

Relative Pixel Sizes

FIGURE 2.5

PLOT

Each point on the screen 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 (light-up) a pixel you use the PLOT command, in the form:

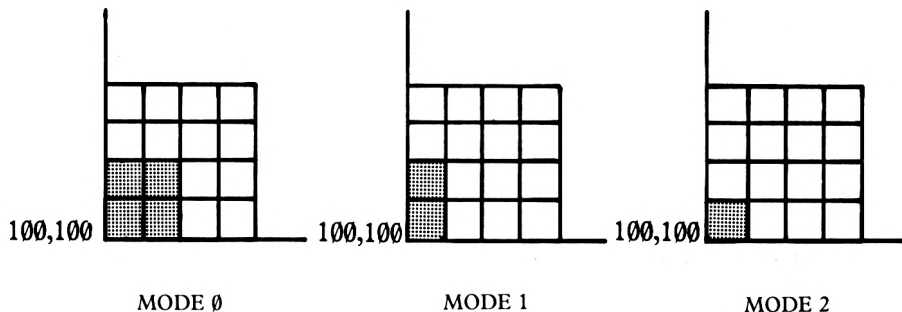
```
PLOT x,y
```

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 co-ordinate value of 0,0, i.e the bottom left-hand corner of the screen. The colour is decided by the value of INK1. (INK1 being the default pixel colour value).

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 the Amstrad will plot only the point specified by the coordinates. However, when MODE 1 is being used, the computer plots two points because each pixel in MODE 1 is twice the size of those in MODE 2. In MODE 1 the points are grouped together in pairs. Consequently, plotting at one location will cause the second point in that group of two to be plotted as well. Thus plotting at location 100,100 will cause location 100,101 to be switched on as well. 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 the plot command in MODE 0 causes four points to be set.



The Three Different Pixel Sizes

FIGURE 2.6

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 points, 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 set certain surrounding points.

To demonstrate this point(?) further, type in and run Program 2.1.

PROGRAM 2.1

```

5 MODE 0
10 FOR x=0 TO 640 STEP 4
20 PLOT x,10
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. When it has finished, change line 5 to the following:

```

5 MODE 1

```

MODE 1 pixels are half the size of MODE 0 pixels, so the program will produce a dotted line with every second point set. When the program finishes, change line 5 again, to:

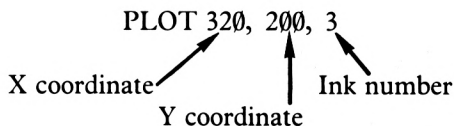
```

5 MODE 2

```

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

It was mentioned briefly that the pixel colour defaults to that of inkpot one (at switch-on, this is bright yellow). The PLOT command can be used to change the graphic colour by including an ink number. For example:



This will PLOT a point at the centre of the screen in the colour stored in inkpot three. The default value of ink 3 in MODE 2 is bright yellow. Changing the graphics colour does not affect the character colour.

PLOTR

PLOTR is a variation of PLOT and sets a point relative to the last point set. For example:

```
PLOT 100,100
```

This is interpreted by the computer as 'move' the graphics cursor to the position one hundred points across and one hundred up from the origin and set the point there.

```
PLOTR 100,100
```

This tells the computer to move the graphics cursor one hundred points across and one hundred points up from 'the last point set.' The result of typing in these two commands is two dots on the screen. The first (PLOT) at coordinates 100,100 and the second (PLOTR) at coordinates 200,200.

DRAW

PLOT only sets one point; thus to fill in a whole line of points you would need a FOR...NEXT loop like the one in Program 2.1. The DRAW command is used to draw a line of points starting from the current position of the graphics cursor and ending at the specified coordinates.

```
DRAW 640,400
```

If you had typed in the previous PLOT commands then the line of points will be drawn from 200,200 to 640,400.

PROGRAM 2.2

```
10 MODE 0  
20 DRAW 640,400
```

When MODE 0 is set, the graphics cursor is reset to coordinates 0,0, so Program 2.2 draws a line from the bottom left to the top right of the screen. The graphics cursor is now at coordinates 640,400. The next DRAW command will begin at this point. The DRAW command can be used to change the colour of the graphics in the same way as PLOT.

PROGRAM 2.2(a)

```
30 DRAW 300,300,3
```

This will draw a red line across the screen. Red is the default colour value for inkpot three.

DRAWR

The DRAWR command works in a similar way to the PLOTTR command. Whereas DRAW produces a line to a specific coordinate, DRAWR draws a line to a position relative to the position of the graphics cursor. This is best illustrated with an example:

PROGRAM 2.3

```
10 MODE 0
30 DRAWR 100,100
```

When run, this program will start at the graphics origin (point 0,0) and draw a line a hundred points up and right from this position. So far so good. Now add line 20 and re-run the program.

PROGRAM 2.3(a)

```
20 PLOT 320,200
```

When run now, the program draws a line one hundred points long starting at the centre of the screen. Line 30 is not drawing a line to the coordinate 100,100 but a line to a position one hundred points right and up from the last position of the graphics cursor.

MOVE

MOVE, as its name suggests, simply positions the graphics cursor at a chosen x and y coordinate; for example:

```
MOVE 100,100
```

will position the graphics cursor 100 points up and 100 points across from the origin. The difference between MOVE and PLOT is that PLOT sets a point at the position to which the graphics cursor is moved, whereas MOVE simply positions the graphics cursor. No point is set.

MOVER

This command has the same relationship with MOVE as PLOTTR and DRAWR have with PLOT and DRAW respectively. One thing about the relative commands that hasn't been covered yet is that the cursor can move to a point behind the present graphics cursor position, e.g.

```
MOVER -20,-100
```

Will move the graphics cursor left twenty points and down one hundred. This applies to MOVER, PLOTTR and DRAWR.

Using the graphics commands one can begin to draw pictures. Drawing a square is simple. You only need three values: the x and y coordinates of one of the corners and the lengths of the sides. Program 2.4 uses the input x and y coordinates for the bottom left hand corner of the screen.

PROGRAM 2.4

```

10 MODE 0
20 INPUT"x value";x
30 INPUT"y value";y
40 INPUT"length";l
50 CLS
60 MOVE x,y:REM start positions
70 DRAWR l,0:REM bottom
80 DRAWR 0,l:REM right
90 DRAWR -l,0:REM top
100 DRAWR 0,-l:REM left
110 LOCATE 1,22:PRINT"finished!"

```

This is all very well if you just want to draw a square, but suppose you wanted to draw a rectangle. In this case you would need to input another value, this being the height and amend lines 80 and 100 accordingly.

PROGRAM 2.4(a)

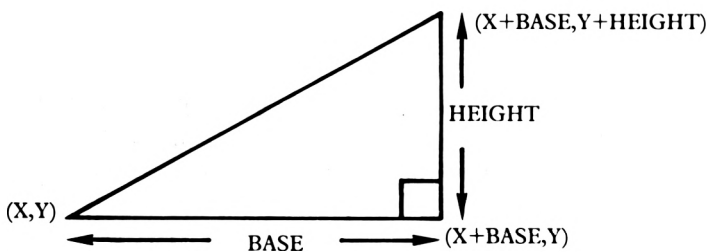
```

45 INPUT"height";h
80 DRAWR 0,h:REM right
100 DRAWR 0,-h:REM left

```

So far so good: drawing boxes is quite easy. The problems arise when you want to start drawing triangles, hexagons, octagons, other assorted polygons, and circles.

Drawing right-angled triangles is very similar to the drawing squares. Again you need the x and y coordinates of a corner, the length of the base side, and the height of the triangle.



The Co-ordinates for a Right-Angled Triangle

FIGURE 2.7

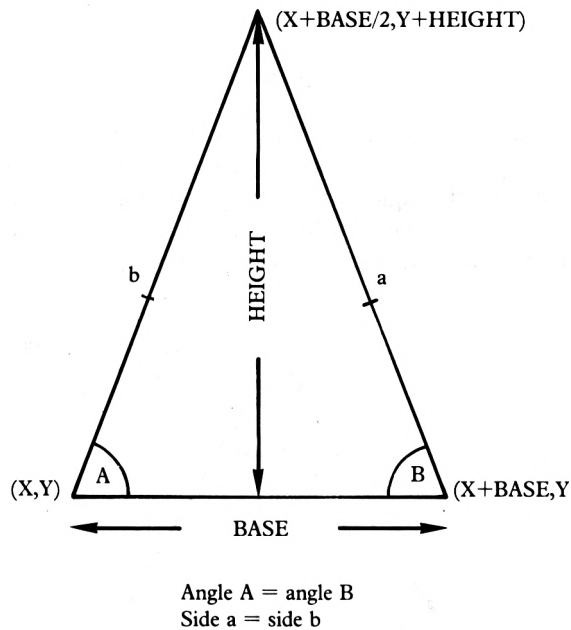
Turning the values in Figure 2.7 into a program gives us the following:

Type in 'NEW' before typing in this program.

PROGRAM 2.4(b)

```
10 MODE 0
20 INPUT"x value";x
30 INPUT"y value";y
40 INPUT"base";base
50 INPUT"height";height
60 CLS
70 MOVE x,y:REM start positions
80 DRAWR base,0:REM base
90 DRAWR 0,height:REM right side
100 DRAW x,y:REM hypotenuse
110 LOCATE 1,22:PRINT"finished"
```

A similar type of routine can be used to draw an isosceles triangle (an isosceles triangle is a triangle with two angles and two sides the same). Figure 2.8 shows the coordinates of an isosceles triangle using the same type of input as Program 2.4(b).



The Co-ordinates for an Isosceles Triangle

FIGURE 2.8

The line from the apex of the triangle has length of 'HEIGHT'. This line bisects the base exactly in half, therefore the coordinate value for the top of the triangle is X+half of the base, Y+height. Translating this into a program yields:

PROGRAM 2.4(c)

```
90 DRAW -BASE/2,HEIGHT
```

Only line 90 needs changing as this draws the vertical side of the triangle. Programs 2.4(b) and 2.4(c) used DRAWR on lines 80 and 90 as opposed to DRAW because it makes the program simpler to write. This point is illustrated by Figure 2.9(a) which has two listings of lines 80 and 90. The one on the left uses DRAW and the one on the right using DRAWR.

80 DRAW X+BASE,Y	80 DRAWR BASE,0
90 DRAW X+BASE/2,Y+HEIGHT	90 DRAWR -BASE/2,HEIGHT

The Advantage of Using DRAWR

FIGURE 2.9(a)

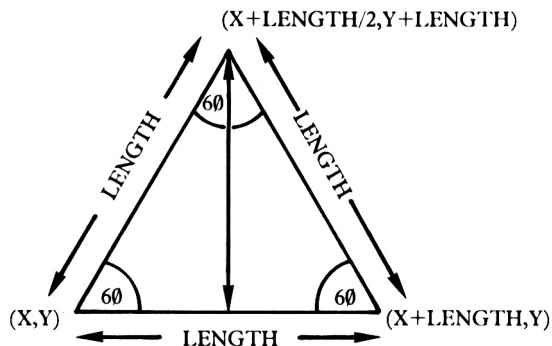
DRAWR doesn't have it all its own way though, because line 100 uses a DRAW command for exactly the same reasons that lines 80 and 90 don't. Figure 2.9(b) shows the advantage of using DRAW.

100 DRAW X,Y	100 DRAWR -BASE/2,-HEIGHT
--------------	---------------------------

The Advantage of Using DRAW

FIGURE 2.9(b)

An equilateral triangle is a triangle that has all three sides and angles the same; consequently it is very simple to draw: slightly easier to do than a square. The input required for an equilateral triangle is the x and y coordinates of one side and the length of the sides. Figure 2.10 shows the coordinates of each side of an equilateral triangle and Program 2.4(d) draws one using the x and y coordinates for the bottom left corner.



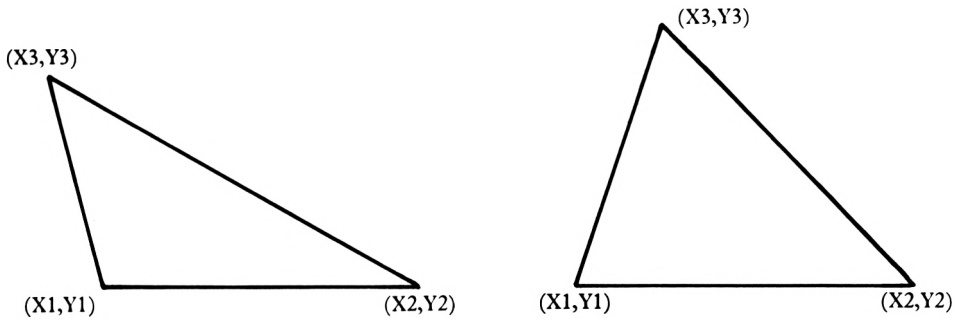
The Coordinates for an Equilateral Triangle

FIGURE 2.10

PROGRAM 2.4(d)

```
10 MODE 0
20 INPUT"x value";x
30 INPUT"y value";y
40 INPUT"length";length
50 CLS
60 MOVE x,y
70 DRAW length,0
80 DRAW -length/2,length
90 DRAW x,y
100 LOCATE 1,22:PRINT"finished"
```

Probably the easiest way to draw other types of triangle is to enter the coordinates for each corner. Figure 2.11 shows some of the triangles that would have to be drawn using this method.



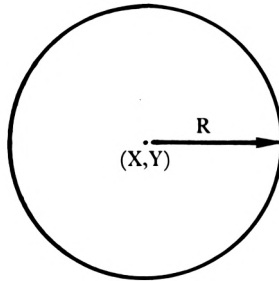
Some Other Types of Triangle

FIGURE 2.11

The program to draw these triangles is quite simple: you need to enter three x values (X1, X2, X3) and three y values (Y1, Y2, Y3) then draw from (X1,Y1) to (X2,Y2), from (X2,Y2) to (X3,Y3) and from (X3,Y3) to (X1,Y1).

CIRCLES

When drawing a circle you need three things: the x and y coordinates of the centre of the circle and the radius. With these values the circle is calculated in the following way.



Circumference of a Circle = $2\pi R$

FIGURE 2.12

To draw a circle, all the points around the circumference need to be set. In order to do this our old friends from school SIN and COS are needed. The x and y coordinates of each point in a circle are calculated via the following method:

$$x = \text{COS}(\theta \text{ to } 2\pi) \text{ times radius}$$

$$y = \text{SIN}(\theta \text{ to } 2\pi) \text{ times radius}$$

As well as the coordinates for the centre of the circle and the radius, a loop is required ranging from θ to 2 times PI. 'PI' is a built-in Amstrad variable which contains a value of 3.14159265.

PROGRAM 2.4(e)

```
10 MODE 1
20 radius=100
30 FOR a=0 TO 2*PI
40 x=COS(a)*radius
50 y=SIN(a)*radius
60 PLOT 320+x,200+y
70 NEXT a
```

The circle that Program 2.4(e) attempts to draw has a radius of one hundred and the centre of it is the centre of the screen (coordinate 320,200). Sadly Program 2.4(e) does not produce a circle but a set of dots vaguely forming a circle shape. To produce a complete circle the default step value of the FOR...NEXT loop in line 30 needs to be changed to a lower value e.g. STEP 0.02. This step size is sufficiently small to fit all the points in a circle. Change line 30 to the following:

PROGRAM 2.4(f)

```
30 FOR a=0 TO 2*PI STEP 0.02
```

Now the program will draw a complete circle. The only problem is that it takes quite a long time before the program actually finishes plotting all the points. There are various methods of speeding up the circle drawing, two of which will now be examined.

The first 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 be no gaps in the circle thus allowing the step size to be increased. Problems will occur if the step size is increased too much, of course, because the circle will look more like a polygon. A suitable step size is about 0.1, five times the size of that used in program 2.4(f). Program 2.4(g) uses DRAW instead of PLOT. Notice the PLOT on line 30. This positions the graphics cursor to the first coordinate of the circle.

PROGRAM 2.4(g)

```
10 MODE 1
20 r=100:REM radius
30 PLOT 420,200
40 FOR a=0 TO 2*PI STEP 0.1
50 x=r*COS(a):y=r*SIN(a)
60 DRAW 320+x,200+y
70 NEXT a
```

The execution of this program presents the user with something of a problem. The program does not produce a full circle. It very nearly draws a whole circle but a little bit is missing on the right hand side. This problem is due to the FOR...NEXT loop. In Program 2.4(g) the counter, A, will have the values 5.9, 6.0, 6.1 and 6.2. The counter is then incremented to 6.3 and as this value is greater than the loop limit ($2*PI=6.28318530$), the FOR...NEXT loop ends. 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. To handle this 'error', the step size needs to be doctored so that it divides exactly into the difference between the loop's start and end values. In the example of Program 2.4(g) the simplest solution however is to add 0.1 to the loop's limit ($2*PI+0.1$) as in Program 2.4(h) below. This will have the effect of pushing the limit to a value just greater than $2*PI$, and consequently the step size will divide into it enough times to produce a complete circle.

PROGRAM 2.4(h)

```
40 FOR a=0 TO 2*PI+0.1 STEP 0
```

The second method of speeding things up paradoxically involves taking more time to some extent. This apparent contradiction will be resolved shortly.

The method involves the use of a 'look-up table'. This is simply a list of data that needs to be referred to a number of times in a program. In Program 2.4(h) the values for the SINE and COSINE of the angles have to be computed every time the circle is drawn. However, the values will be the same every time regardless of the radius of the circle. So, instead of calculating the values each time a circle is drawn we just calculate them once and store the result in a 'look-up-table'. Once the values are stored we will be able to look them up when needed. The calculation of SINES and other complicated mathematical functions is a slow process so the look-up table will produce a circle much more quickly. However (and this is the paradox) creating the look-up table will initially slow things down. It is, obviously enough, not suitable for a Program that only draws one circle. However, when more than one circle is to be produced the advantages become apparent.

To create the look up table we will store the calculated values in two one-dimensional arrays – SN() and CS() – to hold respectively the SINE and COSINE values. This gives the following line:

PROGRAM 2.5

```
1000 DIM sn(50),cs(50)
```

50 represents the number of sides in the circle, which is actually a polygon. Calculating the values is done in the same way as before, but this time we enter the result in the appropriate elements of the arrays. If we are to have a 50 sided circle then, given that there are 2π radians (i.e. 360 degrees) in a circle, each side will cover $2\pi/50$ radians or $\pi/25$. This will be the step value used for calculating the SINES and COSINES. Also, some variables, STP and ANG, need to be initialised for the STeP size and ANGLE respectively, giving the following line:

PROGRAM 2.5(a)

```
1010 stp=PI/25:ang=0
```

Now to deal with the main calculation loop which will assign values to the look-up tables. This is done in Program 2.5(b):

PROGRAM 2.5(b)

```
1020 FOR a=1 TO 50
1030 ang=ang+stp
1040 sn(a)=SIN(ang)
1050 cs(a)=COS(ang)
1060 NEXT a
1070 RETURN
```

Once the look-up table has been set up it needs to be incorporated into the program. Line 40 in Program 2.5(c) below sets up a loop from 1 to 50 to read each of the array values, line 50 multiplies the values of SN(A) and CS(A) by the radius value and then the next 'side' of the circle is drawn (line 60).

PROGRAM 2.5(c)

```
10 MODE 1
15 GOSUB 1000:REM initialise look-up tab
16 le
20 r=100:REM r=radius
30 PLOT 320,300
40 FOR a=1 TO 50
50 x=sn(a)*r:y=cs(a)*r
60 DRAW 320+x,200+y
70 NEXT a
80 STOP
```

It was mentioned earlier that the look-up table can only save time when more than one circle is drawn so Program 2.5(c) needs to be changed so that it draws more than one circle. The best way of doing this is to draw circles of random radius and random starting positions:

PROGRAM 2.5(d)

```
20 r=INT(RND*200)+1
25 sx=INT(RND*640)+1
27 sy=INT(RND*400)+1
30 PLOT sx,sy+r
60 DRAW sx+x,sy+y
80 GOTO 20
```

PART TWO

TEST

The TEST command is used to find the colour of a pixel. The coordinate value of the pixel to be tested is defined by the test coordinates i.e.

```
TEST(100,100)
```

will look at the pixel which is one hundred units right and one hundred units up from the origin. The test command reports back the ink number that the pixel is coloured with. By way of example Program 2.6 clears the screen, setting the paper to the colour in inkpot three. Line twenty then tests a pixel (a purely arbitrary choice of pixel) and reports back the value three.

PROGRAM 2.6

```
10 PAPER 3:CLS
20 PRINT TEST(197,304)
```

The TEST command has a 'sister' relative command, TESTR where:

```
A=TESTR(10,-10)
```

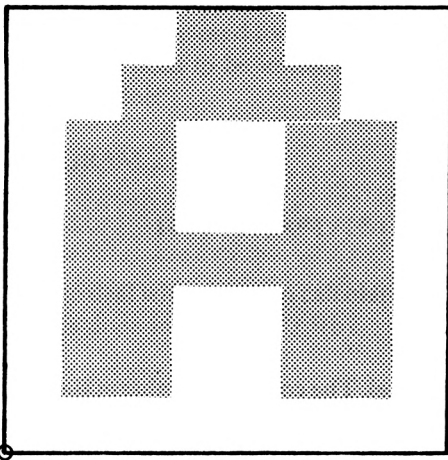
will give the variable A the ink colour of the pixel 10 to the right and 10 down from the current graphics cursor position.

One use for TEST is to produce a type of screen dump where a section of the screen will be tested and when a pixel is set the program will perform some action. The program we will develop will read a section of the screen one character wide and reproduce whatever is there onto the screen again, but twice as big. The first stage of this is to clear the screen and display a printed character.

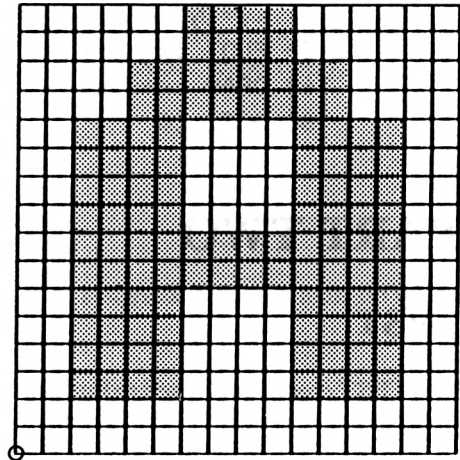
PROGRAM 2.6(a)

```
10 MODE 1
20 LOCATE 3,23:PEN 1
30 PRINT CHR$(65)
```

The character displayed is a capital A. The bottom left corner of the character is at coordinate 32,32. The area that 'A' takes up is a grid of sixteen by sixteen pixels; see Figure 2.13.



Location 3,23



Co-ordinate (32,32)

Capital A as Pixels

FIGURE 2.13

The TEST routine will start at coordinate 32,32 (the bottom left of the 'A' matrix) and will work its way along to coordinate 48,48, (the top right of the 'A' matrix) testing all the values on the y axis before increasing the x. Thus two loops are required, one controlling the y coordinate and one controlling the x:

PROGRAM 2.6(b)

```
40 FOR x=32 TO 48
50 FOR y=32 TO 48
```

Now the actual testing can begin. If the result of testing pixel (X,Y) is one, then it is part of the character A, INK 1 being the colour that the character was printed in. If this condition is true a flag (F) is set to one.

PROGRAM 2.6(c)

```
60 IF TEST(x,y)=1 THEN f=1
```

While F is equal to one, part of the character needs to be drawn. To do this the graphics cursor needs to be moved to the required place, the draw coordinates calculated and then the line drawn.

PROGRAM 2.6(d)

```
70 WHILE f=1
80 FOR l=1 TO 2
90 MOVE 10+(x-32)*2,(100+(y-32)*2)+1
100 DRAW 2,0
110 NEXT l
120 f=0
130 WEND
140 NEXT y,x
```

If you run this program, you will see that it does indeed draw a double-sized letter A. Line 90 is the one that positions the graphics cursor and involves something of a calculation. The first part of the expanded character is drawn at the coordinates:

```
10+(32-32)*2,(100+(32-32)*2+1
10+( 0 )*2,(100+( 0 )*2+1
10      ,101
```

I.e. the start coordinate for the enlarged picture equals (10,101)

From this point a line is drawn relative, two across and none up; i.e. coordinates (12,101). When the loop L is increased the graphics cursor is moved to the position 10,102 and another relative line is drawn. When the L loop has been completed a block has been drawn on the screen. This block is twice the size of the tested pixel. When the program is finished there will be two A's on the screen. The printed one will be a normal mode one character size and the drawn one will be exactly twice the size of a mode 1 character.

The program can, of course, be adapted so that the character to be enlarged and the size of the enlargement can be input by the user. Program 2.6(e) demonstrates this with the user being able to type in a four character string.

PROGRAM 2.6(e)

```
10 MODE 1
12 LOCATE 1,1:INPUT"a string";a$
14 IF LEN(a$)>4 THEN 10
20 LOCATE 3,23
30 PRINT a$
32 LOCATE 1,2:INPUT"size";size
40 FOR x=32 TO 32+16*LEN(a$)
50 FOR y=32 TO 48
60 IF TEST(x,y)=1 THEN f=1
70 WHILE f=1
80 FOR l=1 TO size
90 MOVE 10+(x-32)*size,(100+(y-32)*size)
+l
100 DRAWR size,0
110 NEXT l
120 f=0
130 WEND
140 NEXT y,x
```

To draw a four-character string in characters eight times as big will take just under a minute.

PART THREE

Windows

A window is a user defined area of the screen that is used to highlight certain items of text. Windows act like individual screens. Text can be displayed in them and their colours can be changed independently of the main screen. When designing windows you need five things:

- stream number range (0-7)
- coordinate of left side of window
- coordinate of right side of window
- coordinate of top side of window
- coordinate of bottom side of window

The coordinate values have the same range as for the locate command

The syntax of the WINDOW command is:

WINDOW stream,left,right,top,bottom

Streams

Streams are used to tell the computer where to display text. There are ten different stream values and the first eight (0-7) are reserved for use with windows. The two other stream values, '8' and '9', are used to send text to the printer and cassette unit respectively. Stream 0 is the default value and unless defined otherwise by a window command, is the main screen.

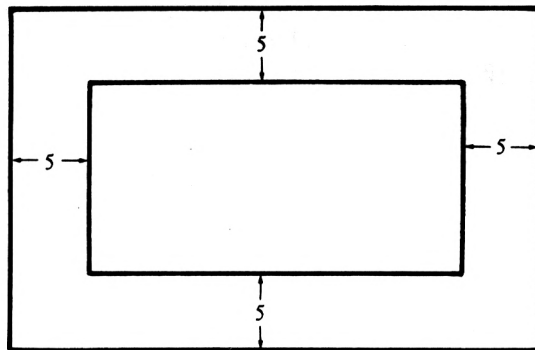
A stream is indicated by placing a hash (#) after the stream-using statement, followed by the stream number and then a comma. For example:

```
PRINT#1,"ROW, ROW, ROW, THE BOAT"
```

This would display the string "ROW,ROW, ROW, THE BOAT" in window one. It would, if window one had been set up. Because there is no window for this message to appear in, it is displayed in the default stream, stream zero (the screen).

Designing Windows

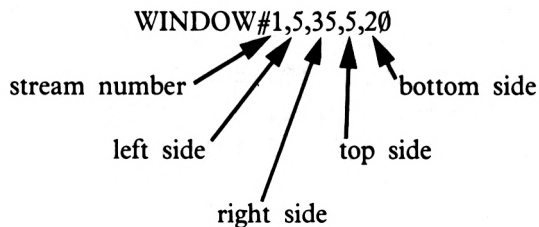
Once you have decided which stream number you want to use you have to decide how big you want the window to be and where it will go. Figure 2.14 shows the window that we will attempt to design.



A Window to Be Designed

FIGURE 2.14

The command to design this particular window will look like this



If you type in that command, you will notice no change on the screen. Now type in:

```
PRINT#1,"ROW, ROW,ROW THE BOAT"
```

The string will be displayed starting at the 5th location across and the 5th down. This is the first location in window one. To see the window in all of its glory, type in the following direct entry commands:

```
PAPER#1,3:CLS#1
```

This sets the PAPER in window one to the colour contained in inkpot 3. Window one is then cleared, giving it the colour of ink three. If you have just switched the Amstrad on the screen will now have two background colours. Window one contains the colour red and the main screen is coloured blue.

The following commands can all be used with a stream number.

CLS	PAPER
INPUT	PEN
LIST	PRINT
LOCATE	

These commands are quite easy to use with different streams, you simply place a hash, stream number and in most cases a comma after the command. For example

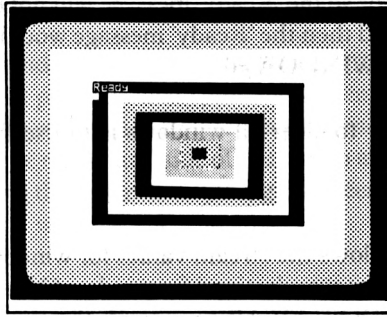
CLS#1	PAPER#1,2
INPUT#1,A\$	PEN#1,0
LIST#1,100-200	PRINT#1,"WINDOW 1"
LOCATE#1,5,3	

Program 2.7 demonstrates the kind of effect that can be produced using windows.

PROGRAM 2.7

```
10 REM window demonstration
20 INK 0,0:INK 1,24:INK 2,17:INK 3,6
30 PAPER 1:CLS
40 WINDOW#0,6,34,5,19:PAPER#0,0:CLS#0
50 WINDOW#1,8,32,6,18:PAPER#1,1:CLS#1
60 WINDOW#2,10,30,7,17:PAPER#2,3:CLS#2
70 WINDOW#3,12,28,8,16:PAPER#3,0:CLS#3
80 WINDOW#4,14,26,9,15:PAPER#4,1:CLS#4
90 WINDOW#5,16,24,10,14:PAPER#5,2:CLS#5
100 WINDOW#6,18,22,11,13:PAPER#6,3:CLS#6
110 WINDOW#7,20,20,12,12:PAPER#7,0:CLS#7
```

This program sets up eight different windows, producing the following pattern:



Windows of Program 2.7

FIGURE 2.15

When program 2.7 has finished, the 'Ready' message appears in the top left hand corner of window#0. If you type in LIST the program will be displayed in that window, overwriting all the other windows. Notice that the display in window zero scrolls and the program lines wrap around as on the normal screen. Although the other windows have been overwritten they are still set up. To prove this, type in the following direct entry commands:

```
CLS#2  
LIST#3  
CLS#6
```

Windows are very handy for highlighting items of text. Windows can also be used to create some interesting effects, one of which is demonstrated in Program 2.8 below. This program uses all of the eight available windows to draw a series of boxes diagonally down the screen.

PROGRAM 2.8

```
10 BORDER 17  
20 INK 3,10:PAPER 3  
30 MODE 1:CLS  
40 INK 0,1:INK 1,26  
50 FOR x=0 TO 7  
60 xx=x*5:yy=x*3  
70 IF x=0 THEN xx=1:yy=1  
80 WINDOW#x,xx,xx+5,yy,yy+3  
90 c=c+1  
100 IF INT(c/2)=c/2 THEN PAPER#x,0:PEN#x  
    ,1:ELSE PAPER#x,1:PEN#x,0  
110 CLS#x  
120 LOCATE#x,2,2:PRINT#x,x  
130 NEXT x  
140 IF INKEY$<>" " THEN 140 ELSE WINDOW#  
    0,1,40,1,25  
150 LOCATE#0,2,20
```

Lines 140 and 150 are used to re-define window zero back to its original size (i.e the screen size) and to position the cursor so the 'Ready' message will not be printed over the picture. If these lines were not included then the 'Ready' message would appear in the top left hand corner of the newly defined WINDOW#0.

The quickest and easiest way to clear all windows is to use the MODE command.

WINDOW SWAP

There is a command which allows the stream values for windows to be swapped round so that (for example) anything that was destined for window#3 would instead be displayed in window#7. This is the WINDOW SWAP command, the syntax of which is:

```
WINDOW SWAP source , destination
              window   window
```

The source window is that to which messages were initially going and the destination window is that to which the messages are to be redirected. For example:

```
WINDOW SWAP 3,7
```

This will cause any command using stream three to be directed to stream seven instead; e.g.

