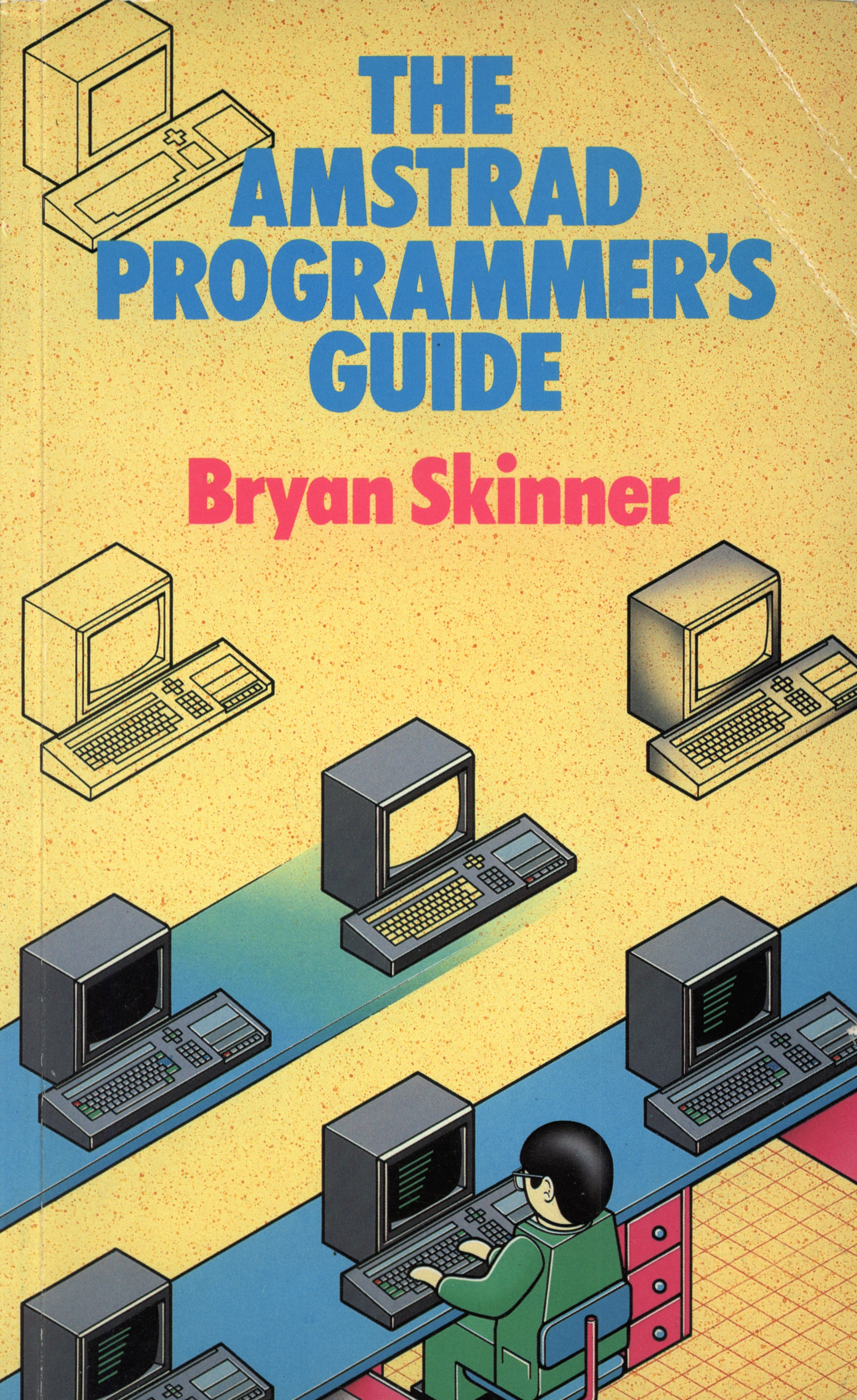

THE
AMSTRAD
PROGRAMMER'S
GUIDE
Bryan Skinner

THE AMSTRAD PROGRAMMER'S GUIDE

The Amstrad Programmer's Guide

Bryan Skinner

Duckworth

First published in 1985 by
Gerald Duckworth & Co. Ltd.
The Old Piano Factory
43 Gloucester Crescent, London NW1

© 1985 by Bryan Skinner

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publisher.

ISBN 0 7156 1984 5

British Library Cataloguing in Publication Data

Skinner, Bryan
Amstrad programmer's guide.—(Duckworth home computing)
1. Amstrad computers—Programming
I. Title
001.64'2 QA76.8.A4

ISBN 0-7156-1984-5

Photoset in North Wales by
Derek Doyle & Associates, Mold, Clwyd.
Printed in Great Britain by
Redwood Burn Ltd, Trowbridge

Contents

Introduction 11

1 Getting Started 13
Immediate and deferred modes 13
Basic words 13
PRINT 14
  Print separators 14
Screen modes 15
LOCATE 18
Programming 18
LIST and RENUM 19
Editing 20
  Copy editing 20
  The control key 21
AUTO 22
NEW 22

2 Variables and Loops 23
Variables 23
  CHR$ 24
  ASC 25
FOR ... NEXT loops 25
  Nested loops 27
Redefining characters 28
Numeric and string variables 31
Integers and integer variables 32
DEFSTR, DEFINT and DEFREAL 33

3 Strings and Keys 34
INPUT 34
IF, THEN and GOTO 35
ELSE 36
INKEY$ 37
String handling 38

LEFT$ 38
RIGHT$ 39
MID$ 39
UPPER$ and LOWER$ 39
STRING$ 40
LEN 40
INSTR 40
Examples 41
Concatenation 42
SPACE$ 43
Data type conversion 43
STR$ 43
VAL 44
Some number-handling operations 45
RND 45
INT 45
FIX 45
ROUND 45
REM statements 46
ZONE 47
WIDTH 47
Examples 47
Keys 50
KEY 50
KEY DEF 51
SPEED KEY 52
Disabling ENTER 53
INKEY 53

4 Subroutines, Arrays and System Functions 55
Subroutines 55
WHILE and WEND 56
ON GOTO/GOSUB 57
Functions 58
DEF FN 58
Arrays 59
Multi-dimensional arrays 62
Animals game 65
System functions 68
ERASE 68
READ and DATA 69
RESTORE 72
TIME 73

Trigonometric functions 74
Memory 75
PEEK and POKE 75
Database program 76

5 Designing a Game 81
Coding the game 81
A border 81
The characters 82
PEEKing the screen 83
Movement 83
Changing direction 85
Generating random numbers 88
Decrementing the numbers 90
Scoring 91
The complete listing 92

6 Numbers and Logic 98
Notation 98
Decimal 98
Binary 99
Hexadecimal 100
Boolean logic 101
Truth and falsehood 103
Bit testing 104
Bit mapping 105
Number-handling functions 108
MOD 109
MIN and MAX 110
ABS 110
SGN 110
PI 110
LOG, EXP and LOG10 111
CINT and CREAL 111
UNT 112

7 Machine Code 113
Registers 113
Machine code instructions 113
Op-codes 113
Mnemonics 114
Mixing Basic and machine code 114
Machine code routines 115

Scrolling the screen 115
A more complex version 117
PEEKing text on the screen 119
Filling boxes 121
Stripy inks 123
ROM calls 125

8 Introduction to Graphics 127
Colours 127
Resolution 128
Border colour 129
Background and foreground colours 129
INK 129
PAPER 131
PEN 131
Graphics 132
Co-ordinates 132
POS and VPOS 132
MOVE 133
MOVER 133
PLOT 134
PLOTR 134
DRAW 134
DRAWR 134
TEST and TESTR 135
Erasing pixels and lines 135
XPOS and YPOS 136
Examples 136
Squares 136
Graphics subroutines 138
Circles 139
Filling shapes 140
Speeding up graphics 141
ORIGIN 141
Ellipses 145
Spirals 145

9 Advanced Text and Graphics 147
Linking text and graphics 147
TAG and TAGOFF 147
Screen write operations 150
Curvestitch program 151
Moire program 153

Non-printing control codes 154
Extra-large characters 154
Bouncing ball routine 157
Windows 160
Setting up windows 160
Windows and colours 161
Using windows 162
Window overlaps 163
Contours 163

10 Sound 166
Basic sound 166
Channels and duration 167
Sound in full 169
Frequency 170
Notes 170
Chords 172
Tone envelopes 173
Designing a tone envelope 176
Volume envelopes 178
Experimenting with volume envelopes 179
Noise period 181
Channels, rendezvous and holds 182
Sound queue 183
RELEASE 184
ON SQ ... GOSUB 185
Ring modulation 185

11 The Cassette System 187
Loading programs 187
Saving programs 187
SPEED WRITE 188
Saving options 188
Protecting programs: the ,P option 188
Files 189
ASCII format : the ,A option 189
Recording (saving) data 190
WRITE 191
Retrieving data 192
EOF 192
LINE INPUT 193
Saving blocks of memory 193
Saving characters 195

Saving machine code 195
Cataloguing files 196
Chaining programs 197
CHAIN 197
CHAIN MERGE 198
ROM calls 199
Cassette error messages 199

12 Interrupts 201
Timers 201
EVERY 202
Disabling and enabling interrupts 203
Timer priority 205
AFTER 207
ON BREAK GOSUB 208
ON ERROR GOTO 210
RESUME 211
ERR and ERL 211
REMAIN 212

Introduction

The Amstrad CPC464 has a well-designed, fast and powerful version of Basic. To beginners it can seem rather complex, and those used to other dialects of the language may find it hard to adapt to the Amstrad. This book is aimed at those who wish to extend their understanding and skills in Basic programming on the Amstrad. As well as explaining each Basic command or instruction, the book outlines useful programming techniques which make complex programs easier to write and may also save time and memory.

Many listings are included. Some serve as examples or demonstrations of points raised in the text, while others are intended to form a framework on which you can build to create your own personalised programs.

Chapters 1-3 introduce the fundamental concepts of programming on the Amstrad. Arrays and programming techniques for making the most of them are dealt with in Chapter 4, which ends with a listing for a simple database. Chapter 5 shows how the Basic commands so far described can be combined to create a game. Chapter 6 explains the relationship between the decimal, binary and hexadecimal number systems. It also covers Boolean operators and shows how information can be encoded by bit-mapping. Chapter 7 introduces machine code and assembly language programming and includes several useful routines. Chapters 8 and 9 show how the Amstrad graphics system operates, and Chapter 10 deals with sound generation, describing how to transcribe music and demonstrating some of the Amstrad's unique facilities. Chapter 11 focusses on the cassette unit for recording programs, data or screen pictures, and Chapter 12 discusses Basic interrupts and suggests ways in which they can be used to improve your programs.

B.S.

1
Getting Started

Immediate and deferred modes

You can instruct your Amstrad to calculate, display words, draw lines and so on by entering commands directly at the keyboard; this is called 'immediate' or 'direct' mode. Alternatively, you can type a series of such commands (a program), tell the computer to carry them out and sit back and wait until it has finished. This is called 'program' or 'deferred' mode. Most of the operations you make the computer perform will be part of a program, because this method allows you to recall the instructions so you can change them, or record them on cassette tape for future use. Neither of these is possible with commands entered in direct mode. More important, programs allow complex series of operations to be performed, which is impossible in direct mode. There are some commands which cannot be used in direct mode, but it is useful for checking the progress of a program, as we will see later. The main use of direct mode is for entering and altering the commands which make up a program.

Basic words

Your Amstrad 'understands' a limited set of instructions or commands which make up the programming language called Basic. These are words like PRINT, LET, LOCATE, and MODE. Using simple words like these you can build up a series of instructions (a program) which are stored in the computer's memory and which can be used to make it perform intricate operations very quickly.

Sometimes you may mis-spell a word – for example, you may type PRNIT and because the computer cannot understand this it will display an 'error message', in this case 'Syntax error'. There are a number of error messages that the computer may produce when you try to make it perform impossible tasks, such as dividing

by zero, and these will be discussed in later chapters. If at this stage you come across a syntax error, simply retype the command.

PRINT

Whatever sort of program you write on your Amstrad, you'll want to see something on the screen. You may want to display numbers, words or some of the other available symbols. To begin with, we'll concentrate on the simplest method of controlling the display: making 'characters' (letters, numbers or other symbols) appear on the screen.

After each command has been entered, you *must* press the large blue key marked ENTER, which tells the computer that you want it to act on what you have just typed. We'll show this as ‹ENTER› for a while, until you get into the habit. Although command words are shown here in capital letters, you can use lower case if you want, as the Amstrad makes no distinction. To switch between ('toggle') upper and lower case, press the key marked CAPS.

The easiest way to make characters appear is to use the command PRINT. For example, the direct command 'PRINT 7 ‹ENTER›' will produce the number seven at the left-hand side of the screen. You can use your machine like a calculator: type 'PRINT 7 * 9 ‹ENTER›' and the number 63 appears. Notice that Basic uses the asterisk instead of 'x' to mean 'multiplied by'; similarly, '/' means 'divided by'. If you type 'PRINT 18 / 6 ‹ENTER›' the number 3 will appear at the left of the display. It's not necessary to put spaces between the numbers and the symbols; this has been done here for clarity. However, you must leave a space after command words: 'PRINT7' is 'illegal' and will produce the 'Syntax error' message. You can abbreviate PRINT to the question mark (?), which will save time.

Print separators

You'll probably have noticed that after the computer has obeyed your command it displays the message 'Ok' underneath the result of the operation. What if you wanted the results of the calculations 7 * 9 and 18 / 6 on the same line? To do this you have to understand that after printing something on the screen,

the computer moves the cursor to the beginning of the next line down (the cursor is the steady square which indicates where the next item will be displayed). However, you can tell your Amstrad not to do this by using the semi-colon. Try 'PRINT 7 * 9 ; 18 / 6 ‹ENTER›'. You should find that the numbers 63 and 3 appear on the same line. The semi-colon is called a 'print separator' because it is used to separate items in a single PRINT command.

The comma is another print separator and is used to help tabulate results. If you want words to appear in neat columns you can use the comma rather as you might use the tab key on a typewriter. The comma tells the computer to move the cursor to the next screen 'zone' before printing the next item. The normal zone width is 13 character places or columns, but you can change this if you wish by using the ZONE command (e.g. ZONE 10). To see how the comma works, try changing the semi-colon in the above example to a comma, or even two commas (PRINT 7 * 9,,18/6).

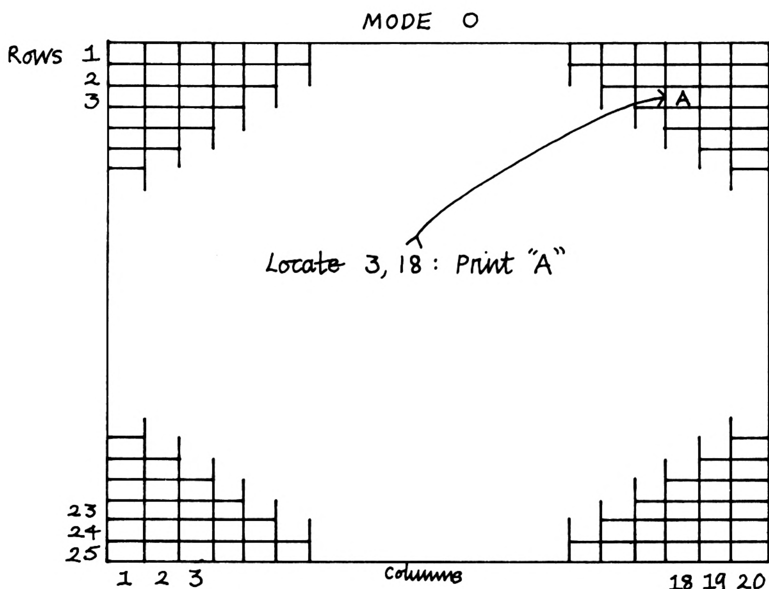
Of course, most of the time you won't want to have just numbers displayed, you'll want to see letters and words as well. To do this you need to use the quotation mark (") on each side of the item you want displayed. Try 'PRINT "AMSTRAD CPC464" ‹ENTER›'. The string of letters between the quote marks is displayed, again starting at the left-hand side of the next line down.

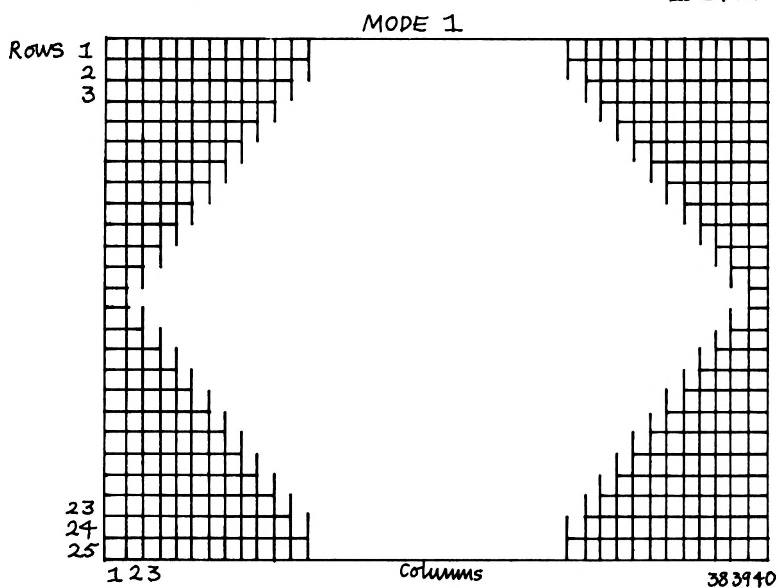
You can mix letters and numbers in a PRINT command by using a print separator. Try 'PRINT "One hundred divided by nine =";100 / 9 ‹ENTER›'. You can put numbers inside quotation marks: for example, 'PRINT "376 – 67" ‹ENTER›'. Whatever you put inside quotes in a PRINT statement (numbers, letters, or other symbols) will appear on the screen.