```
PRINT#3,"THIS IS THREE"
```

will be displayed in window seven.

Lines 60 to 90 in Program 2.9 below print '3's and '7's in windows three and seven respectively. Then line 100 swaps the windows round. Next, lines 110 to 140 perform exactly the same operation as 60 to 90 only this time the sevens appear over the threes and the threes replace the sevens.

PROGRAM 2.9

```
10 MODE 2:CLS
20 INK 0,26:INK 1,2
30 WINDOW#3,1,39,1,25:PAPER#3,1:PEN#3,0
40 WINDOW#7,40,80,1,25:PAPER#7,0:PEN#7,1
50 CLS#3:CLS#7
60 LOCATE#3,1,1:LOCATE#7,1,1
70 FOR x=0 TO 324
80 PRINT#3,3;:PRINT#7,7;
90 NEXT x
100 WINDOW SWAP 3,7
110 LOCATE#3,1,1:LOCATE#7,1,1
120 FOR x=0 TO 324
130 PRINT#3,3;:PRINT#7,7;
140 NEXT x
```

A split-screen similar to that used in Program 2.9 can be a useful tool for developing MODE 1 screen displays.

The character size of each window defined by Program 2.9 is that of the mode one screen. Program 2.10 below is a short program that splits the screen in half. Window#0 is the left half and is used to develop the program. All system messages will appear in this window as per normal. Window one is the right hand side of the screen and should be used to format a mode one display. When using the program anything that needs printing should have a '#1,' before it, sending the data to window one.

PROGRAM 2.10

```
10 GOSUB 9000
8999 END
9000 MODE 2
9010 WINDOW#0,1,39,2,25:PAPER#0,0:PEN#0,
1:CLS#0
9020 WINDOW#1,40,80,2,25:PAPER#1,1:PEN#1
,0:CLS#1
9030 WINDOW#2,1,39,1,1:PAPER#2,1:PEN#2,0
:CLS#2
9040 WINDOW#3,40,80,1,1:PAPER#3,0:PEN#3,
1:CLS#3
9050 LOCATE#3,14,1:PRINT#3,"The display"
9060 LOCATE#2,14,1:PRINT#2,"The program"
9070 RETURN
```

This program, if used, should be added as a subroutine to your own programs.

Window Graphics

So far all the windows used have been text windows; however, it is possible to create a graphics window using the 'ORIGIN' command. Only one graphics window can be used in a program and consequently there is no need to assign it a stream number. What the ORIGIN does is to move the graphics start position from the bottom right hand corner of the screen to anywhere on the screen and then designate boundaries in the same way that text window boundaries are defined.

ORIGIN x,y,left,right,top,bottom

The x and y coordinate value dictate the new graphics start position. If x and y had the values of 50 and 100 respectively then the graphics cursor will be moved from the bottom left up one hundred points and right fifty. This point will then be given a coordinate value of 0,0 making it the ORIGIN. To demonstrate this, type in the following program.

PROGRAM 2.11

```
10 INK 0,26:INK 1,2
20 BORDER 17:MODE 0
30 DRAW 0,200,1:DRAWR 200,0
40 DRAWR 0,-200:DRAWR -200,0
50 LOCATE 1,1:PRINT"Press the space bar"
60 IF INKEY$<>" " THEN 60
70 ORIGIN 100,50
80 DRAW 0,200,1:DRAWR 200,0
90 DRAWR 0,-200:DRAWR -200,0
```

This program draws a square starting at the default origin, coordinate 0,0. Then when the user has pressed the space bar, the origin is moved to location 100,50 (line 70). Lines 80 and 90 are an exact copy of the lines that drew the earlier square. This time the box is drawn starting at the new origin, therefore the command has worked.

The window limits are an optional extra to the origin command. If left out then the normal screen limits will be assumed. These default graphics screen limits are 640 points across and 400 points up.

Amend line 70 of Program 2.11 to that of Program 2.11(a). This restricts the window to drawing only in the centre of the screen.

PROGRAM 2.11(a)

```
70 ORIGIN 100,50,100,540,50,350
```

When the program is re-run no difference is apparent. To see the graphics window displayed we need to make use of an as yet undiscovered Amstrad graphics command.

CLG

The CLG command is used to CLear the Graphics window, setting all points therein to a designated colour. For example:

```
CLG 2
```

will clear the screen to the colour in inkpot two. CLG clears only a graphics screen. If the origin command has not been used then the graphic screen is the same size as the text screen and so both will be cleared. Because Program 2.11(a) sets a graphics window, only the points in that window will be cleared. To demonstrate this add line 75.

PROGRAM 2.11(b)

```
75 CLG 2
```


Once again run the program. The darker area is the graphic window created in line seventy of Program 2.11(a). Notice that the area outside the window has not been cleared even though it contains graphics. CLG only clears the area in the graphics window.

TAG

Amstrad Basic provides us with a command that allows characters to be printed where the graphics cursor is. This is the TAG command. TAG simply causes the print and graphics cursor to be one and the same. To demonstrate this, type in the following short program:

PROGRAM 2.12

```
10 MODE 1
20 TAG
30 PLOT 200,200
40 PRINT"You are here"
```

When you run this, not only will 'YOU ARE HERE' be displayed, but another two characters as well. These characters '←' and '↓' are the computer's code for a carriage return and line feed respectively. To prevent these extra characters from being displayed a semi colon should be put after every printed item. Thus line 40 should be changed to read:

```
PRINT"YOU ARE HERE";
```

The TAG command can be used with any of the streams, for example

```
TAG#2
```

will display all the data printed, to window two, at the graphics cursor. This point is illustrated in Program 2.12(a) below where a window is created in the centre of the screen. Nothing is printed in the window though. Stream one has been tagged to the graphics cursor in line 50.

PROGRAM 2.12(a)

```
10 MODE 1
20 INK 2,3:INK 3,26,17:INK 0,26:INK 1,2
30 WINDOW#1,10,30,4,20
40 PAPER#1,1:PEN#1,2:CLS#1
50 TAG#1
60 FOR x=0 TO 640 STEP 4
70 PLOT x,13:DRAW x,400,3
80 PRINT#1,CHR$(248);
90 NEXT x
```

Did you take notice of the character printed? Did you notice its colour? Line forty of Program 2.11(a) set the window one PEN to the colour in inkpot two (red). However, the character when printed was flashing white and pink, the same colour as the lines being drawn. When a stream is TAGged all the characters printed in that stream will be printed the same colour as the graphics cursor.

TAGOFF

This command quite simply switches off the TAG operation and can be used as in line 100 of Program 2.12(b):

PROGRAM 2.12(b)

```
100 TAGOFF
```

When TAG is used with WINDOW#0 all system messages (i.e. ready, syntax error etc.) override the TAG command and appear normally on the screen. This is an effective TAGOFF. This is only true of WINDOW#0: all the other windows are tagged until an appropriate TAGOFF or mode change is executed.

Chapter 3

One of the major features of the Amstrad personal computer is its musical abilities. The Amstrad has been specially designed to make it easy for the user to produce simple notes and complex tunes. The quickest way to produce a sound is to hold down the CTRL key and press the letter P. This produces a small beep. If you cannot hear it then turn the volume on your computer up. The volume control is the wheel located on the right hand side of the Amstrad. Having turned this up if you again hold down CTRL and press P, you should be able to hear the beep. As mentioned, this is the simplest way of producing a sound: the most versatile of the Amstrad's sound commands is called, unsurprisingly:

SOUND

There are a total of seven arguments for the SOUND command, five of which are optional. Figure 3.1 shows the sound arguments. The two highlighted must be included.

```
S O U N D <channel>,<tone>,<length>,<volume>  
          <volume beep>,<tone envelope>,<noise period>
```

The Sound Command

FIGURE 3.1

Before going into a lengthy explanation of each argument, it would be a good idea to produce a sound first. In order to do this a brief explanation of the first two arguments would be helpful.

The first value is the channel. This is used to indicate the way in which the SOUND command will be processed. For the convenience of this simple explanation a channel value of 1 will be used. The channel argument will be covered in greater detail shortly.

The second value is the tone, which controls the frequency of the note produced. Tone is a numerical value where a small number, e.g. 10, will produce a shrill note, sort of like a squeak, and a large number, e.g. 3000, will produce a flat, raspberry type, noise. The range of tones available extends from 0 (no tone at all) to 4096 (a very, very flat note). The tone values from 1000 onwards sound very similar to each other.

To demonstrate the type of sounds these two arguments produce, type in the following program:

PROGRAM 3.1

```
10 FOR x=0 TO 4095 STEP 5
20 SOUND 1,x
30 LOCATE 10,5
40 PRINT"tone:";x
50 NEXT x
```

Once the tone value reaches the thousand plus mark, the difference in tone is not so easy to identify and at this stage you might like to stop the program. This is done by pressing the ESC key twice. If you do let the program continue to its logical conclusion you will notice that the 'Ready' message appears before the music stops, which will be explained soon. Those of you who want to use the sound command just for simple additions to games, i.e. beeps and pings, might like to skip the detailed explanation of channels and tone and turn to the section entitled 'Length'.

Channel

The channel argument is used to control the way a note is played by the computer. The computer has three sound 'channels' or 'voices', making it possible to play up to three individual notes at once. The value is an integer (whole number) ranging from 1 to 255. The value is referred to as 'bit significant'. This means that the value of the channel argument can be looked upon as a binary number, where every binary one or zero is represented as a decimal number. This point is illustrated in Figure 3.2.

BIT	7	6	5	4	3	2	1	0
Decimal	128	64	32	16	8	4	2	1

Bit 0: Sends sound to channel A
Bit 1: Sends sound to channel B
Bit 2: Sends sound to channel C
Bit 3: Rendezvous with sound A
Bit 4: Rendezvous with sound B
Bit 5: Rendezvous with sound C
Bit 6: Hold the note
Bit 7: Flush

FIGURE 3.2

Figure 3.2 shows quite clearly that to send a sound through channel A a value of 1 is used. To send a sound through channel B a value of 2 is used and for channel C a value of 4. Before looking at the other values, a brief explanation of the differences between each channel is in order.

Each channel uses the sound command in the same way and the three channels can produce exactly the same sounds. To demonstrate this, type in Program 3.2.

PROGRAM 3.2

```
10 SOUND 1,90,50,12
```

If you run this you will hear a short beep. Now add line twenty:

PROGRAM 3.2(a)

```
20 SOUND 2,90,50,12
```

Run this by typing in RUN 20. This causes the program to begin execution at line twenty. This will produce an identical beep to that produced by line 10. Lastly, but not least, type in line 30. This, using a channel value of 4 (if confused see Figure 3.1), sends the note to channel C.

PROGRAM 3.2(b)

```
30 SOUND 4,90,50,12
```

Run this program by typing in RUN 30. Once again you will hear that familiar beep. once again run the program, only this time run the whole program. You will hear that familiar beep, once only – not three times as you might expect. Clearly the three channels have no difference, playing notes identically. The program currently in memory should look like this:

PROGRAM 3.2(c)

```
10 SOUND 1,90,50,12
20 SOUND 2,90,50,12
30 SOUND 4,90,50,12
```

As Program 3.2(c) shows that there are three sound commands, if the program is run as a whole you would expect three sounds to be produced. Not so! This is because all the three sounds are being played at the same time. The Amstrad's three channels each play their notes at simultaneously, thus allowing you to mix notes. If the tone values are changed so that each command plays a different note then you could hear a 'harmony'.

PROGRAM 3.2(d)

```
10 SOUND 1,90,50,12
20 SOUND 2,50,50,12
30 SOUND 4,20,50,12
```

Program 3.2(d) played three different notes all at the same time, turning a boring beep into something a bit like a train whistle.

Because the channel value is bit significant, a value of '3' (1+2) will cause the note to be played by channels 'A' and 'B'. Quite simply, the channel or channels used is defined by the sum of the various bits: thus, in this example, to get sound from channels A and B, the values 1 and 2 are used together, making 3. The first seven values for the channel argument are outlined in Figure 3.3.

Channel value	Operation
1	send sound to channel A
2	send sound to channel B
3	send sound to channels A and B
4	send sound to channel C
5	send sound to channels A and C
6	send sound to channels B and C
7	send sound to channels A, B and C

The First Seven Channel Values

FIGURE 3.3

Rendezvous

The rendezvous feature is used to synchronize sound commands on different channels. Try in Program 3.3:

PROGRAM 3.3

```
10 SOUND 1,50,50
20 SOUND 2,100,100
30 SOUND 4,150,200
```

When run, this program will start by playing all three notes. Then the duration of the note in channel A, will run out leaving the Amstrad playing notes in channels B and C. When channel B's duration value, is reached, channel C is left playing by itself. If the same sound commands are added again (Program 3.3(a)) the Amstrad will play channels A,B and C as before. However, when the duration value of channel A (line 10) runs out, the Amstrad will begin to play the note on line 40 right away.

PROGRAM 3.3(a)

```
40 SOUND 1,50,50
50 SOUND 2,100,100
60 SOUND 4,150,200
```

Suppose you wanted to halt the program until the note in channel C (line 30) has finished, then play the notes from line 40 onwards. This is where rendezvous comes in. The rendezvous function halts the playing of notes in one channel until a note has finished playing in another channel. Figure 3.4 shows the channel values.

Bit 3: Rendezvous with channel A: Decimal 8
Bit 4: Rendezvous with channel B: Decimal 16
Bit 5: Rendezvous with channel C: Decimal 32

Rendezvous

FIGURE 3.4

The rendezvous values are **added** to the channel values, so to rendezvous channel A to C the following sound command will replace line 40.

PROGRAM 3.3(b)

```
40 SOUND 33,50,50
```

Because the value of channel A is 1 and the rendezvous value for channel C is 32, the channel is given the value 33 (32+1). If you run the program now, however, you will hear no difference in the notes played. Rendezvous has not worked because the note played in channel C has not been told that it is going to rendezvous with channel A. This is done by changing line 60:

PROGRAM 3.3(c)

```
60 SOUND 12,150,200
```

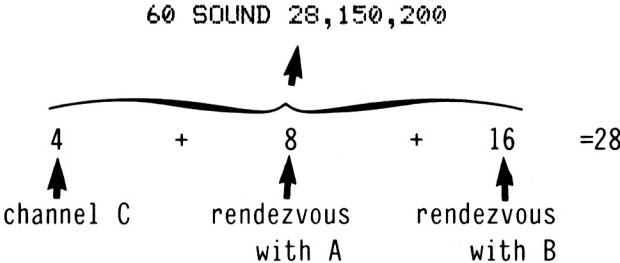
To get the program to play the same sequence of notes twice, channel B to be synchronized with 'C'. To do so change line 50 accordingly:

PROGRAM 3.3(d)

```
50 SOUND 34,100,100
```

Once again the sound command on line sixty needs to be changed, this time the value for channel B needs to be added:

PROGRAM 3.3(e)



The full program should look like this:

PROGRAM 3.3(f)

```
10 SOUND 1,50,50
20 SOUND 2,100,100
30 SOUND 4,150,200
40 SOUND 33,50,50
50 SOUND 34,100,100
60 SOUND 28,150,200
```

When run, Program 3.3(f) will play the same sequence of notes twice.

Hold

As seen, the rendezvous-channel argument allows notes to be played in synchronization with each other. There is another channel value that holds the playing of a note until it is required. Hold is bit six and thus has a decimal value of 64. To hold a note in any of the three sound channels you add 64 to 1, 2 or 4, channels A ,B or C respectively. To demonstrate this, type in the following short program.

PROGRAM 3.4

```
10 SOUND 65,100
20 SOUND 66,150
30 SOUND 68,200
```

If you run this program you will not hear a single note, because all three sound channels are held. To play a held note you need to use the RELEASE command.

RELEASE

Release is a BASIC command whose sole purpose is to work with the hold function of the SOUND command. Like the sound channel values, RELEASE is bit significant. This time just the first three bits have a recognised value:

Bit	2	1	0
Decimal	4	2	1

Bit 0: Release sound channel A

Bit 1: Release sound channel B

Bit 2: Release sound channel C

The Release Bit Values

FIGURE 3.5

To release just one of the three channels, you can add line 40 to Program 3.4

PROGRAM 3.4(a)

```
40 RELEASE 1
```

This will release only the note held in channel A. To release only the note in channels B or C the release values should be 2 or 4 respectively.

To release all three channels at the same time you should type in the following line.

PROGRAM 3.4(b)

```
40 RELEASE 7
```

The value of seven is obtained by adding the release values for A,B and C together, i.e. $1 + 2 + 4 = 7$.

To hold a whole sequence of notes, you only need to add the hold value to the first SOUND command. After this, all notes in that channel will remain on hold until a release command is encountered. This principle is demonstrated in Program 3.5

PROGRAM 3.5

```
10 SOUND 66,100,50
20 SOUND 2,125,50
30 SOUND 2,150,50
40 PRINT"Press space bar to hear notes"
50 IF INKEY$("<>") THEN 50
60 RELEASE 2
```

Flush

The final channel value, bit seven, is called 'flush'. When this is used, a specified channel, A,B or C, is emptied of all notes. All values in the 'sound queue' for that channel are removed. The other two channels will not be affected. If you know about binary arithmetic, then you will know that flush, being bit seven, has a decimal value of 128. If you didn't know that, you do now! To flush sound channel A, the command would read:

```
SOUND 129,100,50
```

This flushes channel A, playing a last note with a tone value of 100 and a duration of half a second.

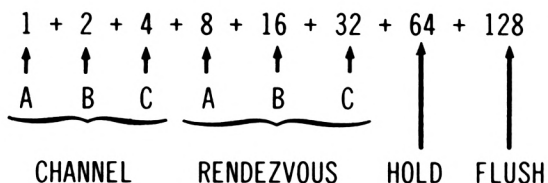
To illustrate this, change line 60 of Program 3.5 to that of program 3.5(a) below. Run the program and press the space bar, instead of the three notes that were on HOLD you will need only one. The flush used on line sixty removes the sound held and plays tone 100 on channel B.

PROGRAM 3.5(b)

60 SOUND 130,100,50

All of the sound channel values can be used together. If you wanted to you could put all the sound channel features into one sound command, not that you will hear much if you do:

SOUND 255,200,50



Tone

As was mentioned earlier, the tone argument controls the frequency of the note produced. The tone value ranges from 0 (no sound) to 4096 (a very deep note). The tone value is used to calculate the frequency of a value in the following way:

$$\text{Frequency} = 125000/\text{tone}$$

Therefore the frequency of a sound command with a tone value of 50 would be:

$$\begin{aligned}\text{Frequency} &= 125000/50 \\ &= 2500 \text{ Hertz}\end{aligned}$$

(Note: Hertz is the metric unit for cycles per second). Appendix seven of the Amstrad user manual contains a concise list of the frequency and necessary tone values for each note.

The following sound arguments are optional and can be used at the readers discretion.

Length

The length of a note is controlled in one hundredths of a second, so a length value of 50 will play a note for half a second. If no value is given, the Amstrad defaults to twenty, each note lasting for a fifth of a second.

Length may take any integer value from -32768 to +32767, where:

- A positive length value will play the note for length/100 seconds. For example

SOUND 1,50,200

will play a note lasting two seconds.

- A length value of 0 will cause the note to last until the end of the volume envelope (volume envelope will be discovered soon).
- A negative length value causes the Amstrad to play a notes ABS(length) times, and repeat the volume envelope ABS(length) times. (Note: ABS takes the positive value of a number; thus ABS(-5) = +5.) For example:

SOUND 1,50,-100

will play a note lasting one second and will repeat the volume envelope one hundred times.

Volume

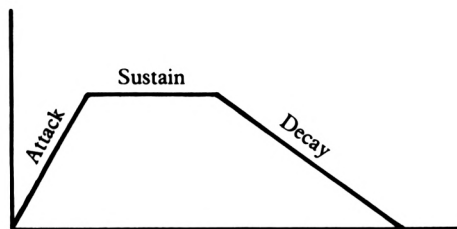
The volume argument, unsurprisingly, deals with the noise level of each note. The volume value ranges from 0 to 15. 0 denotes no volume, consequently you cannot hear the note being played. Fifteen is the highest volume value and plays a loud note. Remember there is a volume wheel on the right hand side of the computer as well. The default volume value is 12.

Volume Envelope

This is used to specify the way in which the volume of a note is played. That is, whether the note is to be constantly at the maximum volume specified in the envelope, or whether it is to build up gradually to that volume from that specified in the volume argument of the sound command as it is played, or whatever. To understand how this works, one must first understand sound envelopes in general.

Envelopes

'Envelope' is a term used to describe the shape of a sound. A sound can be thought of as having three sections: Attack, Sustain and Decay. These are shown in Figure 3.6



Sound Envelope

FIGURE 3.6

'Attack' is the speed at which the peak value is reached. 'Sustain' is the length of time the peak volume value is held, and 'Decay' is the time it takes for the note to fade out. The Amstrad comes equipped with an envelope command called:

ENV

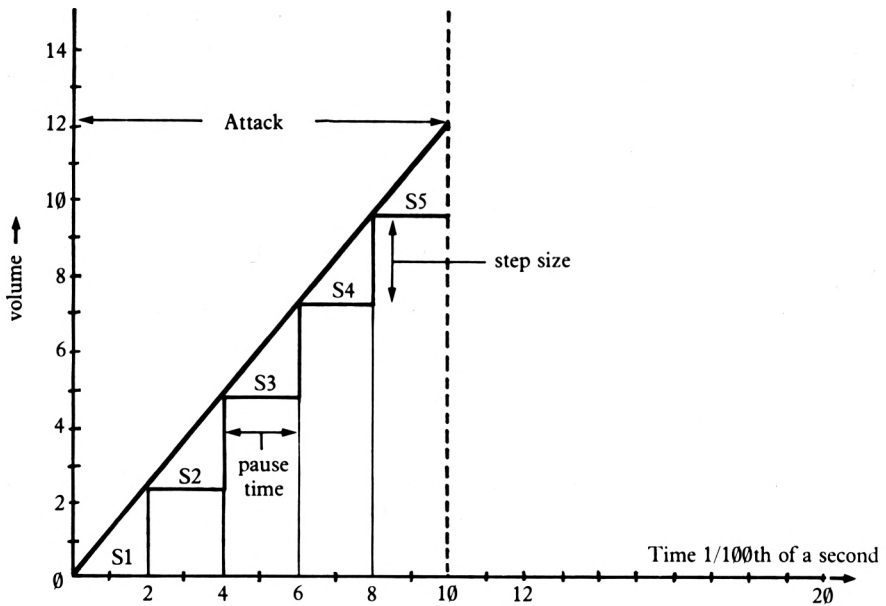
The ENV command makes it quite simple for the user to program attack, sustain and decay into a musical note. There are fifteen possible ENVelopes to use, ranging from ENV 0 to ENV 15, and each is defined by a number. For example:

ENV 1

indicates the first envelope. Attack, sustain and decay values follow the envelope number.

Attack

The attack value dictates the time it takes for a note to reach its peak value. Three arguments are used to specify an attack value; the step count, step size and pause time. The meaning of these values is indicated in Figure 3.7



The Attack Values

FIGURE 3.7

The ENV command works in one hundredths of a second, so the attack outlined above lasts for one tenth of a second.

The step count in this particular example is 5, each step being marked by an 'S' (S1-S5). The step count can be any value from 0 to the attack length value. The step count is simply the number of steps you have chosen to use in that section of the envelope. It determines how smoothly the change in volume is to take place.

The step size is calculated simply by dividing the required volume increase value by the step count. In the above example with a peak volume of 12 and a step count of five, this gives the following:

$$\begin{aligned}\text{step size} &= \frac{\text{volume increase}}{\text{step count}} \\ &= 12 \div 5 \\ &= 2.4\end{aligned}$$

Therefore the step size is 2.4.

The pause time is calculated by dividing the attack time by the step count. In the above example this gives the following:

$$\begin{aligned}\text{Pause time} &= \frac{\text{attack time}}{\text{step count}} \\ &= 10 \div 5 \\ &= 2\end{aligned}$$

Therefore the pause time is 2 hundredths of a second.

So far, the ENV command looks like this:

```
10 ENV 1,5,2.4,2
```

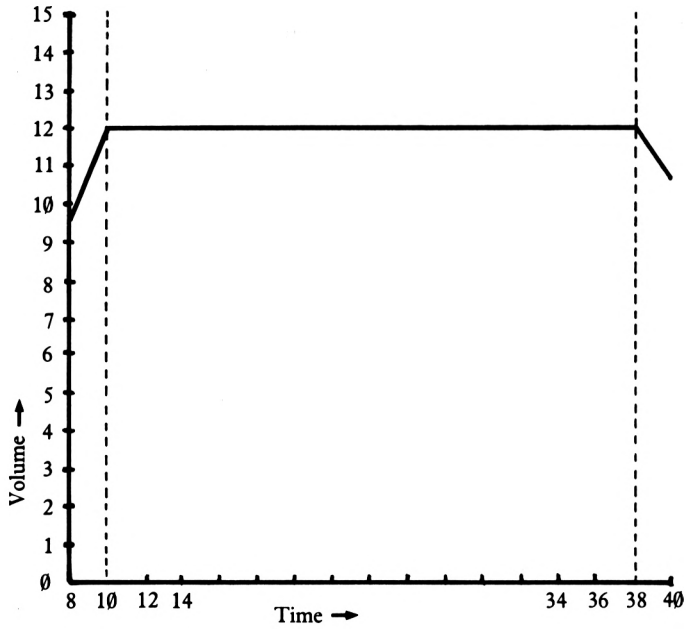
It can be utilised with the following SOUND command:

```
20 SOUND 1,100,0,1,1
```

A duration of 0 tells the computer to play the note for the duration specified in the ENV command. When seeing a volume envelope value of one, the computer uses the values in ENV 1, if such an envelope has been defined. In this example the note will last one tenth of a second (step count * pausetime), although subjectively, it sounds longer than this.

Sustain

The sustain phase is the period in which the note remains at its peak value. Once again the arguments are step count, step size and pause time. Figure 3.8 shows the sustain values.



The Sustain Values

FIGURE 3.8

Sustain, by definition, holds the volume that it begins with: therefore there is only one step, starting where the attack ends and finishing where the decay begins.

$$\text{Step size} = \frac{\text{volume increase}}{\text{step count}}$$

Because the previous volume is maintained during the sustain phase there is no volume increase, so step size = 0 (0/1=0).

$$\begin{aligned} \text{Pause time} &= \frac{\text{sustain time}}{\text{step count}} \\ &= 28 \div 1 \\ &= 28 \end{aligned}$$

Therefore the pause time equals 28 hundredths of a second in this case.

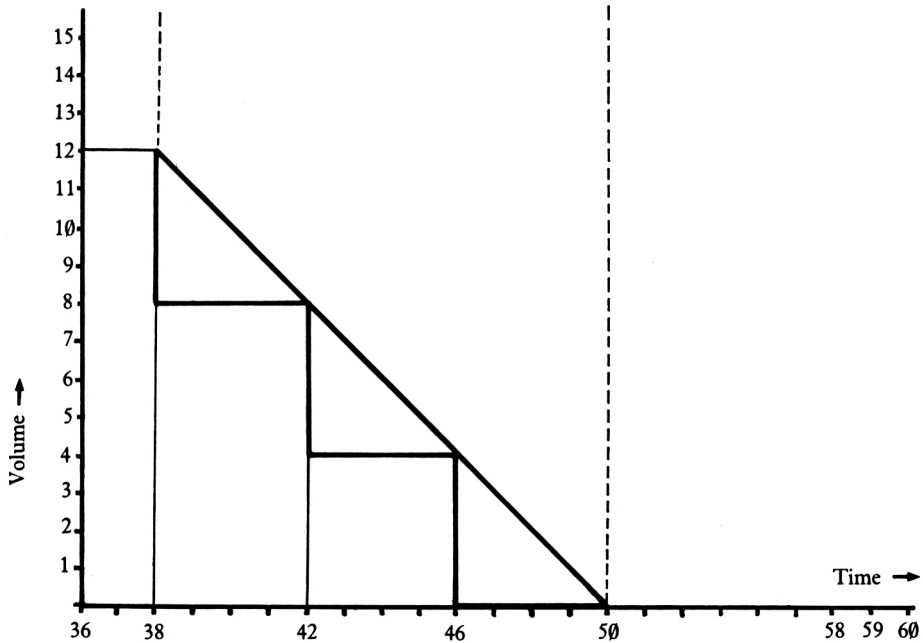
These extra values are added to line 10 giving us a new line 10:

10 ENV 1,5,2,4,2,1,0,28

the only thing left to calculate now is:

Decay

The decay time is the time taken for the note to fade out. Once again there are three arguments, only this time the step size is negative because the volume is decreasing. Figure 3.9 shows the decay values.



The Decay Values

FIGURE 3.9

In the example shown in Figure 3.9, the volume fades out at a rate of one unit of volume per hundredth of a second. The step size can be any of a number of values, the extremes being determined by the sustained volume and the step count which can take values from one to twelve. A step count of one will cause the note to decay in twelve different steps. This will cause the decay to appear very smooth. The other extreme value of twelve will cause the note to end suddenly. To compromise, a step count of three will be chosen, as in the above Figure 3.9. If you feel confident enough you might like to substitute your own step count value.

$$\text{step size} = \frac{\text{volume increase}}{\text{step count}}$$

$$= -12 \div 3$$

$$= -4$$

Also,

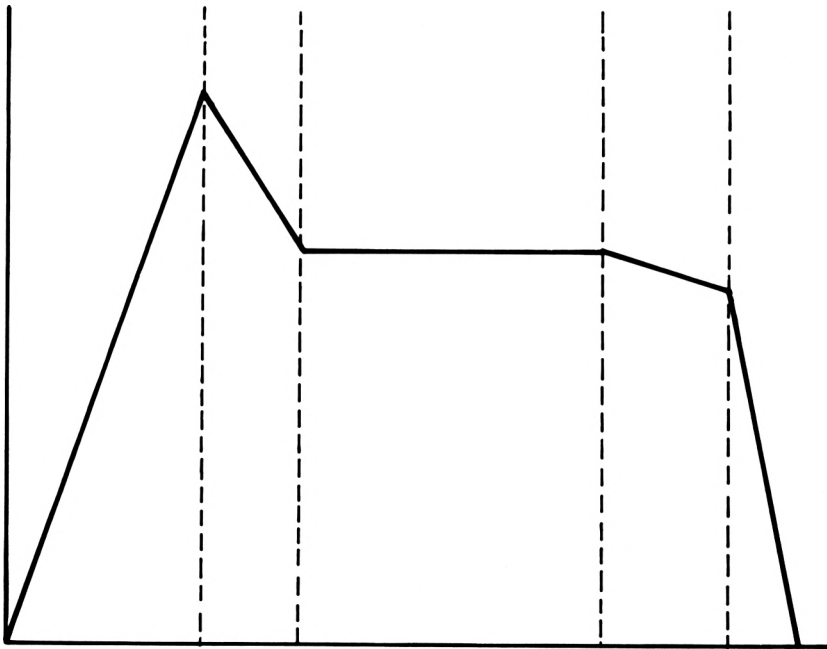
$$\text{pause time} = \frac{\text{decay time}}{\text{step count}}$$

$$= 12 \div 3$$

$$= 4$$

This produces a new line 10:

10 ENV 1,5,2.4,2,1,0,28,3,-4,4



A Five Section Envelope

FIGURE 3.10

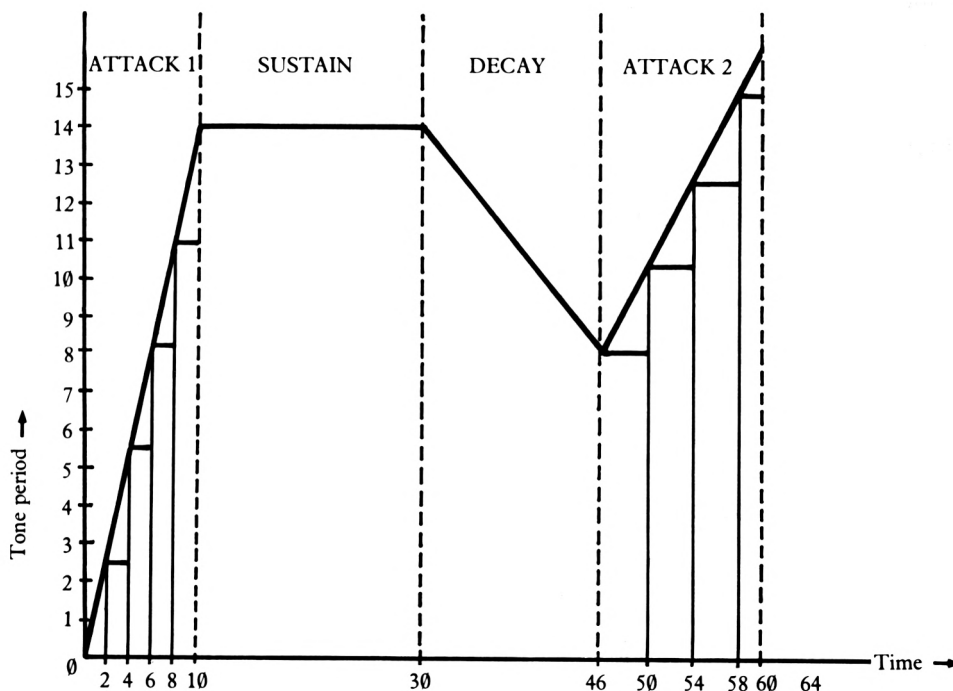
Just in case you've forgotten, to tell the computer to actually use the envelope that you've defined, the volume envelope argument in the SOUND statement (fifth number) should be the same as the first number in the ENV statement. A volume envelope value of one means use ENV 1, etc.

The Tone Envelope

The tone envelope is used to vary the frequency of a note as it is playing. The tone envelope works in much the same way as the volume envelope, and comes equipped with a statement similar to ENV, namely ENT.

ENT

Figure 3.11 shows an example tone envelope.



The Tone Shape

FIGURE 3.11

The particular tone shape chosen has two attacks, the tone ending in an attack form.

The calculations for the ENT statement are described in the following sections. They are very similar to the corresponding calculations for the ENV, as you will see.

Attack 1

$$\text{step count} = 5$$

$$\begin{aligned}\text{step size} &= \frac{\text{frequency increase}}{\text{step count}} \\ &= 14 \div 5 \\ &= 2.8\end{aligned}$$

$$\begin{aligned}\text{pause time} &= \frac{\text{attack time}}{\text{step count}} \\ &= 10 \div 5 \\ &= 2\end{aligned}$$

Thus the first section of the ENT command looks like this:

15 ENT 1,5,2.8,2

Sustain

$$\text{step count} = 1$$

$$\begin{aligned}\text{step size} &= \frac{\text{frequency increase}}{\text{step count}} \\ &= 0 \div 1 \\ &= 0\end{aligned}$$

$$\begin{aligned}\text{pause time} &= \frac{\text{sustain time}}{\text{step count}} \\ &= 20 \div 1 \\ &= 20\end{aligned}$$

Thus the ENT command so far looks like this:

15 ENT 1,5,2.8,2,1,0,20

Decay

$$\text{step count} = 16$$

$$\text{step size} = \frac{\text{frequency increase}}{\text{step count}}$$

$$= -6 \div 16$$

$$= -0.375$$

$$\text{pause time} = \frac{\text{decay time}}{\text{step count}}$$

$$= 16 \div 16$$

$$= 1$$

At this stage, the ENT command looks like this:

```
15 ENT 1,5,2.8,2,1,0,20,16,-0.375,1
```

Attack 2

$$\text{step count} = 4$$

$$\text{step size} = \frac{\text{frequency increase}}{\text{step count}}$$

$$= 8 \div 4$$

$$= 2$$

$$\text{pause time} = \frac{\text{attack time}}{\text{step count}}$$

$$= 14 \div 4$$

$$= 3.5$$

The completed ENT command looks like this:

```
15 ENT 1,5,2.8,2,1,0,20,16,-0.375,1,4,2,3.5
```

The major difference between ENT and ENV is that ENT does not affect the duration of a note. If the time allowed for the ENT command is greater than the sound length (as specified by the third parameter of the SOUND command) then the last values will not be played. If it is too short then the tone value used in the sound command will play only until the end of the note.

The SOUND command calls up each tone envelope by its number: in this example we are using 'ENT 1'. The SOUND command that calls it up will look like this:

```
20 SOUND 1,100,0,1,1,1
```

Although the ENT command does not affect the length of a note, it is possible to force the tone envelope to last to the end of the note. This is done by placing a minus sign at the beginning of the ENT number, e.g.

```
15 ENT -1, etc.
```

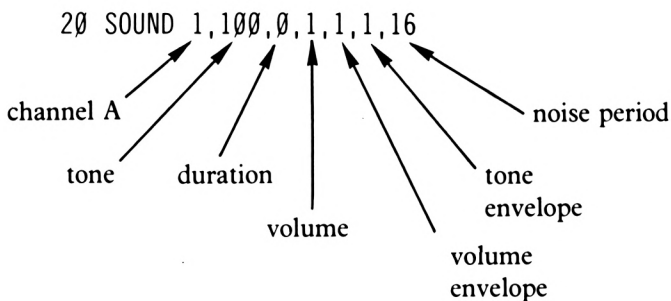
will cause the tone envelope to last the full duration of the note. This will only work if the tone envelope's duration is less than the note's length.

It is important to note that when using envelopes, the attack, sustain and decay values do not have to increase, remain the same or decrease, respectively. It is perfectly allowable for the first values of an envelope command to have a minus step size, i.e. a decreasing attack. The results will be perfectly logical, depending on what has gone before.

Another thing you may have noticed is that if you play the above sound and ENT envelope, the sound will behave according to both the ENT envelope and the earlier ENV envelope – assuming you haven't switched the computer off before trying this out. If the program currently in your computer still contains the ENV definition, try deleting it and running the program again. The actual sound produced does not change. This is because envelopes stay set until they are deliberately reset. To reset an envelope, just type in, for example 'ENV 1', with no further arguments, to cancel volume envelope 1.

Noise period

The final SOUND channel argument is noise. This is a value in the range 0 to 31. The noise produced is called 'white noise' and sounds like radio static. A noise value of 1 has a higher pitch than that of 31. To demonstrate noise, edit line 20 so that it looks like this:



That is all the arguments for the sound command. The next stage is to use the ENV and ENT commands to play music.

To illustrate the versatility of the SOUND command arguments type in Program 3.6. This is a short easy way to play a sequence of notes.

PROGRAM 3.6

```
10 FOR X=1 TO 5
20 READ TONE
30 SOUND 1,TONE
40 NEXT X
400 DATA 32,28,36,71,47
```

Upon running this program you may recognise the tune and notice that the notes are played too quickly. This is easily remedied by including a duration value on line 30. Now each note will last half a second.

PROGRAM 3.6(a)

```
30 SOUND 1,TONE,50
```

Now the tune sounds almost perfect, played at just the right speed. The next step is to add a volume envelope. Type in the following:

PROGRAM 3.6(b)

```
5 ENV 1,10,5,2
30 SOUND 1,TONE,50,0,1
```

This distorts the sound quite a bit producing a most peculiar version of the tune. Add the rest to line 5 and a longer duration in line 30 and the sound will appear a little more distorted:

PROGRAM 3.6(c)

```
5 ENV 1,10,5,2,2,0,8,12,3,10
30 SOUND 1,TONE,100,0,1
```

To create even more of a distortion add the following line which makes use of a tone ENTelope (!).

PROGRAM 3.6(d)

```
7 ENT 1,4,5,2,0,16,4,3,12,1
30 SOUND 1,TONE,50,0,1,1
```

There is one final trick that can be used to destroy the tune even further and that is to play the notes as random noise. This is done by placing a '16' at the end of line 30 as follows:

PROGRAM 3.6(e)

```
30 SOUND 1, TONE, 50, 0, 1, 1, 16
```

At this point, the program sounds nothing like it used to. You might like to play the tune to someone and see if they can guess what it is.

Chapter 4

When the first colour computers were launched upon the home computer market, the video-game industry boomed. It is doubtful that computers would have had the same massive impact on the public if it hadn't been for computer games. Playing computer games can be very enjoyable and, perhaps surprisingly, so can writing them. The object of this chapter is to demonstrate what goes into computer games and how they are developed.

Part one of this chapter will look at some general techniques and simple routines which are needed in one form or another in most video games. Part two of this chapter will go on to use some of these ideas in developing a new game, 'Albert'.

PART ONE

Video-Game Mechanics

The main ingredient of every video game is a character that the user moves around the screen. The character we will attempt to move around the screen is called Albert and can be viewed by typing in:

```
PRINT CHR$(249)
```

On the screen you will see a small humanoid character; this is Albert. This character can be placed anywhere on the screen simply by locating the print cursor and printing the character. For example Program 4.1 clears the screen and then randomly displays CHR\$(249) – Albert – all over the screen, X and Y being randomly variable coordinates.

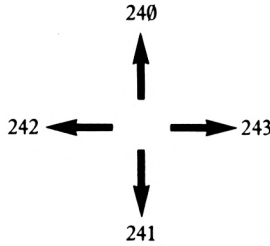
PROGRAM 4.1

```
10 MODE 1
20 x=INT(RND*40)+1
30 y=INT(RND*25)+1
40 LOCATE x,y:PRINT CHR$(249)
50 GOTO 20
```

Program 4.1 is a very simple program used to display a character randomly on the screen. The only user participation in this program is the guessing of where Albert will appear next – not much fun. The program can be stopped by pressing ESC twice.

Moving Albert

ALBERT has to be moved by the user. When one of the arrow keys is pressed, Albert will move in the indicated direction. Figure 4.1 illustrates the arrow keys, Albert's direction and the character values of the arrow keys.



The Arrow Keys and Character Values

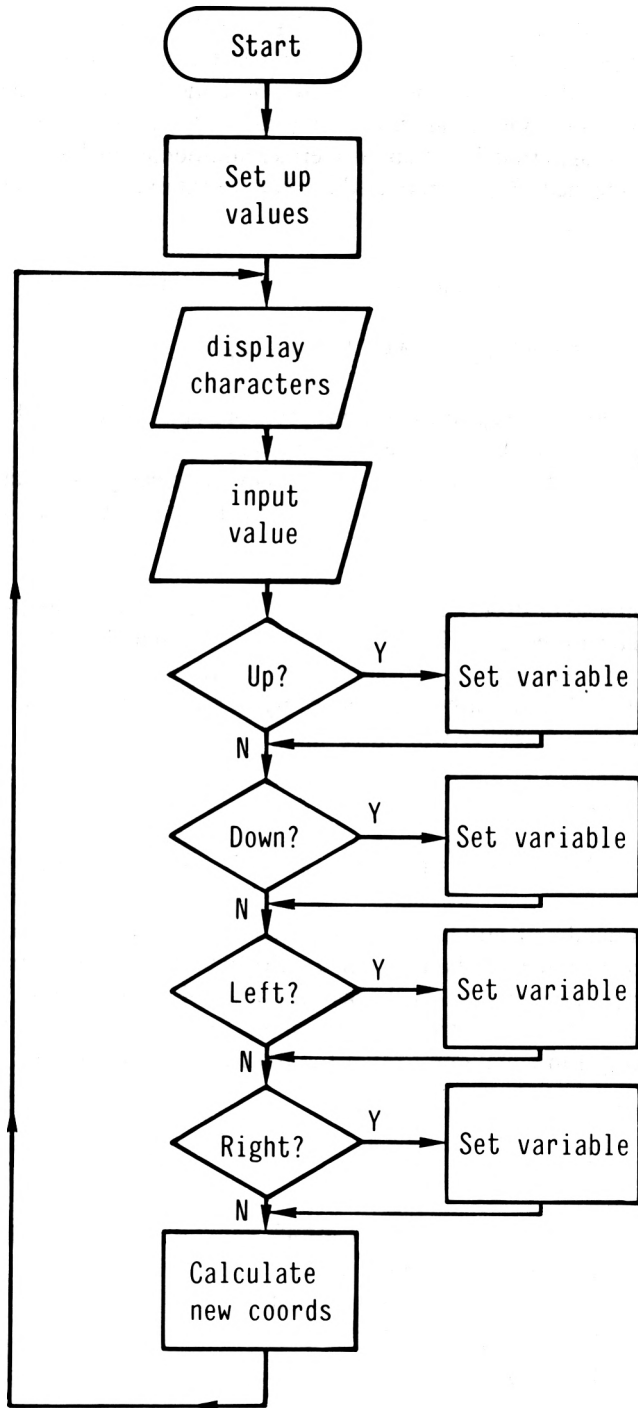
FIGURE 4.1

If the CHR\$ value of the character input is 240 then Albert moves up the screen, if the input is CHR\$(241) then Albert moves down the screen, etc. Translating this into BASIC yields Program 4.1(a). The variables X and Y are used as location pointers. When the program is first run their values are set to 20 and 12 respectively; therefore the character is first printed in the twentieth column across and the twelfth row down.

PROGRAM 4.1(a)

```
10 MODE 1
20 x=20:y=12:REM start location
30 LOCATE x,y:PRINT CHR$(249)
40 a$=INKEY$
50 IF a$=CHR$(240) THEN yadd=-1
60 IF a$=CHR$(241) THEN yadd=+1
70 IF a$=CHR$(242) THEN xadd=-1
80 IF a$=CHR$(243) THEN xadd=+1
90 x=x+xadd:y=y+yadd
100 GOTO 30
```

When run, this program will display a character in the centre of the screen, nothing happens until you press one of the arrow keys. As soon as a key is pressed the character shoots off in the direction chosen. One key press causes the character to continue moving until it runs off the edge of the screen, producing an 'Improper argument' error message to be displayed. To understand why the character continues to move even when the key is no longer being pressed, have a look at Figure 4.2:



Flow Chart of Program 4.1(a)

FIGURE 4.2

As Figure 4.2 indicates, Program 4.1(a) continually loops around lines 30 to 100 even if no key has been pressed. When first run, XADD and YADD have no values. Therefore the additions on line 90 do not change the values of X and Y and, subsequently, the character does not move. Once one of the arrow keys is pressed, e.g. the left-arrow key, the character continues moving in that direction until either another arrow key is pressed or the edge of the screen is reached. What is required is some way of moving the character only when an arrow key is being pressed.

A solution to this problem is to include a test on line forty to see if any key was pressed i.e.

```
40 A$=INKEY$:IF A$="" THEN 40
```

When this is added to Program 4.1(a) one press of a key will move Albert one position. The program appears to work, but there is one slight fault. Pressing a key will move Albert in the required direction, but once one of the arrow keys has been pressed (and XADD or YADD given a value), pressing any other key will move the character in the same direction as the previous one.

To fix this, line 45 of Program 4.1(b) needs to be added. This tests the ASCII value of the input character to see if it is less than 240 or greater than 243. If it is, then the character pressed was not an arrow so the Program loops back to line 40 for another input. In this way the character will move only upon the press of an arrow key. If the right-arrow key is pressed twice the character will move twice!

PROGRAM 4.1(b)

```
45 IF ASC(a$)<240 OR ASC(a$)>243 THEN 40
```

This slight amendment still leaves one problem: the character can still wander off the edge of the screen. To remedy this situation, a series of tests needs to be incorporated at the end of the program. Program 4.1(c) checks the values of X and Y and if they are outside the allowed range, sets them to the extreme values. For example, when Y is less than one, its value is changed to equal one so that the Amstrad doesn't attempt to LOCATE off the screen.

PROGRAM 4.1(c)

```
100 IF x>40 THEN x=40
110 IF x<1 THEN x=1
120 IF y>24 THEN y=24
130 IF y<1 THEN y=1
140 GOTO 30
```

Note that line 120 tests to see if the Y value is greater than 24, not 25. If you move the character along the twenty fifth line of the screen, when it reaches the far right of the screen (location 40,25), the computer performs a scroll – i.e. it moves the screen display up one line. The Amstrad interprets the printing of a character at this location as a cry for more room and so makes the screen scroll up one line.

Now that we have a moving character program that actually works, we can begin to tidy it up somewhat. The two main inconveniences with it are that when the character moves across the screen, it leaves a trail, and that the movement quickly becomes restricted to a diagonal direction. Upon the press of an arrow key, a variable (either XADD or YADD) is set and this variable remains constantly set to either -1 or +1. Thus it is possible for both variables to be set at the same time. When this happens the character moves diagonally. To prevent this from happening the variable YADD needs to be set to zero whenever XADD is set and XADD set to zero whenever YADD is set.

PROGRAM 4.1(d)

```
50 IF a$=CHR$(240) THEN yadd=-1:xadd=0
60 IF a$=CHR$(241) THEN yadd=+1:xadd=0
70 IF a$=CHR$(242) THEN xadd=-1:yadd=0
80 IF a$=CHR$(243) THEN xadd=+1:yadd=0
```

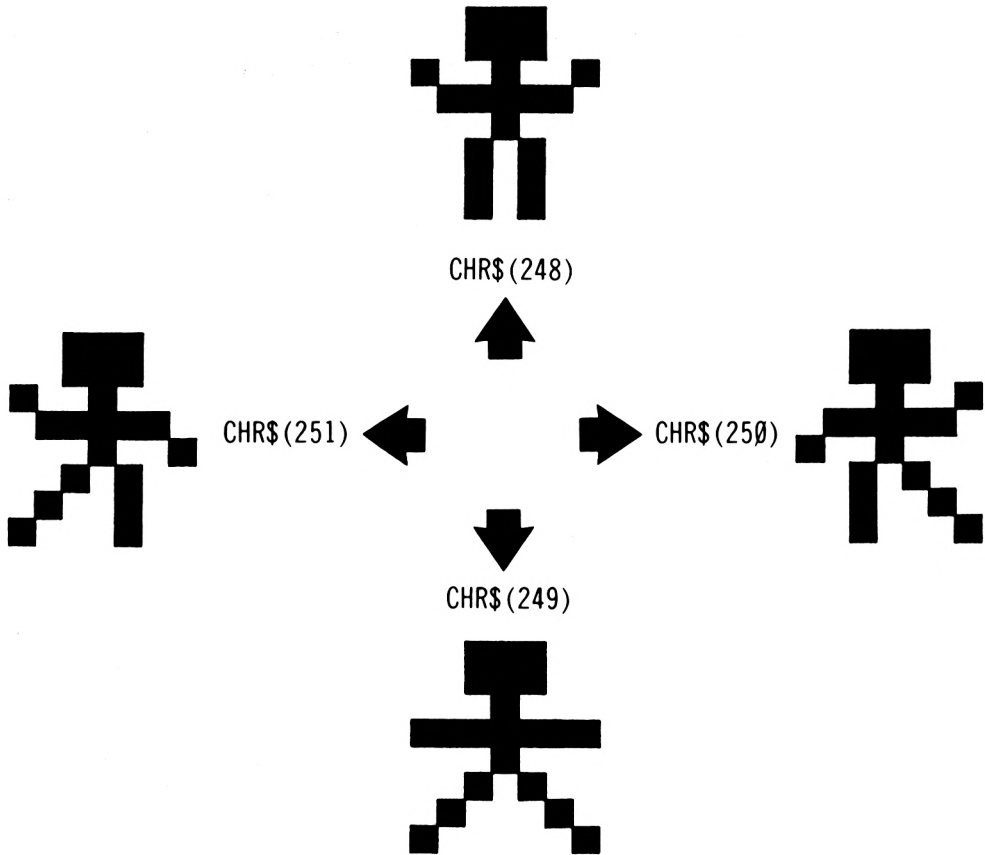
The modifications added to lines 50 to 80 make sure that the character can only move in the vertical/horizontal directions.

Now to get rid of the character's trail. The trail is caused by leaving a copy of the character in the position last indicated by the X and Y coordinates. This problem can be overcome by simply printing a space in the character's location prior to moving to the new X,Y location. To do this, two new variables are needed to remember the old values of X and Y. These variables will be called OLDX and OLDY. Line 85 sets them to equal X and Y respectively before the values of X and Y are changed. Line 135 then prints a space in the old location before line 140 directs the program back to line thirty where the new character position is located and Albert printed.

PROGRAM 4.1(e)

```
85 oldx=x:oldy=y
135 LOCATE oldx,oldy:PRINT " "
```

With the inclusion of Program 4.1(e) the character moving routine is almost perfect. Albert can be moved anywhere on the screen without leaving a trail behind. Before leaving this program, however, let us look at one possible way of improving it. At the moment the same character is printed irrespective of the direction in which Albert is moving. In the Amstrad's character set there are four different Albert characters, one for each direction, as shown in Figure 4.3.



The Characters For Each Direction

FIGURE 4.3

The character that is to be printed is controlled by the direction in which Albert is to be moved. The variable C (Character) is set to the ASCII value of the appropriate character following a press of one of the arrow keys. For example if the left arrow is pressed the value of C is set to 251 and the character printed is a left-facing Albert.

PROGRAM 4.1(f)

```

50 IF a$=CHR$(240) THEN yadd=-1:xadd=0:c=248
60 IF a$=CHR$(241) THEN yadd=+1:xadd=0:c=249
70 IF a$=CHR$(242) THEN xadd=-1:yadd=0:c=251
80 IF a$=CHR$(243) THEN xadd=+1:yadd=0:c=250

```

Line 30 needs to be amended so that it prints the CHR\$ value of C. C is initially set to 249 at the start of the program so that the first character displayed will be the downwards moving Albert. These changes are included in Program 4.1(g) which is a listing of the complete character moving Program.

PROGRAM 4.1(g)

```
10 MODE 1
20 x=20:y=12:REM start location
22 c=249:REM first character value
30 LOCATE x,y:PRINT CHR$(c)
40 a$=INKEY$
45 IF ASC(a$)<240 OR ASC(a$)>243 THEN 40
50 IF a$=CHR$(240) THEN yadd=-1:xadd=0:c
   =248
60 IF a$=CHR$(241) THEN yadd=+1:xadd=0:c
   =249
70 IF a$=CHR$(242) THEN xadd=-1:yadd=0:c
   =251
80 IF a$=CHR$(243) THEN xadd=+1:yadd=0:c
   =250
85 oldx=x:oldy=y
90 x=x+xadd:y=y+yadd
100 IF x>40 THEN x=40
110 IF x<1 THEN x=1
120 IF y>25 THEN y=25
130 IF y<1 THEN y=1
135 LOCATE oldx,oldy:PRINT" "
140 GOTO 30
```

Firing

The second most important ingredient of a video game is the ability to destroy the baddies. Albert will be able to fire if the COPY key is pressed. Program 4.1(g) can be modified to check if the COPY key is being pressed by testing the INKEY\$ value to see if it equals CHR\$(224), the value of the copy key.

PROGRAM 4.2

```
42 IF ASC(a$)=224 THEN GOSUB 1000
```

Now that the program includes a test for the fire button, the actual fire routine needs to be developed. The player can fire in four directions (left, right, up and down), the direction of fire being controlled by the direction the player was moving in when the COPY key was pressed. The variable C (the ASCII value of Albert's character) can be used to work out the player's direction: if C=251 then the player is moving to the left, or if C=249 the player is moving down the screen, etc. Line 1010 subtracts 247 from C and stores the value in variable T. Thus when T=1 the player is moving up the screen, when T=2 the player is moving down the screen etc. etc. The ON GOTO command on line 1020 sends the program to the relevant fire routine, (one fire routine per direction).

PROGRAM 4.2(a)

```
1000 REM open fire
1010 t=c-247:REM t controls fire directi
on
1020 ON t GOTO 1100,1200,1300,1400
```

When firing a 'missile', it should reach the edge of the screen and then stop. Because there are four possible directions there are four screen edges, four separate edge tests and consequently four individual firing routines.

The routine for firing **up** the screen needs to print a bomb character in every row from the position of the character up to the top row:

```
LOCATE X,Y to LOCATE X,1
```

X and Y being Albert's current coordinates.

The firing routine can use a loop:

PROGRAM 4.2(b)

```
1110 FOR fy=y TO 1 STEP -1
```

Inside this loop, three things need to be printed. The missile, a blank space over the last position of the missile and finally Albert. The Albert character needs to be printed inside the loop because the first missile will be printed at the same coordinate position as Albert, thus effectively removing Albert from the screen. Unless the Albert character is reprinted, it will not reappear on the screen until one of the arrow keys is pressed. Line 1140 is the line that prints Albert back onto the screen.

PROGRAM 4.2(c)

```
1120 LOCATE x,fy:PRINT CHR$(231)
1140 LOCATE c,y:PRINT CHR$(c)
```

If the program as a whole is run and the character moved, at least once, upwards before pressing the copy key, the missile will be drawn from the current character position to the top line of the screen (directly above the Albert character). Once the missile reaches this point the program will stop. What is left to do is remove the last copy of the bomb from the screen and then RETURN back to the character moving routine.

PROGRAM 4.2(d)

```
1160 LOCATE x,1:PRINT" "
1170 RETURN
```

The routine from lines 1100 to 1170 only works when the player is firing up the screen. Any attempt to fire whilst moving in another direction (down, left or right) will result in the computer displaying a 'Line does not exist' error message. This is because the other firing direction routines have not been written yet, of course. A fire down routine differs from a firing up routine in that the loop is from Y to 24.

PROGRAM 4.2(e)

```
1200 REM fire down the screen
1210 FOR fy=y TO 24 STEP +1
1220 LOCATE x,fy:PRINT CHR$(231)
1230 LOCATE x,fy-1:PRINT" "
1240 LOCATE x,y:PRINT CHR$(c)
1250 NEXT fy
1260 LOCATE x,24:PRINT" "
1270 RETURN
```

When firing left and right (as one might expect) it is the X coordinate that needs to be increased or decreased, depending on the fire direction. When firing right, the missile's X coordinate needs to start at X and increase in steps of one until it reaches a value of 40, line 1310 of Program 4.2(f):

PROGRAM 4.2(f)

```
1300 REM fire right of the screen
1310 FOR fx=x TO 40 STEP +1
1320 LOCATE fx,y:PRINT CHR$(231)
1330 LOCATE fx-1,y:PRINT" "
1340 LOCATE x,y:PRINT CHR$(c)
1350 NEXT fx
1360 LOCATE 40,y:PRINT" "
1370 RETURN
```

The last firing routine is that which fires to the left. To do this the missile's X coordinate needs to be decreased by one until reaches a value of one.

PROGRAM 4.2(g)

```
1400 REM fire left
1410 FOR fx=x TO 1 STEP -1
1420 LOCATE fx,y:PRINT CHR$(231)
1430 LOCATE fx+1,y:PRINT" "
1440 LOCATE x,y:PRINT CHR$(c)
1450 NEXT fx
1455 SOUND 1,50,50,15,0,1,1
1460 LOCATE 1,y:PRINT" "
1470 RETURN
```

With the inclusion of Program 4.2(g), the user can move the Albert character around the screen and fire a missile in any direction other than diagonally. This firing mechanism, delightful though it is, is a bit primitive. At the moment, all that happens is a missile (a cobalt bomb actually) is released and rushes to the end of the screen where it disappears. As most of you are probably aware, most cobalt bombs when launched emit a sound something like a broken police siren, and upon reaching their destination, explode. To reproduce these sound effects on the Amstrad we need to use the sound command and design some envelopes. These elements are included by adding Program 4.2(h) to the program currently in memory.

PROGRAM 4.2(h)

```
5 GOSUB 2000:REM set envelopes

1005 SOUND 2,50,50,15,0,2

1155 SOUND 1,50,50,15,0,1,1
1255 SOUND 1,50,50,15,0,1,1
1355 SOUND 1,50,50,15,0,1,1
1455 SOUND 1,50,50,15,0,1,1

2000 REM set envelopes
2010 ENT 1,10,4,2,20,-4,1,10,4,3
2020 ENV 1,20,30,10,20,-10,30,40,10,50
2030 ENT 2,0,4,10,40,-8,5,0,4,10
2040 RETURN
```

The Enemy

The third most important element in a computer game is the opponent. In most computer games, the opponent is the computer itself. Games like space invaders and pacman have the computer moving multiple characters on the screen in order to destroy the player's character. To develop an enemy routine you must first delete the program currently in memory (you can save it first if you wish!). When your computer has been NEWed you are ready to begin.

Movement

The movement of a computer-controlled character is very similar to that of the player's character except the computer chooses the direction. The computer's choice of direction can be calculated using a random number: if the random value equals one, then the computer's character will move up the screen, etc. The structure and content of the enemy-moving routine is so similar to the player movement routine that the only real change is that the direction is chosen with a random number (lines 30-70) and the X and Y variables are different. BX and BY are the coordinates of the baddie, BX2 and BY2 remember the previous position of the baddie and finally BXADD and BYADD control the actual direction in the same way that XADD and YADD controlled the direction of the player's characters.

PROGRAM 4.3

```
10 MODE 1
20 bx=20:by=20
30 d=INT(RND*4)+1
40 IF d=1 THEN byadd=-1:bxadd=0
50 IF d=2 THEN byadd=+1:bxadd=0
60 IF d=3 THEN bxadd=-1:byadd=0
70 IF d=4 THEN bxadd=+1:byadd=0
80 bx2=bx:by2=by
90 bx=bx+bxadd:by=by+byadd
100 IF bx>40 THEN bx=40
110 IF bx<1 THEN bx=1
120 IF by>24 THEN by=24
130 IF by<1 THEN by=1
140 LOCATE bx2,by2:PRINT " "
150 LOCATE bx,by:PRINT CHR$(224)
160 GOTO 30
```

When run, this program will continually display a happy smiling face moving in a random direction all over the screen. At the moment the computer's character, like the user controlled character, can only move up, down, left and right, not diagonally. The object of any computer game is for the computer to prevent the player from winning; to facilitate this the computer's character needs to have an edge. In this case the advantage the computer will have is the ability to wander off one edge of the screen and to re-appear on the other side of the screen: when BX becomes greater than forty it is reset to a value of one and vice versa. The same idea applies to the Y direction variables. Thus lines 100 to 130 need to be amended accordingly.

PROGRAM 4.3(a)

```
100 IF bx>40 THEN bx=1
110 IF bx<1 THEN bx=40
120 IF by>24 THEN by=1
130 IF by<1 THEN by=24
```

Now the character can be seen flickering from side-to-side of the screen; so far so good. It would be an interesting feature of the game if the computer's character deliberately left a trail behind it.

This will present the player with double jeopardy: not only will the moving smiling face have to be avoided, but also the stationary happy faces will have to be avoided.

The character being printed is a happy face, but it would be nice if an unhappy face (CHR\$(225)) was printed in the old position (location BX2,BY2). Not only can different characters be printed, but they can also have different colours; for now, the moving face will be red and the stationary face will be printed in yellow. Only lines 140 and 150 need to be changed. Type in these modifications and then run the program.

PROGRAM 4.3(b)

```
140 LOCATE bx2,by2:PEN 1:PRINT CHR$(225)
150 LOCATE bx,by:PEN 3:PRINT CHR$(224)
```

Not only does the baddie run around the screen in continual pursuit of the prey, but it also leaves a messy trail, until eventually the screen will be full of unhappy yellow faces.

Multiple Enemies

It was mentioned previously that the computer needed to have an advantage when playing in a video game; often this is done by giving the computer more than one computer-controlled character. Moving multiple objects across the screen creates a few problems – for example, where do the characters start from? How can a program remember the coordinates for umpteen characters? Also, how can the characters be moved simultaneously? The answers to these questions are: randomly, arrays and they can't.

Random Start

For demonstration purposes, it will be assumed that the computer is controlling five characters on the screen. These five characters need to begin moving in five different locations. If the start locations are chosen using random numbers then the chances of two characters appearing at the same location are slight.

Remember Arrays

The problem of remembering the X and Y coordinates of each character can be easily solved using arrays; one array to remember the X coordinate of each character and another array to remember the Y coordinate. Thus the first part of a multi-movement routine is the setting up of the character arrays, BX() and BY(), and then placing a random X and Y coordinate in each array value.

PROGRAM 4.4