One final point for the moment about PRINT: positive numbers are always printed with a leading space. This is to make them line up with negative numbers, which of course have a leading minus sign. Dealing with this, which can be a problem, will be dealt with later on.

Screen modes

The Amstrad can display characters on the screen in a number of sizes; it has three different screen 'modes'. These are put into effect with the command MODE n, where n is 0, 1 or 2. MODE 0 is the lowest 'resolution'. If you type 'MODE 0 ‹ENTER›', the screen


MODE 0
Rows 1
2
3
A
Locate 3, 18 : Print "A"
23
24
25
1 2 3
columns
18 19 20


MODE 1
Rows 1
2
3
23
24
25
1 2 3
columns
38 39 40

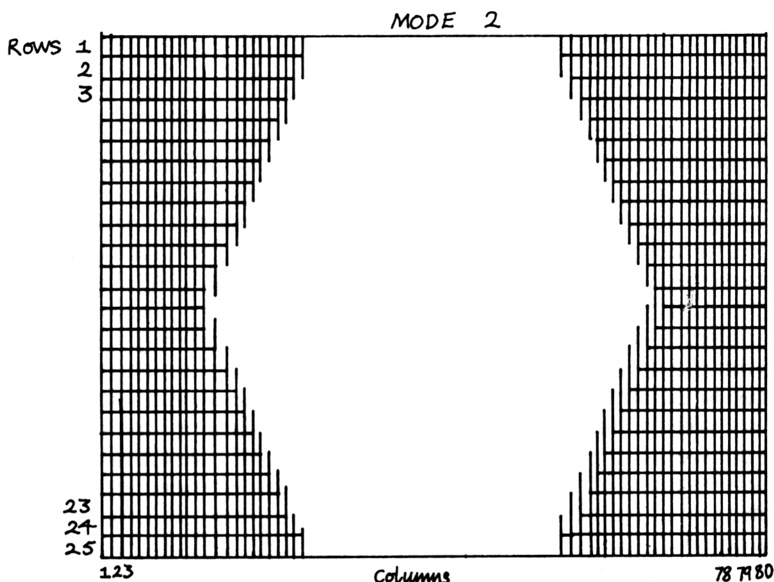


Figure 1.1. Text resolution in MODES 0,1 and 2

clears and then the message 'Ok' appears. Note that the letters making up the 'Ok' message are very large indeed. Try 'MODE 1 ‹ENTER›', then 'MODE 2 ‹ENTER›'. Notice how these commands give successively smaller characters. What is happening here is that the machine is allocating different numbers of characters per row or line in the different modes. In MODE 0, there are 20 characters per line; in MODE 1 there are 40 such character positions per line; while MODE 2 gives what is called an 80 column display. MODE 1 is the 'default' screen mode, i.e. the mode which the computer automatically adopts when it is switched on. In all modes there are 25 lines, but the penalty of higher resolution is that fewer colours are available.

MODE	Columns	Colours
0	20	16
1	40	4
2	80	2

LOCATE

When you write a program to display information, it's very important to control exactly where the information appears. To do this you can use the word LOCATE. This word must be followed by two numbers, which we'll call c and r (for column and row), and the numbers must be separated by a comma – this has nothing to do with using the comma as a print separator. LOCATE c,r places the cursor at the position specified by the two numbers. The easiest way to imagine this is to think in terms of graph paper, with the co-ordinates 1,1 referring to the character position at the top left of the screen. The column number is always given first, so to make the letter 'Z' appear at the tenth column of the third row you would type 'LOCATE 10,3 : PRINT "Z" ‹ENTER›'. In this example we've also introduced the fact that you can enter more than one command at a time, provided that they are separated by a colon.

When using LOCATE you must take into account which mode the screen is in, or you'll get some odd results. Obviously there will be problems if you tell the computer to LOCATE the cursor at column 75 of line 1 if it's in MODE 0, which only has 20 columns per line.

Programming

So far all the commands you've entered have been in direct mode, i.e. the computer did what you told it as soon as you pressed ‹ENTER›. Program or deferred mode allows you to place a series of commands into the computer's memory which it will act on ('execute') in order. You can also record the sequence of instructions on cassette tape so you can re-use the program without having to type it in all over again. The instructions in a program will be the same as the commands you've just been entering, except that each command will begin with a number. You can turn some of the commands you've entered so far into a program quite easily. For example, type in exactly what follows:

```
1 PRINT 7<ENTER>
2 PRINT 7 * 9<ENTER>
4 PRINT 7 * 9 ; 18 / 6<ENTER>
3 PRINT 18 / 6<ENTER>
5 PRINT "AMSTRAD CPC464"<ENTER>
6 PRINT "One hundred divided by nine ="; 100 /
9<ENTER>
7 PRINT "376 - 67 =";376 - 67<ENTER>
8 LOCATE 10,3:PRINT "Z"<ENTER>
```

To see this simple program work, type 'CLS:RUN ‹ENTER›'. These two commands will Clear the Screen (CLS), then make the computer RUN through the instructions one by one, until there are no more. Then it will display the 'Ok' message, telling you that you're back in control – direct mode. Note that the order in which you enter program lines doesn't matter (lines 3 and 4 are the wrong way round), the computer will automatically sort them into ascending line numbers.

LIST and RENUM

There are a number of commands you can now use to look at your program, and change it if you want to. Try typing 'LIST ‹ENTER›' – your program appears on the screen just as you typed it in. Try 'RENUM ‹ENTER›', then 'LIST ‹ENTER›'. You'll see that the computer has RENUMbered the lines of your program from 1, 2, 3, 4, etc. to 10, 20, 30, 40, and so on. It's usual to use line numbers in increments of 10 when entering program lines because this makes it easy to insert other lines, say 15, 22 or 23, if you need to.

You can use LIST in a number of ways. If you only want to see a part of your program, say lines 10 to 40, you can add these line numbers to the LIST command, as in 'LIST 10-40'. If you wanted to see all the lines up to and including line 100, the instruction would be 'LIST -100'. If you're listing a long program you'll find that it 'scrolls' off the top of the screen too fast to read. You can pause a listing by pressing the escape key once. Any other key thereafter will continue the list, while pressing escape again will stop the listing with the message 'Break'.

Editing

There are a number of ways to alter (edit) program lines. Obviously you could retype a line together with any changes, but this can be tedious if it's a long line. The simplest editing method is to type the command EDIT n ‹ENTER›, where n is the line number you want to change. If you try to edit a non-existent line, the Amstrad will tell you with the error message, 'Line does not exist'. When you edit a line, a copy of the line appears on the screen with the cursor over the first character, the first digit of the line number. You can now use the left and right arrow keys to move the cursor over the line. If you type other characters, they'll be inserted into the line and any characters to the right of the cursor will be moved to the right. Try 'EDIT 10 ‹ENTER›', then move the cursor to the left of the 7, and press the key marked 6. The number 7 becomes the number 67. Pressing ‹ENTER› will make the change permanent. To check this, type 'LIST ‹ENTER›' to see the new version of the program.

You can also use the key marked DEL to delete characters to the left of the cursor and the CLR key to delete the character under the cursor. In either case, characters to the right of the line will be dragged to the left to fill up the space. You can press ‹ENTER› as soon as all the changes you want to make are complete: there's no need to move the cursor to the end of the line. With a bit of practice you'll find that these methods allow you to make rapid changes to any of the lines in a program.

Copy editing

The other method of making changes to program lines is to use the arrow keys, one of the shift keys and the green COPY key in the centre of the cursor key cluster. This process is called copy editing. Let's assume that you want to alter line 20 of the program, the one which reads '20 PRINT 7 * 9', and you want it to read '20 PRINT "Seven times nine equals";7 * 9'. First type 'CLS:LIST ‹ENTER›' to clear the screen and list the program lines. Now hold down one of the shift keys and press the up arrow until it's on the 2 of 20. If you overshoot, press the down arrow (still keeping the shift key pressed) until the cursor is where you want it. You can either tap the arrow key to move up one line at a time,

or keep it depressed which makes the key 'repeat'. Now release the shift key. Pressing the copy key will move the 'copy cursor' over the line in the list and produce a copy of the characters of the line at the 'real' cursor line underneath the program. Copy the digits two and zero and the space after the zero, then release the copy key and type ' "Seven times nine equals";'. Now press the copy key until the rest of the original line appears on the new line 20 at the bottom of the program listing. When this is done, you can press ‹ENTER›, and the new version of the line will replace the old. Again you should check this by typing 'CLS:LIST ‹ENTER›'.

If you get into a mess when copy editing, press the key marked ESC (short for ESCape); this leaves the original line intact. You can now type 'CLS:LIST ‹ENTER›' to start editing again. As with EDIT, the best way to get to grips with this method is plenty of practice. You should soon find that you become quite skilled at mixing the methods and using them to make changes quickly. You can use copy editing to copy parts of other lines into new lines, making the task of replicating sections of your program quick and easy.

The control key

There are two other cursor movements available which move the cursor to the start or end of a line. After typing 'EDIT n ‹ENTER›', holding down the CTRL (Control) key and pressing the left arrow or the up arrow moves the cursor to the start of the line, while CTRL and the right arrow or the down arrow will move the cursor to the end of that line. In copy editing, only CTRL/left arrow and CTRL/right arrow have any effect, both on the copy of the line.

After entering the EDIT command, pressing the CTRL key together with the TAB key will switch between two editing modes, 'overstrike' and 'insert'. Overstrike means that characters under the cursor are overwritten by characters typed at the keyboard. Insert mode allows characters to be inserted into the current line. You can switch between the two modes at any point with the CTRL/TAB combination.

During any editing process the Amstrad will 'beep' if you try to move the cursor to an 'illegal' position.

AUTO

When you begin typing program lines into your Amstrad you'll probably find that your most common mistake is to forget to put a line number at the front of the line, so you may get an error message, or the 'Ok' message, even though the line hasn't been entered into the program. One way to avoid this is to use the AUTO command. This will automatically produce the next program line number, so you can type in each program line, press ‹ENTER› and not have to worry about numbering the lines. You can also tell the Amstrad where you want to start, so if you've already entered lines 10 to 100, you could add to the program with the direct command 'AUTO 110'.

To get out of AUTO mode, press ‹ENTER› followed by the escape key.

NEW

To clear out the Amstrad's memory, type 'NEW ‹ENTER›'. The NEW command erases any program in the Amstrad's memory. Use it with care, because there's no simple way to get a NEWed program back.

2

Variables and Loops

Variables

One of the most powerful aspects of a programming language like Basic is that it enables you to store numbers which can be referred to by letters or words. To see what this means, type as a direct command, 'LET A = 100 ‹ENTER›'. Now type 'PRINT A ‹ENTER›'. The number 100 should appear. This is because the computer has stored the number 100 somewhere, and labelled it as 'belonging' to the 'variable name' A. It's important to recognise that this has nothing whatever to do with the letter A itself, you could just as well have used the letter B or X, for example.

You can also make the computer perform operations on the values associated with or assigned to variables stored in its memory. Type 'LET A = A + 10 ‹ENTER›'. Now type 'PRINT A ‹ENTER›'. You'll see that the computer has 'updated' the value associated with the variable A to 110; it has looked for and found the original value of the variable A (100), added 10 to this to make 110, and stored this new value under the name A.

Some versions of Basic only take into account the first two letters of a 'variable' name, so for them DIVISOR would be the same as DIVIDEND. Fortunately for the Amstrad Basic programmer, you can use variable names up to forty characters long. The only proviso is that the first character must not be a number. This 'long variable name' facility makes programming much easier when it comes to doing mathematical operations such as calculating profit and loss or keeping track of the score in a game.

Reserved words, commands and other instructions, like LET, DEF, etc., must not be used as variable names – that's why they're called reserved words. However, whereas many micros won't even let you use them as part of variable names, the Amstrad is very tolerant. You could use 'LETTER' or 'DEFINE$' without problems.

The following program shows how you can use variable names

to make your programs easier to understand, an important factor when you're trying to make sense of a program you wrote several months ago.

```
10 LET GROSSINCOME = 500
20 LET TAXRATE = 30/100
30 LET TAX = GROSSINCOME * TAXRATE
40 LET NETINCOME = GROSSINCOME - TAX
50 PRINT "Gross income:";GROSSINCOME
60 PRINT "Tax rate:";TAXRATE * 100;"%"
70 PRINT "Tax paid:";TAX
80 PRINT "Net Income is:";NETINCOME
```

The word LET is optional, i.e. line 30 could read '30 TAX = GROSSINCOME * TAXRATE'. When you're first starting to program, it can be very confusing to read lines like '100 value = value * 2' – how can a number be equal to the same number times two? The word LET reminds you that the equals sign can be used in Basic to give a variable a value: '100 LET value = value *2' makes more sense, but as you become used to the idea you can leave out LET.

In the examples of numeric variables in the program above, the Amstrad assumes that we are dealing with what are known as 'real' numbers. The largest number the Amstrad can deal with is 1.78E+38; the smallest (apart from zero) is 2.9E-39. (This method of notation is called 'exponential' or 'scientific'. 1.78E+38 means 1.78×10 to the power 38; 2.9E-39 means 2.9 divided by 10 to the power 39.) If the result of a calculation is greater than 1.78E+38, or an attempt is made to assign a value above this limit, the error message 'Overflow' will occur.

When a variable name is first used, providing that no value has been assigned to it, the Amstrad assumes it to have a value of zero. Thus the program line, '1 PRINT VALUE' will produce 0.

CHR$

The 'character set' is the collection of characters comprising letters, numbers, mathematical symbols and many others. You can make the computer display the letter 'A' with the command PRINT "A".

Another way to produce a character shape is to use the built-in function CHR$, pronounced 'character string'. You have to give

('pass') this function a number ('parameter' or 'argument') in the range 32 to 255. For example, 'PRINT CHR$(66) ‹ENTER›' will cause the computer to display the capital letter 'B'. Each character has a number associated with it. 'A' is 65, 'B' is 66, 'C' is 67, and so on, to 'Z' which is 90. Numbers start at 48 with zero and end at 57 with nine, while the lower case alphabet runs from 97 to 122. The numbers between 32 and 127 are defined as the set of 'ASCII' codes. ASCII stands for American Standard Code for Information Interchange and is useful because it provides a standard code which allows data to be transferred between one computer and another. There are a number of other predefined symbols in your Amstrad (between ASCII codes 0 and 31 and 127 to 255), some of which you may find useful in your programs. Strictly speaking, these are not part of the conventional ASCII set, but are specific to the Amstrad. Try 'PRINT CHR$ (249) ‹ENTER›', for example. A full list of these appears in the Amstrad manual, Appendix 3, pp. 2-13.

ASC

This is a useful function which reverses that of CHR$. It is named after the first three letters of the acronym ASCII and is used to find out the ASCII code for a character. Try 'PRINT ASC("%") ‹ENTER›', for example. The difference between ASC and CHR$ is that CHR$ produces or 'returns' a character, while ASC returns a number.

While their value may not be immediately apparent, ASC and CHR$ are very useful functions whose use will be described in more detail in later chapters.

FOR...NEXT loops

To see which codes produce which symbols you could type in a series of commands like 'PRINT CHR$(32) ‹ENTER› PRINT CHR$(33) ‹ENTER›', and so on. This would be very tedious indeed. Making these commands into a program doesn't help much either:

```
1Ø PRINT CHR$(32) <ENTER>
2Ø PRINT CHR$(33) <ENTER>
RUN <ENTER>
```

This requires even more typing.

What we want is for the computer to show us all the characters associated with the numbers 32 to 255. To do this we can make use of a 'control structure' called a loop. The one we're going to use tells the Amstrad to perform the same action a certain number of times. It takes the general form:

```
FOR COUNT = (START) TO (FINISH) STEP n
(do operation)
NEXT (COUNT)
```

COUNT, START and FINISH are numeric variables, n may be any number, and the operation may involve a number of program lines.

This sort of loop is often referred to as a FOR...NEXT loop, for obvious reasons. In this particular case we want to set the value of START to 32, that of FINISH to 255 and that of STEP to 1. The program comprises three lines:

```
10 FOR COUNT = 32 TO 255 STEP 1
20 PRINT CHR$ (COUNT)
30 NEXT COUNT
```

This will first set the value of the variable COUNT to 32, then display the ASCII symbol of that code number (a space). When the NEXT COUNT instruction is encountered in line 30, the computer adds the value following STEP to COUNT, jumps back to line 10, checks to see if the value associated with COUNT is greater than that of the variable FINISH (255), and if not it repeats the operation, i.e. it executes lines 20 and 30 again. This cycle will repeat, with the variable COUNT taking on the values 32, 33, 34 and so on up to 256, which is greater than the value of FINISH so the program will end. On each 'pass' through the loop, the character-shape of each ASCII code will be displayed by line 20.

It would be more useful to see which ASCII code was being used, so add a new line, '15 PRINT COUNT;'. This will display the value of the loop counter or index. The shape will be immediately next to the number, making it difficult to distinguish, so alter the new line to '15 PRINT COUNT;" ";' which inserts a space between the number and the character. You could also insert a PRINT statement, '16 PRINT', to make each number/shape pair appear on alternate lines for clarity. PRINT on its own moves the cursor to the beginning of the next line down.

To pause a FOR...NEXT loop during its execution without stopping it completely, press ESC once. Pressing any other key will now resume the program, and pressing ESC again will stop the program with the message 'Break in line XX', where XX is the number of the line which was being processed or executed when you interrupted the computer.

FOR...NEXT loops may be made to step backwards through a range of numbers by setting the value of START greater than that of FINISH, and making STEP negative. For example, we could step backwards through the ASCII set using the following fragment:

```
10 FOR COUNT = 255 TO 1 STEP -1
20 PRINT COUNT;" ";CHR$(COUNT)
30 NEXT COUNT
```

Moreover, the value associated with STEP may be fractional. To see this, type:

```
10 FOR FRACT = 0 TO 10 STEP 0.57
20 PRINT FRACT
30 NEXT
```

Note that in this example we have not used NEXT FRACT in line 30. It's not necessary to 'declare' the loop counter variable name after NEXT, but it can help to make programs easier to understand. Omitting STEP makes the Amstrad assume a value of one, so our first example could have been written:

```
10 FOR COUNT = 32 TO 255
20 PRINT CHR$ (COUNT)
30 NEXT
```

Nested loops

It is possible to use one FOR...NEXT loop inside another; this is referred to as 'nesting'. You can use this technique to perform simple operations such as displaying 'times tables'.

```
10 FOR OUTER = 1 TO 12
20 FOR INNER = 1 TO 12
30 PRINT INNER;" times";OUTER;" =";INNER*OUTER
40 NEXT INNER
50 NEXT OUTER
```

While it's difficult to grasp the concept of nested loops at first, this example shows what happens quite clearly. For every single pass through the outer loop, the inner loop is executed 12 times. In this example, the outer loop counter will first be set to 1, then the inner loop counter will cycle through values of 1 to 12. When the inner loop counter exceeds 12, the NEXT OUTER command will set the outer loop counter to 2, and the inner loop counter will again cycle through 1 to 12. This will repeat until the value of the outer loop counter exceeds 12: line 30 will be executed 144 times.

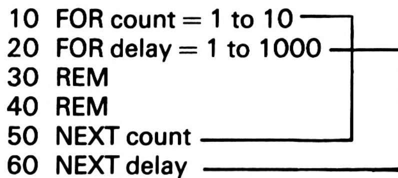


Figure 2.1. Illegal FOR...NEXT loop crossing

When using nested loops you must make sure that the loops don't cross over. A loop structure such as the one shown in Figure 2.1 will cause the error message 'Unexpected NEXT', because the computer gets confused between inner and outer loops. This problem can be avoided by omitting loop counter variable names after NEXT – letting the computer keep track of the correct order itself.

Redefining characters

Although the Amstrad has a number of built-in character shapes, you may want to define your own. The Amstrad provides a simple means of doing this. You can redefine any of the symbols as any shape that you can make from an eight row by eight column matrix. If you want to redefine a character you should begin by drawing the grid as shown in Figure 2.2. In this case we will show you how to redefine the exclamation mark as a square outline with a diagonal line through it. The next step is to shade in the cells of the matrix which you want to be 'set', i.e. appear on the screen (Figure 2.3). Next you have to work out eight numbers, one for each row. Each number is the sum of the cells 'set' in the various columns. Each column has a number associated with it:

Figure 2.2. The character matrix

Figure 2.3. Designing a character

from right to left these are 1, 2, 4, 8, 16, 32, 64 and 128. For each row, work out the value of the set cells, and add these values together (see Figure 2.4).

The next stage is to decide which symbol you're going to redefine. In this case we have decided on the exclamation mark, whose ASCII code is 33. Remember, an easy way to find this out is 'PRINT ASC("!")'. To redefine a character you have to issue two instructions. The first of these is SYMBOL AFTER n, which tells the computer that you're about to redefine a character whose ASCII code is greater than the value of n. In our example you could use SYMBOL AFTER 31 or SYMBOL AFTER 32. The command to redefine the exclamation mark as we want it is 'SYMBOL 33, 255, 193, 161, 145, 137, 133, 131, 255'. The first number, 33, is the ASCII code of the character to be changed. The eight numbers which follow are the sums of the column values of each cell in each row (Figure 2.4). Once you've redefined the character, only new appearances of that shape (produced by 'PRINT CHR$(33)') will have the new pattern: existing ones will not change. To reset the exclamation mark or any other redefined character back to its original shape, you must issue a SYMBOL AFTER n command, where n is any positive integer.

bit positions	7	6	5	4	3	2	1	0	
decimal values	128	64	32	16	8	4	2	1	sum of 'set' cells
Rows 1	128	64	32	16	8	4	2	1	255
2	128	64						1	193
3	128		32					1	161
4	128			16				1	145
5	128				8			1	137
6	128					4		1	133
7	128						2	1	131
8	128	64	32	16	8	4	2	1	255

Figure 2.4. Calculating values for the SYMBOL command

Numeric and string variables

At the beginning of this chapter we discussed variables. These were simple numeric variables, used for storing numbers in the computer's memory. As useful as this may be, you can also store letters, words and even phrases in the Amstrad during a program. The only difference between variables which refer to numbers and those which refer to a character or string of characters is that the latter are identified by the dollar sign ($). This allows you to use statements like 'LET A$ = "AMSTRAD CPC464" ', either in a program or as a direct command. The command PRINT A$ will produce the string of letters assigned to the 'string variable' A$. (This is pronounced 'A dollar' or 'A string'.) The number of characters assigned to a string variable is limited to 255, and a string variable with no assigned characters is described as empty or 'null'.

String variables can 'contain' numbers, in a way. The assignment statement 'LET NUMBER$ = "7" ' is allowed, but it is very important to realise that "7" is quite different from the number whose value is seven. "7" in this case is just a symbol or shape, not a real number.

Like numeric variables, string variables can have long names, and may not begin with a number. You could use string variables like NAME$ or ADDRESS$ in a database program to store information about friends or customers. You may have numeric and string variables which use the same combinations of letters; the computer will keep track of the difference. This means you can use the variable names A and A$ or NUMBER and NUMBER$. Like numeric variables, string variables which have not had a string of characters assigned to them are assumed to be null.

You cannot mix string and numeric variables. Attempting to do so will produce a 'Type mismatch' error message. An assignment statement like A = "HELLO" is 'illegal' because "HELLO" is a string of characters, while 'A' is a numeric variable, and numeric variables may only contain values. Similarly, WORD$ = 45 is not allowed, because 'WORD$' is a string variable and may only 'contain' strings of characters, not numeric values. NAME$ = VALUE and PHONENO = NUMBER$ will also generate the error message, because you cannot cross-assign numeric and string variables. Numbers and letters are treated as quite distinct types of data and may not be intermixed, though there are ways of converting the one to the other.

Integers and integer variables

As well as the 'real' numeric variables described above, the Amstrad has special facilities for dealing with integers (whole numbers). The advantages of this are economy of storage and speed. If a variable is specified ('declared') as an integer variable with the per cent sign (%), then less memory is needed to store the contents of that variable, because the Amstrad can ignore anything after the decimal point. Because less memory is needed to store integers, less time is taken when the values associated with their variables have to be retrieved from memory and manipulated. As with string and numeric variables, the Amstrad can keep track of the difference between A and A%, but we humans may find this difficult, so try to use different variable names which show clearly what they 'stand for' to make your programs more understandable.

Integer variables are used just like numeric variables, but you must remember to use the integer sign (%). Also remember that if you try to assign a decimal fraction to an integer, the value stored will not be accurate. The other limitation is that integer variables cannot be used to store numbers outside the range –32768 to +32767. Again, attempting to assign a value outside this range to an integer variable will produce an 'Overflow' error message.

To compare the speed of integer variable handling with that of real numeric variables, try the following programs:

```
1 REM Normal numeric variables
10 CONST = TIME
20 FOR COUNT = 1 TO 1000
30 NEXT
40 CLS:DURATION = (TIME - CONST)/47
50 LOCATE 0,10:PRINT"That took";DURATION;"
seconds"
```

```
1 REM Integer numeric variables
10 CONST = TIME
20 FOR COUNT% = 1 TO 1000
30 NEXT
40 CLS:DURATION = (TIME - CONST)/47
50 LOCATE 0,10:PRINT"That took";DURATION;"
seconds"
```

As the two programs are identical except for the integer symbol you needn't type them both in, simply edit the first to get the second.

DEFSTR, DEFINT and DEFREAL

These three words allow you to declare single-letter variable names for use with strings or integers. The best place for them is at the start of a program. For example, 'DEFSTR A' means 'define the variable A to be a string variable'. This command can be used to save on memory (because you don't need to use $ every time you refer to the variable thereafter). Try this:

```
10 DEFSTR A
20 A = "This is a string"
30 PRINT A
40 DEFINT A
50 A = 32000
60 PRINT A
70 DEFREAL A
80 A = 1.2E ^ 25
90 PRINT A
```

You can use these words to define groups of variable names: 'DEFSTR A,G,W-Z' means that the letters A,G and those between W and Z (inclusive) will be string variable names.

It may take you some time to get used to using these commands, and we don't recommend them for beginners both because it restricts variable names to a single character and because it can get very confusing trying to make sense of a program in which these commands have been used.

3
Strings and Keys

INPUT

In direct mode you can enter commands, statements, or program lines. It's also useful to be able to 'collect' input from the keyboard during a program. This would be necessary in a game, for example, where the player has to press a key to start, or in a database where data has to be entered by the user.

The word needed to accept characters from the keyboard during a program is INPUT. This can be followed by a numeric or string variable, e.g. INPUT value or INPUT name$. The word INPUT causes the computer to print a question mark on the screen, then wait until the user has typed in a series of characters at the keyboard, the last of which is ‹ENTER›. It then assigns whatever characters the user has typed to the variable following INPUT. So, if your program includes INPUT name$, whatever characters the user has typed will be assigned to the string variable name$.

Because data types are incompatible, while it's quite all right for the user to enter a number when the variable following an INPUT is a string variable, the reverse is not true. If your program contains the instruction 'INPUT number', and the user types in 'seven ‹ENTER›', your program will stop with the error message 'Redo from start'. This message prompts the user to re-enter the information as numbers. The error occurs because, as explained in Chapter 2, strings of characters cannot be assigned to variables designed to deal with numeric values only. On the other hand, as we saw earlier, assigning numbers (when they are represented by strings) to string variables (e.g. 'LET this$ = "1234" ') is quite feasible and, as we'll see later, very useful.

INPUT on its own isn't much use, it only displays a question mark, giving the user no clue as to what is supposed to be entered. What's needed is a 'prompt', a message that appears on the screen and tells the user what information is required. The simplest way of doing this is to use a PRINT statement in the line

immediately before the INPUT. For example:

```
100 PRINT "Please enter your name ";
110 INPUT name$
```

(We will not be using the ‹ENTER› convention from here on, it's up to you to remember.)

Note that the cursor is left on the same line as the prompt, because of the semi-colon following the PRINT statement. You can get similar effects by expanding the INPUT statement;

```
200 INPUT "Please enter your name ";name$
```

or

```
300 INPUT "Please enter your name ",name$
```

In line 200, the question mark will still appear, but the comma in line 300 suppresses it.

IF, THEN and GOTO

IF, its associated word THEN, and the word GOTO, allow you to compare items such as variables with one another, and control the order in which program lines are executed.

Normally the computer starts processing program lines from the first line number, proceeds to the second, third and so on until there are no more program lines. As we saw in Chapter 2, the FOR...NEXT loop allows a group of lines to be repeated a certain number of times, and this 'redirection' can be very useful.

Let's suppose that you want a program to use a password, so that anyone who doesn't know the password cannot use it. The password may be a number, or a word, or a group of words, but for simplicity we'll use the word 'secret'. The first few lines of your program might look like this:

```
10 CLS:INPUT "Please enter the password
",password$
20 IF password$ <> "secret" THEN GOTO 10
30 PRINT "Ok, let's get on with the rest of the
program"
```

Line 10 clears the screen and prints the prompt enclosed in quotation marks. The computer then waits for the user to enter one or more characters followed by <ENTER>. Line 20 is processed next, and if the characters entered by the user do not form the word 'secret', the computer will jump back to line 10, clear the screen, print the prompt and wait for an input again. This will go on until the user enters the correct word, when the computer will process line 30 of the program, and carry on with the rest of the program. The symbol '<>' means 'not equal to', or 'not the same as' and can be used to compare string variables with strings of characters, string variables with one another, numeric variables against numbers, numeric variables with each other, and so on.

The general form of an IF statement or 'clause' is: IF (condition) THEN (operation). The condition part of the clause may compare numeric variables with numbers or string variables with groups of characters to see if they are the same or different; it could subtract one number from another, e.g. 'IF (value – 3) = 4 THEN number = 10'. The operation part of the clause may be a GOTO command, a PRINT statement, or many other operations. You can also use multiple statements after THEN, all of which will be executed if the condition is true, as in:

```
1000 IF reply$ = answer$ THEN score = score +
10:LOCATE0,20:PRINT "Score so far";score;:GOTO
500
```

As you read this book you'll come across many examples of the IF...THEN clause and will soon feel quite at home with it.

ELSE

The word ELSE can follow an IF...THEN clause, and is like the English word 'otherwise'. The word must be on the same program line as the conditional clause to which it refers, and is used to dictate what will happen if the IF clause is found to be false. In its simplest form it's used like this:

```
100 IF reply$ = answer$ THEN PRINT "Correct" ELSE
PRINT "Wrong"
```

It can be followed by multiple statements, just like THEN, for example:

```
750 IF reply$ = answer$ THEN PRINT
"Correct":score = score + 10 ELSE PRINT
"Wrong";:score = score - 5
```

Complex conditional clauses can be built up using one or more IF ... THEN clauses within another. For example:

```
830 IF a > b THEN q = q + 1:a = b ELSE IF a < b
THEN q = q - 1:b = a
```

INKEY$

Sometimes you won't need the user to type in a whole string of characters, you may just want a single key press. One typical example of this is at the end of a game, when you want the program to ask if the player wants another go. Often this is done by asking 'Do you want another go? Press Y for yes, N for no', or even abbreviated to 'Another go? ... Y/N'.

You could use INPUT for this, but if you've used much software you'll know how tedious it can be to have to keep pressing <ENTER> when you've made a choice. The word we need to collect a single key press is INKEY$. This tells the computer to look at the keyboard to see if a key is being pressed. One useful aspect of INKEY$ is that, unlike INPUT, the characters collected by the function are not displayed on the screen.

Because the Amstrad can test the keyboard very quickly, we have to combine INKEY$ with IF...THEN and GOTO to ensure that it waits until a key is pressed, INKEY$ only tests the keyboard once, which is why the following routine will not work:

```
100 PRINT "Press Y for yes, N for no"
110 response$ = INKEY$
120 IF response$ = "Y" THEN GOTO 10
130 IF response$ = "N" THEN END
```

The reason this won't work is that unless the user happens to be pressing either 'Y' or 'N' the instant the computer executes line 110, the program will continue through lines 120 and 130, to line 140, if there is one, because response$ has nothing assigned to it

– it is an empty string. The way round this is to make sure that the program keeps on repeating line 110 until the user presses one of the two keys. This can be done by adding a line: '115 IF response$ = "" THEN GOTO 110', which means, 'if the user isn't pressing a key, GOTO line 110 again'. The double quotes ("") indicate a 'null' or empty string – one with no characters assigned to it. We also have to cope with the fact that users may not press the 'Y' or 'N' keys.

The whole routine now looks like this:

```
100 PRINT "Press Y for yes, N for no"
110 response$ = INKEY$
115 IF response$ = "" THEN GOTO 110
120 IF response$ = "Y" THEN GOTO 10
130 IF response$ = "N" THEN END
140 GOTO 110
```

Alternatively, one could use a line like 'IF response$ <> "Y" AND response$ <> "N" THEN 110' in line 115, and omit line 140. Yet another method would be to use INSTR, e.g. IF INSTR ("YN", response$) = 0 THEN 110. INSTR is described below.

String handling

Basic has many built-in functions for manipulating groups of characters and the contents of string variables. These go under the general title of 'string handling' and include words like LEFT$, RIGHT$, MID$, LEN and INSTR. The first three produce strings, the last two return numbers.

LEFT$

LEFT$ is used to take copies of the left-hand characters of a string. For example, 'PRINT LEFT$ ("Example", 3)' will display the letters 'Exa': the leftmost three characters of the string 'Example'. If you were to pass a number greater than the length of the string to LEFT$, such as 'PRINT LEFT$("Example",9)', the whole string would be printed. The only limitations on numbers used in LEFT$ is that they may not be negative, fractions will be rounded to the nearest whole number (integer) and using zero returns a null string. All these considerations apply to RIGHT$.