```
10 REM multi baddie movement
20 bad=5:REM number of baddies
30 DIM bx(bad),by(bad)
40 FOR loop=1 TO bad
50 r=INT(RND*24)+1:by(loop)=r
60 c=INT(RND*40)+1:bx(loop)=c
70 NEXT loop
```

The number of characters to be moved is set as a variable (BAD) in line twenty of Program 4.4. Because this is a variable, the number of baddies can be increased or decreased at will: only line twenty needs to be changed.

Simultaneous Movement?

Five characters cannot all be moved at exactly the same time, but they can be moved one position at a time in sequence. Using a loop it is possible (in the same way as before) to calculate the position of a character, locate it and then print the character there. The loop counter is then increased and the process repeated for each of the characters.

PROGRAM 4.4(a)

```
90 MODE 1
100 REM time to move
110 FOR loop=1 TO bad
120 d=INT(RND*4)+1
130 IF d=1 THEN byadd=-1:bxadd=0
140 IF d=2 THEN byadd=+1:bxadd=0
150 IF d=3 THEN bxadd=-1:byadd=0
160 IF d=4 THEN bxadd=+1:byadd=0
170 bx2=bx(loop):by2=by(loop)
180 bx(loop)=bx(loop)+bxadd
190 by(loop)=by(loop)+byadd
200 IF bx(loop)>40 THEN bx(loop)=1
210 IF bx(loop)<1 THEN bx(loop)=40
220 IF by(loop)>24 THEN by(loop)=1
230 IF by(loop)<1 THEN by(loop)=24
240 LOCATE bx2,by2:PEN 1:PRINT CHR$(225)
250 LOCATE bx(loop),by(loop):PEN 3:PRINT
    CHR$(224)
260 NEXT loop
270 GOTO 100
```

The multi-baddie movement routine is very similar to the single-baddie movement routine. The only programming differences are that the movement routine is enclosed in a loop (from one to the number of characters) and the values of BX and BY in Program 4.3 have been replaced with their array equivalents, BX(loop) and BY(loop). When this program is run you will see five red happy faces moving around the screen, all five leaving a trail of yellow unhappy faces. Have you noticed how much slower these characters move compared with the character in the single-character moving routine? The speed of movement for each of the five characters is approximately 1/5th of the speed of the character in the single-character moving routine, simply because the computer is having to do five times as much work.

PART TWO

Albert: Writing A Jigsaw Puzzle

Before work on a video game can begin, the programmer has to make some important decisions, the most important being the object of the game. It is not a good idea to attempt a large scale program without being absolutely sure what the program should do. A complex program can be regarded as being like a jigsaw puzzle. Without the picture (the idea) it is much more difficult to connect all the pieces (the subroutines) together correctly. So, before we can begin to consider developing a video game, four major things need to be considered:

1. The object of the game.
2. Screen display
3. Player participation.
4. Computer participation

There is a well known, but unspoken law in computing: 'A program is never finished, merely completed to a satisfactory level'. It is very easy when writing a program (especially a game program) to get sidetracked from your original objective. It is important that the program should be completed to the first specification you write before any extra features are added. Not only does this provide you with a 'completed' program, but if the program was well structured, it also makes it easier to add extra features.

The Object of the Game

There is a well-known law in software development companies: 'a program is never finished, just completed to a satisfactory level'. The object of Space Invaders is to shoot all the invaders, in Pacman you have to eat all the energy dots.

There are two important considerations to bear in mind when developing a video game concept: practicability and time. The idea you have has to be practical, i.e. if you can't program it or if the program needs more memory than is available then it is not practical. Time needs to be considered because slow games are usually boring. Have you ever tried to play a slow Pacman? Have you ever seen a slow Pacman? If the object of your game was to dodge one hundred bouncing oranges for as long a time a possible did you consider how long it would take to make all one hundred oranges bounce? Also, if it is going to take you too long to write the program, that can be boring too! Think about it before starting.

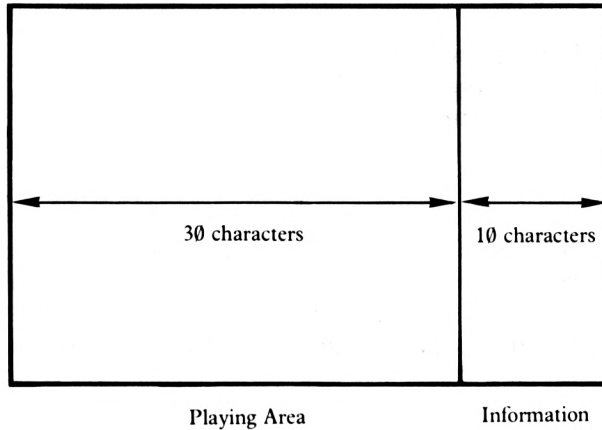
After much thought along these lines the author has come up with a game object that is very simple and easy to develop. The object of the game is to:

Move a character around the screen, picking up objects randomly displayed there. When all the objects have been picked up, the game is over.

That is the essence of the game. Features like how the characters will appear on the screen and how they will be picked up will be dealt with in the relevant sections ('Computer Participation' and 'Player Participation' respectively).

Screen Display

The screen is to display two things: the characters to pick up and player information (score, time, etc.). Because there are two types of information to be displayed, the screen will be split into two. The left side will be the playing area, and the right the player information screen. Figure 4.4 shows what the split screen will look like.



The Split Screen Display

FIGURE 4.4

Player Participation

The player has to run around the screen picking up randomly displayed objects. As well as this, the player is required to avoid contact with a character the computer will move around the screen.

In Part One of this chapter, a player movement routine was developed (Program 4.1(g)) and the routine used in this game will be very similar to that.

Computer Participation

The computer has to do all the work and that's the way it should be. Firstly the computer needs to clear the game area of the screen and display the random characters. Then the computer needs to keep a constant check on three things:

1. The player's character:

- Has it picked up an object?

- Has it wandered off the edge of the screen?

- Has it bumped into the computer's character?

2. Each of the computer's characters:

Has it picked up an object?
Has it wandered off the edge of the screen?
Has it bumped into the player's character?

3. Is the game over:

Has it picked up an object?
Has the player been killed?
Has it bumped into the player's character?

All of these conditions have to be sorted out by the computer between the movements of the characters. The placing of characters at random locations on the screen only needs to be performed once, at the start of the game. A baddie movement routine has already been developed in Part One of this chapter (Program 4.3(b)) and a similar routine will be used in the actual game.

Having considered these things, work can begin in earnest on the actual writing of the game. The first part of the game is to display the objects on the screen. Because the characters will be displayed in random locations, the X and Y coordinates are calculated using the random number command.

PROGRAM 4.5

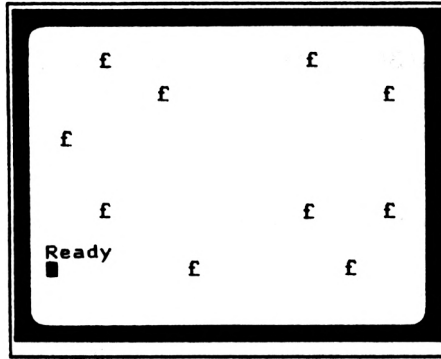
```
10 MODE 1

100 GOSUB 1000:REM display screen

999 END

1000 REM *****display screen*****
1010 FOR I=1 TO 10:REM ten objects
1020 X=INT(RND*30)+1
1030 Y=INT(RND*24)+1
1040 LOCATE X,Y:PRINT CHR$(163)
1050 NEXT I
1060 RETURN
```

The display routine from lines 1000 to 1060 places ten characters randomly on the screen. The actual character to be printed is a pound sign (£). The X coordinate is in the range of one to thirty because the last ten characters of the screen (locations 31 to 40) will be used to display game information. Figure 4.5 shows a typical display after running program 4.5.



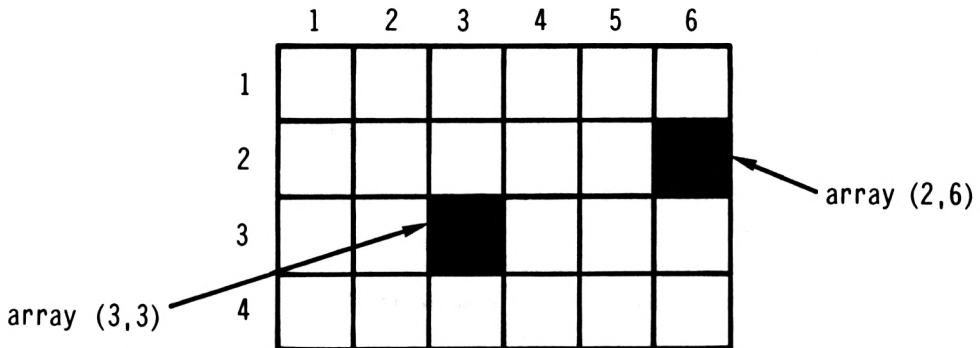
A Typical Display After Running Program 4.5

FIGURE 4.5

There is one slight problem with program 4.5: how can the computer tell where a pound sign has been printed? The most obvious answer is to use the TEST command but the problem with test is that it tests one pixel and we are dealing with whole characters having a total of two hundred and fifty six pixels! (16*16). The answer is to remember the locations, in an 'array'.

Two Dimensional Arrays

A two dimensional array can be regarded as a rectangular matrix of cells with X columns in one direction and Y rows in the other. See Figure 4.6



A Four By Six Two Dimensional Array

FIGURE 4.6

In a two dimensional array, each cell can be addressed individually by defining its coordinates in each direction. This is demonstrated in Figure 4.6 above. In order to remember the locations in which the pound signs are displayed we can set up a two dimensional array where each element represents a location on the screen. Then, when printing the pound characters, a value can be inserted into the array element corresponding to the location in which the pound sign is displayed. For example: if the X and Y locations of a pound sign is 10,12 then the screen array element (10,12) will be set to equal one, indicating a pound sign is at that location.

PROGRAM 4.5(a)

```
20 DIM sa(30,24):REM screen array

1035 IF sa(x,y)=1 THEN 1020
1045 sa(x,y)=1
```

Line 1035 tests to see if a pound sign has already been printed at location X,Y. If one has then the program loops back to choose another location.

The array can also be used to keep track of which screen locations are occupied by the baddies. This idea will be elaborated in the baddie movement section.

Having placed all the characters on the screen, the next step is to add a player-moving routine. This routine will allow the player to move Albert around the screen. This routine will differ from the one displayed in Part One of this chapter, (Program 4.1(g)) in that the routine needs to include a test to see if the player has moved into a location occupied by a pound sign. This is done by looking at the array element that corresponds with the player's location: if this value is equal to one, then the player is in a location occupied by a pound sign. This condition can be remembered by setting a flag, called SF (Score Flag).

PROGRAM 4.5(b)

```
2000 REM *****player movement*****
2010 a$=INKEY$:IF a$="" THEN 2010
2020 IF ASC(a$)<240 OR ASC(a$)>243 THEN
RETURN
2030 IF a$=CHR$(240) THEN my=-1:mx=0:c=2
48
2040 IF a$=CHR$(241) THEN my=+1:mx=0:c=2
49
2050 IF a$=CHR$(242) THEN mx=-1:my=0:c=2
51
2060 IF a$=CHR$(243) THEN mx=+1:my=0:c=2
50
2070 manx2=manx:many2=many
2080 manx=manx+mx:many=many+my
2090 IF manx>30 THEN manx=1:manx2=30
2100 IF manx<1 THEN manx=30:manx2=1
2110 IF many>24 THEN many=1:many2=24
2120 IF many<1 THEN many=24:many2=1
2130 IF sa(manx,many)=1 THEN sf=1
2140 LOCATE manx2,many2:PRINT " "
2150 LOCATE manx,many:PRINT CHR$(c)
2160 RETURN
```

Lines 2090 to 2120 tests to see if the player's character has reached the edge of the screen. If it has then it reappears at the other side of the screen. The player movement routine needs to be incorporated into the main program.

In Program 4.5(c) below, line 110 chooses a random start for the player's character, then line 120 tests to see if a pound sign has already been printed at this location. If one has, then the player's start position is recalculated and will continue to be recalculated until the player's character starts at an empty location.

PROGRAM 4.5(c)

```
110 manx=INT(RND*30)+1:many=INT(RND*24)+1
120 IF sa(manx,many)=1 THEN 110
130 LOCATE manx,many:PRINT CHR$(249)
```

If you run the program now, the screen will clear, then ten pound signs will be randomly displayed and then the player's character printed. At this point, because the player's movement subroutine has not been included, the program stops. The player movement subroutine needs to be called up continuously, to allow continuous movement. Line 200 of Program 4.5(d) below calls up the movement routine and line 390 loops back to line 200. The area between 200 and 390 is the main part of the program where all the flag testing will take place.

PROGRAM 4.5(d)

```
200 GOSUB 2000:REM player movement
390 GOTO 200
```

Now, when run, the program allows you to move your character around the screen. To experiment, move the character off each edge of the screen. If Program 4.5(b) was typed in correctly, Albert will appear at the other side of the screen.

The next section of the program to add is the scoring routine, which is called when the player has picked up a pound sign.

The scoring routine needs to do three things: increase the player's score, subtract one from the object count and set the appropriate element in the screen array to zero. The player's score will increase by a random amount each time a pound is picked up – the actual value being anything from ten to one hundred (line 1110 of Program 4.5(e) below). Line 1120 subtracts one from a variable called LO, for 'Left Objects'; this variable will have to be initialised at the start of the program to a value of ten (because ten objects are printed on the screen).

PROGRAM 4.5(e)

```
1100 REM *****score section*****
1110 score=score+10*INT(RND*10)+1
1120 lo=lo-1:REM object count
1130 sa(manx,many)=0
1140 SOUND 1,100,50,12
1150 RETURN
```

Line 1140 provides a rather unimaginative bleep whenever the player has scored some points. The sound should be left as it is for the moment because at this stage we are attempting to develop a game in its simplest form only. The core section fits into the main program via line 210. Upon returning from the scoring routine, the score flag is reset to zero: this prevents the scoring routine being called up even when a pound sign has not been picked up.

PROGRAM 4.5(f)

```
30 lo=10:REM ten objects
210 IF sf=1 THEN GOSUB 1100:sf=0
```

Another part of the game that can now be added is that which tells the player how many objects are left on the screen (the value of LO) and the score so far. This information will be displayed in the player information section of the screen.

PROGRAM 4.5(g)

```
370 LOCATE 33,6:PRINT"Score"
375 LOCATE 34,8:PRINT score
380 LOCATE 32,2:PRINT"Object"
385 LOCATE 34,4:PRINT lo
```

The only part of the game that is missing is that which announces that the player has won. To remedy this situation, add Program 4.5(h) which, upon a test of the variable LO, informs the player that all the pound notes have been picked up. The player is then asked whether 'another go' is required.

PROGRAM 4.5(h)

```
200 IF lo=0 THEN 800:REM you have won
800 REM *****you have won*****
810 CLS
820 LOCATE 12,2:PRINT"**You have won!**"
830 LOCATE 4,4
840 PRINT"Congratulations, Congratulatio
ns"
850 LOCATE 6,6
860 PRINT"and then more congratulations"
900 REM *****Another go?*****
910 LOCATE 6,22
920 PRINT"Do you want another game (Y/N)
?"
930 a%=INKEY$:IF a%="" THEN 930
940 a%=UPPER$(a%)
950 IF a%="Y" THEN RUN
960 IF a%="N" THEN CLS:LOCATE 18,12:PRIN
T"Goodbye!":END
970 GOTO 930
```

There are two particular features about the 'Another Game' routine that might interest you. The first is the command 'UPPER\$' on line 940. This converts the character input into an upper-case character. By way of an example, if the player was in lower-case mode, the 'another game' routine would receive an input of a lower-case 'y' or 'n'. This lower case value is then changed to upper case so that it becomes 'Y' or 'N'. The input testing lines only test for an upper case input. The second point of interest is that a Yes response to the 'Another game' condition results in the game being RUN. RUNning a program is the quickest and most efficient way of initialising all variables. The RUN command sets all numeric variables to zero and all strings to null strings.

At the moment we have the nucleus of the game: boring is it not? Looking back at the game specification you will see that we still have to include a baddie routine. This is a very important feature of the game. Having something to compete against (in this game something to avoid) prevents the player from leisurely strolling along with no incentive to pick the pounds up quickly. The difference between the baddie movement routine, in Program 4.5(i), and that developed in Part One of this chapter (Program 4.3(b)), is that the baddie moves diagonally around the screen instead of horizontally and vertically. The second important difference is that this new movement includes tests to see if the location the baddie is about to move into is occupied by the player's character or by a pound sign.

PROGRAM 4.5(i)

```

140 badx=INT(RND*30)+1:bady=INT(RND*24)+
1
145 IF sa(badx,bady)=1 THEN 140
3000 REM *****baddie movement*****
3010 m=INT(RND*4)+1
3020 IF m=1 THEN by=-1
3030 IF m=2 THEN by=+1
3040 IF m=3 THEN bx=-1
3050 IF m=4 THEN bx=+1
3060 badx2=badx:bady2=bady
3070 badx=badx+bx:bady=bady+by
3080 IF badx>30 THEN badx=1:badx2=30
3090 IF badx<1 THEN badx=30:badx2=1
3100 IF bady>24 THEN bady=1:bady2=24
3110 IF bady<1 THEN bady=24:bady2=1
3120 IF badx=manx AND bady=many THEN df=
1:RETURN:REM df= death flag
3130 IF sa(badx,bady)=1 THEN lo=lo-1:SOU
ND d1,100,50,12
3140 sa(badx,bady)=-1
3150 PEN 1:LOCATE badx2,bady2:PRINT CHR$(
225)
3160 PEN 3:LOCATE badx,bady:PRINT CHR$(2
24)
3170 RETURN

```

Line 140 gives the baddie a random start location and this is checked to make sure that a pound has not already been placed there (line 145). Line 3120 of Program 4.5(i) compares the coordinates of the enemy with that of the player. If they are the same then the player's character has been killed and a flag (DF – Death Flag) is set to one. Once the baddie movement routine has returned to the main program the death flag needs to be tested. If the death flag is set then the player is informed and then asked if another go is desired.

PROGRAM 4.5(j)

```
230 GOSUB 3000:REM baddie movement
240 IF df=1 THEN 700:REM you have lost
700 REM *****you have lost*****
710 LOCATE manx,many:PEN 3
720 PRINT CHR$(238)
730 LOCATE 4,6
740 PRINT"You have been killed"
750 GOTO 900:REM Another go?
```

If you run the program as it stands, you will see that the baddie movement routine works and it is possible to move the player's character in subjective simultaneity with the baddie. There are, however, four major problems with the program and these are:

- 1) When you move the yellow player character, it turns red.
- 2) The player character can walk through any obstruction of yellow unhappy faces.
- 3) Because the baddie is moving diagonally it cannot fill each location, only every other one.
- 4) If the player chooses to play another game then all the pound signs are displayed red.

Problems 1 and 4 are caused, indirectly, by the baddie's movement routine. After printing the happy face, the current pen colour is red and so each subsequent character displayed is red with the exception of the unhappy face because the pen colour is changed immediately before printing it. To remedy the problem, the player movement and screen display routines should set the pen colour before they display anything. The player's character and the pound signs will be printed using the colour in inkpot two, i.e. light blue. Program 4.5(k) shows you which lines need to be changed.

PROGRAM 4.5(k)

```
130 LOCATE manx,many:PEN 2:PRINT CHR$(249)

1040 LOCATE x,y:PEN 2:PRINT CHR$(163)

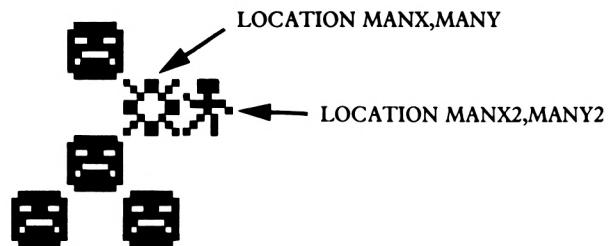
2150 LOCATE manx,many:PEN 2:PRINT CHR$(c)
```

As far as problems 1 and 4 are concerned, the game is cured. Problem 2 is just as simple to correct; the reason why the player's character can walk through yellow unhappy faces is that the player movement routine does not contain a test to see if the value of the next location is -1 (an unhappy face). Line 2135 of Program 4.5(1) contains the required test; if the result is true (the player has crashed into a yellow face) then the Death Flag is set and the program RETURNS.

PROGRAM 4.5(1)

```
2135 IF sa(manx,many)<0 THEN df=1:RETURN
```

Having added line 2135, run the program and attempt to get yourself killed by moving Albert into a yellow face. Figure 4.7 shows what happens:

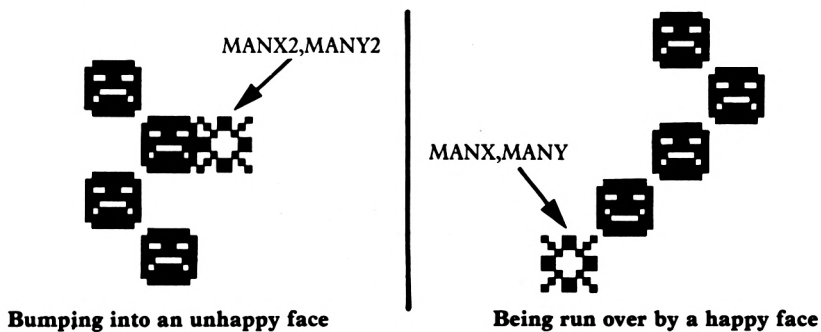


What Happens When Albert Collides

FIGURE 4.7

The 'explosion' character is printed at the location MANX,MANY, where the player's character was going to be displayed. The player's character, still on the screen, has the location value MANX2,MANY2. Normally this character would be removed by line 2140 of the player movement routine, but because the player has been killed the program never reaches that stage.

There are two ways in which the player can be killed; by being hit by the red happy face or by bumping into a yellow unhappy face. There now needs to be two different sets of coordinates for the printing of the explosion character (CHR\$(238)). Figure 4.8 shows the two types of death and the explosion character's coordinates for each.



Killing Coordinates

FIGURE 4.8

Lines 2135 and 3120 (those which set the death flag) can be edited to include the crash character coordinates (CX and CY). Line 710, the one that locates the crash character, also needs changing to include the variables CX and CY.

PROGRAM 4.5(m)

```

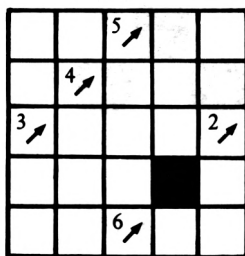
710 LOCATE cx,cy:PEN 3

2135 IF sa(manx,many)<0 THEN df=1:cx=man
x2:cy=many2:RETURN

3120 IF badx=manx AND bady=many THEN df=
1:cx=manx:cy=many:RETURN

```

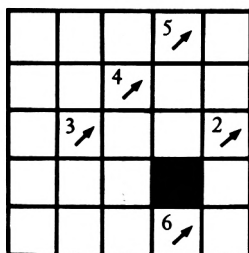
The last problem to correct is problem 3. Because the baddie is moving diagonally, it cannot fill each screen location, only every other one. Figure 4.9 demonstrates the problem. A diagonally moving object on the screen will eventually retrace its steps.



The Problem of Diagonal Movement

FIGURE 4.9

This can be solved by fixing the baddie's X coordinate when it moves from one side of the screen to the other. At the moment (as Figure 4.9 demonstrates) when the baddie reaches the edge of the screen (stage 2) it reappears at the extreme edge on the other side of the screen (stage 3) i.e. when BADX is greater than thirty it becomes one (and vice versa). If the X coordinate of the baddie is altered so that when it moves from one edge to another it doesn't go from say 30 to 1 but from 30 to 2. This point is illustrated in Figure 4.10.



Fixed Diagonal Movement

FIGURE 4.10

This value represents the length of time your computer has been switched on in 300th's of a second. To find out how long this is in a more user-friendly form, type in one of the following direct entry commands:

```
PRINT"I SWITCHED ON";TIME/300;"SECONDS AGO"  
PRINT"I SWITCHED ON";(TIME/300)/60;"MINUTES AGO"  
  
PRINT"I SWITCHED ON";((TIME/300)/60)/60;"HOURS Ago"
```

It is possible to time the length of anything by taking the value of TIME before the operation and again after the operation. The first TIME value is then subtracted from the second to give the time taken in 300ths of a second.

To time the length of the game it is necessary to first of all take the value of the variable TIME just before the player movement routine and then check the TIME constantly in the main game loop (lines 200 to 390).

TIME

TIME is a special variable that is controlled by the computer and is called a 'system variable'. The Amstrad's TIME variable is used (among other things) to time how long the computer has been switched on. To see the time value type in:

```
PRINT TIME
```

The value displayed depends upon how long your computer has been switched on, but could look something like this:

```
1592519
```

Only two lines need changing to incorporate this into the program; these are 3080 and 3090. When the baddie moves to the right edge of the screen (BADX>30) then BADX is set to equal two, and when the baddie reaches the left edge of the screen (BADX<1) BADX is set to twenty nine.

PROGRAM 4.5(n)

```
3080 IF badx>29 THEN badx=3:badx2=29  
3090 IF badx<2 THEN badx=28:badx2=2
```

Even with the addition of these two lines the baddie character eventually retraces its steps – it is impossible to prevent it from doing this – but the inclusion of lines 3080 and 3090 does make it possible for the baddie to fill every screen location with yellow unhappy faces. The program is now beginning to look more and more like a proper game. Referring back to the game outline there is still one more thing to add to the program.

The value of time is stored in the variable START on line 190, directly before entering the main game loop. If the value of TIME was taken at the start of the program then the initialise loops would be timed. Because we are timing the player, it would be harsh to add on the time taken by the computer in setting up the screen. Testing the time elapsed can be done in a subroutine from line 1500 onwards. The player is allowed three minutes (180 seconds) to pick up all the objects. If more than three minutes elapse then a flag is set (ET, End of Time).

PROGRAM 4.5(o)

```
190 start=TIME:REM start the timer
1500 REM *****check the time*****
1510 stp=TIME-start
1520 stp=INT(stp/300)
1530 IF stp>180 THEN et=1:ELSE et=0
1540 REM the player is allowed three minutes
1550 RETURN
```

This routine needs to be incorporated into the main game loop. Line 250 GOSUBs to the time-checking routine and line 260 tests the value of the End of Time flag. If the flag is set then the program GOTOs line 600 where a 'Run out of time' message is displayed. Line 640 directs the program to the 'Another go?' routine.

PROGRAM 4.5(p)

```
250 GOSUB 1500:REM check time
260 IF et=1 THEN 600:REM time out

600 REM *****run out of time*****
610 LOCATE 4,10
620 PRINT"You have run out of time"
630 FOR x=1 TO 1000:NEXT x
640 GOTO 900:REM do you want another go
```

The last thing to add to the time section is lines 360 and 365. These report the amount of time the player is taking to pick up all the objects.

PROGRAM 4.5(q)

```
360 LOCATE 34,10:PRINT"time"
365 LOCATE 34,12:PRINT stp
```

When Program 4.5(q) has been added, run the program. You are able to move the character all around the screen picking pound signs up. Take care while you are doing this because there is a baddie moving diagonally across the screen. If this bumps into Albert or if Albert bumps into one of the yellow unhappy faces then Albert will be killed. Be as quick as you can when picking up the pound signs because you only have three minutes in which to do so. You may notice whilst playing the game that it fulfills all of the original plans made for it. Now that a working version of the game exists, work can begin on improving it.

PART THREE

Improving The Game

Screen Display

There are three major ways in which the screen display can be improved: a greater variety of colour, a border around the game screen and the use of windows in the player's information section of the screen. At the moment the game uses the colours in inkpots zero to three. When the Amstrad is switched on the colours in those inkpots are blue, yellow, light blue and red respectively. If you load the game without first resetting the Amstrad then the actual colour values of inkpots zero to three could be any combination, which might have disastrous effects on the game display. By way of example, type in the following direct entry commands and RUN the program.

```
INK 0,17,26
INK 1,19
INK 2,16
INK 3,17
```

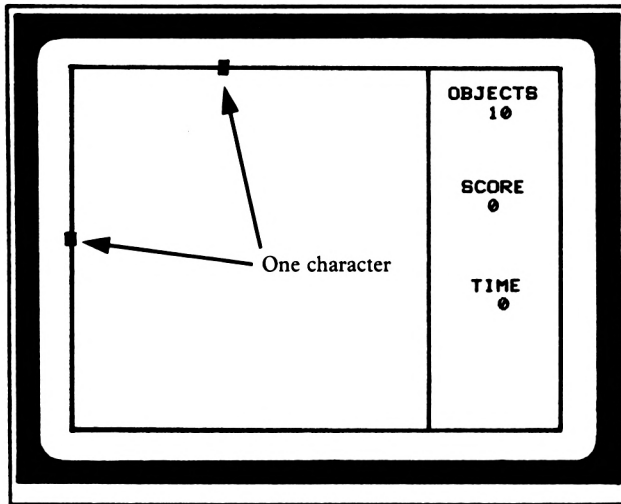
A suggested colour scheme is to have a light blue background, a white player and pound signs and the same yellow unhappy faces with a red smiling one. Not only is this colour scheme visually appealing but the colour differences are noticeable on a monochrome monitor. The ink setting routine is included as a subroutine starting at line 1200 and ending at line 1260 which RETURNS back to the main control program. Line 40 of Program 4.6 calls the colour setting routine before the display screen routine is called up, thus making sure that the characters are displayed on the screen in the correct colour.

PROGRAM 4.6

```
40 GOSUB 1200:REM set the colours

1200 REM *****set the colours*****
1210 INK 0,11:REM the screen is blue
1220 INK 1,24:REM old baddy is yellow
1230 INK 2,26:REM albert is white
1240 INK 3,6:REM baddy header is red
1250 MODE 1:BORDER 17
1260 RETURN
```

Another improvement to the screen display would be a drawn border separating the game and player-information areas of screen. The inclusion of a border will change the screen layout to that of Figure 4.11:



The Screen Border Display

FIGURE 4.11

The border is quite simple to include. You simply MOVE the graphics cursor to the border's bottom left-hand corner and DRAW four sides. The DRAW command on line 1320 sets the graphics colour to that in inkpot two. This means that the border will be white.

PROGRAM 4.6(a)

```

45 GOSUB 1300:REM draw border

1300 REM *****draw a border*****
1310 MOVE 10,10
1320 DRAW 10,390,2:REM bottom side
1330 DRAW 470,390:REM right side
1340 DRAW 470,10::REM top side
1350 DRAW 10,10:::REM left side
1360 RETURN

```

If you run the program, you will see the border. However, if you continue to watch you will see the baddie move all over the border removing whichever part it comes in contact with. Not only does the baddie move over the border but so does the player's character and it is possible for the pound notes to be displayed over it too. The inclusion of a graphics border has effectively moved the playing screen boundary in by one character. Consequently the player and baddie movement routines need to be amended and a line needs to be inserted into the screen display routine to make sure the pounds' X and Y coordinates are within the new range.

PROGRAM 4.6(b)

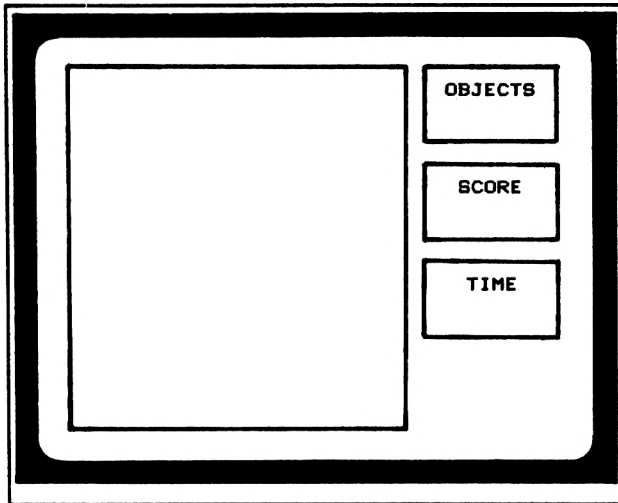
```
1032 IF x>29 OR x<2 OR y>24 OR y<2 THEN
1020

2090 IF manx>29 THEN manx=2:manx2=29
2100 IF manx<2 THEN manx=29:manx2=2
2110 IF many>24 THEN many=2:many2=24
2120 IF many<2 THEN many=24:many2=2

3080 IF badx>29 THEN badx=3:badx2=29
3090 IF badx<2 THEN badx=28:badx2=2
3100 IF bady>24 THEN bady=2:bady2=24
3110 IF bady<2 THEN bady=24:bady2=2
```

You should not have to type in any of the lines again, simply edit the already existing lines in the program. Once Program 4.6(b) has been included in your program, the program will run without affecting the white border.

The last piece of screen display that could do with smartening up is the player-information area of the screen. At the moment this section of the screen displays three pieces of information: objects left, score and time. These three pieces of information can be placed in three separate windows creating the following display.



The Player Information Windows

FIGURE 4.12

The window-setting routine is located at lines 4000 onwards and is called up by line 50 of Program 4.6(c). The three windows have different colours, the first and third windows have a red background with white writing and the second window has a white background with red writing. This produces a very nice visual effect.

PROGRAM 4.6(c)

```
50 GOSUB 4000:REM set up windows

4000 REM *****set the windows*****
4010 WINDOW#1,31,39,2,6
4020 PAPER#1,3:PEN#1,2:CLS#1
4030 LOCATE#1,2,2:PRINT#1,"objects"
4040 WINDOW#2,31,39,8,12
4050 PAPER#2,2:PEN#2,3:CLS#2
4060 LOCATE#2,3,2:PRINT#2,"score"
4070 WINDOW#3,31,39,14,18
4080 PAPER#3,3:PEN#3,2:CLS#3
4090 LOCATE#3,3,2:PRINT#3,"time"
4100 WINDOW#4,31,39,20,24
```

Lines 4000 to 4100 set up the windows, but it is currently lines 360 to 385 which actually display the player information. These lines need to be changed so that the values are displayed in the correct windows. To this end, type in the following DELETE command (taking care that you type it in correctly) and then add Program 4.6(d).