RIGHT$

As you might expect, this function takes characters from the right-hand side of a supplied string, so 'PRINT RIGHT$("Second example",5)' would produce the letters 'ample'.

MID$

MID$ is a little more complex. It requires a string and two numbers to work with. The first number is the character position to start from, the second the number of characters to take. An example should clarify this: 'PRINT MID$("Third example",5,4)' will display 'd ex'. These are the four characters taken from the string "Third example", starting from the fifth character. Similarly, 'PRINT MID$("Third example",7,4)' will yield 'exam'. You can use MID$ to extract single characters from a string: 'PRINT MID$("One character",3,1)' returns 'e'.

MID$ can also be used to insert characters in a string. Its use as an assignment statement is quite unusual among Basic dialects, and very convenient it is too. It's used as described above, except that it appears to the left of the equals sign, with the string or string variable which is to be spliced into the string on the right. For example:

```
10 a$ = "The first string"
20 PRINT a$
30 b$ = "final"
40 MID$(a$,5,5) = b$
50 PRINT a$
```

UPPER$ and LOWER$

As you might expect from their names, these two commands convert all the characters in any string to upper or lower case. 'PRINT UPPER$("Amstrad")' will produce 'AMSTRAD', and 'PRINT LOWER$("Amstrad")' will produce 'amstrad'.

STRING$

STRING$ produces a string which is a repetition of one character. It needs two arguments: the number of times to repeat the character, and the character itself, its ASCII value, or its associated string variable. One of its most common uses is for producing borders. Try the following:

```
10 MODE 0
20 PRINT STRING$(20,"A")
30 LOCATE 0,25:PRINT STRING$(19,67);
40 man$ = CHR$(249)
50 LOCATE 0,2:PRINT STRING$(15,man$)
```

Note that in line 20, STRING$ is given the character itself in quotes, in line 30 the character argument is an ASCII code, while in line 50 it is a string variable. STRING$ is quite versatile in this respect.

LEN

LEN returns the length of the string passed to it. 'PRINT LEN("A string is a group of characters")' would produce the number 33 (remember that spaces are characters too). Because a string cannot be more than 255 characters, and an empty string contains no characters, LEN will always return an integer in the range 0 to 255.

INSTR

INSTR is a particularly useful command. It's short for 'in string' and is used to search a string of characters for another set. Like MID$ you have to provide three 'arguments' and in this case they are: the character position to begin at, the string itself, and the character(s) to search for. 'PRINT INSTR(1, "Amstrad", "tra")' will yield the number 4, because 'tra' begins at the fourth character position in the string 'Amstrad'. Omitting the first number causes the computer to assume the first position (the start of the string), so the example above is exactly the same as 'PRINT

INSTR("Amstrad", "tra")'. 'PRINT INSTR(5,"Amstrad","tra")' returns 0 because the pattern 'tra' begins before the fifth character position. You could use INSTR to test whether the user *had entered* a sentence *with* a *given* word *in it*, e.g.:

```
980 IF INSTR(reply$,"please") = 0 THEN PRINT
"You'll have to be more polite"
```

Examples

All the functions outlined above may be passed string variables as their arguments – you don't have to specify the literal string itself in quotes. Similarly, although we've been using PRINT to show how the functions work, you can assign the result of applying a string function to a string variable. The next two programs demonstrate these points.

```
10 LET surname$ = "THATCHER"
20 PRINT LEFT$(surname$,4)
30 PRINT RIGHT$(surname$,3)
40 PRINT MID$(surname$,2,3)
```

As another example, here's a short program which requests the user to type in his or her full name – christian and surname, separated by a space, and then splits the single INPUT string into the two names. See if you can work out how it does this before reading the explanation below, or try writing your own routine to achieve the same effect.

```
10 CLS:MODE 2
20 LOCATE 0,10:INPUT"Please type your full name,
first and last, then ENTER ",full.name$
30 length = LEN(full.name$)
40 space.pos = INSTR(full.name$," ")
50 IF space.pos = 0 THEN GOTO 10
60 first.name$ = LEFT$(full.name$,space.pos - 1)
70 last.name$ = RIGHT$(full.name$$,length -
space.pos)
80 CLS
90 LOCATE 0,10:PRINT"Thank you ";first.name$
```

Using a combination of INPUT, FOR...NEXT STEP, MID$ and LEN we can get the computer to perform simple tasks such as reversing someone's name. Let's follow through the constructs

and code words we'll need to achieve this. To begin with, we'll want to clear the screen, then have the user enter his name, and for this he'll have to be prompted. We can use INPUT with a prompt and store whatever is entered in the string variable name$. This suggests a line of code like:

```
10 MODE 2
20 INPUT"Please type in your first name, then
press ENTER ",name$
```

Note that we use the comma form of INPUT to avoid having the question mark appear, and that we remind the user to press ENTER when he's typed his response – never assume that a user knows anything about computing.

Next we'll need a loop, which starts with the last letter of the string called name$, and this suggests a FOR...NEXT loop, working backwards with a negative step value, i.e. taking characters from the right to the left of the name. We'll need to know the length of the string to find the last character position, and we can use LEN for this. See if you can work out how the program should go before looking at the solution. Don't worry if your solution isn't exactly the same as ours – Basic is such a flexible language that there are many ways of achieving the same end. The important question is not so much 'Is it right?', as 'Does it work?'

```
10 MODE 1
20 INPUT"Please type your first name, then press
ENTER",name$
30 FOR letter = LEN(name$) TO 1 STEP-1
40 PRINT MID$(name$,letter,1);
50 NEXT letter
```

Two points about this: note the semi-colon at the end of line 40, to ensure that the letters are printed on the same line, and the use of long variable names to make the program easier to understand.

Concatenation

This long word means nothing more than adding strings together with the plus sign. For example:

```
10 LET surname$ = "Jones"
20 LET first.name$= "David"
30 LET full.name$ = first.name$ + " " + surname$
```

Lines 10 and 20 define the two strings which are 'concatenated' in line 30 and assigned to a third string variable. Note that a space has to be added between the two strings, or the result would be 'DavidJones'. Strings cannot be subtracted, you have to use combinations of MID$, LEFT$ and RIGHT$ for this.

SPACE$

SPACE$ produces a string of spaces of the given length. Note that as with all string-handling functions, the number in brackets may not exceed 255. For example 'alongblank$ = SPACE$(250)'. An undocumented function, SPC, can be used with PRINT, but the number in brackets is MODded with 40 (number MOD 40). So 'PRINT SPC(89)' will produce 9 spaces. Because SPC can only be used with PRINT, you can't use it like SPACE$. 'LET along$ = SPC(250)' will produce a 'Syntax error' message.

Data type conversion

There are two functions available for converting strings to numbers and vice versa. These are STR$ and VAL.

STR$

STR$ is used to translate a number into its string representation. Numbers cannot be directly assigned to string variables: 'LET a$ = 7' is an illegal command and will produce an error message. However, STR$ can be used so that the character '7' is assigned to a string, complete with its leading space for a positive number and minus sign for a negative number. The function is used as in 'LET a$ = STR$(7)', and the argument in the brackets may be a numeric variable, e.g. 'LET value = 19:number$ = STR$(value)'. To prove to yourself that a leading character is added, try the following:

```
10 FOR value = 2 TO -2 STEP -1
20 number$ = STR$(value)
30 PRINT "/";number$;"/"
40 NEXT value
```

VAL

VAL is the complement to STR$ and returns the value of a string. It tests a string or string variable for numeric content and produces a number based on that test. For example, 'PRINT VAL("123")' returns the number 123. 'PRINT VAL("12A3")' returns 12. If the string begins with the '&' symbol, VAL attempts to evaluate the rest of the string as a hexadecimal number. The instructions 'PRINT VAL("&A")' and 'PRINT VAL("&" + chr$(65))' will produce the number 10, because the letter A represents the decimal number 10 in the hexadecimal system of counting (see Chapter 6).

VAL is particularly useful for converting single key presses, collected from the keyboard via INKEY$, to numbers. For example:

```
10 MODE 1
20 LOCATE 0,3:PRINT"Please press a number between
1 and 9"
30 akey$ = INKEY$:IF akey$ = "" THEN 30
40 number = VAL(akey$)
50 IF number = 0 THEN 10
60 LOCATE 0,7
70 PRINT"That number squared is";number * number
```

In this example, the square of a number is calculated by multiplying the number by itself; another method would be to use the 'exponential' sign, the up arrow and the number two, as in: 'square = number ^ 2'. The symbol can be used for cubes (cube = number ^ 3), and so on.

Some number-handling operations

RND

In many programs it's useful to have the computer produce a number at random, and the Amstrad has a special function do to this called RND. This returns a more or less random number between 0 and 1, so if you want larger numbers you have to multiply the result, as in 'LET random = RND(1) *10'. This would produce numbers between 0 and 9.99999999. To turn these to whole numbers you could use the functions INT, ROUND, or FIX.

INT

INT converts a number with one or more decimal places to an integer – a whole number. 'PRINT INT(3.3)' produces 3, as does 'PRINT INT(3.6)'.

FIX

FIX has the same effect on positive numbers, but negative numbers treated with FIX are one more that those treated with INT.

ROUND

ROUND rounds numbers with decimal parts to the nearest whole number, so 'PRINT ROUND(3.3)' produces 3 but, unlike INT or FIX, 'PRINT ROUND(3.6)' will produce 4.

To get random numbers in the range 1 to 6, which would be useful in a game involving dice, you could use operations like the following:

```
100 LET random.number = RND(1) * 5
110 LET random.number = random.number + 1
120 LET random.number = ROUND(random.number)
```

Line 100 places a value between 0 and 4.999999999 in the variable 'random.number'. Line 110 adds one to this, making the value range of 'random.number' between 1 and 5.999999999. Line 120 rounds the value in 'random.number' to between 1 and 6. The three lines can be condensed to:

```
100 random.number = ROUND((RND(1) * 5) + 1)
```

Using INT this would be:

```
100 random.number = INT((RND(1) * 6) + 1)
```

REM statements

REM is short for REMark, and is used in a program to make it more understandable to another programmer, or even to yourself! It can be used anywhere in a program and is, effectively, ignored by the computer. It tells the computer to ignore the rest of the characters in that program line, and is used like this:

```
10 REM sample program
20 REM
30 REM     Initialise Variables
40 score = 0: max.goes = 10
50 REM max.goes is the highest number of 'turns'
allowed
60 players = 5:title$ = "HANGMAN":REM title$ is
the name of the game
```

Note that REM can be used to break up a listing (line 20), may be followed by a number of spaces (line 30), can be used after a colon (line 60) and so on. REM may also be abbreviated to the apostrophe ('), as in:

```
100 '      Calculate average
110 '    Result will be average
120 IF number = 0 THEN average = 0:GOTO 170: ' If
number is zero, need to avoid
130 ' calculation or we'll get a
140 ' division by zero error
150 average = total / number
160 REM Calculation complete
170 IF average = 0 THEN PRINT "Error"
180 REM Rest of program
```

When you first start programming you should make liberal use of REMs to help you find problems in your programs.

ZONE

ZONE sets up the tab fields used to calculate where to display items following the comma print separator in PRINT statements. ZONE can only be passed numbers in the range 1 to 255, and real values will be rounded. ZONE is normally set to 13: try 'PRINT "a", "a", "a" '. You can alter ZONE to any value between 1 and 255: try 'ZONE 7:PRINT "a", "a", "a" '. Coupled with PRINT USING, ZONE allows you to format tables or columns of figures with relative ease.

WIDTH

WIDTH is used to set the line width of the printer. This means that after the Amstrad has sent the number of characters indicated to the printer, it will send a carriage return/line feed (CR/LF), so that the next line can be sent and printed. WIDTH 255 effectively turns off this automatic CR/LF, leaving it up to software or the printer to decide when to take the print head back to the left and advance the paper by a line. Many printers have a DIP switch to control whether the computer should send CR/LF, or whether the printer will decide for itself (usually after 80 characters).

If you have trouble getting single line spacing from your Amstrad/printer system, try cutting line 14 of the printer cable. You'll need a scalpel and a printer manual, and the Amstrad manual contains details which should help you work out which one it is. Line 14 carries the AUTO FEED XT signal. A more serious problem is that the Amstrad only sends the bottom 7 bits of each byte, so ASCII codes 128 to 255 can't be sent as printer control codes. If graphics characters don't come out as you hope, this could be the reason.

Examples

Here's a simple program which demonstrates some of the principles outlined in this chapter and shows how to use some of the functions.

```
10 MODE 1:LOCATE 15,0:PRINT "GUESSING GAME":tries
= 0
20 anumber = ROUND((RND(1) * 8) + 1)
25 REM Select a number between 1 and 9
30 LOCATE 0,5:PRINT "I've thought of a number
between 1 and 9"
40 LOCATE 0,7:PRINT "Press a number to guess it"
50 guess$ = INKEY$:IF guess$ = "" THEN GOTO 50
55 ' Trap null $ in GUESS$
60 guess = VAL(guess$):IF guess = 0 THEN GOTO 50
65 ' Convert $ to number with VAL, if zero go back
to line 50 - get another key press
70 tries = tries + 1:REM Update number of attempts
80 IF guess = anumber THEN GOTO 110: 'correct
90 IF guess < anumber THEN LOCATE 0,10:PRINT
guess; " is too low":GOTO 50
100 IF guess > anumber THEN LOCATE 0,10:PRINT
guess;" is too High":GOTO 50
105 REM ******************************
107 ' Routine for a correct answer
108 '
110 CLS:LOCATE 0,10:PRINT guess;" is right"
120 LOCATE 3,12:PRINT "You got it in";tries;"
tries"
130 LOCATE 10,15:PRINT "Another go...y/n"
140 akey$ = UPPER$(INKEY$):IF akey$ = "" THEN 140
150  'Collect upper case character & convert to
caps
160 IF akey$ = "Y" THEN 10
170 IF akey$ = "N" THEN CLS:END
180 GOTO 140:' akey$ neither Y or N, so repeat
line 140 until it is
```

Line 10 sets the screen mode, displays the title of the program starting from the fifteenth column of line zero and 'initialises' the number of tries to zero. Line 20 puts a random number between 1 and 9 into the variable 'anumber'. Lines 30 and 40 display information and a prompt. Line 50 tests the keyboard repeatedly until a key is pressed, the character of that key is then assigned to the string variable 'guess$'. Line 60 tests the value of the character in 'guess$', if this is 0, i.e. zero or non-numeric, line 50 is repeated. Line 70 updates the number of tries so far. Line 80 tests for a correct guess and redirects the program flow to line 110 if the guess is right. Line 90 checks whether the guess is too low, if so it displays an appropriate message and redirects flow to line 50 for the next guess. Line 100 tests for the guessed number being greater than the random number, displaying a message and

redirecting flow accordingly, just like the previous line. The lines following 110 deal with a correct answer, and use the 'Y/N' test described earlier. Rather than having to test for upper or lower case answer (Y/y and N/n) we use UPPER$ to convert any key press to upper case.

You'll also note that we don't use LET and that GOTO after a THEN is optional, i.e. 'IF (condition) THEN GOTO 50' is the same as 'IF (condition) THEN 50'. Be careful not to place a NEXT on the same line as an IF...THEN, because the NEXT will not be executed if the condition following the IF clause is false.

Here's a listing for a routine which uses INKEY$ to replace INPUT. INPUT is all very well, but users can make mistakes and, when entering a long string or number, can overwrite the screen display. Commercial software often displays a 'prompt' followed by two angle brackets, e.g. 'Please enter a 3-digit number ‹ ›'. The cursor is initially placed at the left-hand side of the space between the angle brackets, the user is free to move the cursor between them but not beyond, and this technique can easily be adapted to the Amstrad. Because it uses INKEY$, you could tailor it in various ways to suit your needs. For example, you could use a flag when calling the routine such that it ignores non-numeric or non-alphabetic characters (ASCII codes 48 to 58), redefine the cursor character, input length markers and so on. As written, the algorithm ignores ASCII codes less than 32 and greater than 122, so it disables the CLR and other keys.