```
DELETE 360-385
```

PROGRAM 4.6(d)

```
360 LOCATE#1,3,4:PRINT#1,lo
370 LOCATE#2,3,4:PRINT#2,score
380 LOCATE#3,3,4:PRINT#3,stp
```

Sound

Now that the screen display has been tidied up we can proceed onto the next improvement, that of SOUND. At the moment the only sound used is when the player or the computer picks up an object. There are two other cases where sound can be used. When the player has lost (killed by a baddie or run out of time) and when the player has won. This requires two separate tunes, one for the death sequence (a sad tune) and one for the winning sequence (a happy tune).

Lines 60,70 and 80 set the envelopes that will be used to produce the two sounds. Envelopes one (ENT 1,ENV 1) are for the losing sound and envelope two (ENV 2) is the winning sound.

PROGRAM 4.6(e)

```
60 ENT 1,12,3,4,21,3,4,2,1,12
70 ENV 1,8,7,6,7,8,6,5,6,8
80 ENV 2,11.5,-2.5,11,11.5,-2.5,22,6,2.5,8
```

The different sound routines now have to be placed in the relevant subroutines; the loser's sound goes onto the 'You have lost' message section (lines 732–736) and the winner's sound at the 'congratulations' section (lines 870–878).

PROGRAM 4.6(f)

```
732 FOR x=10 TO 50 STEP 10
734 SOUND 7,x,10,0,1,1,0
736 NEXT x

870 DATA 71,63,119,71,63,239,71,63,119,2
39,95,89,0
872 FOR x=1 TO 13
874 READ tone
876 SOUND 7,tone,10,0,2,0,0
878 NEXT x
```

There is no sound produced when the player has run out of time. This has been left for you to include if you so wish. You can also change the sounds produced by the winning and losing sections if you want to – at the moment they are rather spartan.

Best Score

The last improvement to the game that will be covered here is the inclusion of a 'best score' feature. A best score routine compares the current player's score with previous players' scores. To include a best score routine, quite a large piece of program needs to be added to the already existing 'You have won' routine. The best score routine is too big to fit into the limited space between the 'well done' and 'another go?' routines so it will be added as a subroutine called up by line 880 of Program 4.6(g).

The routine at 5000 will calculate the player's score in a slightly different way. At the moment the player's score at the end of the game is the sum of all the random numbers scored for each object.

The routine at 5000 will have a second calculation which will multiply the amount of time left (180-STP) by two; this value will then be added to the player's score producing a total score value. This will give the player an extra incentive to pick up all the objects as quickly as possible.

PROGRAM 4.6(g)

```
880 GOSUB 5000:REM BEST SCORE

5000 REM *****BEST SCORE*****
5010 score=score+(180-stp)*2
```

Once the new score value has been calculated, it needs to be checked to see if it is a 'best score'. An array called BEST is used to store the top ten scores and a second array, NAME\$, is used to store the scorers' names. Because both of these arrays only have ten elements there is no need to dimension them.

Program 4.6(h) below checks each value in the BEST array to see if the current player's score is greater than any of them. If the current score is greater, a flag, (BF) is set. When BF is tested (line 5060), the program returns to the 'Another go' routine If it is not set and continues if it is.

PROGRAM 4.6(h)

```
5020 bf=0:REM Best Score Flag
5030 FOR x=1 TO 10
5040 IF score>best(x) THEN bf=1
5050 NEXT x
5060 IF bf=0 THEN RETURN
5070 LOCATE 14,9
5080 PRINT"NEW BEST SCORE"
```

A LINE INPUT statement can be used for inputs of player's names so that they can type in any of the keyboard characters, although line 5120 ensures that only the first ten characters are remembered.

PROGRAM 4.6(i)

```
5090 REM *****whats in a name*****
5100 LOCATE 2,11
5110 LINE INPUT"what is your name ";name
5120 name$=LEFT$(name$,10)
```

The player's name and score need to be inserted in the appropriate place in the BEST and NAME\$ arrays. Program 4.6(h) simply checked to see if the players score was a best one, not where it goes. If the score is the second best then the current second best becomes the third best, the third best becomes the fourth best, etc.. Program 4.6(j) uses a WHILE...WEND loop to sort through the array of best scores.

PROGRAM 4.6(j)

```
5130 FOR x=1 TO 10
5140 WHILE score>best(x)
5150 temp$=name$(x):REM temporary name
5160 temp =best(x):REM temporary score
5170 best(x)=score:name$(x)=name$
5180 score=temp:name$=temp$
5190 WEND
5200 NEXT x
```

Having rearranged the top ten scores and scorers, the program then has to display them.

PROGRAM 4.6(k)

```
5210 CLS
5220 LOCATE 14,4
5230 PRINT"Ten top scores"
5240 FOR x=1 TO 10
5250 LOCATE 10,8+x
5260 PRINT name$(x),best(x)
5270 NEXT x
5280 RETURN
```

Having displayed the best scores, this section of program returns to line 900 where the player is asked if another go is required. If you run the program and complete the screen you will see the familiar congratulatory messages and hear the winning tune. No matter how bad your score is it will be the top score (when the array is set up all the best values have a default value of zero) and you will be asked to type in your name. After doing so the top ten will be displayed. Only the name you typed in will be displayed because the other elements of NAME\$(X) all contain the default value of nothing (a null string). When asked if you would like another game, reply 'Y'es. As soon as you have chosen to play again, the test on line 950 is true and the program is RUN.

```
950 IF A$="Y" THEN RUN
```

When a RUN command is executed, the computer is instructed to forget the value of any variables. This, unfortunately, includes the best scores and the scorers' name. RUN was used in the first place because it provided the quickest way of initialising the screen array. There are other variables, aside from the screen array, that need initialising before the game can be played again. These are the death and win flags. Program 4.6(l) extends the 'Another go?' routine to include an initialisation section.

PROGRAM 4.6(l)

```
950 IF a$="Y" THEN 980
960 IF a$="N" THEN CLS:LOCATE 18,12:PRIN
T"Goodbye!":END
970 GOTO 930
980 CLS:LOCATE 8,12:PRINT"Please wait"
982 FOR y=1 TO 24:FOR x=1 TO 30
984 sa(x,y)=0
986 NEXT x,y
988 sf=0:REM score flag
990 et=0:REM End of Time flag
992 df=0:REM Death Flag
994 RESTORE
996 score=0
998 GOTO 30
```

It takes about twenty seconds for the whole of the screen array to be initialised. Line 980 informs the player that something is happening. Now the best score routine is incorporated into the program, but it is not fully operational. There are one or two slight additions that should be made.

The first addition would be to initialise the NAME\$ and BEST arrays. At the moment when the game is run there are no names in NAME\$() and no scores in BEST(). This situation is remedied by the routine at 6000. Each element of NAME\$(X) is given a different name (line 6020) and a different score (line 6030). The array initialisation routine is called up by line twenty five.

PROGRAM 4.6(m)

```
25 GOSUB 6000:REM initialise score array

6000 REM *****score array*****
6010 FOR x=1 TO 10
6020 name$(x)="albert#"+STR$(x)
6030 best(x)=1200-(100*x)
6040 NEXT x
6050 RETURN
```

Now your score will not automatically go into the top ten unless it is better than one of the scores already there.

The second improvement is to display the current best score and scorer (BEST(1),NAME\$(1)) in the player information section of the screen. With the current player-information screen layout there is room for another window at the bottom right of the screen. The display will be the same size as the other three windows and can have a white background with red ink. Because the value of the best score will not change whilst the game is in operation, this window only needs to be dealt with once, at the window setting routine.

PROGRAM 4.6(n)

```
4100 WINDOW#4,31,39,20,24
4110 PAPER#4,2:PEN#4,3:CLS#4
4120 LOCATE#4,1,2:PRINT#4,name$(1)
4130 LOCATE#4,3,4:PRINT#4,best(1)
4140 RETURN
```


At last the game can be considered complete. It is, however, not finished. There are many things that can be added to enhance the game's playability. If you wish to continue developing this program then please do, the chapter itself ends here with a few hints as to further development.

- Multi-baddies
- Different objects to pick up
- Player firing
- Baddie firing

Chapter 5

Data Handling

In this chapter, we shall look at what is probably the main use to which computers are put: data storage and manipulation. The ideas behind the READ...DATA statements will be enormously extended to allow the computer to handle a much larger amount of data, without it being necessary to alter the program every time the data is altered. Firstly, a program will be developed to store, as an example, names and ages on cassette or disk.

The program developed in this chapter might appear quite complicated, but you will be surprised to find that it's not as tough as it looks. For a start, the job is made easier by taking advantage of structured programming. The program is developed as a series of short modules, each module, except for a couple of complex bits, being relatively easy to follow on their own. In this way a quite complicated program can be built up relatively painlessly, as previously demonstrated with the Albert game.

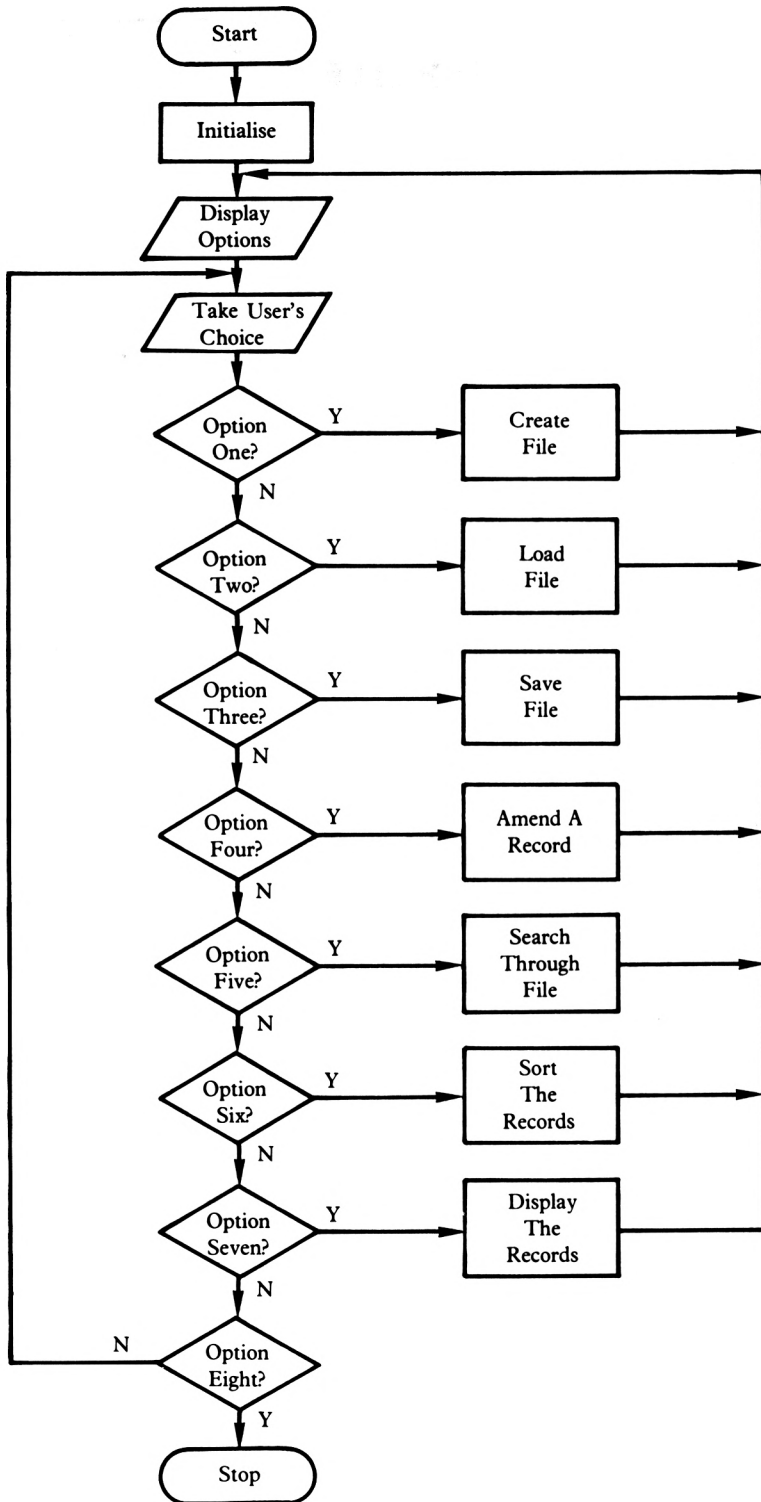
Part One

Sequential Cassette Filing

A sequential file is a file in which the data is stored in sequence – i.e. in list form. For example, suppose you have a cassette with various tunes recorded on it. These are stored 'sequentially'. To listen to any one particular tune, you must wind the tape past any previous tunes. Tapes are ideal for sequential storage of information, and the Database program will demonstrate this.

The first thing to do in designing the program is to decide exactly what is required of it, and draw up a general flow chart.

The program must allow the user to: Create a file, Load a file, Save a file, Add, Delete and Alter a record, Search the file for a particular record, Display the file and, perhaps, sort the records into alphabetical or numerical order.



Flowchart of the Database Program

FIGURE 5.1

The flowchart in Figure 5.1 shows the general structure of the program. In order to make the program 'user friendly', i.e. easy to use, it is to be 'menu driven'. A menu-driven program is one in which several options are presented on the screen by the program, and the user chooses one by pressing an appropriate key.

Part one of this chapter will deal with each of the items in the flowchart in the order shown. Before this is done, however, some terminology must be defined, and the structure of the data file itself must be looked at.

Fields, Records and Files

Take a look at the following address:

Ms Joanne Bloggs
29 Acacia Crescent,
Riverton,
London
W12 6DN

01-441-4130

This consists of several separate items of information:

- Title – Ms
- The first name, Joanne
- The surname, Bloggs
- The house number, 29
- The name of the road, Acacia Crescent
- The town, Riverton
- The postal district, London
- The postcode, W12 6DN
- The telephone number, 01-441-4130

If, in the program, we treat each of these as a separate item, then they are each known as a 'field'. The whole entry, i.e. the name, address and telephone number, is known as a 'record'. A set of records is known as a 'file'

Perhaps a better way to think of it is like this: imagine a filing cabinet, containing information about a company's customers, for instance. This is the 'file'. Inside the cabinet are lots of folders, one for each customer. These are the customer's 'records'. Each record consists of various sheets of paper, with information written on them. Each sheet, or group of sheets, possibly arranged under headings, is a 'field'. This is how the data file will be arranged in the program.

It would be possible to define the fields differently, for example, '29 Acacia Crescent' could be one field instead of two. It is purely a matter for the programmer or user to decide. The important point is that a field will be treated by the computer as one data item, and when you are entering data into the computer, it will expect one complete field at a time.

How will the file be stored in the computer's memory? The plan is to store each record as an element in a string array, where each element is divided up into a number of individual fields. The computer will then be able to look up, say, record number 6 simply by saying something like `PRINT DAT$(6,n)` where `DAT$()` is the file array and where `n` is the number of the particular field you wish displayed.

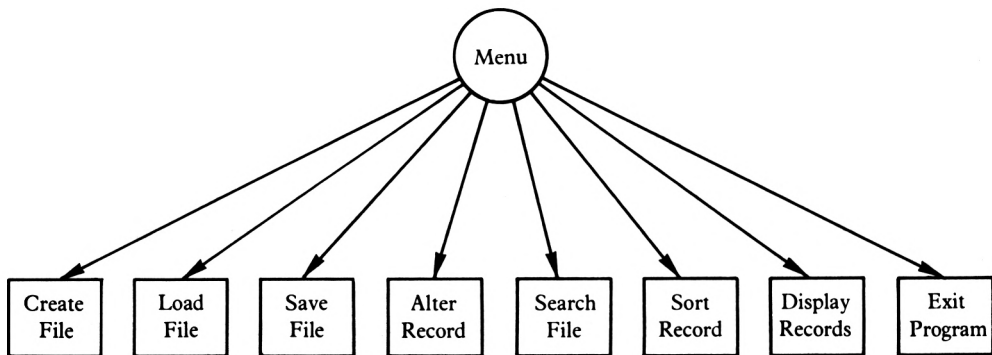
With a database program, one objective is to be able to create as many records as possible. The main limitation in this respect is the amount of memory the computer has available. When switched on, the Amstrad allocates 64k for your use. Some of this available memory will be taken up by the database program itself, thus restricting the number of records that can be created. The amount of memory occupied by the program can be slimmed down a little bit by the use of 'integer' variables – a new concept.

An integer variable is a special numeric variable that can only store whole numbers (i.e. integers) in the range -32768 to $+32767$. Integer variables use less memory than normal numeric variables because their size is restricted to five characters (six, if you want to include the $+/-$ sign). To tell the computer that you are going to use an integer variable, the name should end with a percent sign (%):

`AGE%=12`

Database: The program

The database program developed in this chapter consists of eight modules (a module being a section of programming that has been written to perform independently of the main program) and a menu (a section used to select each module). There will be other smaller routines used and these will be covered when necessary. Figure 5.2 shows the structure of the database program developed in this chapter and shows you what the eight modules are.



The Database Program Structure

FIGURE 5.2

Menu

The menu program needs to perform three tasks: initialise variables, display options available and direct the program to the selected module. This is perhaps the simplest part of the database program.

Create File

The Create File module offers the user the chance to design both the number and the names of the file's fields. The user is also asked if the field is string or numeric. i.e. will it contain strings or numbers?

Load File

This allows the user to load in previously stored data from the Amstrad's tape or disk unit.

Save File

In order to use the load module, you must first save a file onto tape or disk!

Amending Records

Having created your fields, this routine gives you the option of either adding records or changing records. Naturally, you cannot delete or change records until you have added them, so there is, therefore, a certain amount of checking involved in this routine.

Add Records

The Add Records section invites you to type in records under the field headings. The user is informed how much memory is available and how many records have already been created.

Delete Records

Once the Add section has been executed and records created, the user can delete them. This routine allows single or batch record Deletion.

Alter Records

If you made a mistake whilst typing in your record information, the Alter section offers you a chance to correct it. The user is asked for the record number, the file they want to change and the replacement data. If the field was set up as a numeric field, then the replacement data is checked to ensure that it contains numbers only.

Search File

This module allows the user to locate an item of data anywhere in a file. The program searches a specified field and displays all occurrences of the chosen string. The user is informed if no match is found.

Sort Records

Here the user is given the chance to sort their file into ascending (smallest first, largest last) or descending (largest first, smallest last) order. The sorting is conducted on a chosen field and the whole file arranged accordingly. For example, if there are three fields and the user chooses to sort field two in ascending order, the whole file will be rearranged so that the first record has the lowest field two value and the last record has the highest field two value.

Display Records

Having created or loaded a file, the user is now able to view each record individually. The record number is displayed and the user is informed as to whether the record has been deleted or not. The field number is then displayed along with the information contained for that particular record.

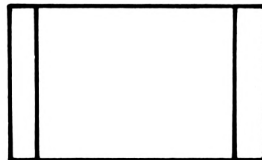
Exit Program

The last of the eight modules is that which ends the program. This module will clear all variables (set them to zero, strings will equal null strings), reset any windows and clear the screen before handing control of the Amstrad over to the user.

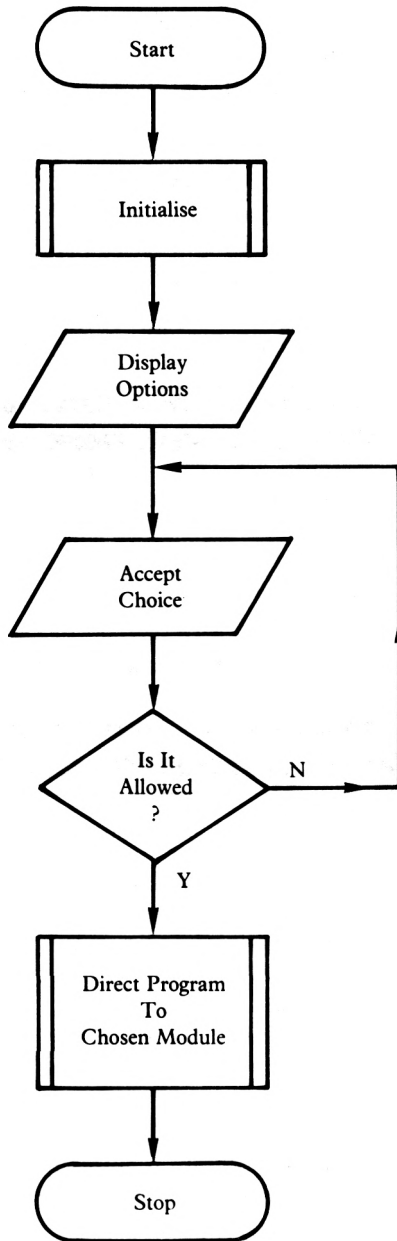
Each module of this program will be developed individually but in a slightly different order from that shown. It would be quite ridiculous to attempt to test a load routine when we haven't saved any data onto tape. Likewise, the save routine cannot be used until at least one record has been created.

Menu

As mentioned earlier, the Menu has to initialise variables, display the options that the user has and direct the program to the module of their choice. The flowchart in Figure 5.3 shows the general operation of the MENU program. One of the symbols may not be familiar to you. A process box with a vertical line on each side, i.e.



indicates that the process in the box is a routine on its own. In the database program the initialise section will be GOSUBbed to by the Menu routine.



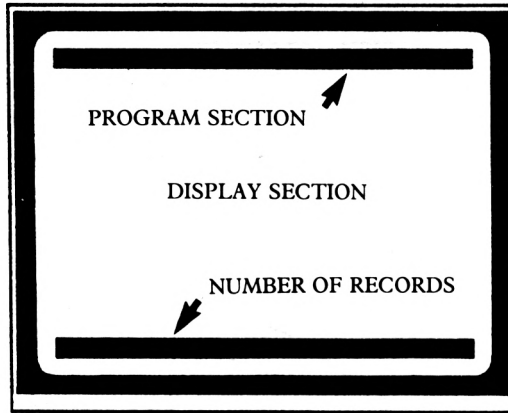
Flowchart of the Menu Routine

FIGURE 5.3

Initialisation

The first job of the initialise routine is to select the mode (MODE 1) and set up the ink values. The colours chosen are the Amstrad's default colours, therefore it is absolutely certain that both colour and monochrome monitors will be able to display the colour changes.

Next, three windows are set up, windows one, two and three. Window one occupies the top line of the screen and is used to display which program module the user is in. For example, when the user is creating a file, window one will display 'new file'. Window two fills lines 2 to 24. This is the main area of the screen where everything (well almost everything) will be displayed. The last window, window three is used to display the number of records that the screen has created. This gives an effective screen display of:



Screen Display

FIGURE 5.4

When displaying data on the screen, the cursor is not positioned at stream two but positioned at its absolute screen position (e.g. the top left-hand corner of window #2 would be located with 'LOCATE 1,2' instead of 'LOCATE#2,1,1'). The display area of the screen has been designed as a window to make it possible for the screen to be cleared (CLS#2) without removing the values in windows one and three.

The other half of the initialise subroutine deals with variables, dimensioning arrays and setting start values. The database program requires two arrays, these being FDN\$ and DAT\$. FDN\$ is used to store the Field Names (FDN\$(X,1)) and the field types (FDN\$(X,2)). DAT\$ is the array that stores the DATA for each field. Due to the Amstrad's memory size, the database is limited to ten fields and four hundred records, which should be more than adequate for the average user's requirements. The way data is stored in these arrays will be explained in the relevant sections. (FDN\$() in create file and DAT\$() in Alter file).

The two variables set are FC% and AC%. These are used to indicate whether there are any fields or records in memory respectively. At the start of the program these two variables are set to minus one (true), indicating that there are no fields (FC%) and no records (AC%) at that point in time.

Having performed all these operations, the initialise section returns to the Menu.

PROGRAM 5a

```
9000 REM *****initialisation*****
9010 MODE 1
9020 INK 0,1:INK 1,24:INK 2,11:INK 3,6
9030 BORDER 1:PEN 2
9040 WINDOW#1,1,40,1,1
9050 WINDOW#2,1,40,2,24
9060 WINDOW#3,1,40,25,25
9070 DIM dat$(400,10)
9080 DIM fdn$(10,2):REM Field Name$
9100 fc%=-1:REM No data as yet
9110 ac%=-1:REM No records as yet
9120 RETURN
```

After initialising, the user needs to be told what options there are. This is performed by lines 70 to 110. Each option title is stored as a piece of DATA on line 220 or 230. All this data is read via line 80 and displayed on the screen with the relevant module number. Line 150 asks the user for a choice. The input is checked (line 170) and (via the ON GOSUB at line 200) the program is directed to the chosen routine.

```
10 REM *****Data Base*****
20 REM *****By J.Irwin*****
30 GOSUB 9000:REM initialisation
40 REM *****Main Menu*****
50 PAPER#1,2:PEN#1,1:CLS#1:LOCATE#1,16,1
60 PRINT#1,"Main Menu":PEN 2
70 RESTORE:CLS#2:LOCATE 1,4
80 FOR x=1 TO 8:READ a$
90 PEN 1:PRINT x;
100 PEN 2:PRINT"...";a$:PRINT
110 NEXT x
120 PAPER#3,2:PEN#3,1:CLS#3:LOCATE#3,8,3
130 PRINT#3,"Total number of records ";1
r%
140 LOCATE 2,22:PEN 2
150 PRINT"Enter your choice please ";
160 c$=INKEY$:IF c$="" THEN 160
170 IF c$>"8" OR c$<"0" THEN 160
180 PEN 1:PRINT c$
190 c%=VAL(c$)
```

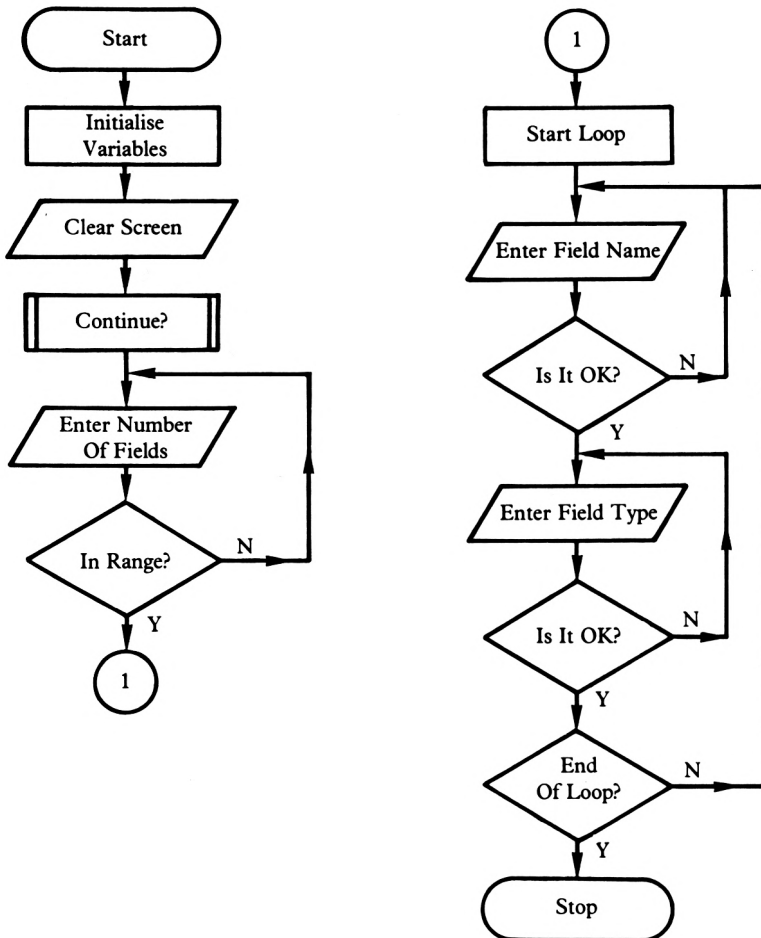
```

200 ON c% GOSUB 1000,2000,3000,4000,5000
,6000,7000,8000:REM Each routine
210 GOTO 50
220 DATA Create file, Load file from tap
e, Save records to tape,"Add, Delete or
Alter a record"
230 DATA Search file for a record, Sort
records in memory, Display records in me
mory, Exit program

```

Module 1:Create file

The Create File Module asks the user how many fields are wanted and the name and type (string or numeric) of each field. Figure 5.4 is the flowchart for this section.



Create File Flowchart

FIGURE 5.5

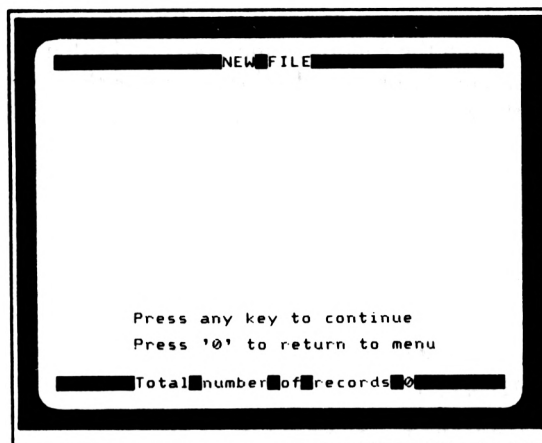
Remember that the flowchart is very general, and quite complex error trapping methods are used in the actual Create File Module. Before delving any further into this module, let us first look at the 'continue?' subroutine. Each module will ask users if they are sure they want to continue. This prevents the user from accidentally entering a routine, which could be disastrous if the delete routine was called up unexpectedly. The subroutine at line 9200 performs this task. The user is told to press '0' to return to the Menu or any other key to continue. The value of R\$ is tested in the module that calls up the input routine. In the case of module one, this line is 1070.

PROGRAM 5.1a

```
1000 REM *****Create File*****
1010 CLS#1:CLS#2
1020 lr%=0:REM Last Record
1030 nr%=0:REM Number of records
1040 LOCATE#1,16,1
1050 PRINT#1,"NEW FILE"
1060 PEN 2:GOSUB 9200:REM Continue?
1070 IF r$="0" THEN CLS#2:RETURN