To use the routine, you have to define 'prompt$' as the message to be displayed, 'maxlen' as the maximum number of characters to be entered and 'row' and 'col' for the row and column from where the prompt and input are to be displayed. When the user presses ‹ENTER›, the subroutine returns with any characters input in the string variable 'enter$'.

```
10 MODE 1
20 col = 1:row = 10:maxlen = 5
40 prompt$ = "Enter a 5 digit number"
50 GOSUB 10000
60 MODE 1
70 PRINT"You entered ";entry$
80 END
10000 akey$ = "":entry$ = ""
10010 enter$ = CHR$(13):leftarrow = 242
10020 curschar = 95
10030 cursor$ = CHR$(32)+CHR$(8)+CHR$(curschar)+CH
R$(8)
10040 blank$ = STRING$(maxlen+1,8)
```

```
10050 LOCATE col,row
10060 PRINT prompt$;" <";SPACE$(maxlen);">";
10070 PRINT blank$;cursor$;
10080 akey$ = INKEY$
10090 IF akey$ = "" THEN 10080
10100 akey = ASC(akey$)
10110 entrylen = LEN(entry$)
10120 IF akey = leftarrow AND entrylen>0 THEN GOSU
B 10220:GOTO 10080
10130 IF akey = leftarrow  AND entrylen<1 THEN 100
80
10140 IF akey$ = enter$ AND LEN(entry$)= maxlen TH
EN RETURN
10150 IF (akey <32 OR akey>122) THEN 10080
10160 entry$ = entry$+akey$
10170 entrylen = entrylen + 1
10180 IF entrylen>maxlen THEN entry$ = LEFT$(entry
$,maxlen):GOTO 10080
10190 PRINT akey$;
10200 IF entrylen<maxlen THEN PRINT cursor$;
10210 GOTO 10080
10220 entry$ = LEFT$(entry$,entrylen-1)
10230 PRINT CHR$(8);cursor$;
10240 RETURN
```

Keys

KEY

The Amstrad keyboard is 'soft' – you can alter the character generated by any key. This may seem a somewhat redundant feature – after all, what's a keyboard for if not generating a standard character set? – but there are a number of ways in which this feature can be turned to the programmer's advantage. For example, while a program which uses MODE 2 is under development it can get very tedious having to type in 'MODE 1:LIST' every few minutes, and there's no need. What you can do is to assign a command string to any of the keys on the numeric keypad using the keyword KEY. The string just quoted can be assigned to the 7 key with the command:

```
KEY 7,"MODE 1:LIST" + CHR$(13)
```

Note that the instructions assigned to the key must be in quotes and that to get an automatic ‹ENTER› at the end of the string you must add CHR$(13) with the '+' sign for string concatenation. The key number to use is the code given in the Amstrad manual, Appendix 3, p. 15. In this example you can see that the Amstrad has automatically added 128 to the number supplied. 'KEY 135, "MODE 1:LIST" + CHR$(13)' does exactly the same thing, but makes less sense.

You can only use 31 expansion strings, numbered 128 to 159. Note that all keys of the keypad (except the small ENTER key) generate the same value if SHIFT or CTRL is pressed as well. The small ENTER key generates 139 normally and with SHIFT, but if CTRL is pressed as well it produces a value of 140. When the Amstrad is turned on, this key is set up to produce RUN"‹ENTER› when pressed with CTRL (the ASCII hex sequence for this is given in the Amstrad manual, Appendix 3, p.15). There are 120 bytes allotted for all the KEY assignments.

KEY DEF

As well as being able to set up 'function' keys, you can actually alter the ASCII code generated by an key using the reserved words KEY DEF. These can be used to alter whether the key repeats as well as the 'shifted' and control values generated by each key. For example, to make the large ENTER key produce the capital letter 'A', you would use 'KEY DEF 18,0,65,65,65'. The first number refers to the key (see the Amstrad manual, Appendix 3, p.16), the second number dictates whether or not the key will repeat. A zero means no repeat, a one means repeat and other numbers will give the error message 'Improper argument'. The three numbers which follow are the ASCII codes the key should generate. The first of these is the 'normal' character, the second is the 'shifted' character and the last is if the CTRL key is pressed as well. If you only want to change one of these, you simply omit the values you don't want altered and whatever is being used will continue to be used. For example, if you wanted to alter the CTRL value of the 'z' key, you could use 'KEY DEF 71,,,,32'. This makes CTRL-z generate a space (ASCII code 32), but doesn't affect key repeat, normal or shifted character generation.

KEY DEF can be very useful. If you get tired of hitting ESC instead of '1', you can reset the escape key so that it only works

as normal if shift or CTRL are pressed as well. To do this, use KEY DEF 66,,0.

SPEED KEY

Most of the Amstrad's keys will auto-repeat: when a key is pressed there is a short delay, and then the character is repeated at regular intervals. Both these values can be altered via the reserved words SPEED KEY. Setting a faster repeat rate can be useful if you have a lot of copy editing to do, or if you want a fast keyboard response for a game.

SPEED KEY needs two arguments: the delay before the repeating starts, and the interval between repeats. Both are measured in multiples of 0.02 of a second (1/50th sec.). Thus 'SPEED KEY 100,50' means that keys will start to repeat after they've been held down for two seconds, and will repeat at one-second intervals. You must be careful if you set up a short delay and a fast repeat in a program, e.g. with 'SPEED KEY 1,1', because when the program ends, or if you break into it, you may find that the keyboard is unusable. The keys will begin to repeat virtually the instant they're pressed, and will repeat so quickly that you end up with half-a-dozen characters at every press. Always restore the two functions to their default values or your preferred values at the end of a program. A quick way to reset the default value is 'CALL &BB00', and you should remember to trap errors and breaks, redirecting flow to the program's end, e.g.

```
10 ON ERROR GOTO 10000
20 ON BREAK GOSUB 10000
30 REM Program
40 '
999 GOTO 10000:REM END
9999 ' Reset key delay and repeat
10000 CALL &BB00
```

Note that 'CALL &BB00' also resets all keys to their default ASCII codes (as listed in the Amstrad manual, Appendix 3, p.14). The ROM routine completely resets the keyboard management system, including key assignments, repeat speeds, and so on.

Disabling ENTER

Using KEY DEF you can protect programs, at least once they're running, in a subtle and frustrating way. For example, 'KEY DEF 18,0,0,0,0' will make the large enter key generate nothing at all! Similarly, 'KEY 139,"" ' disables the small enter key of the numeric keypad. To complete this protection you also have to 'KEY DEF 38,1,109,77,0', which means CTRL-M will not generate a carriage return either (109 and 77 are the ASCII codes for lower and upper case 'm' respectively). Now, while a user may break into a program, there's no way it can be listed, because no one can enter a command – none of the enter keys generate anything. As with other protection techniques, you must make sure that your program is finished before you use these commands, because you're excluded just as much as anyone else. You'll also have to rewrite the INPUT routine, adopting some other convention for ENTER so that programs can accept user inputs.

Although there are 31 expansion keys available (128-159), the manual only shows key numbers to 140 (shifted small enter key) – so where are keys 141 to 159? The answer is that you have to assign them to keys yourself. For example 'KEY DEF 38,1,109,77,155' assigns 'key' number 155 to the 'm' key. Then you'd use something like 'KEY 155, "RENUM"+CHR$(13)' to assign a function string to CTRL-M.

INKEY

INKEY allows you to test the state of a given key, so it is rather like a specialised INKEY$. INKEY is used like this:

```
100 IF INKEY (38) = 0 THEN GOTO 1000
```

INKEY tests the state of the key number given in brackets – in the example this is the 'm' key – and the number you use is the same as the one used for KEY DEF. INKEY may return one of four values, according to whether the key is not pressed, is pressed, is pressed with shift held down, or is pressed with CTRL pressed as well. These values are given in the following table.

INKEY value	Meaning
–1	Key not pressed
0	Key pressed
32	Key pressed + SHIFT
128	Key pressed + CTRL
160	Key pressed + SHIFT + CTRL

Because it's possible to use INKEY to establish whether a key is pressed, regardless of whether the SHIFT and/or CTRL keys are pressed as well, INKEY can often be used instead of INKEY$. For example, if you want a program to GOTO line 5000 if key number 43 ('y') is pressed, you would use 'IF INKEY (43) <> –1 THEN GOTO 5000'. INKEY also allows you to check whether a combination of the required key, SHIFT and/or CTRL are pressed. This sort of thing would be very awkward to do using INKEY$, particularly if you had redefined any of the keys with KEY DEF. Together with KEY DEF, INKEY allows you to use complex keyboard handling in your programs.

4
Subroutines, Arrays and System Functions

Subroutines

Subroutines are sections of code which can be called up from any part of a program. They can save you having to type in identical program lines over and over again and thus reduce the space taken up by a program. For example, you may have a program which stops at various points and asks the user to press the space bar to continue, and this operation is best handled with a subroutine. The code at the heart of this might be:

```
1000 LOCATE 25,1:PRINT "Please press the space
bar to continue"
1010 A$ = INKEY$:IF a$ = "" THEN 1010
1020 IF A$ <> CHR8$(32) THEN 1010
1030 'On with the program
```

This fragment displays a prompt at the foot of the screen, then cycles between lines 1010 and 1020 until the user presses the space bar, before continuing with the program. It would be tedious to have to write these lines half-a-dozen times or more in the program every time you needed the pause. An alternative is to convert the section of code into a subroutine with the word RETURN at line 1030. Then, when you want the prompt displayed, and your program to pause until the space bar is pressed, all you have to do is use the command GOSUB 1000.

GOSUB tells the computer to jump to the line number indicated, and to process the program from there onwards. When it meets a RETURN command, it will jump back to the statement following the GOSUB command. The computer remembers where it encountered the GOSUB command and RETURNs to the command after it. To see how the GOSUB process works, enter the subroutine and then run the following:

```
10 CLS
20 GOSUB 1000:REM space bar subroutine
30 CLS
40 PRINT "Once more, please"
50 GOSUB 1000:REM space bar subroutine
60 CLS
70 LOCATE 10,10
80 PRINT "THE END"
90 END
```

It's a good idea to use self-contained subroutines like this throughout your programs. They can make a listing much easier to follow, but do mean that you have to take great care over documenting the listing with plenty of REMs.

Operations can be repeated a number of times using a FOR...NEXT loop, and there are ways of achieving the same effect with other reserved words. One method is to use an IF...THEN clause with a GOTO: the following two fragments are identical in effect.

```
10 FOR COUNT = 10 TO 100 STEP 10
20 PRINT "The square root
of";COUNT;"is";SQR(COUNT)
30 NEXT
```

```
10 COUNT = 10
20 PRINT "The square root
of";COUNT;"is";SQR(COUNT)
30 COUNT = COUNT + 10
40 IF COUNT <= 100 THEN GOTO 20
```

The second example is longer, but this method can be useful when you're not certain how many times you want an operation repeated, or if you want to change the number of operations during the loop.

WHILE and WEND

There are two other words which go together like IF...THEN to control loops. These are WHILE and WEND. The fragments above are replicated in the next example.

```
10 COUNT = 10
20 WHILE COUNT <= 100
30 PRINT "The square root
of";COUNT;"is";SQR(COUNT)
40 COUNT = COUNT + 10
50 WEND
```

These sorts of 'control structures' have a number of advantages over FOR...NEXT loops. First and foremost is the fact that all too often we do not know exactly how many times we may wish to have a particular set of operations performed. The structures outlined here avoid the problem of having to 'leave' a FOR...NEXT loop before its exit condition has been fulfilled (i.e. before the loop counter has reached the upper limit). For example, consider the following:

```
10 FOR avalue = 1 TO 10000
20 akey$ = inkey$
30 IF akey$ = CHR$(32) THEN 50
40 NEXT
50 REM On with the program
```

This routine creates a pause in a program until the user presses the space bar, or until the loop has been repeated 10000 times. However, if the user presses the space bar before the loop has finished, then we still have a number of FOR...NEXT loops 'unresolved', and this can cause problems if it occurs too often.

ON GOTO/GOSUB

ON GOTO and ON GOSUB don't really do much more than allow you to condense lines of code like this:

```
1000 IF option = 1 THEN GOTO 2010
1010 IF option = 2 THEN GOTO 5350
1020 IF option = 3 THEN GOTO 6870
1030 IF option = 4 THEN GOTO 8000
```

to this:

```
1000 ON option GOTO 2010,5350,6870,8000
```

The latter is much neater, and therefore it's harder to make a mistake when following the program or typing it in.

ON GOSUB works in the same way as ON GOTO. For both, if the value of the variable 'option' (or whatever numeric expression you use) is zero, control will 'fall through'. That is, none of the list of line numbers will be processed – the system will ignore the ON ... statement. The same will happen if the expression evaluates to a number greater than the number of line numbers in the list.

ON GOSUB doesn't return control to the next line number in the list when a RETURN is met. Control is returned to the statement following the statement which called the subroutine. ON GOSUB is of particular value in menu selection. Using the first lettter of each option, INSTR and GOSUB, menus become simple to design:

```
10 CLS
20 PRINT "Q = Quit"
30 PRINT "N = Next"
40 PRINT "P = Previous"
50 PRINT "D = Delete"
60 Print "A = Amend"
70 menu$ = "QNPDA"
80 akey$ = INKEY$:IF akey$ = "" THEN 80
90 choice = INSTR(menu$,akey$)
100 IF choice = 0 THEN 80
110 IF choice = 1 THEN CLS:END
120 ON choice GOSUB 2000,3000,4000,5000
130 GOTO 10
```

Functions

DEF FN

DEF FN can be used to set up user-defined functions. While it's a very useful feature, it's rather limited. Functions can't do things like make sounds, draw lines, print results, and so on. They're generally used for numeric or string operations. One very common use is for generating random numbers at any point in a program, without having to repeat statements like 'randomnumber = INT(RND(1) * 100) + 1'. Using DEF FN you'd set up a function at the start of a program like this:

```
10 DEF FN rand(number) = INT(RND(1)* number) + 1
```

Then, whenever you needed a random number in the range 1 to number, you could call the function rather as you would a subroutine:

```
100 X = FN rand(100)
```

This passes the number 100 to the function 'rand', which then substitutes the current value of 'number' in the expression 'INT(RND(1) * number) + 1', which gives the variable 'randnum' a randomly selected integer value in the range 1 to 100.

Functions are not restricted to operating on the number you pass to them in parentheses; they can use any variable in your program, but they can only return one item: for example, if you wanted to generate random numbers between two values, to be stored in the variables 'high' and 'low', like this:

```
10 DEF FN rand(number) = INT(RND(1) * (high-low))
+ low
```

Functions can also operate on strings. If a function is to be passed and return a string, its name must be followed by the $ or string identifier. The following returns the string representation of numbers, stripped of any leading sign (space or minus):

```
DEF FN strip$(number) =
MID$(STR$(number),2,LEN(STR$(number)) - 1)
```

To use this function, you'd 'call' it like this:

```
1000 number$ = FN strip$(number)
```

If an error is found in a function the error message will report the line number of the statement which called the function, not, as you might expect, the line number of the function definition itself.

Arrays

Array handling is one of Basic's most important facilities. Once you grasp the concepts and become familiar with manipulating arrays you'll find you use them in most of your programs because they make programming much easier.

An array is a structure for storing data, and can be thought of

as a list. You can have arrays for real numbers, integers or strings. Arrays are defined with the reserved word DIM, which sets up space for an array in memory.

The simplest sort of array is an integer array of one dimension. This can be imagined as a list of values, or series of boxes, each of which holds an integer value. When we define an integer array (called 'array%') with 'DIM array%(10)', the Amstrad sets up 10 pigeon holes for data storage and we can load information into them with the usual assignment statement, '='. Thus 'LET array%(1)=99' puts the value 99 into the first cell or box of the array. The figure in brackets is known as a subscript. Similarly, 'array%(5) = ROUND(divisor / dividend)' places the result of the expression into the fifth cell of the array. We can access the values in this integer array rather as we would with normal variables: 'LET number = array%(7)' assigns to the variable 'number' whatever value is contained in the seventh element of the array. See Figure 4.1.

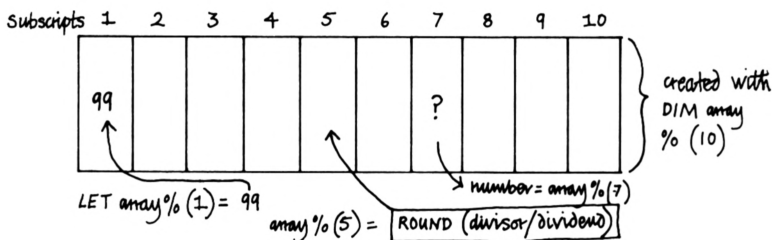


Fig. 4.1. Schematic representation of the integer array 'array%', with some array operations

Arrays are very useful because they allow the programmer to store data and access or update it very quickly. For example, we can use an array to store the squares of numbers, so that when we need to find squares, there is no need to recalculate the various values:

```
10 DIM square%(30)
20 FOR number = 1 TO 30
30 square%(number) = number ^ 2
40 NEXT
```

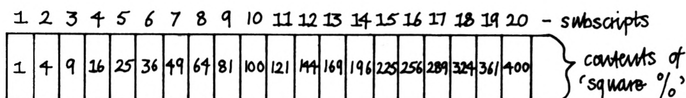


Figure 4.2. Schematic representation of the array 'square%'

Lists like this (Figure 4.2) are known as 'look-up' tables – once the values have been calculated and stored, they can be accessed much faster than if they had to be calculated each time they were needed. A look-up table could be loaded into an array from a cassette file at the beginning of a program (see Chapter 11).

The listing shows how variables may be used as subscripts. The variable 'number' is used to reference all the cells in the array, because 'number' is the loop counter of the FOR...NEXT structure, is used as a subscript, and takes on values of 1, 2, 3, ... 19, 20.

Integer arrays can only cope with integer values in the range –32768 to +32767. Attempts to assign values outside this range to cells of an integer array will generate an 'Overflow' error message, and if real values are assigned to integer arrays, they will first have the function 'INT' applied.

Real arrays are used for numbers with decimal places, as in:

```
10 DIM realarray(100)
20 FOR eachcell = 1 TO 100
30 realarray(eachcell) = RND(1)
40 NEXT
```

Real arrays occupy more memory and therefore take longer to access than integer arrays, so should only be used when necessary.