9200 REM *****Input*****
9210 LOCATE 8,20
9220 PRINT"Press any key to continue"
9230 LOCATE 8,22
9240 PRINT"Press '0' to return to menu"
9250 IF INKEY$<>"" THEN 9250
9260 r$=INKEY$:IF r$="" THEN 9260
9270 RETURN
```

When this section of the program is run, the screen display will be:

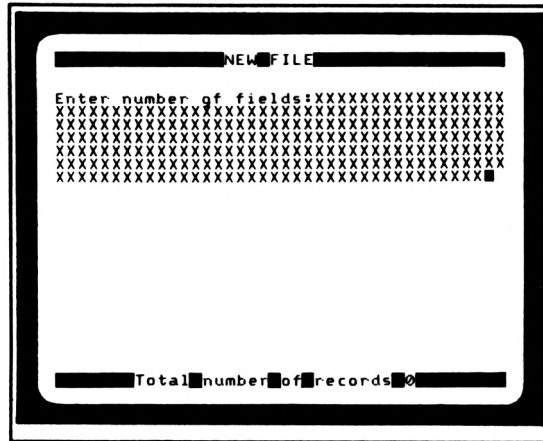


Continue Screen Display

FIGURE 5.6

If the user does want to continue, the screen is cleared and the user is asked to type in the number of fields required. The maximum number of fields that the user can create is 10, so the input needs to be tested to make sure that it is in the range of one to ten.

When INPUT is used, it is possible for the user to type in two hundred and fifty five characters. At this particular input, if the user was to type in 255 characters, the screen display would look like this:



Too Many Fields Requested

FIGURE 5.7

This messes up the screen display. It is pointless allowing the user to type in all these characters when the maximum number of characters allowed is two (i.e. 1 and 0). It would improve the screen display greatly if the user's input was restricted to a specific area of the screen.

In this program, a special form of window will be used to restrict the area of screen that an input can effect. Figure 5.8 shows the window that this particular input will use. It is still possible for the user to type in two hundred and fifty five characters: however, any wraparound will occur within the window. Line 1140 checks that the input contains the required number of characters (1 or 2), and a further test on the input checks to see if the number typed in is in the required range (1 to 10 inclusive).

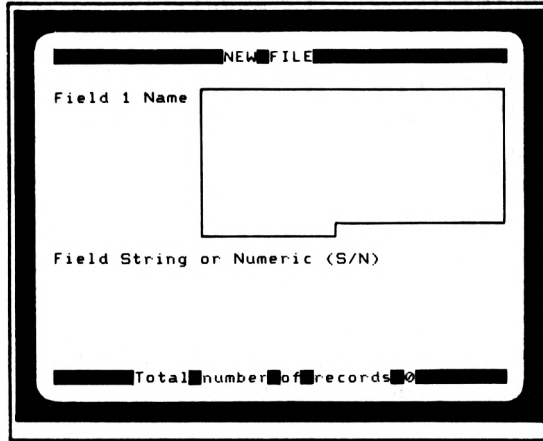
PROGRAM 5.1b

```

1080 CLS#2:LOCATE 1,4
1090 PRINT"Enter number of fields:"
1100 LOCATE 15,5:PRINT"Max.(10)"
1110 WINDOW#7,24,40,4,4:CLS#7
1120 PEN#7,3
1130 INPUT#7,"", c$:PEN#7, 2
1140 IF LEN(c#)>2 THEN CLS#7:GOTO 1110
1150 f%=VAL(c#):IF f%<1 OR f%>10 THEN 11
10

```

Next comes the most important part of this particular module, that which asks the user for the field name and field type. Line 1170 sets up a loop, from one to the number of fields that the user has specified. Within the loop, the program asks for the field name and field type (String or Numeric). When entering the field name, another window is set up to control wraparound. Figure 5.8 shows the screen display at this stage of the module. The difference between this window and that used for the number of fields is that this window allows you to see all the characters that you type in. This is because the field name is allowed to be 255 characters in length, whereas the number of fields can only be 2 characters.



Field Name Display

FIGURE 5.8

INKEY\$ is used when selecting the field type because the input required is only one character long. This single character input is checked to see if it is an 'S' or an 'N' before the program encounters a NEXT, and the whole process is repeated for the next field.

PROGRAM 5.1c

```

1170 FOR x%=1 TO f%
1180 CLS#2:LOCATE 1,4
1190 PRINT"Field";x%;"Name"
1200 LOCATE 1,15:PEN 2
1210 PRINT"Field String or Numeric (S/N)"
1220 IF x%=10 THEN WINDOW#7,15,40,4,13:CLS#7
LS#7: ELSE WINDOW#7,14,40,4,13:CLS#7
1230 PEN#7,3
1240 LINE INPUT#7, fdn$(x%,1)
1250 IF fdn$(x%,1)="" THEN 1230
1270 LOCATE 31,15:PRINT CHR$(143)
1280 x$=INKEY$:IF x$="" THEN 1280
1290 x$=UPPER$(x$)
1300 IF x$<>"S" AND x$<>"N" THEN 1270
1310 LOCATE 31,15:PRINT x$
1330 fdn$(x%,2)=x$
1340 PEN 2:NEXT x%

```

The last part of this module is that which tells the user that the Create File Module has ended and returns the program back to the Main Menu. When it has finished, the variable FC% is set to equal '0', which indicates the module has been called up (FC% = Fields Created).

PROGRAM 5.1d

```
1360 LOCATE 4,20:PEN 1
1370 PRINT"The Create module is now finished"
1380 LOCATE 2,22
1390 PRINT"The ADD section is used to create data"
1400 FOR x=1 TO 2000:NEXT x
1405 fc%=0
1410 RETURN
```

To test this module you simply type RUN and select option one at the Menu. The computer will then ask you to press any key to continue or '0' to return to the Menu. Press the space bar to continue. The computer will then ask you to:

```
Enter number of fields
MAX. (10)
```

To demonstrate this module we will create two fields, so type '2' and press ENTER. Now the program asks for the field name and type. Type in the example shown below: the darker type is what you type in:

Field 1 Name **Names**

Field String or Numeric (S/N) **S**

Field 2 Name **Age**

Field String or Numeric (S/N) **N**

Having typed in this data, the computer informs you that the Create File Module is finished:

```
The Create module is now finished
```

```
The ADD section is used to create data
```

Now that the fields have been created, it is time to create some records. This is done in module 4.

MODULE 4: Add, Delete or Alter a record

A submenu is used at the start of this module telling the user what options are available, and an input request is displayed. This submenu works in the same way as the main menu, the various options stored as data on line 4665 and read by the program at line 4030.

PROGRAM 5.4a

```
4000 REM *****Add,Delete,Alter*****
4005 CLS#1:CLS#2
4010 LOCATE#1,15,1
4015 PRINT#1,"Amend file":PEN 2
4020 RESTORE 4665
4025 LOCATE 1,3
4030 FOR x%=1 TO 4:READ a$
4035 PEN 1:PRINT
4040 IF x%=4 THEN PRINT 0;:ELSE PRINT x%
;
4045 PEN 2:PRINT"...";a$
4050 NEXT x%:PRINT
4055 LOCATE 2,14:PRINT"Enter choice ";
4060 PEN 3:PRINT CHR$(143):PEN 2
4070 a$=INKEY$:IF a$<>"" THEN 4070
4075 a$=INKEY$:IF a$="" THEN 4075
4080 IF a$<"0" OR a$>"3" THEN 4075
4085 LOCATE 15,14:PRINT a$:a=VAL(a$):PEN
2

4665 DATA Add records, Delete records, A
lter records, Return to menu
```

Line 4080 checks to see if the input is in the allowable range (0-3). If it is, then further tests need to be performed. If the user has selected option one (Add Records), a test is made on the variable FC%. If FC% is true (FC%=-1), then no fields have been created and so the user is not allowed to add any records. Line 4087 performs this test and if the test is true the routine at line 9400 informs the user that no fields have been created. If the user has selected to Delete (option 2) or Amend (option 3) a record, the AC% flag is tested to see if there are any records to delete or amend. If AC% equals -1, there are no records. The routine at line 9300 informs the user of this.

PROGRAM 5.4b

```
4087 IF fc% AND a=1 THEN GOSUB 9400:GOTO
4055
```

```

9400 REM *****No Fields*****
9410 LOCATE 6,18:PEN 1
9420 PRINT"You have not created any Fields";
9430 FOR x=1 TO 2000:NEXT x
9440 PRINT CHR$(17):PEN 2
9450 RETURN

4090 IF ac% AND (a=2 OR a=3) THEN GOSUB

9300 REM *****No Records*****
9310 LOCATE 6,18:PEN 1
9320 PRINT"You have not created any records";
9330 FOR x=1 TO 2000:NEXT x
9340 PRINT CHR$(17);:PEN 2
9350 RETURN

```

If the input passes these tests, it is checked to see if it is '0'. If it is, the program returns to the main menu. This is the final test and if the input is allowable, the program will reach line 4095. For the moment, we only need to concentrate on the Add Record routine so the ON...GOSUB command is not complete at this stage.

PROGRAM 5.4c

```

4093 IF a$="0" THEN RETURN
4095 ON a GOSUB 4105,4295,4470
4100 GOTO 4000

```

Add Records

The start of the Add Records routine is very similar to the start of the Create File Module. The screen is cleared and the user is asked to press any key to continue. All of the modules and routines (Add, Delete and Amend) will begin in this way.

PROGRAM 5.4d

```

4105 REM *****Add record*****
4110 CLS#1:CLS#2
4115 LOCATE#1,14,1
4120 PRINT#1,"Add records":PEN 2
4125 GOSUB 9200:REM Continue?
4130 IF r$="0" THEN CLS#2:RETURN
4135 CLS#2

```

Once the user has decided to continue, the program needs to check if there is any room left in memory to store new records. A new command is used here, the **FRE(0)** command. FRE(0) reports back to the user how much memory the Amstrad has available for use. To illustrate its use, type in the following direct entry commands:

```
PRINT FRE(0)
```

The number your computer displays should be something like:

41057

This number represents the number of bytes available for use. One thousand and twenty four bytes equals 1K, so you have about 40K of memory available. When all of the database program has been typed in, there will be less memory available. The initialise routine restricted the number of records to four hundred, by dimensioning the array DAT\$ to 400. This should provide more than enough room for your records. To prevent the user from running out of memory the add record routine contains two tests. The first test (line 4140) checks the amount of memory left if the number of records is less than 400:

PROGRAM 5.4e

```
4140 IF nr%<>400 AND FRE(0)<=1100 THEN L
OCATE 12,10:PRINT"Limited memory left";
FOR x=1 TO 2000:NEXT x:PRINT CHR$(17):RE
TURN
```

If the number of bytes free is less than 1100, the user is informed that there is 'limited memory left'. CHR\$(17) is one of the Amstrad's special character values (see Appendix five). Printing CHR\$(17) removes everything to the left of the cursor position on one line of the screen. To demonstrate this, type in the following direct entry commands:

```
PRINT"AMSTRAD";CHR$(17)
```

When you press ENTER, you will see 'Amstrad' flash very briefly onto the screen. The computer first prints 'Amstrad' and then prints CHR\$(17) which removes 'Amstrad' from the screen. This method is used to remove the 'Limited memory left' message from the screen.

The second test checks if the number of records equals its limit (400) or if the amount of memory is less than or equal to 1000. If any of these conditions is true, there is no memory available to store another record.

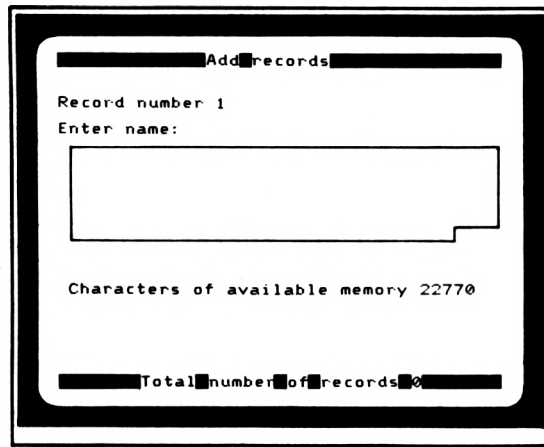
PROGRAM 5.4f

```
.4145 IF nr%=400 OR FRE(0)<=1000 THEN LOC
ATE 14,10:PRINT"No memory left";:FOR x=1
TO 2000:NEXT x:PRINT CHR$(17):RETURN
```

This line of the program also uses character 17 to remove the message displayed. Having decided that you have used up all available memory and you have not used up all the allowable records, the program continues.

Variables NR% (Number of records) and LR% (Last of records), which were set to zero in the Create File Module, are incremented by one. At this point, NR% and LR% have the same value but a situation will arise where the number of records and the value of the last record differ. This will be dealt with shortly.

Lines 4155 to 4245 are the real heart of the Add Record routine. Line 4155 sets a loop up from one to the number of fields (F%). In this loop the user is told the record number, how much memory is available, and asked for data for the 'X%' field. Once again a window is used to control the format of the user's input. Figure 5.8 shows a sample screen display for this stage of the Add Record routine (the bordered area shows the window used for input).



Add Record Display

FIGURE 5.9

PROGRAM 5.4g

```

4150 nr%=nr%+1:lr%=lr%+1
4155 FOR x%=1 TO f%
4160 CLS#2:LOCATE 1,4
4165 PRINT"Record number";
4170 PEN 1:PRINT lr%:PEN 2
4175 LOCATE 2,18:PEN 1
4180 PRINT"Characters of available memor
y";FRE(0)-1000
4185 LOCATE 1,6:PEN 2
4190 PRINT"Enter ";LEFT$(fdn$(x%,1),32);
";"
4195 WINDOW#7,2,38,8,15:CLS#7:PEN#7,3
4200 LINE INPUT#7,dat$(lr%,x%)

```

Having input your data, it needs to be checked. If the field type was specified as numeric (via module one) and the data you typed in was a string, then something ain't quite right and the user needs to be told so. If, however, the field type was specified as a string, there is no need to check the contents of the input.

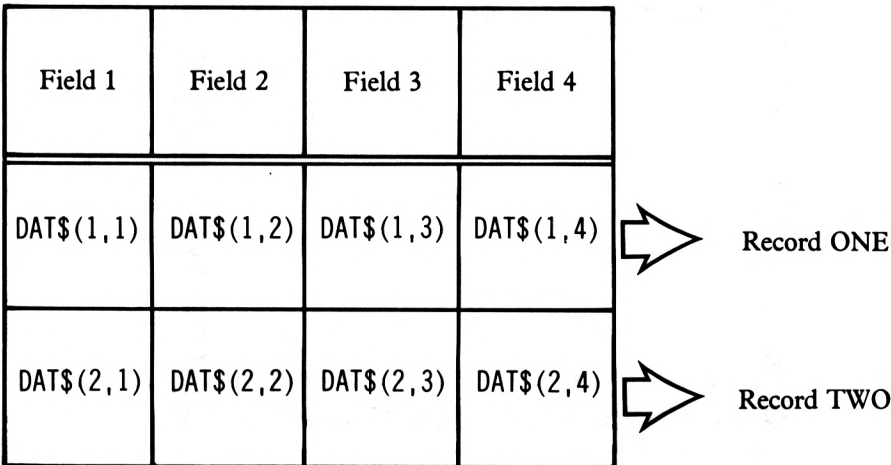
PROGRAM 5.4h

```

4210 IF fdn$(x%,2)="S" THEN 4245
4215 f1=0
4220 FOR l=1 TO LEN(dat$(lr%,x%))
4225 l$=MID$(dat$(lr%,x%),l,1)
4230 IF INSTR("1234567890+-.",l$)=0 THEN
    f1=1
4235 NEXT l:
4240 IF f1=1 THEN LOCATE 9,16:PEN 1:PRIN
T"Numerical data only please";:PEN 2:FOR
    x=1 TO 2000:NEXT x:PRINT CHR$(17):GOTO
4195
4245 NEXT x%

```

When the loop from lines 4185 to 4245 is finished, you have created your first record. The record's data is stored in the two dimensional array DAT\$(). If you had created four fields in the Create File Module, the information for record one would be stored in DAT\$(1,1), DAT\$(1,2) DAT\$(1,3) and DAT\$(1,4). Figure 5.10 attempts to demonstrate this:



Storage Of Data In Fields

FIGURE 5.10

Having created a record, the 'Total number of records' display at the bottom of the screen needs to be updated (lines 4255 – 4260). The computer then displays the 'continue' message and if the user wishes to continue, the program loops back to line 4135 where the screen is cleared and the whole add record process begins again. If the user does not want to continue, the computer informs the user that the Add routine has finished. Before returning back to the Submenu at line 4000, the AC% flag is set, indicating that the Add routine has been used and there are now records in memory.

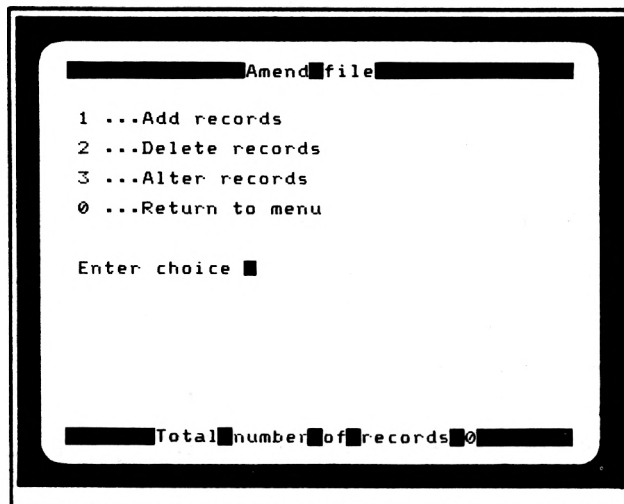
PROGRAM 5.4i

```

4255 CLS#3:LOCATE#3,8,3:PEN#3,1
4260 PRINT#3,"Total number of records";1
r%
4265 GOSUB 9200:REM Continue?
4270 IF r#<>"0" THEN 4135
4275 LOCATE 1,18:PRINT SPACE$(40)
4280 LOCATE 8,18:PEN 1
4285 PRINT"Add routine is now finished"
4287 ac%=0
4290 FOR x=1 TO 2000:NEXT x:RETURN

```

That concludes the Add Record routine, now it needs to be tested. Run the program and select option four at the menu. The computer will then display the following:



Calling The Add Record Routine

FIGURE 5.11

If you select option 1 (Add Records) the following message will be displayed:

You Have not created any fields

Because we have not executed module one (Create File), the flag FC% equals -1 and the condition on line 4087 is true. So far so good. When the message is removed, press 'Ø' which will return you to the Main Menu. When there, select option one and repeat the input on page twelve. As soon as you have created your two fields the program returns to the Main Menu. Select option 4 again and you will see the submenu. If you select option two or three you will see the following message:

You have not created any records

The AC% flag can only be set when the Add Record routine is called up, so the test on line 4090 is true and the message is printed. When the message is removed, select option 1. When asked to press any key, reply by pressing the space bar. The computer tells you that the record number is 1 and asks you to 'Enter names'. Work your way through these sample inputs, the darker type is what the user should type in:

Record number 1

Enter names: **Albert** <ENTER>

Enter Ages: **12** <ENTER>

Press the **SPACE BAR**

Record number 2

Enter names: **Julia** <ENTER>

Enter ages: **Twenty One** <ENTER>

After typing in 'Twenty One', the computer announces:

Numerical data only please

because the 'ages' field was specified as numeric. Now complete the record....

Record number 2

Enter ages: **21**

Select '0' to return to the submenu. The computer will respond with 'Add routine is now finished'. The message at the bottom of the screen should now read:

Total number of records 2

because we have created two records. Now that there are records, the AC% flag is set making it possible to select either the Delete or Alter options. If you press '2' or '3' an error will occur because these sections have not been implemented yet. However, notice that the 'You have not created any records' message did not appear. The Add Record routine has proved itself satisfactorily. Time to move on.

Module four is not finished; only the Add Record routine is. However, for the moment we will leave this module and work on saving and loading files from the tape unit (modules 3 and 2 respectively). By developing these modules now, it will only be necessary to create fields and records once. When a new module needs testing, we can simply load in a previously saved file. We will return to module four shortly.

Module 3: Save Records to Tape

Up to now, we have not covered Saving files. The only use of the cassette unit has been to save and load programs. However, before we rush into the save and load modules, a brief explanation of how to save data onto the cassette drive is required.

Save the database program before continuing.

OPENIN, OPENOUT, CLOSEIN, and CLOSEOUT

We have already seen that the Amstrad has seven streams for use with windows. There are two other streams, 8 and 9, which are for use with the printer and cassette unit. Appendix three demonstrates stream 8, for anyone interested.

The OPEN command tells the Amstrad that you wish to use the cassette unit to save or load data. You have to specify whether you want to save (out) data or load (in) data from the cassette unit and the name under which the data is stored (or to be stored), i.e. the file name.

To prepare the Amstrad for saving data, use 'OPENOUT"<filename>'"

To prepare the Amstrad to load in previously saved data, use 'OPENIN"<filename>'"

OPENOUT is used to send data OUT to the cassette unit and OPENIN reads data IN from the cassette unit.

The companion commands to OPENIN and OPENOUT are CLOSEIN and CLOSEOUT. CLOSEIN informs the Amstrad that no more data is to be loaded and CLOSEOUT says that no more data is to be saved. These commands are very simple so there should be no problem in understanding them.

One important point is that you don't need to specify the filename when closing a file. Only one file can be open at anytime so the Amstrad knows which file to close.

PRINT#9

When saving data onto tape, you simply print it to stream 9, the cassette stream. For example:

```
OPENOUT"ALBERT"  
PRINT#9,"ALBERT"  
CLOSEOUT
```

This sequence of commands will inform the Amstrad that the user wishes to send some data to the cassette unit under the file name of 'ALBERT'. Once the file has been opened, the data is passed to stream 9, which writes the data (in this example, the string 'ALBERT') onto the cassette in the cassette unit. When finished the output file is CLOSED. OPENOUT, PRINT#9 and CLOSEOUT will not do anything as direct entry commands; they have to be used within a program.

INPUT#9

To read data from a tape, you must first use OPENIN with the same filename that you used in the OPENOUT command and then INPUT to stream 9, i.e.

```
INPUT#9,A$
```

The Amstrad interprets this as 'get the first item of data stored under the filename of "ALBERT" and store it in the string variable A\$'. You must always remember to close the file when you have finished with it.

Now that you know how to save and load data to cassettes, we can develop a simple demonstration program. Program 5.d write a name to tape and then reads it back. A very simple use of the cassette commands:

PROGRAM 5.d

```
10 OPENOUT"name"  
20 PRINT#9,"albert"  
30 CLOSEOUT  
40 PRINT"Re-wind tape then press space b  
ar"  
50 IF INKEY$<>" " THEN 50  
60 OPENIN"name"  
70 INPUT#9,a$:PRINT a$  
80 CLOSEIN  
90 STOP
```

When this program is run, you will see the following:

```
RUN
Press REC and PLAY then any key:
Saving NAME block 1
RE-WIND TAPE THEN PRESS SPACE BAR
Press PLAY then any key:
Loading NAME block 1
ALBERT
Break in 90
Ready
```

If you received this screen display, you have successfully saved and loaded an item of data using the cassette unit. Those of you who did not get the above display, but instead somewhere along the line got a 'read error' would be advised to try again with a different cassette.

PRINT#9 can be used to output several variables at once. To do this, each variable has to be separated by CHR\$(44) (not a comma but the actual string 'CHR\$(44)'). By way of example, add Program 5.d2 to Program 5.d

PROGRAM 5.d2

```
20 PRINT#9,"albert"CHR$(44)"and the"
75 INPUT#9,b$:PRINT b$
```

When using PRINT#9, the Amstrad looks at 'CHR\$(44)' and interprets it not as a comma but as an end of variable marker. So line twenty reads Save 'ALBERT' as one piece of data and then Save 'AND THE' as another. You may ask yourself what will happen if you replace 'CHR\$(44)' with a comma (.). Try it and see.

PROGRAM 5.d2

```
20 PRINT#9,"albert","and the"
```

When you have changed line twenty, run the program. You will see the following display:

```
RUN
Press REC and PLAY then any key:
Saving NAME block 1
RE-WIND TAPE THEN PRESS SPACE BAR
Press PLAY then any key:
Loading NAME block 1
ALBERT AND THE
EOF met in 75
Ready
```

The Amstrad has read line twenty as:

PRINT TO TAPE"ALBERT AND THE"

EOF is an acronym for End Of File. Line seventy five attempted to read in a second piece of data when only one was saved. The Amstrad informs us that we have reached the end of the file at line 75. The program stops at this line and the file is automatically closed.

Having seen how to save and load data to and from tape, we can now add these routines to our database program.

The Save Module begins in the same way as all the other modules, by clearing the screen and checking if the user wishes to continue. Line 3035 makes sure that the user has created records that can be saved. If records have not been created, the routine at line 9300 tells the user this and the program returns to the Main Menu. Assuming that records have been created, the program can then continue to save the data.

PROGRAM 3.5a

```
3000 REM *****Save file*****
3010 CLS#1:CLS#2
3020 LOCATE#1,16,1
3030 PRINT#1,"Saving files":PEN 2
3035 IF ac% THEN GOSUB 9300:RETURN
3040 GOSUB 9200:REM Continue?
3050 IF r$="0" THEN CLS#2:RETURN
```

The Save Module saves everything that the user has created. The first two items saved are the number of fields (F%) and the number of records (NR%).

PROGRAM 5.3b

```
3060 CLS#2:LOCATE 7,9
3070 PRINT"Position tape for saving"
3090 OPENOUT"FILE"
3100 PRINT#9,f%:REM number of Fields
3110 PRINT#9,nr%:REM Number of Records
```

Next there is a short loop (from 1 to the number of fields). This is used to save all the field names and field types (lines 3120-3160).

PROGRAM 5.3b1

```
3120 FOR x%=1 TO f%
3130 PRINT#9,fdn$(x%,1):REM field name
3150 PRINT#9,fdn$(x%,2):REM Field type
3160 NEXT x%
```

Lastly, all the records are output to the cassette unit in two loops. The first loop (line 3170) runs from 1 to LR% (last record) and the loop on line 3190 sets the Amstrad up to output the data stored under each field name, for each record.

PRINT#9,dat\$(x%,z%)

record number field number

Lines 3250–3280 politely tell the user that the save module is finished and sees them safely back to the main menu.

PROGRAM 5.3c

```
3170 FOR x%=1 TO lr%
3180 IF dat$(x%,0)="d" THEN 3220
3190 FOR z%=1 TO f%
3200 PRINT#9,dat$(x%,z%):REM data
3210 NEXT z%
3220 NEXT x%
3230 CLOSEOUT
3250 LOCATE 4,20:PEN 1
3260 PRINT"The Save routine is now finis
hed"
3270 FOR x=1 TO 2000:NEXT x
3280 RETURN
```

To test the save module, we first need to create records. Run the database program and create the following records:

Three fields

Field 1 : Surname, string
Field 2 : First name, string
Field 3 : Age, numeric

Record	Surname	First Name	Age
1	And The	Albert	12
2	Rockerfella	Cinderella	18
3	Beauty	Sleeping	237
4	Beanstalk	Jack	18
5	Young	Arthur	93

When you have typed in the fifth record, return to the Main Menu and select option 3. The program will save your data and return back to the Main Menu. If your program did not follow the above steps, check the module again. Keep the data just saved; it will be used as a test file several times later on in this chapter.

Module 2:Load file from tape

The only way you can find out if the file has been saved properly is to load it back via the load module. The load module is an almost exact copy of the save module, except INPUT# replaces PRINT#. The data is read back in exactly the same order as it is written and it is also read into the same program variables. Lines 2240 and 2250 of Program 5.2a, set the FC% and AC% flags because loading in a file means loading in fields (FC%=0) and records (AC%=0).

PROGRAM 5.2a

```

2000 REM *****Load file*****
2010 CLS#1:CLS#2
2020 LOCATE#1,16,1
2030 PRINT#1,"Loading files":PEN 2
2040 GOSUB 9200:REM Continue?
2050 IF r$="0" THEN CLS#2:RETURN
2060 CLS#2:LOCATE 6,9
2070 PRINT"Position tape for loading"
2090 OPENIN"FILE"
2100 INPUT#9,f%:REM number of Fields
2110 INPUT#9,nr%:REM number of Records
2120 FOR x%=1 TO f%
2130 INPUT#9,fdn$(x%,1):REM Field name
2150 INPUT#9,fdn$(x%,2):REM File type
2160 NEXT x%
2170 FOR x%=1 TO nr%
2180 FOR z%=1 TO f%
2190 INPUT#9,dat$(x%,z%):REM Data
2200 NEXT z%,x%

```

```

2210 CLOSEIN
2230 lr%=nr%:REM Last Record
2240 fc%=0:REM There are fields now
2250 ac%=0:REM You can add fields now
2260 LOCATE 4,20:PEN 1
2270 PRINT"The Load routine is now finis
hed"
2280 FOR x=1 TO 3000:NEXT x
2290 RETURN

```

To test the load module you simply run the program and select option 2. When the load module is finished, the program will return to the main menu. At the bottom of the screen you will see the following message:

Total number of records 5

This tells us that we have successfully loaded the file's records that were saved earlier. However, the only proof we have that our records have been safely loaded is the number of records display at the bottom of the screen. It would be handy if we could see the actual records.

Module 7: Display records in memory

The display records module allows the user the option of looking at records that have already been created or loaded from tape. It is up to the user which records to display. Via the inputs on lines 7110 and 7210, the user selects the first and last record to be displayed. The start display record number (FIRST%) is checked to make sure it is not less than 1 (the first record) or greater than CR% (the last record). The stop display record number (LAST%) is checked to make sure that it is not less than FIRST% (you cannot display backwards) and not greater than LR% (you cannot display records you have not created). If the values of FIRST% and LAST% are the same, only one record is displayed.

PROGRAM 5.7a

```

7000 REM *****Display Record*****
7010 CLS#1:CLS#2
7020 LOCATE#1,13,1
7030 PRINT#1,"Record display":PEN 2
7035 IF ac% THEN GOSUB 9300:RETURN
7040 GOSUB 9200:REM Continue?
7050 IF r$="0" THEN CLS#2:RETURN
7060 CLS#2:LOCATE 1,4
7070 PRINT"Number of first Record"
7080 LOCATE 8,5
7090 PRINT"to be displayed:"
7100 WINDOW#7,24,39,5,5:CLS#7

```

```

7110 PEN#7,3:INPUT#7,"",c$
7130 c$=LEFT$(c$,4)
7140 first%=VAL(c$):REM FIRST Record to
be displayed
7150 IF first%<1 OR first%>1r% THEN CLS#
7:GOTO 7110
7160 LOCATE 1,7
7170 PRINT"Number of last Record"
7180 LOCATE 8,8
7190 PRINT"to be displayed:"
7200 WINDOW#7,24,39,8,8:CLS#7
7210 PEN#7,3:INPUT#7,"",c$
7230 c$=LEFT$(c$,4)
7240 last%=VAL(c$):REM LAST record to be
displayed
7250 IF last%<first% OR last%>1r% THEN C
LS#7:GOTO 7210

```

Both the first record and last record inputs use a formatting window – WINDOW#7, see Figure 5.12.

Display Record Input Windows

FIGURE 5.12

The FIRST% and LAST% values are used in a loop which controls the records to be displayed. The actual display routine is a subroutine from line 7500 to 7650 and is called up by line 7280 of Program 5.7b. The display subroutine displays the record number, field number and data stored for that field. Due to the restricted screen space, only the first 34 characters in each field are displayed. This should be more than adequate to give the user some idea of what data is stored under that particular field.

PROGRAM 5.7b

```
7260 FOR x%=first% TO last%
7270 CLS#2:LOCATE 1,4
7280 GOSUB 7500:REM Display record time
7290 LOCATE 8,22
7300 PRINT"Press space bar to continue"
7310 IF INKEY#<>" " THEN 7310
7320 NEXT x%
7330 LOCATE 4,22:PEN 1
7340 PRINT"The Display module is now finished"
7350 FOR x=1 TO 2000:NEXT x
7360 RETURN
7500 REM *****Display record*****
7510 LOCATE 2,4:PRINT"Record number ";
7520 PEN 1:PRINT x%;
7530 IF dat$(x%,0)="d" THEN PEN 3:PRINT"
  (Dead Record)":ELSE PRINT
7540 LOCATE 1,6
7550 FOR dr%=1 TO f%
7560 PEN 1:PRINT dr%;
7570 PEN 2:PRINT LEFT$(dat$(x%,dr%),34)
7580 NEXT dr%
7590 RETURN
```

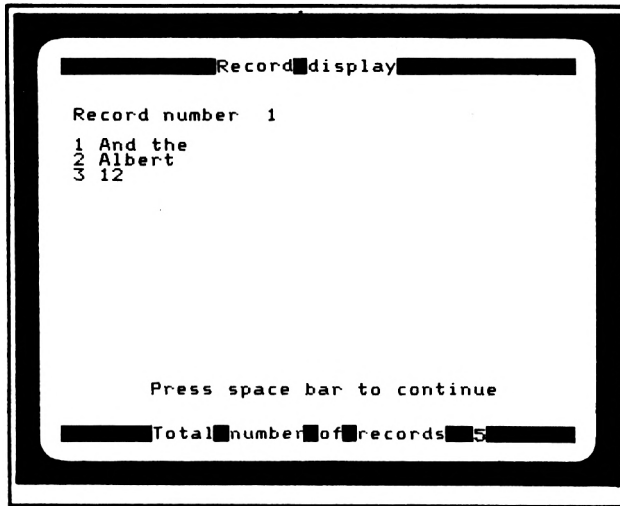
Testing this module is very simple. Run the program and load in the previously saved file (Via option 2). When the file has been loaded in, the program will return to the main menu. The user then selects option 7 to 'Display records in memory' and is asked to press any key to continue (as usual). When the user has decided to continue, enter the number of the first record to be displayed. Any input less than 1 or greater than 5 will be ignored. (One is the number of the first record and 5 the number of the last record). Type in the following value:

Number of first record
to be displayed: **1**

The number of the last record to be displayed should not be less than the number of the first record to be displayed and should not be greater than the last record number. Type in the following value:

Number of last record
to be displayed: **5**

This informs the computer that the user wishes to see all of the records, i.e. records one to five in this case. Input the number of the last record to be displayed and press ENTER. The screen will clear and you will see the following display:



Display Of A Record

FIGURE 5.13

After looking at a record, press the space bar and the next record will be displayed, until the last record is displayed and the following message appears:

The Display module is now finished

The screen is then cleared and the main menu is displayed. The display module has worked perfectly, and not only that, but because the records loaded correctly, we can say that the Save and Load modules worked as well.

Module 4: Add, delete or alter records

Now, at long last, we can return to module four. The next stage of this module is the delete routine and so, without further ado....

Delete

This is a very simple routine which only requires the user to type in the record number to begin deleting at and the record number to finish deleting at. The input of this data is a similar routine to that used in the display record module. The first record number that you want to delete is tested to make sure it is not less than one and not greater than the last record number. The last record number is tested to make that is it not less than the first record number and not greater than the last record. Any input that fails this test will be ignored.

PROGRAM 5.4i

```
4295 REM *****Delete record*****
4300 CLS#1:CLS#2
4305 LOCATE#1,13,1
4310 PRINT#1,"Delete record":PEN 2
4315 GOSUB 9200:REM Continue?
4320 IF r$="0" THEN CLS#2:RETURN
4325 CLS#2:LOCATE 1,4
4330 PRINT"Enter first record number"
4335 LOCATE 16,5:PRINT" to delete:"
4340 WINDOW#7,27,39,5,5:CLS#7
4345 PEN#7,3:INPUT#7,"",c$
4355 c$=LEFT$(c$,4)
4360 dr%=VAL(c%):REM Delete record
4365 IF dr%<1 OR dr%>1r% THEN 4340
4370 LOCATE 1,8
4375 PRINT"Enter last record number"
4380 LOCATE 17,9:PRINT"to delete:"
4385 WINDOW#7,27,39,9,9:CLS#7
4390 PEN#7,3:INPUT#7,"",c$
4400 c$=LEFT$(c$,4)
4405 ld%=VAL(c%):REM Last record to delete
4410 IF ld%<dr% OR ld%>1r% THEN 4385
```

The actual deleting of the record is something of a fraud because records are not deleted but a value is set to indicate that you don't want them any more. The value that is set is the zeroth value of array DAT\$, i.e.

If DAT\$(Record number,0)="d" then the record has been deleted.

By keeping a record's information, it is possible for a record to be recovered. To recover a record, the 'd' from the DAT\$ arrays must be removed. More of that later.

Program 5.4j sets up a loop from the delete start value (DR%) to the delete finish value (LD%) to perform the actual deleting. A small display is used to tell the user which records are being deleted and the 'number of records count' (NR%) is decremented each time through the loop. When the delete routine is finished, the program returns to the submenu.

PROGRAM 5.4j

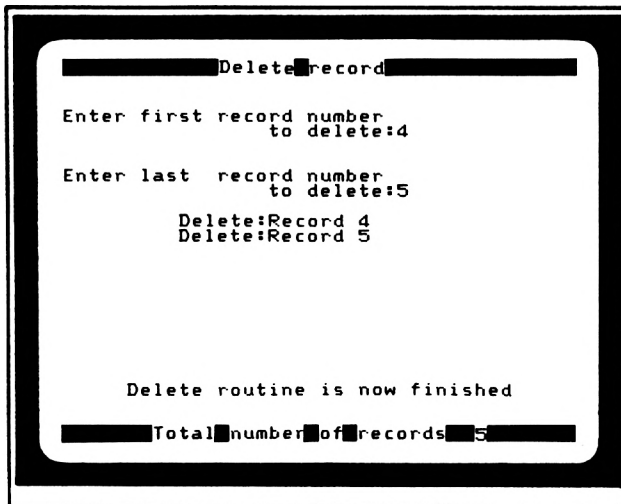
```
4415 WINDOW#7,10,29,11,20
4420 FOR x%=dr% TO ld%
4425 PEN#7,1:PRINT#7,"Delete:";
4430 PEN#7,3:PRINT#7,"Record";x%
4435 dat$(x%,0)="d":REM Dead record
4440 nr%=nr%-1:REM Number of Records
4445 NEXT x%
4455 LOCATE 6,22:PEN 1
4460 PRINT"Delete routine is now finishe
d"
4465 FOR x=1 TO 2000:NEXT x:RETURN
```

Testing the delete routine is simple. First load the previously saved file and then select option four at the main menu. When faced with the submenu, select option 2. Type in these example inputs:

Enter first record number
to delete: 4

Enter last record number
to delete: 5

This tells the computer that we no longer want records four and five. Once this has been typed in, you will see the following display:



Delete Display

FIGURE 5.14

At this stage, it appears that the delete routine has worked. Upon returning to the main menu, select option seven (display records) and display records four and five. The whole purpose of a delete routine is to remove unwanted records from a file; unfortunately our delete routine does not quite do this. At present, records chosen for deletion are only marked, not removed. The removal of deleted records takes place in the save module where only records without a 'd' at DAT\$(X%,0) are saved, thus deleting all unwanted records.

To test these changes you must perform the following functions:

1. Load data file
2. Delete records 4 and 5
3. Display records 4 and 5.

With these records you will see:

(Dead Record)

4. Save the file
5. Load the file back into memory

At this point, the number of records at the bottom of the screen will tell you that there are three records in memory. We had five records and then deleted two which left three. If you are still not convinced that the delete records routine worked, then try and display records four and five. They are not there.

The next part of module four is the Alter routine.

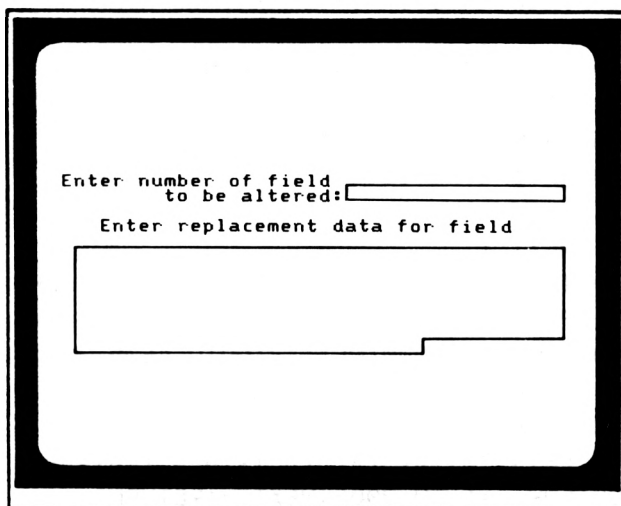
Alter Record

The Alter record routine would be used if you made a mistake whilst adding a record or if you wanted to update an existing record. The user is asked to 'Enter record number to be altered'. This value is tested to make sure it is not less than one and not greater than the last record number. As per normal, any value that fails the test is ignored.

PROGRAM 5.4k

```
4470 REM *****Alter record*****
4475 CLS#1:CLS#2
4480 LOCATE#1,14,1
4485 PRINT#1,"Alter record":PEN 2
4490 GOSUB 9200:REM Continue?
4495 IF r$="" THEN CLS#2:RETURN
4500 CLS#2:LOCATE 1,4
4505 PRINT"Enter record number"
4510 LOCATE 7,5:PRINT"to be altered:"
4515 WINDOW#7,21,39,5,5:CLS#7
4520 PEN#7,3:INPUT#7,"",c$
4530 c$=LEFT$(c$,4)
4535 ar%=VAL(c$):REM Alter Record
4540 IF ar%<1 OR ar%>1r% THEN 4515
```

Having selected the record that the user wishes to amend, the computer clears the screen and displays the chosen record. The user is then asked to 'Enter number of field to be altered'. This value undergoes tests to make sure it is in the required range (not less than one and not greater than the number of fields). Lastly, the user is asked to type in replacement data for the chosen field. All the inputs in this module use formatting window seven and Figure 5.15 shows the windows for each input:



Enter number of field
to be altered:

Enter replacement data for field

Alter Record Input Windows

FIGURE 5.15

When the user has entered the replacement data, the field type is checked to see if it is numeric (FND\$(AF%,2)="N") or string (FDN\$(AF%,2)="S"). If it is a string field, the program simply continues. However, if the field has been specified as numeric, the user's input needs to be checked to make sure that it contains only numeric characters. If this test fails, FL is set and the user is asked to input the data again. The Alter routine ends with a polite message:

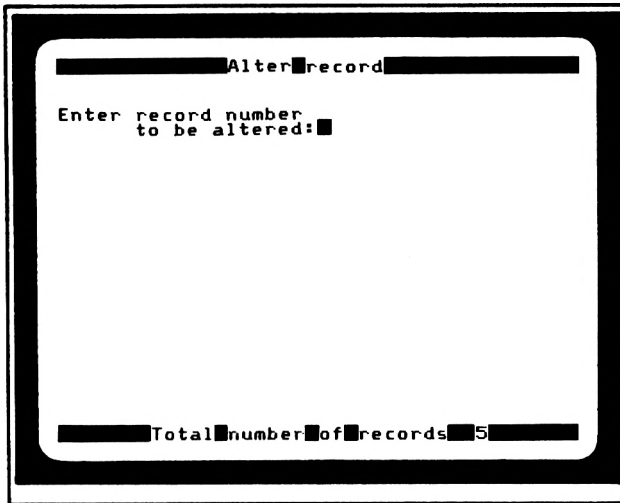
Alter record is now finished

and the program is returned to the Submenu.

PROGRAM 5.41

```
4543 CLS#2
4545 x%=ar%:GOSUB 7500:REM Display record
4550 LOCATE 1,12
4555 PRINT"Enter number of field"
4560 LOCATE 9,13:PRINT"to be altered:"
4565 WINDOW#7,23,39,13,13:CLS#7
4570 PEN#7,3:INPUT#7,"",c$
4580 c%=LEFT$(c$,4)
4585 af%=VAL(c%):REM Alter Field
4590 IF af%<1 OR af%>f% THEN 4565
4595 LOCATE 4,15
4600 PRINT"Enter replacement data for field ";
4605 WINDOW#7,2,39,17,24:CLS#7
4610 PEN#7,3:INPUT#7,"",dat$(ar%,af%)
4615 IF fdn$(af%,2)="S" THEN 4645
4620 f1=0:FOR l=1 TO LEN(dat$(ar%,af%))
4625 l%=MID$(dat$(ar%,af%),l,1)
4630 IF INSTR("1234567890",l%)=0 THEN f1
=1
4635 NEXT l
4640 IF f1=1 THEN CLS#7:LOCATE#7,7,1:PEN
#7,1:PRINT#7,"Numerical data only please
":FOR x=1 TO 2000:NEXT x:GOTO 4605
4645 CLS#7
4650 LOCATE 7,20:PEN 1
4655 PRINT"Alter record is now finished"
4660 FOR x=1 TO 2000:NEXT x:RETURN
```

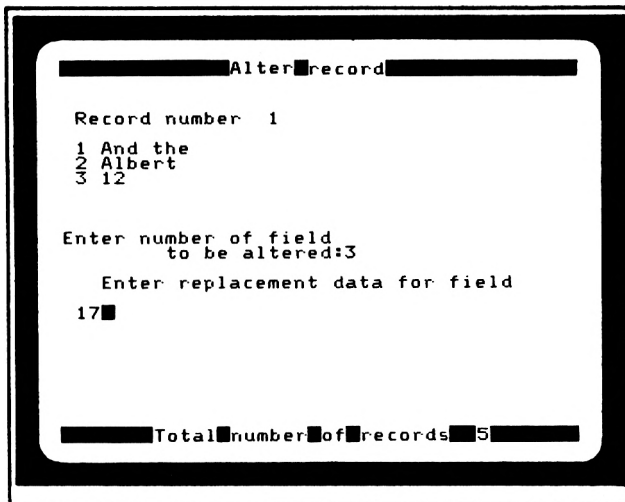
The testing of this routine is quite simple. First load in one of the previously saved files. At this stage, it does not matter which one you load in. When loaded, select option 4 at the main menu and option 3 at the submenu and press any key other than '0' to enter the alter routine. You will see the following display:



Alter Display

FIGURE 5.16

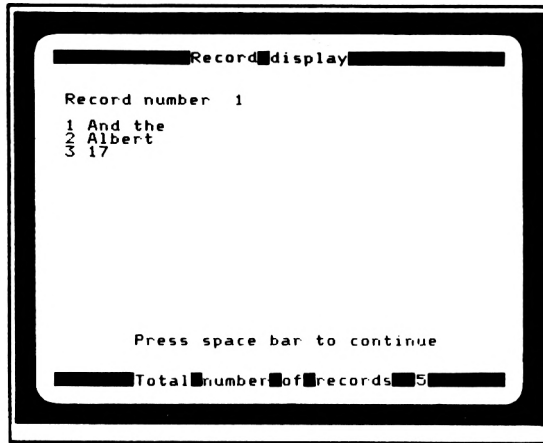
Select record 1 as the record to be altered, and it will appear on the screen. Next, the user must decide which field to change. The example records have three fields, so the value input has to be from one to three. In this example, we will change the third field (age), therefore type in 3 and press ENTER. This field is a numeric field so the replacement data has to be numeric. Replace the value in field 3 with '17'. The screen will look like this:



Replacing A Field

FIGURE 5.17

When you press ENTER, the program returns to the submenu. Select '0' and you will be returned to the main menu. From here, go to the display record module (option 7) and display record one. You will see the following:



Record One After Alteration

FIGURE 5.18

The old data for record 1 field 3 has been replaced by the new data.

Now, all the major modules have been developed and there are three minor ones left to do.

Module 5: Search file for a record

The search module allows the user to check every item of data in a specified field throughout the file to see if it matches a given search string. First the user has to specify which field to search through. The value input by the user is checked to make sure it is not less than 1 and not greater than the number of fields (F%). Any value that fails this test is simply ignored and the user invited to type in another value.

PROGRAM 5.5a

```

5000 REM *****Search file*****
5010 CLS#1:CLS#2
5020 LOCATE#1,13,1
5030 PRINT#1,"Record search":PEN 2
5035 IF ac% THEN GOSUB 9300:RETURN
5040 GOSUB 9200:REM Continue?
5050 IF r$="0" THEN CLS#2:RETURN
5060 CLS#2:LOCATE 1,4
5070 PRINT"Enter number of search field:
"
5080 WINDOW#7,30,39,4,4:CLS#7
5090 PEN#7,3:INPUT#7,"",c$
5110 c$=LEFT$(c$,2)
5120 sf%=VAL(c$):REM Search Field
5130 IF sf%<1 OR sf%>f% THEN 5080
  
```


The next stage is for the user to decide what to search for. The search string (SS\$) is compared with each record's entry under the chosen field. If a match occurs, the record with the matching data is displayed. Because more than one match can occur in a file, the loop continues until all the records (including dead records) have been checked. A flag (FL) is used to indicate whether any matches were found. If no occurrence of the search string was found amongst the data stored in the specified fields, the user is informed 'No matches found'. When the search routine is finished, program control is returned to the main menu.

PROGRAM 5.5b

```
5140 LOCATE 10,6
5150 PRINT"Enter search string ";
5160 WINDOW#7,2,39,8,15:CLS#7
5170 PEN#7,3:LINE INPUT#7,"",ss$:REM SS$
=String Search
5190 fl=0
5200 FOR x%=1 TO 1r%
5210 CLS#2:LOCATE 1,4
5220 IF INSTR(dat$(x%,sf%),ss$) THEN fl=
1:GOSUB 7500:GOTO 5240
5230 IF NOT INSTR(dat$(x%,sf%),ss$) THEN
5270
5240 LOCATE 1,16
5250 PRINT"Press spacebar to continue"
5260 IF INKEY$("<>") " THEN 5260
5270 NEXT x%
5275 IF fl("<>") THEN 5290
5280 LOCATE 13,18:PRINT"No matches found
"
5290 LOCATE 4,20:PEN 1
5300 PRINT"The search module is now fini
shed"
5310 FOR x=1 TO 2000:NEXT x:RETURN
```

To test this module, load in the first file that was saved to cassette (the one with five records). When loaded, select option 5 – search file for a record. Because there are three fields (surname, first name and age), the search field value (SF%) must be between 1 and 3. For this demonstration select field 3, AGE. The search string will be '18'. The program will be searching through all the records to see if anybody has an age of 18.

Searching

FIGURE 5.19

There are two records that have a field 3 value of 18, and these will be displayed on the screen. When finished, the search module will announce:

The Search module is now finished

and the program will return to the main menu.

Module 6:Sort records in memory

The sort module is a helpful option when attempting to organise your file. A file can be sorted into ascending (up) or descending (down) order using any of the available fields.

Once in the sort module, the field numbers, names and types are all displayed showing the user the sort options available.

PROGRAM 5.6a

```

6000 REM *****Sort file*****
6010 CLS#1:CLS#2
6020 LOCATE#1,15,1
6030 PRINT#1,"Sort records":PEN 2
6035 IF ac% THEN GOSUB 9300:RETURN
6040 GOSUB 9200:REM Continue?
6050 IF r$="0" THEN CLS#2:RETURN
6060 CLS#2:PEN 1:LOCATE 4,4:PRINT"Field
name"

```

```

6070 LOCATE 31,4:PRINT"Type"
6080 FOR x%=1 TO f%
6100 PEN 1:PRINT x%;
6110 PEN 2:PRINT LEFT$(fdn$(x%,1),26);
6120 IF fdn$(x%,2)="S" THEN p$="String":
ELSE p$="Numeric"
6130 PRINT TAB(31);p$
6140 NEXT x%

```

The user is asked to enter the sort field number. This value must not be less than one or greater than the number of records (F%). If a value outside this range is input, it is ignored and the user is requested to type in more data. Once the decision is made on which field to sort, the user is asked whether it is to be sorted into ascending or descending order.

PROGRAM 5.6b

```

6150 LOCATE 1,21
6160 PRINT"Enter sort field number:"
6170 WINDOW#7,25,39,21,21:CLS#7
6180 PEN#7,3:INPUT#7,"",c$
6200 c$=LEFT$(c$,2)
6210 fst%=VAL(c%):REM Field sort
6220 IF fst%<1 OR fst%>f% THEN 6170
6230 LOCATE 1,23
6240 PRINT"Ascending or Descending sort
(A/D)";:PEN 3:PRINT CHR$(143)
6250 IF INKEY#<>"" THEN 6250
6260 a$=INKEY$:IF a$="" THEN 6260:ELSE a
$=UPPER$(a$)
6270 IF a$<>"A" AND a$<>"D" THEN 6260
6280 LOCATE 35,23:PEN 3:PRINT a$:PEN 2

```

Both ascending and descending sorts follow the procedure of comparing two different records using the chosen field. When performing a descending sort, the test is as follows:

IF DAT\$(record,sortfield)<DAT\$(record+1,sortfield) then...

If this condition is true, the values stored for DAT\$(record) are swapped with those stored for DAT\$(record+1). The swapping over of entire records is performed by the subroutine at lines 6599 to 6640. An ascending sort follows the same pattern as a descending sort only the actual test is different.

IF DAT\$(record,sortfield)>DAT\$(record+1,sortfield) then..

The ascending and descending sort routines use a FOR...NEXT loop and a WHILE...WEND loop. The FOR...NEXT loop runs through all the records once and if any records are swapped, the S flag is set. The WHILE...WEND loop forces the actual field checking loop to be repeated until no more swaps are necessary. If there are no swaps, S is set to zero and the WHILE condition proves false, thus the program stops its looping and returns. At the end of the sort routine, the user is informed that the sort routine has finished and the program returns to the main menu.

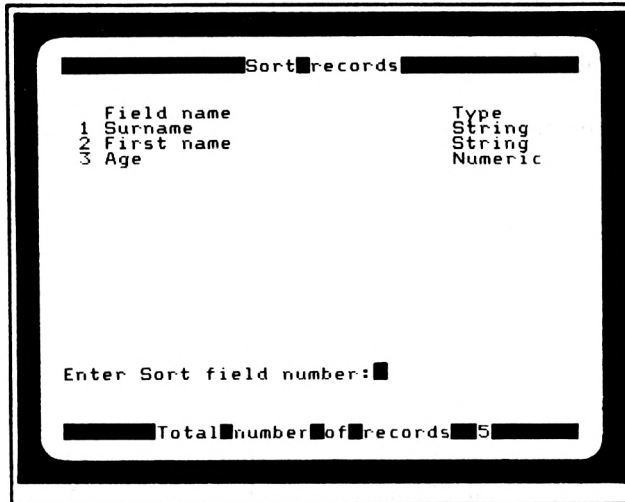
PROGRAM 5.6c

```

6290 IF a$="D" THEN GOSUB 6340:REM Desce
nding
6300 IF a$="A" THEN GOSUB 6460:REM Ascen
ding sort
6310 LOCATE 1,21:PRINT SPACE$(40)
6320 LOCATE 1,23:PRINT SPACE$(40)
6325 LOCATE 4,22:PEN 1:PRINT"The sort mo
dule is now finished"
6330 FOR x=1 TO 2000:NEXT x:RETURN
6340 REM *****Descending Sort*****
6343 s=-1
6345 WHILE s
6347 s=0
6350 FOR i%=1 TO lr%-1
6360 IF dat$(i%,fst%)<dat$(i%+1,fst%) TH
EN GOSUB 6600:s=-1
6370 NEXT i%
6375 WEND
6380 RETURN
6460 REM *****Ascending sort*****
6463 s=-1
6465 WHILE s
6467 s=0
6470 FOR i%=1 TO lr%-1
6480 IF dat$(i%,fst%)>dat$(i%+1,fst%) TH
EN GOSUB 6600:s=-1
6490 NEXT i%
6495 WEND
6500 RETURN
6599 REM *****Swap Time*****
6600 FOR l%=0 TO f%
6610 tran#=dat$(i%,l%)
6620 dat$(i%,l%)=dat$(i%+1,l%)
6630 dat$(i%+1,l%)=tran#
6635 NEXT l%
6640 RETURN

```

To test this module, load in a previously created file, the one with five records and then select option six. After the user tells the Amstrad to continue, the following display appears:



Choosing A Sort Field

FIGURE 5.20

This tells us what fields we have available to sort. In this example we have a choice of three fields. To demonstrate the sort module we will sort the first field into ascending order. This means that the records will be sorted into alphabetical order by surname. Type in the following responses to the computer's questions.

Enter sort field number:1

Ascending or Descending sort (A/D):A

There will be a slight moment of apparent idleness whilst the program sorts out your records. When it has finished, you will be returned to the main menu. From here, select option 7 and display all five records. They will be displayed in the following order:

Record	Surname	First name	Age
1	And the	Albert	12
2	Beanstalk	Jack	18
3	Beauty	Sleeping	237
4	Rockerfella	Cinderella	18
5	Young	Arthur	93

To test the descending sort routine we will arrange the file starting with the highest age and ending with the lowest. Go back to the sort module by choosing option six at the main menu and type in the following values:

Enter sort field number:**3**

Ascending or Descending sort (A/D):**D**

Once again there is a slight pause and the program returns to the main menu. When the user displays the records they appear in the following order:

Record	Surname	First name	Age
1	Beauty	Sleeping	287
2	Young	Arthur	93
3	Beanstalk	Jack	18
4	Rockerfella	Cinderella	18
5	And the	Albert	12

When two records have the same value (in this example records 3 and 4) their positions remain the same in relation to each other.

Module 8: Exit program

At long last we arrive at the last module, which will only be used when the user has finished with the database program. The exit module is the easiest of all the modules. The user is simply asked to confirm the requested exit from the program. If the user does, then 'THE END' is displayed and the the program ends. If the user does not want to finish, the main menu is re-entered.

PROGRAM 5.8

```
8000 REM *****The end*****
8010 CLS:LOCATE 12,12:PEN 1
8020 PRINT"Program Termination"
8030 PEN 2
8040 GOSUB 9200:REM Continue?
8050 IF r$="0" THEN CLS#2:RETURN
8060 REM *****Close down sequence*****
8070 CLEAR:MODE 1
8080 LOCATE 17,12:PEN 1
8090 PRINT"The End"
8100 END
```

Appendix 1

Amstrad Arithmetic Statements

ABS()

ABS returns the ABSolute value of the contents of a variable, or the value of an expression inside brackets. In other words it makes the expression positive.

```
PRINT ABS(8) will return 8
```

```
PRINT ABS(0) will return 0
```

ATN()

ArcTanGent will convert a tangent value back into radians. The tangent value goes inside the brackets.

```
PRINT ATN(2) will return 1.10714872
```

BIN\$()

BIN\$ is a function that will display the value of a hexadecimal or decimal number in BINary. The first argument represents the number to be converted, and the (optional) second argument represents the number of digits in the answer. Decimal places are rounded. See CINT.

```
PRINT BIN$(64) will produce 01000000
```

```
PRINT BIN$(64,7) will produce 1000000
```

Only 7 digits appear after typing in the last BIN\$ because the number of digits was specified as seven. If too few digits are asked for, e.g. BIN\$(64,4), then the minimum number necessary to display the correct answer is used.

CINT()

CINT converts a given value to its nearest integer value

```
PRINT CINT(74.352) will produce 74
```

```
PRINT CINT(74.5) will produce 75
```

COS()

The COS function returns the COSine of the variable or expression within the brackets. The result will be for radians unless otherwise specified prior to the execution of the COS function.

```
PRINT COS(50) will produce 0.964966027
```

CREAL()

CREAL converts a value to a **REAL** number. In other words, a number with decimal places.

DEG

DEG changes the default RADian mode to DEGree mode. So if you wanted to know what COS(50) is with 50 in degrees, you would type

```
PRINT COS(50) will return 0.64278761
```

or for 50 in radians:

```
RAD
```

```
PRINT COS(50) will return 0.964966027
```

EXP()

The EXPonential function raises 'e' to the power of the variable or expression within the brackets, where 'e' = 2.7182818....

```
PRINT EXP(10) will return 22026.4658
```

FIX()

The FIX function returns the value of the integer part of a decimal number. It differs from the CINT function in that the decimal number is truncated not rounded, e.g.

```
PRINT FIX(10.752) will return 10
```

```
PRINT CINT (10.752) will return 11
```


HEX\$()

HEX\$ is a function that will convert a binary or decimal value into HEXadecimal. As with BIN\$, the first argument represents the number to be converted, and the optional second argument represents the number of digits in the answer.

```
PRINT HEX$(15,4) will return 000F
```

INT()

The INTeger function rounds a decimal number DOWN to the next smaller integer. It gives the same result as FIX for positive numbers, but one less than FIX for negative decimal numbers.

```
PRINT FIX(-4.592) will return -4
```

but

```
PRINT INT(-4.592) will return -5
```

LOG()

The LOG function returns the LOGarithm, to base e, of the variable or expression within the brackets.

LOG10()

The LOG10 function returns the natural LOGarithm of the variable or expression within the brackets.

```
PRINT LOG10(245) will return 1.39794001
```

MAX()

The MAX function will return the highest or MAXimum value of the numbers and/or the expressions within the brackets.

```
10 M=55
20 PRINT MAX(10,20,M)
30 PRINT MAX(10,20,M,2*100)
```

```
RUN
55
200
```

MIN()

The MIN function will return the lowest or MINimum value of the numbers and/or the expressions within the brackets.

```
10 M=1
20 PRINT MIN(30,M,2)
30 PRINT MIN(30,M,2,0)

RUN
1
0
```

PI

PI (π) is the sixteenth letter in the Greek alphabet and represents the ratio between the circumference and diameter of a circle.

PRINT PI will return 3.14159265...

Try this

```
10 RADIUS=10
20 PRINT"CIRCUMFERENCE=";2*PI*RADIUS
30 PRINT "AREA=";PI*(RADIUS^2)
```

when run:

```
CIRCUMFERENCE= 62.8318531
AREA= 314.159265
```

RAD

RAD, an abbreviation of RADians, is the default unit of angular measurement when the Amstrad is turned on. RAD only needs to be used if DEG has previously been used. See DEG.

RND()

'RND' produces a RaNDom number between but not including 0 and 1. 'RND(0)' returns the last random number generated.

ROUND()

ROUND works in the same way as CINT unless the optional second argument is included. The first variable or expression is the number to be rounded, and the second variable or expression is the number of digits to which the answer is to be rounded.

For example

```
PRINT ROUND(-1.23) will return -1
```

```
PRINT ROUND(8.653,2) will return 8.65
```

```
PRINT ROUND(8.6549,2) will return 8.65
```

SGN()

The SGN function is concerned with finding the SiGN of the number or expression within the brackets.

If the number is negative, -1 is returned

If the number is 0, 0 is returned

SIN()

The SIN function returns the SINE of the variable or expression inside the brackets. The result is for radians unless otherwise specified by the DEG command.

```
PRINT SIN(25) will return -0.132351752
```

SQR()

SQR returns the SQuare Root of the value within the brackets. The value inside the brackets must not be less than zero because negative numbers do not have real square roots.

```
PRINT SQR(9) will return 3
```

TAN()

The TAN function returns the TANgent of the value within the brackets. The value must be in the range -200000 to +200000. TAN defaults to radians unless otherwise specified by DEG.

```
PRINT TAN(25) will return -0.133526409
```

UNT()

UNT converts a number in the range 0 to 65535 to the range -32768 to +32767. Mainly used to convert a number in the first range to a two's complement number by reversing the bits and adding 1. All that jargon is concerned with Assembly language and machine code: it doesn't have much to do with BASIC.

Appendix 2

Using The Data Recorder and Disc unit.

When switched on, the Amstrad checks its input ports to see what is plugged in. If you have a disc unit plugged in then all load and save operations default to the disc unit. If no disc unit is plugged in then all load and save operations default to the tape unit. All the commands in this appendix work in the same way for tape and disc except where indicated.

SAVE

The SAVE command can be used to save three types of files, each type selected by a code.

SAVE 'filename',A

When using SAVE,A a BASIC program is 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 on the disc/tape.

SAVE 'filename',P

The 'P' 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:

```
1Ø REM PROTECTED PROGRAM
2Ø PRINT"I AM SAFE"
```
- `SAVE"PROTECT",P`
- The computer 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 computer 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 start 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 variation of the LOAD command. When CHAIN is used the loaded program automatically RUNS from a specified line number.