String arrays are also possible, and here we can think of each entry not as a number, but as a string, and any of these may be up to 255 characters long. For example:

```
10 DIM French$(20)
20 French$(1) = "un"
30 French$(2) = "deux"
40 French$(3) = "trois"
50 REM rest of definitions
1000 FOR count = 1 TO 20
1010 PRINT count;" = ";French$(count)
1020 NEXT
```

You may use arrays of 10 cells or less, without declaring them using DIM, thus:

```
10 FOR cell = 1 TO 10
20 array(cell) = SQR(cell)
30 NEXT
```

Note that arrays have a zero subscript which can be very useful for storing data, e.g. the total of the values in an array, calculated by the subroutine.

```
1000 total = 0
1010 FOR cell = 1 TO no.cells
1020 total = total + array(cell)
1030 NEXT
1040 array(0) = total
1050 RETURN
```

Multi-dimensional arrays

So far we've only looked at arrays which can be imagined as lists. Of far more use are what are known as multi-dimensional arrays. The simplest of these is the two-dimensional array. This is easiest to imagine rather as the text screen, or graph paper: each entry is referred to by two numbers, column and row. However, for arrays, the order is the reverse of that of the text screen, the values being given in the order 'row, column' (see Figure 4.3).

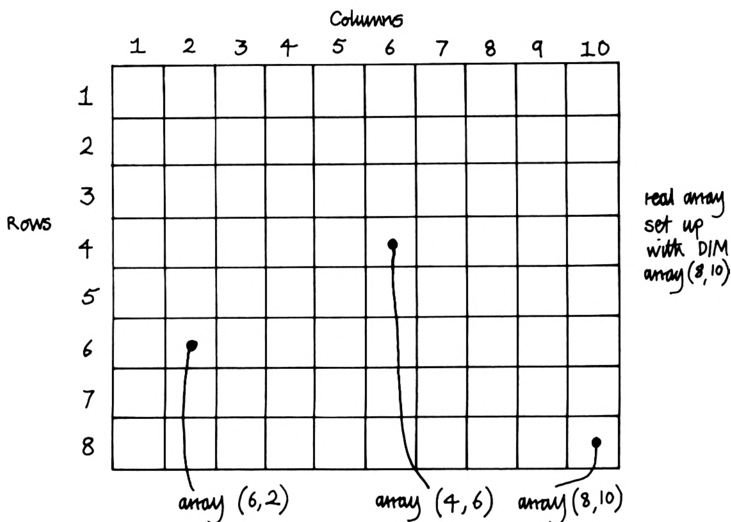


Figure 4.3. Schematic representation of a two-dimensional numeric array

Setting up such an array in memory involves extending the DIM command slightly. For example, we can use a two-dimensional string array to hold names, addresses and phone numbers for a simple database such as that given at the end of this chapter. The array might be set up by the statement, '10 DIM address-book$(100,3)', to allocate space for a string array of 100 rows by 3 columns – 100 names, their addresses and phone numbers. We can now store names in column 1, addresses in the second column and telephone numbers in the third.

To insert data into this array, we might have lines like:

```
200 LET addressbook$(1,1) = Fred Smith
210 LET addressbook$(1,2) = 10, The Avenue
220 LET addressbook$(1,3) = 01-678-5478
```

Figure 4.4 shows how the data can be imagined to help you grasp the principles. (In fact the machine stores all its data as a long list, but the mechanics of this are beyond the scope of this book.)

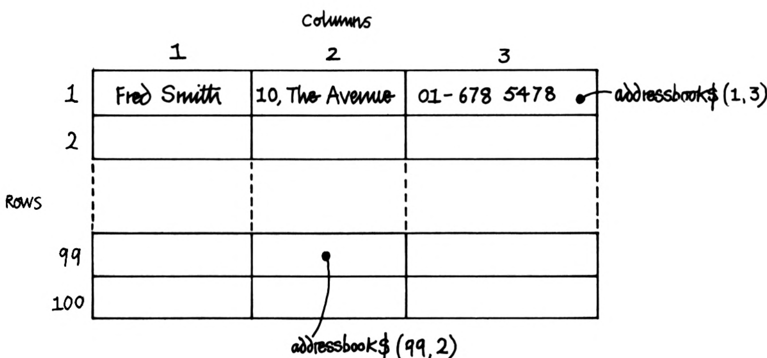


Figure 4.4. Schematic representation of the string array 'addressbooks$' with contents

As an example of array handling, imagine a numeric array of 100 rows, each of 20 columns. It is quite simple to code a subroutine to calculate the totals and averages of each column of figures, and place the results in the last two row entries for each column (rows 101 and 102):

```
1000 FOR column = 1 TO 100
1010 total = 0
1020 FOR row = 1 TO 20
1030 total = total + array(row,column)
1040 NEXT row
1050 array(101,column) = total
1060 average = total / 100
1070 array(102) = average
1080 NEXT column
1090 RETURN
```

With this subroutine we can constantly update the averages and totals of the array with 'GOSUB 1000', then extract the relevant values from the last two rows of the array (Figure 4.5).

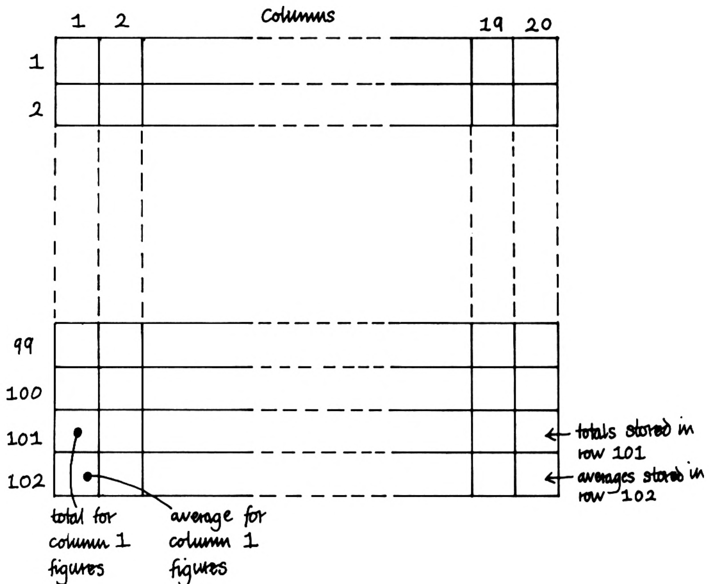


Figure 4.5. Using rows of an array to store 'extra' information

It is possible to have arrays with more than two dimensions: 'DIM largearray$(20,20,20)' is quite legal, but takes enormous quantities of memory and is quite difficult to conceptualise. Such arrays might be used when you need to refer to data with three dimensions, and this is easiest to imagine as a cube (see Figure 4.6). Arrays with more than three dimensions are rarely found in

Basic programs, though the capability for handling them exists in most dialects of the language.

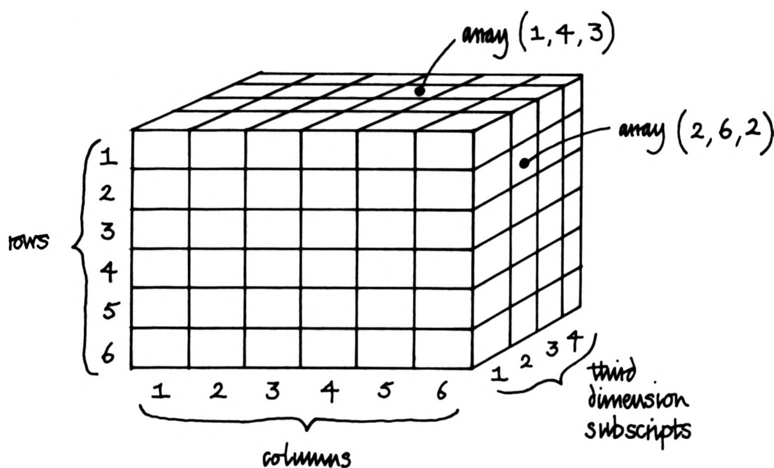


Figure 4.6. Conceptual representation of a three-dimensional array

The Animals Game

What looks at first like a difficult programming problem can often be solved very simply with arrays. One such example is the Animals Game. In this, you think of an animal and the computer tries to guess which one. The game is programmed in such a way that the computer 'learns' new animals and new questions to ask.

In the example which follows, the computer starts by 'knowing' two animals only – moose and whale, and that the difference between them is that a whale lives in water. When it asks if the animal lives in water and the answer is 'yes', the computer will ask if the animal being thought of is a whale. If it's not, the computer will ask what the animal is, and will also ask for a question that will distinguish between a whale and the given animal. It then starts again, asking you to think of an animal, and so on. Simple as the game may be, working out how to program it isn't easy. The routine presented here gives a very simple, if rather

limited solution, which you can tailor to suit your needs. For example, it would be a simple matter to arrange to have the array saved to tape as a sequential data file (see Chapter 11) which could be loaded into the machine at the start of a session. Because the computer doesn't in any way 'understand' the words used, you could alter the initial contents of the array so that the program dealt with plants, aircraft or whatever takes your fancy. The 'yes/no' nature of the program reflects the method of binary classification used in Biology and other subjects, and thus it could be used to help those learning the technique.

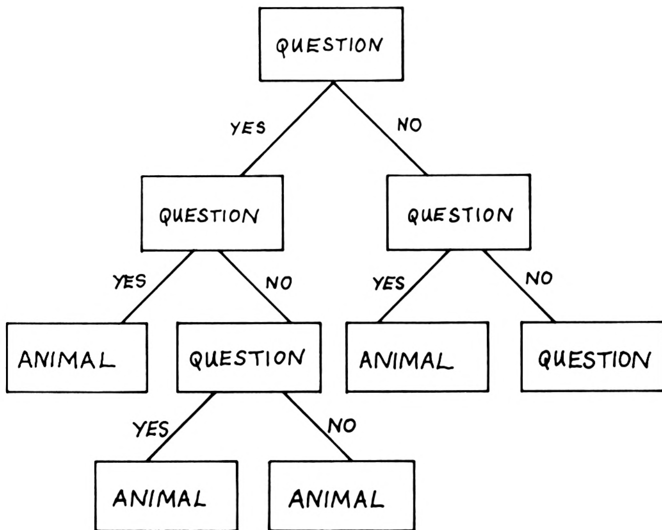
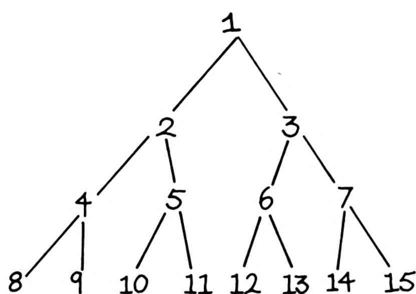


Figure 4.7. Binary tree of the sort used for the Animals Game

The program uses a single-dimensional array to store the data and questions. It's easiest to imagine the way the information is stored as a 'binary tree' (see Figure 4.7). The problem is to represent this two-dimensional structure in an array of only one dimension. (It is possible to use a two-dimensional array, but that method wastes a lot of memory.) Each 'node' of the tree stores either a question or an animal. Animal names are 'tagged' with the symbol "I", and as new animals are 'learned', they and their associated questions are stored in the array, and the other items adjusted accordingly. That is, when the computer finds a string in the array which begins with "I", then that is the last node of a

branch and it can ask if that's the animal being thought of. If it's not, then the distinguishing question is placed in that node, and the old animal and the new animal are placed in the nodes which lead from the branches of the old animal's node. The structure of the binary tree is such that from any node 'n', the nodes which terminate the two branches leading from it are found by 2 * n and (2 * n) + 1. Thus node number 1 branches to nodes two and three, and node 4 branches to nodes eight and nine (see Figure 4.8). The new and old animals must be placed in the correct order, and in this example, the 'yes' branch is always the left-hand one. Figure 4.8 shows how the single dimension mimics the structure of the tree.

```
10 DIM array$(1000):array$(1) = "Does it live in
water"
20 array$(2) = "!moose":array$(3) = "!whale"
30 CLS:element = 1
40 this$ = array$(element)
50 IF LEFT$(this$,1) = "!" THEN GOSUB 80:GOTO 30
60 PRINT this$;"?":GOSUB 180:IF answer$ = "y" THEN
element = (2 * element) + 1 ELSE element =
(element * 2)
70 GOTO 40
80 this$ = RIGHT$(this$,LEN(this$)-1):PRINT"Is it
a ";this$;"?":GOSUB 180
90 IF answer$ = "y" THEN PRINT"I guessed it - want
another go?":GOSUB 180:IF answer$ ="y" THEN 30
ELSE END
100 PRINT"I give up, what is it?"
110 INPUT it$
120 PRINT"Type in a question which
will":PRINT"distinguish between a ";it$;" and a
";this$:PRINT"The question should start with, Does
it, Is it etc.":PRINT"The answer must be either
yes or no"
130 INPUT query$
140 PRINT"What's the answer for a ";it$;"?":GOSUB
180:it$ = "!" + it$:this$ = "!" + this$
150 IF answer$ = "y" THEN array$(2 * element) =
this$:array$((2 * element) + 1) = it$ ELSE
array$(2 * element) = it$:array$((2 * element) + 1
) = this$
160 array$(element) = query$
170 RETURN
180 answer$ = LOWER$(INKEY$):IF answer$ = "" THEN
180
190 IF INSTR("yn",answer$) = 0 THEN 180 ELSE
RETURN
```



Numbering the 'nodes' of a binary tree helps to mimic its structure in a single dimension array (below)

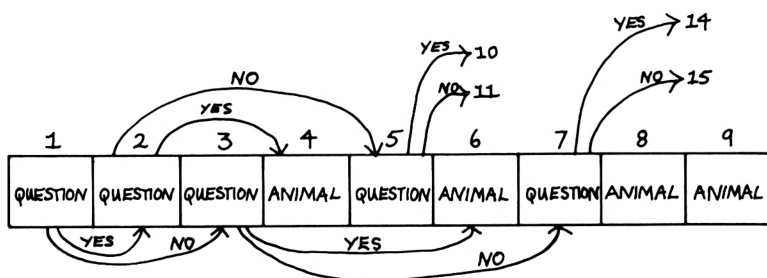


Figure 4.8. Mapping a binary tree structure with a single-dimension array

System functions

The Amstrad has a number of system, or built-in functions which are of great value to the programmer. In this section we'll introduce some of them, and these and others will be explained in routines in later chapters.

ERASE

Arrays can take up a lot of memory, and sometimes it can be useful to reclaim this for use by other arrays, for another program to be merged from tape, and so on. The command ERASE wipes out the specified arrays, freeing the memory they occupied and

allowing the programmer to use the space for other purposes.
The command is used like this:

```
1000 ERASE real.array
1010 ERASE integerarray%
1020 ERASE array$
```

Alternatively, ERASE may be followed by a list:

```
1000 ERASE real.array,integerarray%,array$
```

Any attempt to erase non-existent arrays will result in an 'Improper argument' error message.

READ and DATA

READ and DATA are essential tools for the Basic programmer. They allow you to store data in a program without having to use multiple assignment statements, like the following:

```
100 xcentre = 320
110 ycentre = 200
120 radius = 10
```

We could replace this simple set of statements with:

```
100 READ xcentre
110 READ ycentre
120 READ radius
130 DATA 320
140 DATA 200
150 DATA 10
```

This can be made even simpler because both READ and DATA may be followed by lists. The above becomes:

```
100 READ xcentre,ycentre,radius
110 DATA 320,200,10
```

To understand how READ and DATA operate, imagine a 'data pointer'. At the start of a program this 'points to' the first item of the first DATA statement (if any). If there are no DATA statements and you issue a READ, your program will stop with a 'Data

exhausted' error message. The same will happen if you try to READ more items than there are in the DATA statements. Every time a READ statement is encountered, the value or string pointed to by the data pointer is taken into memory and assigned to the variable indicated by the READ command. In the examples above 'READ xcentre' takes the DATA value (320), and assigns this to the variable 'xcentre', and the data pointer is moved to point to the next DATA item. The next READ takes the next DATA item (200), and assigns the value 200 to the variable 'ycentre'. The pointer now indicates the last item of DATA – the value 10 which is assigned to 'radius' with the last READ command.

READ can be used to assign strings to string variables, for example:

```
100 READ name$,address$,phones$
5000 DATA Fred Bloggs,13 The Crescent Bury
Lancs.,0893-65734
```

Note that DATA statements can appear anywhere in a program. As far as the computer is concerned, DATA statements form one long list, so it will work through them in order, from lowest program lines to highest. However, it's often a good idea to keep DATA statements near their associated READ comments to simplify a listing.

There is a problem with the example just given. How can you put commas into strings in DATA statements without having the Amstrad assume that the commas separate items in the DATA list? The answer is that strings may be enclosed in quotes, and the Amstrad takes whatever's between these as a string, including commas. The above could be re-written:

```
100 READ name$,address$,phones$
5000 DATA Fred Bloggs,"13, The Crescent, Bury,
Lancs.",0893-65734
```

You can mix string and numeric data types in a single DATA list, but you must be careful not to try to make the Amstrad READ a numeric value when the data pointer is pointing to a string. The following produces an error message:

```
1600 READ number
7000 DATA word
```

The problem arises because data-types aren't entirely interchangeable, as explained earlier. The problem will not arise if a 'READ word$' encounters a numerical value, because the character of the number will be assigned to the string variable. It's curious that the error message generated by this situation isn't 'Type mismatch', as you might expect. The Amstrad will report a 'Syntax error' in the line number of the DATA statement where the problem arises. This makes life awkward when you're trying to de-bug a program whose various sections may all access common DATA statements. The moral is to make sure that READ and DATA statements tally.

READ and DATA are very useful for filling arrays. In a simple database you could define an array, e.g. 'DIM data$(100,3)' to hold information about records, books or stamps. The information needed for the arrays would be kept in DATA statements, and READ into the array at the start of the program.

Let's look at a simple example, where we want to store album titles, artists and type of music (e.g. pop, jazz, classical). The array will be three columns 'wide' by up to 100 rows 'deep', and we want to read information into it from DATA statements. The routine to load the array could begin like this:

```
10 DIM array$(100,3)
20 row = 1
30 READ array$(row,1)
40 IF array$(row,1) = "ZZZ" THEN 100
50 REM "ZZZ" signifies end of data
60 READ array$(row,2),array$(row,3)
70 row = row +1
80 GOTO 30
100 REM all DATA read in
110 'Rest of program
1000 REM data in order title,artist,type
1010 DATA Beat,King Crimson,Jazz
1020 DATA Holland,Beach Boys,Pop
1030 DATA Sketches of Spain,Miles Davis/Gil
Evans,Jazz
1040 DATA The Four Seasons,Vivaldi,Classical
1050 DATA ZZZ
```

To make life even easier, you could incorporate a coding system, using the letters 'C', 'P' or 'J' in the DATA statements according to the type of music. You could then decode them like this:

```
1000 type$ = array$(row,3)
1010 IF type$ = "C" THEN type$ = Classical"
1020 IF type$ = "P" THEN type$ = "Pop"
1030 IF type$ = "J" THEN type$ = "Jazz"
1040 array$(row,3) = type$
```

Filling two-dimensional arrays is often easier with nested FOR...NEXT loops. Using the example data above we can see that there are four rows, and we know that each row has three entries. Therefore, loading the array can be done with:

```
100 FOR row = 1 TO 4
110  FOR column = 1 TO 3
120   READ array$(row,column)
130  NEXT column
140 NEXT row
```

But this technique relies on your counting up the number of rows in your DATA statements, rather than just setting an end of data marker.

RESTORE

One very useful facility associated with READ is that you can reset the data pointer to the first item of data in any DATA statement. 'RESTORE 1000' moves the data pointer so that it is indicating the first item of data in a DATA statement at line 1000. If there isn't a line 1000, a 'Line does not exist' error message will be generated. However, if the statement at line 1000 isn't a DATA statement, the DATA pointer will run through the program to the next DATA statement, and point to that. Unfortunately, you cannot use RESTORE with an expression such as 'RESTORE n * 1000', as numbers are the only acceptable argument. This rather restricts what you can do with READ and DATA, but RESTORE does allow you to define or redefine all sorts of programming details. You could use it to reload an array when a program has to be re-run; different data can be assigned to variables at will; specific location descriptions can be READ and PRINTed in an adventure, and so on. Note that in the latter case, it may be wasteful to have data stored in DATA statements in a program and in an array as well. If data need only be accessed, keep information in DATA statements, if it's to be manipulated as well, then it should be READ into arrays.

TIME

When the Amstrad is switched on, a counter begins to count upwards from zero. Four bytes (32 bits) are used for the counter, which means that its highest count is roughly 4.3E+09, and on reaching this it begins again at zero. Knowing that the counter is updated every 1/300 second, you can use it quite easily to maintain an accurate 'real time' clock or simply to time events, like setting a time limit to a game. To time an event you must store the value of the timer in a real numeric variable before the event begins, as in: 'start = TIME'. Then, when the event has finished, you subtract this value from the current time: 'length = TIME – start', and divide this by 300 for seconds: 'secs = length/300'. Note that the clock is not updated during cassette operations.

TIME can be used for a number of purposes. In the listing which follows it's used to time reactions. The program will clear the screen and then, after a random interval, display the message 'HIT KEY'. It counts the time between displaying the prompt and the next key press and displays the result, together with a running average. You'll find that most people have a reaction time of about 0.25 seconds for this task, but this can be reduced with practice.

```
10 ON BREAK GOSUB 510
20 ON ERROR GOTO 510
30 MODE 0:PEN 7:PAPER 6
40 INK 4,26,15:SPEED INK 10,10:CLS
50 LOCATE 4,1
60 PRINT"Reaction Timer"
70 DEF FN rand(n) = INT(RND(1) * n) + 1
80 LOCATE 2,5:PEN 3
90 PRINT "When the message";
100 LOCATE 2,7:PEN 1
110 PRINT "'HIT KEY' appears,"
120 PEN 3
130 LOCATE 3,9::PRINT "press any key"
140 PEN 4
150 LOCATE 1,24
160 PRINT"Press space to start"
170 IF INKEY(47) = -1 THEN 170
180 CLS
```

```
190 SPEED KEY 1,1
200 FOR pause = 1 TO FNrand(10000):NEXT
210 CALL &BB03:MODE 0
220 LOCATE 6,10:PRINT"HIT KEY"
230 const = TIME
240 a$ = INKEY$:IF a$ = "" THEN 240
250 reactiontime = TIME - const
260 goes = goes + 1
270 rtime = reactiontime / 300
280 totime = totime + rtime
290 avtime = totime / goes
300 MODE 1::LOCATE 10,10
310 PRINT"Your reaction time was"
320 LOCATE 10,12:PRINT USING"##.###";rtime;
330 PRINT" seconds";
340 LOCATE 10,15
350 PRINT"Average so far: ";
360 PRINT USING"##.###";avtime;
370 PRINT" seconds";
380 LOCATE 13,25:PRINT"Another?...y/n"
390 CALL &BB00
400 encore$ = UPPER$(INKEY$)
410 IF encore$ = "Y" THEN 180
420 IF encore$ <> "N" THEN 400
430 CLS
440 LOCATE 6,10
450 PRINT"Your average reaction time"
460 LOCATE 11,12
470 PRINT"over";goes;"tries was"
480 LOCATE 10,15
490 PRINT avtime;"seconds"
500 'Restore keyboard to normal
510 CALL &BB00:CALL &BB03
```

Trigonometric functions

The Amstrad has a variety of trigonometric functions which are of particular value in graphics routines such as drawing circles, spirals and the like. SIN, COS, TAN, arctangent (ATN) are all implemented (see Chapter 8). Most other home computer Basics require you to deal with radians, rather than degrees (which often requires a degree to radian conversion function or routine), but the Amstrad allows you to swap between these with the two instructions DEG and RAD. Radians are units of measurement, like degrees. The radian is based on the numerical construct PI, there are 2 * PI radians in a circle, so one radian is about 360/6.283

(57.296) degrees. To convert degrees to radians, you could use the relationship 'radians = (PI/180) * degrees', but as the Amstrad allows you to select one system or the other it's not worth it unless you want to make your programs 'portable', i.e. easy to convert to run on another machine.

Memory

As you know, the Amstrad has 64K of memory. 1K is 1024 bytes, not 1000 because computer people like to think in terms of twos, *and 1024 is 2 ^ 10, so your machine has 64 * 1024 bytes* (65536) available for your use. Each byte can hold a number between 0 and 255, and to get numbers outside this range Basic may use two bytes together and other tricks. More information on this is given in Chapter 7.

PEEK and POKE

These two words allow you direct access to the Amstrad's RAM. As their names suggest, PEEK allows you to examine the contents of an address or memory location, while POKE allows you to place a value in the memory. Thus the command line, 'POKE 43800,255' places a value of 255 in address 43800, and 'PRINT PEEK(43800)' displays the contents of the address given in brackets.

The Amstrad has two forms of memory: ROM and RAM. ROM stands for Read Only Memory, RAM for Random Access Memory. The Amstrad's ROM is built-in in the sense that it is always there – turning the machine off doesn't erase information from it and the information in it cannot be altered. RAM is used for the temporary storage of information such as programs and their data, but the contents of RAM are lost when the Amstrad is switched off. This is why we have to rely on mass-storage devices like tape recorders to store programs. You cannot alter the ROM at all, it contains a complex program which interprets your programs and direct commands, can read from or write to the cassette deck, control a printer and so on. ROM really is read only – information can be extracted from it, various of its subroutines called up, but you cannot put information into it as you can with RAM.

RAM can be imagined as separate compartments – one block

of it is used to store programs, another to store data, yet another part maps out the video display, so any changes here alter what you see on the screen. From this you should be able to see that POKE will allow you to alter the screen displays, while PEEK will provide information about what's on the screen. POKE is often used to load machine code programs into RAM, by READing machine code instructions and data from DATA statements, then placing them in a spare set of addresses somewhere in RAM (see Chapter 7).

Each memory cell is called an address or location. For example, the top left-hand point of the screen is at memory address &C000, while the highest address available to Basic programs and their data is 43907. This can be altered with the word MEMORY, and again this is discussed in Chapter 7.

Note that you cannot POKE a number larger than 255 into a memory address, nor will PEEK produce a number greater than 255.

Database program

Finally, here is a fairly simple program which demonstrates many of the concepts explained in this chapter. It's a three-column database which could be used as an address book, record collection index, etc. It will save data to tape and load it in again, so the same program could be used for a variety of applications. It is intended as a vehicle for demonstrating programming techniques, not a complete program. You are invited to make amendments to improve it and tailor it to your needs.

```
10 ' Simple Database
20 'with tape facilities
30 '
40 'Initialisation
50 CLS
60 DIM array$(300,10)
70 numberofentries=0
80 numberofcols=3
90 '
100 field$(1)="Name"
110 field$(2)="Address"
120 field$(3)="Phone"
130 '
140 FOR i=1 TO numberofcols
150 find$=find$+LEFT$(field$(i),1)
```

```
160 NEXT
170 '
180 option$="LSFEQ"
190 'Load,Save,Find,Enter,Quit
200 submenu$="NADM"
210 'Next/Alter/Delete/Menu
220 '
230 REM Entry to Main Menu
240 menucol=17:CLS
250 LOCATE 17,1:PRINT"Main Menu";
260 LOCATE menucol,5:PRINT"L....Load"
270 LOCATE menucol,7:PRINT"S....Save"
280 LOCATE menucol,9:PRINT"F....Find"
290 LOCATE menucol,11:PRINT"E....Enter"
300 LOCATE menucol,15:PRINT"Q...Quit"
310 LOCATE 19,24:PRINT"Select"
320 '
330 'Clear keyboard buffer
340 GOSUB 1450
350 akey$=UPPER$(INKEY$)
360 IF akey$="" THEN 350
370 option=INSTR(option$,akey$)
380 IF option=0 THEN 350
390 IF option=5 THEN GOTO 1150
400 CLS
410 ON option GOSUB 450,560,670,1030
420 GOTO 240
430 '
440 ' Load
450 OPENIN "datafile"
460 INPUT#9,numberofrows
470 numberofentries=numberofrows
480 FOR row=1 TO numberofrows
490 FOR col=1 TO numberofcols
500 IF EOF THEN 530
510 LINE INPUT#9,array$(row,col)
520 NEXT:NEXT
530 CLOSEIN:RETURN
540 '
550 'Save
560 IF numberofentries<>0 THEN 590
570 LOCATE 13,10:PRINT"No data to save"
580 GOSUB 1290:RETURN
590 OPENOUT"datafile"
600 PRINT#9,numberofentries
610 FOR row=1 TO numberofentries
620 FOR col=1 TO numberofcols
630 PRINT#9,array$(row,col)
640 NEXT:NEXT:CLOSEOUT:RETURN
```

```
650 '
660 'Find
670 'IF numberofentries=0 THEN RETURN
680 LOCATE 17,1:PRINT"Find Menu"
690 FOR i=1 TO numberofcols
700 LOCATE menucol,i*2+5
710 PRINT LEFT$(field$(i),1);
720 PRINT"...."; field$(i)
730 NEXT
740 LOCATE 19,24:PRINT"Select"
750 GOSUB 1450
760 akey$=UPPER$(INKEY$)
770 IF akey$="" THEN 760
780 field=INSTR(find$,akey$)
790 IF field=0 THEN 760
800 '
810 CLS:LOCATE 1,10
820 PRINT"Enter ";field$(field);
830 PRINT" to find";
840 INPUT pattern$
850 '
860 'Start on first row
870 row=1
880 afind=INSTR(array$(row,field),pattern$)
890 'No find - so do next row
900 IF afind=0 THEN 970
910 '
920 GOSUB 1380:'Display
930 GOSUB 1490:'Sub Menu
940 '
950 IF choice$="M" THEN RETURN
960 'Next row
970 row=row+1
980 IF row<=numberofentries THEN 880
990 RETURN
1000 '
1010 'Enter
1020 CLS
1030 numberofentries=numberofentries+1
1040 FOR col=1 TO numberofcols
1050 LOCATE 1,col*2+5
1060 PRINT field$(col);"....";
1070 LINE INPUT array$(numberofentries,col)
1080 NEXT
1090 LOCATE 15,20:PRINT"More...Y/N";
1100 GOSUB 1230
1110 IF y.n$="Y" THEN 1020
1120 RETURN
1130 '
```

```
1140 'Quit - Sure? If not, do menu
1150 CLS
1160 LOCATE 10,10
1170 PRINT"Are you sure...Y/N"
1180 GOSUB 1230
1190 IF y.n$="N" THEN 240
1200 CLS:END
1210 '
1220 ' Get yes/no as upper case Y or N
1230 y.n$=UPPER$(INKEY$)
1240 IF y.n$="" THEN 1230
1250 IF y.n$<>"Y" AND y.n$<>"N" THEN 1230
1260 RETURN
1270 '
1280 'Press Space routine
1290 LOCATE 1,24:PRINT SPACE$(38);
1300 LOCATE 7,24
1310 PRINT"Press Space Bar to continue";
1320 GOSUB 1450
1330 a$=INKEY$:IF a$="" THEN 1330
1340 IF a$=CHR$(32) THEN RETURN
1350 GOTO 1330
1360 '
1370 'display a record
1380 CLS
1390 FOR col=1 TO numberofcols
1400 PRINT field$(col),array$(row,col)
1410 NEXT
1420 RETURN
1430 '
1440 'Clear Keyboard Buffer
1450 CALL &BB03:RETURN
1460 '
1470 'Next/Alter/Delete/Menu
1480 'as Find routine sub-menu
1490 LOCATE 1,24:PRINT SPACE$(38);
1500 LOCATE 9,24
1510 PRINT"Next/Alter/Delete/Menu"
1520 GOSUB 1450
1530 choice$=UPPER$(INKEY$)
1540 IF choice$="" THEN 1530
1550 choice=INSTR(submenu$,choice$)
1560 IF choice=0 THEN 1530
1570 'Menu or Next
1580 IF choice$="M" OR choice$="N" THEN RETURN
1590 '
1600 IF choice$="D" THEN GOSUB 1710:RETURN
1610 '
1620 'Amend Entry
```

```
1630 CLS
1640 FOR col=1 TO numberofcols
1650 LOCATE 1,col*2+5
1660 PRINT"Enter new ";field$(col);"...";
1670 INPUT array$(row,col)
1680 NEXT:RETURN
1690 '
1700 'Delete row
1710 FOR col=1 TO numberofcols
1720 array$(row,col)=""
1730 NEXT:CLS:RETURN
```

5
Designing a Game

In this chapter we'll show how to design and code a simple game using text and text-handling commands. The game uses many of the ideas outlined in previous chapters, and following the program lines step by step will help you gain familiarity with programming.

The technique we'll use is to explain programming problems that have to be solved for the game, then give numbered program lines which you can type in. Because we can't always tackle the problems in the order in which the program lines appear, you'll find that the line numbers won't necessarily be in sequence – but as the Amstrad will sort these out you can enter them as you read them. At the end of this chapter there's a complete listing of the game so that you can check what you've entered, or if you don't want to read about how the game works you can turn straight to it now and start typing it in.

In the game you move a snake around the screen, gobbling up points as you go. The snake must not run into the edges of the screen, or try to double back on itself, or run over its trailing body. It can get quite difficult because the snake grows in length by the number of points eaten. Points will appear at random on the screen, and if you don't go after them quickly they will count down to zero and vanish. You could add a time limit to the game quite easily, using TIME.

Coding the game

A border

The first thing to do is to draw a border round the screen. For this we'll use the various arrow characters supplied by ASCII codes 240 to 243. The top and bottom borders need to be 39 characters long – the width of the screen in MODE 1. Rather than use a

FOR...NEXT loop to print 39 downward-pointing arrows, CHR$(241), we can make use of the string function STRING$, which produces strings of a given length of a single character. The top border will be STRING$(39,241), the bottom will be STRING$(39,240). The code for the top and bottom borders is:

```
240 LOCATE 1,1:PRINT STRING$(39,241);
260 LOCATE 1,25:PRINT STRING$(39,240);
```

These STRING$ functions could also be written as STRING$(39,CHR$(241)) and STRING$(39,CHR$(240)), because the second argument in brackets can either be the character itself or its ASCII code.