```
CHAIN "ALBERT",100
```

This will load the BASIC program ALBERT 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 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 first program encountered on the tape.

The RUN" form of loading does not work with discs.

Both CHAIN and RUN can be used in exactly the same way with protected and unprotected programs.

SAVE 'filename',B

This form of SAVE is used to save the contents of specified memory locations. The data is stored on disc/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 execution address. 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:

Dr Watson Series
Amstrad Assembly Language Course
by
Tim Herbertson

This is available from all good computer bookshops, or by mail order direct from the publishers, Glentop Publishers Ltd.

Saving To Disc

The Amstrad has a built-in disc backup system. When you save a program or file to disc, using any of the save forms, a check is made to see if a file already exists with that name on the disc. For example, if you have a program called 'DATABASE' on your disc and try to save an updated version, the one already on the disc is renamed to 'DATABASE.BAK'. The bak is short for backup. The new copy is saved under the name 'DATABASE'.

If, having two copies of the program, an attempt is made to save another program called DATABASE, then the copy called 'DATABASE.BAK' is forgotten – the copy called 'DATABASE' is renamed 'DATABASE.BAK' and the latest copy is saved as 'DATABASE'.

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 disc/tape. The MERGE command loads the disc/tape program line by line. Suppose the first line loaded is line 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 disc/tape, if no line 10 is present then the line from disc/tape is inserted into the program in memory. 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
Disc/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
```

The MERGE Command

FIGURE A2.1

The new program is a combination of the program loaded in and any lines of the program that already in the computer which do not have the same line numbers in the disc/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 tape/disc program ALBERT with the program currently in memory. Once merging is completed the program will automatically RUN starting at the first line of the program.

```
CHAIN MERGE "ALBERT",100
```

This will merge the tape/disc 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 CHAIN MERGE command can be used with a third argument, DELETE:

```
CHAIN MERGE "ALBERT",100,DELETE 300-
```

This will cause the computer to delete lines 300 onwards from the program in memory BEFORE the program on tape/disc is merged.

CAT

The last of the Amstrad's BASIC tape/disc 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, or the disc in the disc drive.

After typing in CAT, if you are using a tape system, the Amstrad will respond with:

Press PLAY then any key:

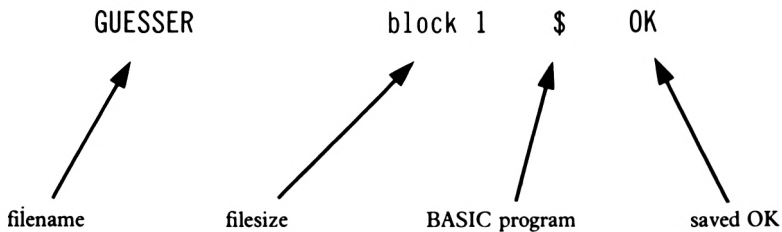
Once a key is pressed, the Amstrad will begin searching through the tape. Every time a program is encountered the following will be displayed:

```
<filename> block <n> <filetype> OK
```

Where <filename> is the name of the file found. <n> is the block number and <filetype> 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

When CAT is used with discs, the catalogue is displayed in Alphanumeric order.

Appendix 3

Printers

Sending information from an Amstrad to a printer is very simple. the printer is assigned a stream value of 8 so anything printed to this stream will be printed out!

Only two commands work with stream 8: LIST and PRINT.

LIST#8

When listing to a printer, the LIST command works the same way as it would for any other stream:

LIST#8	List all program
LIST 30,#8	List only line 30
LIST-30,#8	List up to line 30 inclusive
LIST 30-,#8	List line 30 onwards
LIST 30-60,#8	List lines 30 to 60 inclusive

PRINT#8

PRINT#8 is the same as PRINTing to any other stream.

PRINT#8, "This is a printout"

PRINT#8, P\$,T\$,O\$

Appendix 4

To display characters 0 to 31, CHR\$(1) should be printed before each one (see Appendix 5).

0 = □	32 =	64 = @	96 = `
1 = ▮	33 = !	65 = A	97 = a
2 = ⊥	34 = "	66 = B	98 = b
3 = ⊓	35 = #	67 = C	99 = c
4 = ⚡	36 = \$	68 = D	100 = d
5 = ☒	37 = %	69 = E	101 = e
6 = ✓	38 = &	70 = F	102 = f
7 = ⚙	39 = '	71 = G	103 = g
8 = ←	40 = (72 = H	104 = h
9 = →	41 =)	73 = I	105 = i
10 = ↓	42 = *	74 = J	106 = j
11 = ↑	43 = +	75 = K	107 = k
12 = ⚡	44 = ,	76 = L	108 = l
13 = ←	45 = -	77 = M	109 = m
14 = ⊗	46 = .	78 = N	110 = n
15 = ⊙	47 = /	79 = O	111 = o
16 = □	48 = 0	80 = P	112 = p
17 = ⊙	49 = 1	81 = Q	113 = q
18 = ⊙	50 = 2	82 = R	114 = r
19 = ⊙	51 = 3	83 = S	115 = s
20 = ⊙	52 = 4	84 = T	116 = t
21 = ×	53 = 5	85 = U	117 = u
22 = ∏	54 = 6	86 = V	118 = v
23 = -	55 = 7	87 = W	119 = w
24 = X	56 = 8	88 = X	120 = x
25 = †	57 = 9	89 = Y	121 = y
26 = ?	58 = :	90 = Z	122 = z
27 = ⊙	59 = ;	91 = [123 = {
28 = ⊞	60 = <	92 = \	124 =
29 = ⊞	61 = =	93 =]	125 = }
30 = ⊞	62 = >	94 = ^	126 = ~
31 = ⊞	63 = ?	95 = _	127 = ⌘

128	=		160	=	^	192	=	/	224	=	⊗
129	=	■	161	=	∧	193	=	∖	225	=	⊕
130	=	■	162	=	∩	194	=	/	226	=	+
131	=	■	163	=	£	195	=	∖	227	=	◆
132	=	■	164	=	⊙	196	=	^	228	=	♥
133	=	■	165	=	π	197	=	>	229	=	♠
134	=	■	166	=	5	198	=	√	230	=	○
135	=	■	167	=	'	199	=	<	231	=	◆
136	=	■	168	=	14	200	=	/	232	=	□
137	=	■	169	=	12	201	=	∖	233	=	■
138	=	■	170	=	24	202	=	◇	234	=	♂
139	=	■	171	=	±	203	=	×	235	=	♀
140	=	■	172	=	÷	204	=	/	236	=	J
141	=	■	173	=	→	205	=	∖	237	=	♫
142	=	■	174	=	ó	206	=	✖	238	=	✖
143	=	■	175	=	i	207	=	✖	239	=	↑
144	=	·	176	=	α	208	=	—	240	=	↑
145	=	·	177	=	β	209	=		241	=	↓
146	=	-	178	=	υ	210	=	—	242	=	←
147	=	∪	179	=	6	211	=		243	=	→
148	=	∪	180	=	€	212	=	▾	244	=	▲
149	=		181	=	θ	213	=	▾	245	=	▼
150	=	r	182	=	λ	214	=	▴	246	=	▶
151	=	†	183	=	ρ	215	=	▴	247	=	◀
152	=	-	184	=	π	216	=	✖	248	=	⚙
153	=	∪	185	=	σ	217	=	✖	249	=	⚙
154	=	—	186	=	♂	218	=	✖	250	=	⚙
155	=	⊥	187	=	♂	219	=	✖	251	=	⚙
156	=	∪	188	=	×	220	=	▾	252	=	⚙
157	=	†	189	=	ω	221	=	▾	253	=	⚙
158	=	τ	190	=	Σ	222	=	▾	254	=	⚙
159	=	+	191	=	Ω	223	=	▾	255	=	⚙

Appendix 5

Special Character Codes

CHR\$(0)

This code has no effect whatsoever.

CHR\$(1)

This code enables the first 31, usually unseen, ASCII characters to be displayed on the screen. This is possible by typing 'PRINT CHR\$(1)+CHR\$(n)' where 'n' is the ASCII code number of the character you wish displayed.

CHR\$(2)

This code will make the text cursor the same colour as the paper, in other words making the text cursor invisible. Printing CHR\$(2) does not work in immediate mode, but it can be quite useful in programs as the text cursor will not be displayed flashing around the screen while information is being displayed.

CHR\$(3)

As CHR\$(2) turns off the text cursor within a program, CHR\$(3) will turn the text cursor on again. This may be necessary in a situation where the text cursor is turned off to display information on the screen but needs to be turned on again to take data from an input statement, the displayed cursor acting as a prompt. The cursor is automatically switched on when any system message is displayed.

CHR\$(4)

This code is used as another method of setting the screen mode by typing 'PRINT CHR\$(4)+CHR\$(n)' where n is the screen mode (between 0 and 2).

CHR\$(5)

To send a character to the graphics cursor this code can be used. 'PRINT CHR\$(5) + CHR\$(n)' is typed, where n is the number of the character to be displayed (between 0 and 255). This command is useful when a letter or perhaps a word is to be displayed at an exact position that does not conform to the usual text screen coordinates.

CHR\$(6)

This command will allow text to be displayed after a CHR\$(21) has prevented it. This command cannot be used in immediate mode.

CHR\$(7)

Printing this character will clear all the sound queue waiting to be played by playing a note. The note played is the same as that obtained by holding CTRL and P.

CHR\$(8)

This moves the cursor backwards one character. To demonstrate this, type in the following direct entry command:

```
PRINT "####"+CHR$(8)+CHR$(A)
```

This will display:

```
###A
```

After displaying 4 hashes the cursor is moved backwards one space so that it is on top of the hash. At this point 'A' is printed over the hash.

CHR\$(9)

This does the opposite of CHR\$(8): it moves the cursor forward one position.

CHR\$(10)

This moves the cursor down one line and works in a similar way to CHR\$(8) and CHR\$(9).

CHR\$(11)

This moves the cursor up one line. This works in the same way as the previous three.

```
10 FOR I=1 TO 20
20 CLS
30 RESTORE
40 LOCATE 1,10
50 FOR J=1 TO 29
60 READ A
70 PRINT CHR$(A);
80 PRINT ", ";
90 NEXT J
100 NEXT I
110 DATA 9,9,9,9,10,10,10,11,11,11,11,11,11,11,10,10,
10,1,10,10,10,10,11,11,9,9
120 END
```

CHR\$(12)

When printed, the screen is cleared and the cursor appears at the top left hand location.

CHR\$(13)

This performs a carriage return.

CHR\$(14)

This code is the same as the paper command. With a parameter it could be used instead of the paper command. The syntax is PRINT CHR\$(14)+CHR\$(n) where n is the ink pot, between 0 and 15.

CHR\$(15)

This command is the ink version of the above command. It can be used instead of the ink command.

CHR\$(16)

This could be quite a useful command as it enables the character the cursor is on to be deleted, (by filling the space with the paper colour).

CHR\$(17)

When used in a program (this cannot be used in immediate mode), a whole line of text or whatever is on the line will be cleared; this command clears all the data to the left of the cursor.

CHR\$(18)

This command is similar to the above but works in the opposite direction. It clears from, and including the current cursor position to the right-hand edge of the screen.

CHR\$(19)

This is of the same family as the above two but this one clears everything to the left and above the cursor.

CHR\$(20)

This performs the opposite of the previous character, i.e. it clears everything from the right and down from the cursor.

CHR\$(21)

This character prevents text output to the screen. It can be used in immediate mode. The text screen is automatically switched on when a system message is displayed, or by PRINTing CHR\$(6).

CHR\$(22)

This code can be used with parameters 0 or 1: 'PRINT CHR\$(22)+CHR\$(n)' (where n is either 0 or 1) turns 'transparent mode' off or on respectively. With transparent mode on characters already on the screen can be overprinted.

```
PRINT CHR$(22)+CHR$(1)
```

will switch transparent mode on. To switch it off, use:

```
PRINT CHR$(22)+CHR$(0)
```

CHR\$(23)

This is used to perform an XOR, AND or OR with graphics printing.

To use XOR plotting, you have to print CHR\$(23)+CHR(1). The Amstrad will only draw (or plot) a graphic line if one does not exist at those coordinates, or if one already exists it will be removed.

To illustrate this, type in this short program:

```
5 WINDOW#0,1,40,1,6:CLS#0
10 PRINT CHR$(23)+CHR$(1)
15 PLOT 0,0
20 DRAW 640,200,3
```

When this program is run for the first time, a red line will be drawn because there is no line there. However, if you run the program again the line will be removed from the screen because there is already a line. More generally, the computer does a logical XOR of the ink numbers, and plots the actual colour resulting from this. Thus, if you XOR plot a blue line (ink colour 2) on a red line (colour 3), you get yellow (colour 1).

For AND plotting, `CHR$(23)+CHR$(2)` must be printed. This will cause the Amstrad to draw a line only if a line already exists at the chosen coordinates – i.e change nothing – this assumes you are plotting in the same colour as is already plotted (e.g. red on red).

To demonstrate this, edit line 10:

```
10 PRINT CHR$(23)+CHR$(2)
```

When you run this program there may already be a line on the screen because of the effect of running the previous program. If there is, then the display will not change. If there was no line on the screen, none will be drawn because AND will only draw a line if the points have already been set. In a similar way to XOR plotting, you can AND different colours.

For OR plotting, `CHR$(23)+CHR$(3)` is used. This will cause the Amstrad to draw a line if one is already there OR draw a line if there isn't one – i.e. it will cause it to draw a line! As before, when plotting in different colours, the inks will be ORed.

CHR\$(24)

This command swaps the PAPER and PEN colours. For example: if you are using a red pen with a white background, typing `PRINT CHR$(24)` will cause the pen to write in white on a red background.

CHR\$(25)

This command is the same as the SYMBOL command, enabling the user to define a character. there are 9 parameters to this code and all of them must be used to define the character. The first parameter is the number of the character (ASCII code) being defined; the other 8 parameters are the decimal values of each of the rows of pixels required to define a character.

Appendix six illustrates the use of the symbol command.

CHR\$(26)

This code is the same as the window command except that it only uses 4 parameters. Using CHR\$(26) only allows you to define WINDOW#0. The use of windows is covered in chapter two.

CHR\$(27)

This command has no effect at all.

CHR\$(28)

This code enables the user to set an inkpot to equal two colours; i.e. a flashing ink.

The code uses 3 parameters: the first parameter defines the number of the ink being set, the second and third parameters are the two colours that the ink pot will contain.

```
PRINT CHR$(28)+CHR$(1)+CHR$(15)+CHR$(21)
```

is an attractive combination.

CHR\$(29)

This code is very similar to the one above, setting the border to pairs of colours. There are only two parameters to this command.

CHR\$(30)

This command moves the cursor to top left hand corner of the screen, not much good in immediate mode as the 'Ready' message gets printed underneath and takes the cursor out of the corner. It is therefore only of any use in a program that needs the cursor sent to its origin.

CHR\$(31)

This code with its two parameters is equivalent to the LOCATE command and will send the cursor to a specified position on the screen. The first parameter in the statement is the column, the second parameter is the row.

Appendix 6

PART ONE

The Amstrad has one of the best built-in character sets available in today's home computer range. As well as having the complete alphabet in both upper and lower case, the Amstrad also has various unique characters like a happy and unhappy face, a bomb, a tree and a musical note. Such is the extent of the Amstrad's character set you should find there any character that you are ever likely to need. However, if you need a character not available, the Amstrad has two commands that allow you to change the design of any of the two hundred and fifty five characters the Amstrad starts up with. These two commands are 'SYMBOL' and 'SYMBOL AFTER.'

SYMBOL AFTER

If the user has decided to define a character, then it is necessary to decide which character to replace. When the Amstrad is switched on, sixteen user-definable characters are allocated to the user. These are CHR\$(240) to CHR\$(255). Characters are stored in these values at present and they can be seen by typing in the following direct entry statement:

```
FOR X=240 TO 255:PRINT CHR$(X);:NEXT X
```

Now the screen should display sixteen characters starting with CHR\$(240) and ending with CHR\$(255). Any one of these characters can be replaced with one of your own design. Sadly these sixteen characters are very useful, and you therefore might not want to change them. This is where the SYMBOL AFTER command comes in. At the moment the 'user-definable character pointer' is pointing to character 240, the default value. The symbol after command allows us to reposition the pointer to any point along the character set thus making it possible to redesign all or none of the characters. The symbol after command takes the following form:

```
SYMBOL AFTER XXX
```

Where XXX is a number in the range 0 to 255. As mentioned earlier, the default value is two hundred and forty, so the SYMBOL AFTER command for the default value would look like this:

```
SYMBOL AFTER 240
```

This tells the computer to allocate all the characters starting from 240 onwards as 'user definable'. If there are any characters already defined from 240 onwards they will not be reset as the SYMBOL AFTER command does not reset any characters, it only enables them to be re-defined. They can only be changed by using the 'SYMBOL' command; for the time being let us stick with SYMBOL AFTER.

Having decided not to change the characters from two hundred and forty onwards, one has to decide where to define ones character. You may feel that the characters from two hundred and eight to two hundred and twenty-three are the most dispensable so the user definable pointer will be moved to point to character 208. Type in the following direct entry command:

SYMBOL AFTER 208

When this is typed in and the ENTER key pressed, the computer simply displays the 'Ready' message and the cursor. No apparent change seems to have been made. Appearances are deceptive. Deep in the heart of your Amstrad the user definable character pointer has been moved from its default value of 240 to a new value of 208. This means that a total of 48 characters are now available for redesigning – all those from CHR\$(208) to CHR\$(255).

SYMBOL

Having repositioned the user definable character pointer, the next step is to define a new character. At this point it is assumed, that the reader has a slight knowledge of binary. Each character is designed in an eight by eight grid as modelled in Figure A6.1

128	64	32	16	8	4	2	1

FIGURE A6.1

Each box in each row has a potential numeric value which it takes if it is filled in. The left-most box has a value of one hundred and twenty eight in decimal and the right-most box has a decimal value of one. In Figure A6.1, none of the boxes are set (filled in) and so each row has a value of zero and the character is an empty space. Figure A6.2 shows the grid design for a capital 'S'.

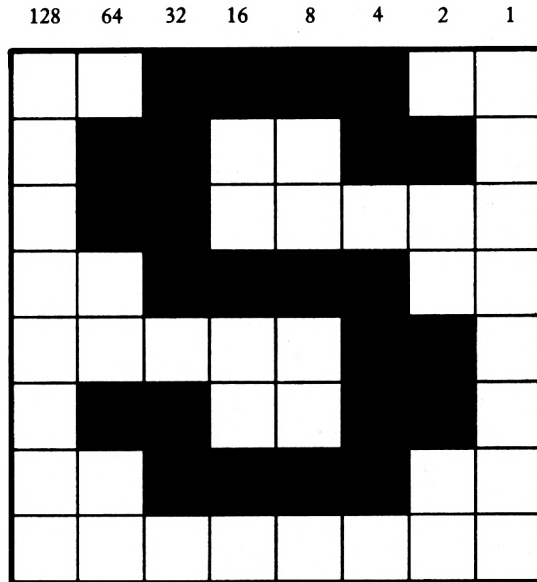


FIGURE A6.2

Each filled-in box represents a set bit so the above character can be represented in binary, using a 0 for an empty box and a 1 for a filled in box:

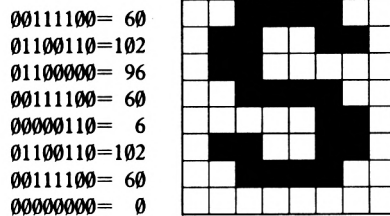


FIGURE A6.3

The figures on the right are the decimal values of each of the rows. It is these values that are used when a symbol is designed. Having got the value of each row, the next thing is to take the CHR\$ value that you want your character stored at and along with the row values complete a SYMBOL command. The symbol to be changed must be AFTER, or the number used in the SYMBOL AFTER statement. The SYMBOL AFTER command typed in earlier positioned the user definable character pointer to character two hundred and eight, so that seems as good as a character as any to change. Thus to change character two hundred and eight the following symbol command would be used.

```
SYMBOL 208,60,102,96,60,6,102,60,0
```

The first number (208) is the character that is being changed and the following eight numbers are the values of each of the rows. Sixty is the value of the top row and zero is the value of the bottom row. The above statement when typed in would change the character in 208 to an uppercase S. This point is easily tested, just type in:

```
PRINT CHR$(208)
```

On the screen will appear an upper case S. Unlike the 'S' at character eighty three this S has been put in its place by you. Having two capital S's is rather pointless so character two hundred and eight will be redesigned again:

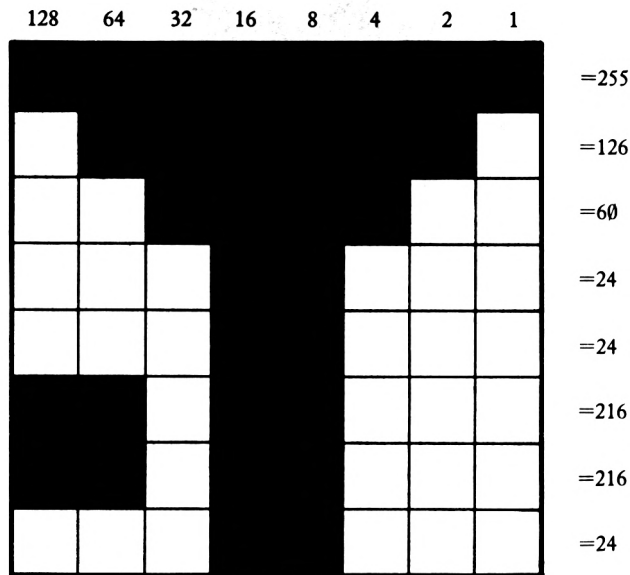


FIGURE A6.4

So, to change character 208 again, the following command can be used

```
SYMBOL 208,255,126,60,24,24,216,216,24
```

PART TWO

Function Key Definition

You may have read in the user instruction manual that you can define 32 'expansion characters' on your Amstrad. This is true, but what are these expansion characters? Partly, the answer is function keys, i.e. keys that can carry out a series of actions upon being pressed. In fact only 13 of these 32 expansion characters are function keys: when the machine is turned on these keys are those on the numeric key pad to the right of the keyboard. There are only 12 keys there though. This is true but the small enter key can be programmed as two keys. The other 19 keys will be covered later. These 13 function keys can be programmed to hold information so that on the press of the particular key the information will be displayed on the screen, and/or action taken.

Below is a list of the function keys defined when the computer is switched on.

Key on numeric pad	Function key number	Expansion tokens	How to obtain programmed information
0	0	128	Press '0'
1	1	129	Press '1'
2	2	130	Press '2'
3	3	131	Press '3'
4	4	132	Press '4'
5	5	133	Press '5'
6	6	134	Press '6'
7	7	135	Press '7'
8	8	137	Press '8'
9	9	136	Press '9'
.	10	138	Press '.'
ENTER	11	139	Press 'ENTER'
ENTER	12	140	Hold 'CTRL' and press 'ENTER'

FIGURE A6.5

As you may have found, these keys on the numeric key pad are already defined when the computer is switched on. They are defined with the values marked on the keys. However, as you can see from the table, the combination of 'CTRL' and 'ENTER' produces 'RUN' and a carriage return: this is the second function of the 'ENTER' key and is not displayed on the key. How to define the function keys (0-13) is demonstrated below.

KEY <expansion token number>,<string>

The <expansion token number> can be in the range 0-31 or in the ASCII range 128-159; both sets of values can be used to program the same expansion character - each range has corresponding values with the other as can be seen from Figure A6.5 above for those values shown.

Thus,

KEY 0,<STRING EXPRESSION>

and

KEY 128,<STRING EXPRESSION>

have the same effect. The string expression part of the command is the information you wish repeated or executed at the press of the key. The expression must be enclosed in quotes. You can add a carriage return to the string expression, in which case the string would act as a command (in a single key) plus an ENTER. To demonstrate this we shall put into function key number 0 'LIST 1-99' with a carriage return:

KEY 0,"LIST 1-99"+CHR\$(13)

The carriage return is the '+CHR\$(13)' part of the above command and is entirely optional.

An important point to remember when defining your expansion characters is that there is a limit to the number of characters that are available for string expressions. The limit is 128 characters in total for all 32 expansion characters put together. If you test this theory by putting 20 characters into each of the first 6 characters you will see that on the 6th entry the Amstrad gives you an 'Improper argument' error message, telling you that you are attempting to use more of the expansion character memory than is available.

However, some space is still available: the numeric keys are already defined when the computer is switched on, so the space used for these definitions can be reclaimed.

To do this, the following short program would suffice.

```
10 FOR X=0 TO 31
20 KEY X, ""
30 NEXT X
```

Having the expansion characters limited to particular key and with the limit on the amount of memory that can be used, is not very convenient so we use 'KEY DEF'. This allows the use of the other 19 expansion characters.

'KEY DEF' allows almost any key on the keyboard to be used as a function key. The way this is done is to define say 1 or some other of the already met expansion characters with your information and then, using KEY DEF, transfer the expansion character to a keyboard character. therefore if we wished the key 'L' to contain 'LIST' we would first define an expansion key with list.

```
KEY 0, "LIST"+CHR$(13)
```

We can then define the keyboard character:

```
KEY DEF 36,1,128
```

This has defined key 'L' with 'LIST' and a carriage return. The syntax for KEY DEF is:

```
KEY DEF <key number>,<repeat>,<key number>
```

The <key number> parameter determines which key is to be programmed with the value of another key; there is a diagram of the keyboard and the keys and their numbers on page 16 of Appendix III of the Amstrad user's manual. The <repeat> parameter's value should either be 1 or 0 as this determines whether the key should have auto-repeat or not: if it has and the key is held down, after half a second or so the value of key is continually re-displayed. If the key does not have auto repeat, then even if the key is held down, the value of the key will not be repeated until the key is released and pressed again. The <key value> parameter is the most difficult to explain as it has a number of sources. Depending on the source of the number the type of number used varies. Don't worry all shall become clear. Refer back to Figure A6.5 and you will see the expansion tokens relate to the function key numbers. If we wish to transfer a function key to an ordinary keyboard key we use this expansion token as the <key value> parameter. If we wish to transfer a value from one key, for instance capital Y, to the M key then we use the ASCII value of 'Y' or 'y', whichever you want to transfer, in <key value>. The process is the same for all keyboard characters. Obviously if you want the upper or lower case of a letter you use the respective ASCII value. The same applies to the numeric keys and their second symbols on the top of the keyboard.

You may now realise the use of the 19 spare expansion characters: you can define all 32 characters and then transfer all or some of these to keyboard characters.

INDEX

A

ABS 157
AFTER 179
AND plotting 176
arithmetic functions 157 et seq.
arrays 22, 93
ASCII 80, 163
attack 66
ATN 157

B

bad subscript 23
BASIC 9
binary files 165
bit significant 58
BIN\$ 157
BORDER 30

C

CAT 168
cassette filing 113, 163
cells, screen 10, 30
CHAIN 165
CHAIN MERGE 167
channel 58
character codes 170, 173
character redefining 179
CHR\$ 77, 173 et seq.
CINT 158
circles 42
CLG 54
CLOSEIN 134
CLOSEOUT 134
CLS 11
colours 30, 33
comments 21
conditional loops 12
continuous loops 11
COS 42, 158
CREAL 158

D

DATA 15, 113 et seq.
DATA error ('syntax') 16
DATA exhausted 16
decay 69
default colours 33
defining characters 179
DEG 158
DIM 22
dimensioning arrays 22
disc, saving to 166
DRAW 36
DRAWR 37

E

ELSE 12
END 18
ENT 71
ENV 66
end of file (EOF) 136
envelope
 tone 71
 volume 65
EOF 136
ESCAPE key 10
equilateral triangle 40
EXP 158
expansion tokens 183

F

field 115
file 115
firing 83
FIX 158
flag 47
flush 63
FOR..NEXT 12
frequency 64
function keys 182

G

GOSUB 17, 18
GOTO 11, 18

H

Hertz 64
HEX\$ 159
hold 62
home 29

I

IF...THEN 11
improper argument 78
INK 31
INKEY\$ 21
inkpots 31
INPUT 9, 20
INPUT# 135
INT 159
integer variables 116
isosceles triangle 39

K

keyboard buffer 21

L

LEFT\$ 23
LEN 26
length
 of input (255) 20
 of note 64
LET 9
line does not exist 17
LINE INPUT 20
LIST# 50, 169
literal string 10
LOCATE 10
LOG 159
LOG10 159
look-up table 44
loops 11
 conditional 12
 continuous 11
 unconditional 11

M

MAX 159
menu driven 115
MERGE 166
MID\$ 24
MIN 160
MODE 29
MOVE 37
MOVER 37

N

names of variables 9
NEXT 12
noise period 74
numeric variable 10

O

ON...GOSUB 18
ON...GOTO 17
OPENIN 134
OPENOUT 134
OR plotting 176
ORIGIN 53

P

PAPER 32
PEN 33
PI() 42, 160
picture elements 33
pixels 33
PLOT 34
PLOTTR 36
PRINT 9
PRINT# 50, 135, 169
prompts 10
protecting programs 163
pseudo random 27

R

RAD 160
RANDOMIZE 27
READ...DATA 15
record 115
rectangle 38
redefining characters 179
relative 36, 37
RELEASE 62
REM 21
RENDEZVOUS 60
resetting envelopes 74
resolution 33
RESTORE 16
RETURN 17
right-angled triangle 38
RIGHT\$ 24
RND 26, 160
ROUND 160
RUN" 165

S

SAVE 163, 165
screen cells 10, 30
scroll 80
seed 27
setting pixels 34
sequential filing 113
SGN 161
SIN 42, 161
SOUND 57
sound queue 63
SQR 161
square 38
STEP 13
STOP 17
streams 49
strings 23
subroutines 17
subscripted variables 22, 93
sustain 67, 72
SYMBOL 180
SYMBOL AFTER 179
syntax error 16

T

TAG 55
TAGOFF 56
TAN 161
TEST 45
TESTR 46
THEN 11
TIME 101
triangle 38, 39, 40
tone 64
 envelope 71

U

unconditional loops 11
unexpected RETURN 17
unexpected WEND 15
UNT 161
UPPER\$ 97
user friendly 115

V

variable 9
 arrays 22
 integer 116
 names 9
 numeric 10
 subscripted 22, 93
video game 77 et seq.
voices 58
volume 65
 envelope 65

W

WEND 14
WEND missing 15
WHILE...WEND 14
windows 48
WINDOW SWAP 52

X

XOR plotting 176

**Other books for the Amstrad
by Glentop Publishers.**

Watson's Workbook Book 1

“Starting BASIC”

by

S. Gray & E. Maddix

ISBN 0 907792 39 1

Dr Watson Series

“Amstrad Assembly Language Course”

by

Tim Herbertson

ISBN 0 907792 41 3

A Dr Watson Logo Book

“Using DR LOGO on the Amstrad”

by

Martin Sims

ISBN 0 907792 56 1

AMSTRAD CPC 464

Sound, Graphics & Data Handling

This book is aimed at the reader who already knows a little BASIC, perhaps through reading book one of this series, 'Starting Basic'. It will also be of use to those who have used BASIC on another computer and wish to know how Amstrad BASIC differs.

The book will show you how to get the Amstrad to produce strange noises and musical sounds. It then explains, with numerous examples, how to move characters around the screen, the use of colour and how to divide the screen into independent sections. The reader is then shown step-by-step how to use all this knowledge to develop a full-blown BASIC video game with both animated graphics and sound.

Computers can also be used for home management, and a general-purpose database program is developed in detail, showing by way of example how it can be used to keep a computerised address book.

Glentop Publishers Ltd,
Standfast House, Bath Place,
High Street, Barnet,
Herts, EN5 1ED

ISBN 0-907792-40-5

£5.95

ISBN 0-907792-40-5



9 780907 792406

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.