The sides of the border pose rather more of a problem. The easiest method is to use a FOR...NEXT loop, stepping down each row at a time and printing a right-pointing arrow at the left of the screen, a left-pointing one at the right:

```
280 FOR row = 2 TO 24
300 LOCATE 1,row PRINT CHR$(243);
310 LOCATE 39,row:PRINT CHR$(242);
320 NEXT
```

Part of the border at the top will be used to display the score:

```
330 LOCATE 15,1:PRINT "Score = 0";
340 score = 0
```

We don't actually have to a assign a value to 'score' explicitly because the very first time the Amstrad meets a new variable, to which no value has been assigned, it automatically assigns zero to it.

The characters

Next we'll need to define our characters. We'll use a lower case 'x' for the head and CHR$(207), a patterned square, for the body:

```
420 top = ASC("x"):body = 207
430 head$ = CHR$(top):body$ = CHR$(body)
```

The reason for using variables here is that later we'll need to check whether certain characters are on the screen at certain

locations, and to do this we'll need to use their ASCII values. It also allows us to alter the characters used by changing just one line rather than many throughout the program.

PEEKing the screen

One facility not directly provided by the Basic on the Amstrad is a screen PEEK. Sometimes it's very useful to be able to establish the ASCII code of the contents of any given screen location. However, as described in Chapter 7, it's possible to write a short machine code routine to do this for us, and we put this at the start of the program:

```
40 MEMORY 43798
50  address = 43800
60 DATA 197,213,229,245,205,96,187
70 DATA 50,23,171,241,225,209,193,201
80 DATA 0
90 READ value
100 IF value = 0 THEN 160
110 POKE address,value
120 address = address + 1
130 GOTO 90
```

Now, when we want to PEEK the screen, we LOCATE the cursor, issue a CALL 43800 to run our machine code, and PEEK(43799) will reveal the contents of the character cell under the cursor.

Movement

Next we need to define where the snake is to start, and its initial direction of travel. Before we do this, let's take a close look at how we're going to move the snake around the screen.

It would be impossibly slow to PRINT the head and body characters as the snake moves. But what we can do is keep track of the head and tail only, because as the head moves, it gets new co-ordinates, while the tail takes on the co-ordinates of the next section towards the head, i.e. earlier head co-ordinates. The method we'll use involves a two-dimensional integer array and two 'pointers' to the array, one for the head's co-ordinates, the other for those of the tail. The array element 'snake%(n,1)' gives the column of part of the snake, 'snake%(n,2)' gives the row.

First we set a limit to the maximum length of the snake:

```
160 maxlen = 300
```

Then we dimension the array:

```
190 DIM snake%(maxlen,2)
```

The variable 'head' will point to the co-ordinates of the snake's head. At first we make this point to the first array element of 'snake%'. The variable 'tail' initially points to the third element of 'snake%' (we start the snake off as three units long: a head, mid-section and tail):

```
480 head = 1:tail = 3
```

As the snake's head moves, we'll calculate new co-ordinates for it, subtract one from 'head' (to make it point to a new element), then put the new co-ordinates in the new element. If the value of 'head' drops below one, we'll force it to 'maxlen', so it starts using array elements at the other end of the array 'snake%', backing up to one again. The same procedure is used to keep track of the tail – we want the co-ordinates that were those of the head to become those of the tail, so we subtract one from 'tail' when we want to find the next screen column and row co-ordinates for the tail's location. In this way, the 'tail' pointer chases the 'head' pointer around the array. This makes life very easy when we want to increase the length of the snake – all we have to do is shift the head pointer along the row as many times as are needed, updating the co-ordinates as we go and placing them in 'snake%', and leaving 'tail' where it is. The only problem here is that if the snake becomes too long (over 'maxlen' units), the new 'head' co-ordinates will overwrite those of the body. We haven't catered for this, and if it becomes a problem you could increase the size of 'maxlen' in line 160.

Initially we make the snake three units long, and put the starting co-ordinates for the head, mid-section and tail into the array:

```
510 snake%(head,1) = 20:snake%(head,2) = 12
520 'starting location for head
540 snake%(2,1) = 20:snake%(2,2) = 13
550 'middle section
```

```
570 snake%(tail,1) = 20:snake%(tail,2) = 14
580'last section - tail
```

The numbers used make the snake's head appear roughly in the centre of the screen, but you can alter this.

As we need to keep track of the column and row of the snake's head at all times, for various calculations, we use four variables:

```
600 oldcol = snake%(head,1)
610 oldrow = snake%(head,2)
640 newcol = oldcol
650 newrow = oldrow
```

Finally, we draw the snake in its starting position:

```
760 LOCATE snake%(head,1),snake%(head,2)
770 PRINT head$;
790 LOCATE snake%(2,1),snake%(2,2)
800 PRINT tail$;
820 LOCATE snake%(tail,1),snake%(tail,2)
830 PRINT tail$;
```

To move the snake, we begin with the tail. First we blank out the existing tail:

```
930 LOCATE snake%(tail,1),snake%(tail,2)
940 PRINT " ";
```

We decrement the pointer 'tail' so that it references the next portion of the body toward the head, not forgetting to force the value of 'tail' to the other end of the array if it drops below one:

```
970 tail = tail-1
1000 IF tail = 0 THEN tail = maxlen
```

And, of course, we have to replace the head with a tail or body character:

```
1040 LOCATE snake%(head,1),snake%(head,2)
1050 PRINT tail$;
```

Changing direction

We'll use the keys "z", "x", "/" and ";" for left, right, up and down:

```
460 keyboard$ = "zx/;"
```

The lines to read the keyboard are:

```
870 akey$ = INKEY$
880 IF akey$ = "" THEN akey$ = lastkey$
```

Here we use the variable 'lastkey$' to log the character of the last key pressed. Then, if no key is currently pressed, we put the value of the last key press into the variable for the current key press, so that the snake will continue moving in its last direction.

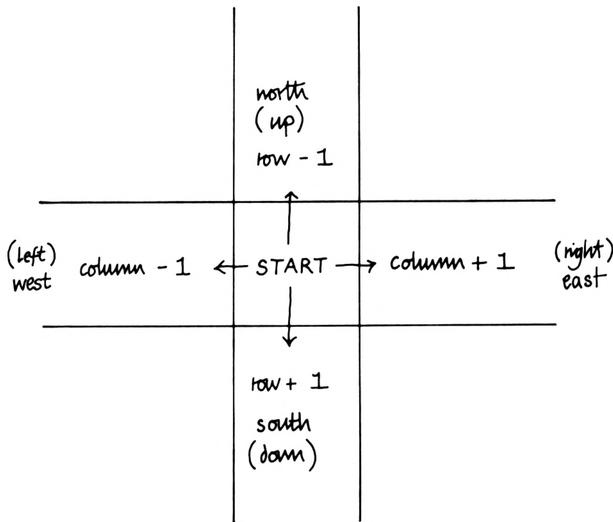


Figure 5.1. Adjusting row/column co-ordinates for moving up, down, right and left

To translate a key press into movement of the snake's head we use the fact that to move right or left we add or subtract one from the current column, while moving up or down involves adding or subtracting one from the current row (see Figure 5.1). We could use code like this:

```
IF akey$ = "z" THEN newcol = oldcol - 1
IF akey$ = "x" THEN newcol = oldcol + 1
IF akey$ = "/" THEN newrow = oldrow + 1
IF akey$ = ";" THEN newrow = oldrow - 1
```

However, the same result can be achieved much faster and in fewer lines with:

```
1150 newcol = oldcol + (akey$ = "z") - (akey$
= "x")
1160 newrow = oldrow - (akey$ = "/") + (akey$ =
";")
```

This technique is discussed more fully in Chapter 6.

Notice that we don't directly update the current co-ordinates of the head as soon as we've read the keyboard. This is because we need to see where the new co-ordinates would put the head – there may be something there already, like part of the body. The variables 'newcol' and 'newrow' are used as temporary stores for the next possible location of the head.

Another reason for using intermediate variables is that we have to make sure that the snake doesn't run into the border, which marks the end of the game. Therefore we add:

```
1180 IF newcol<2 OR newcol >38 THEN 2400
1190 IF newrow<2 OR newrow > 24 THEN 2400
2400 GOTO 200
```

Where line 200 marks the 're-entry point' for the start of another game. Note that it's not necessary to redefine the machine code routine or the array 'snake%', so we don't use RUN.

To test whether the head would be moved to a location which is already occupied by part of the body – an illegal move according to the rules – we move the cursor to the potential location, call the machine code routine, and the contents of address 43799 reveal the ASCII code of any character there:

```
1210 LOCATE newcol,newrow
1220 CALL 43800:check = PEEK(43799)
1230 IF check = top OR check = body THEN 2400
```

Now we can draw the head in its new location, update the head pointer, put the new co-ordinates into the array and update 'lastkey$':

```
1080 head = head -1
1090 IF head = 0 THEN head = maxlen
1250 PRINT head$;
1280 snake%(head,1) = newcol
1290 snake%(head,2) = newrow
```

```
1320 oldcol = newcol
1330 oldrow = newrow
1390 IF INSTR(keyboard$,akey$) = 0 THEN 1450
1420 lastkey$ =akey$
1540 GOTO 870
```

Note that we have to skip the reassignment of 'akey$' to 'lastkey$' in line 1390, because an illegal key press (i.e. not one of "z", "x", ";" or "/") will cause problems in the Boolean logic of 1150 and 1160 (Boolean logic is dealt with in Chapter 6). INSTR will return zero if the second string argument given is not contained in the first.

We'll need to give the snake an initial movement – upwards in this case:

```
700 lastkey$ = ";"
```

You can now test out the routines entered so far. Just add:

```
200 REM
210 MODE 1
```

and RUN the program. At this stage you may find syntax errors, etc., so check carefully what you've entered with the lines shown so far. The most common errors will be mis-spellings, e.g. 'rancol' instead of 'randcol', lines missed, incorrect values typed in and so on. Error messages like 'Improper argument' often betray these. For example, if you have an 'Improper argument' in a line which contains a LOCATE, PRINT the values of the variables used as arguments (e.g. ?randcol). If either of them is zero, check back through the program to find out why.

You should now have a 3-unit shape under keyboard control. The next stages are to enter the routines for scoring points, making the snake 'grow' and so on. You'll notice when you've finished that all the necessary calculations and checking slow down the program slightly.

Generating random numbers

A fundamental subroutine in this game is one to produce and display a random number in the range 1–9, at a random location on the screen. To do this we can make use of the DEF FN facility

to define a function which will generate random numbers in the range 1 to whatever number we pass to it. We do this early in the program, in the 'variable declaration' section:

```
370 DEF FNR(n) = INT(RND(1)*n)+1
```

The random number routine goes like this:

```
1580 randnum = FNr(9)
1600 randcol = FNr(36)+1
1620 randrow = FNr(22)+1
1660 randnum$ = STR$(randnum)
1680 randnum$ = MID$(randnum$,2,1)
1700 LOCATE randcol,randrow
1720 CALL 43800:check = PEEK(43799)
1730 IF check = body OR check = top THEN 1600
1740 PRINT randnum$;
1780 RETURN
```

Line 1580 selects a number between 1 and 9 at random, but we have to convert the variable 'randnum' to its string representation, otherwise it will be printed with its leading and trailing spaces, which could overwrite the snake. This is done using a combination of STR$ and MID$. First we use STR$ to convert the variable randnum to a string of three characters, line 1660. Then in line 1680 we take the middle character using MID$ (i.e. the character of the number itself), PRINT it and RETURN.

We don't want random numbers appearing over any part of the snake, so we use the machine code routine to test a randomly chosen location – lines 1700 to 1730. If part of the snake is there, the routine jumps back to select a new random column and row, and tries again.

We need some way of telling whether a random number is on the screen, because we need to re-use the random number generation/display subroutine for new numbers when the snake runs over a number and erases it. To keep track of random numbers we can introduce a 'flag' which will only ever have two values: 0 or 1, and which we will use to indicate whether a number is on-screen (1) or not (0). We set this flag to zero to start with, by the simple expedient of not declaring it! During the program we will need:

```
1450 IF randflag = 0 THEN GOSUB 1580
```

This calls the routine to select a random number, convert it to a string and display it if no number is currently present.

We'll also have to add a line to the random number routine to set 'randflag' to one to indicate that a number is being displayed:

```
1760 randflag = 1
```

We want to know if the snake has been moved on to a random number, and if so call up a subroutine to update the score, so we add to the snake movement routine:

```
1350 IF newcol = randcol AND newrow = randrow
THEN GOSUB 2000
```

Alternatively we could use PEEK(43799) here, testing for the random number itself, so having LOCATEd the cursor over the appropriate screen cell, line 1350 could read: 'IF randflag = 1 AND PEEK(43799) = randnum—48 then GOSUB 2000'. Note that 48 is subtracted from 'randnum' because the machine code routine returns the ASCII code of characters on the screen. The ASCII code of zero is 48, one is 49, and nine is 57.

Decrementing the numbers

Part of the game is that the numbers which appear are reduced over time, so high scores aren't too easy to get. For this we'll need a routine which is called at random intervals:

```
1500 IF randflag = 1 AND Fnr(20)<3 THEN GOSUB
1820
```

This sends control to a routine at line 1820 if a number is on-screen and a randomly chosen number between 1 and 20 is less than 3. To make numbers reduce at a slower rate, use larger values for the argument in Fnr in line 1500. The routine at 1820 begins like this:

```
1820 randnum = randnum-1
1840 IF randnum>0 THEN GOSUB 1660:RETURN
1880 LOCATE randcol,randrow
1890 PRINT " ";
1910 randflag = 0
1930 randcol = 0:randrow = 0
1950 RETURN
```

This first section subtracts one from the random number. If the result is zero then the number is blanked out, 'randflag' set to zero to indicate no number is present, 'randcol' and 'randrow' set to zero, and the routine returns. If 'randnum' is not zero, a further subroutine at 1660 is called, and this displays the reduced number. In fact this line number is part of the random number display routine entered earlier, so we don't need to write the code to display the new number all over again! What we do is to jump into a subroutine part-way through – a useful trick which can save a lot of code. Actually, this will slow down the program a little, because the number display routine checks to see that part of the snake isn't at the location given, which isn't necessary here, but moving the test lines to an earlier point in the routine would eliminate that. Alternatively, the number-string conversion routine could be rewritten and spliced in here. Either way, the time saving is hardly worthwhile.

Scoring

The 'update score' routine itself is quite simple, all we have to do is add the random number to the score, display the new score and reset the random number flag to zero to show that no number is now present on screen.

```
2000 score = score + randnum
2020 LOCATE 22,1:PRINT score;
2040 randflag = 0
2050 randcol = 0:randrow = 0
2370 RETURN
```

The main body of the routine deals with making the snake 'grow' by the number of points 'eaten'. To do this we simply repeat the bulk of the code we used for making the snake move earlier, but missing out the 'erase tail' code:

```
2080 FOR inc = 1 TO randnum
2150 akey$ = inkey$
2160 IF akey$ = "" THEN akey$ = lastkey$
2170 newcol = oldcol + (akey$ = "z") - (akey$ =
"x")
2180 newrow = oldrow - (akey$ = "/") + (akey$ =
";")
2190 IF newcol<2 OR newcol >38 THEN 2400
```

```
2200 IF newrow<2 OR newrow >24 THEN 2400
2210 LOCATE snake%(head,1),snake%(head,2)
2220 PRINT tail$;
2230 LOCATE newcol,newrow
2240 CALL 43800:check = PEEK(43799)
2250 IF check = top OR check = body THEN 2400
2260 PRINT head$;
2280 head = head -1
2290 IF head=0 THEN head = maxlen
2300 snake%(head,1) = newcol
2310 snake%(head,2) = newrow
2320 oldcol = newcol
2330 oldrow = newrow
2340 IF INSTR(keyboard$,akey$) = 0 THEN 2360
2350 lastkey$ = akey$
2360 NEXT
2370 RETURN
```

You now have a complete program. There's a full listing at the end of this chapter, in which the line numbers are the same as the ones given here. It also contains many REM statements to help you understand how the game works.

If you want to make the game slightly faster, you could alter as many variables as possible to integer: 'head', 'tail', 'top' and 'body' are prime candidates. You could also condense statements so that you cram as many as possible on to one program line. Using integer variables for the screen limits in lines 1170, 1180, 2190 and 2200 will also help. Leaving out REMs will speed up the program, but at the expense of readability.

You should now find it relatively easy to modify the program to your taste. You could set a time limit to the game or use your own redefined characters, and later you'll be able to add sound. Experimenting with someone else's program is a good way to learn Basic programming. The best way is to set yourself a problem and try to code it.

The complete listing

```
10 REM  ++++++++++ Snake +++++++++++
20 ' Set up machine code routine
30 ' to PEEK the text screen
40 MEMORY 43798
50 address = 43800
60 DATA 197,213,229,245,205,96,187
70 DATA 50,23,171,241,225,209,193,201
```

```
80 DATA 0
90 READ value
100 IF value = 0 THEN 160
110 POKE address,value
120 address = address+1
130 GOTO 90
140 '
150 'Set up snake array
160 maxlen = 300
170 'maxlen is longest snake can be
180 DIM snake%(maxlen,2)
190 'snake array, holds coords of head
200 MODE 1
210 '
220 '          Draw a Border
230 LOCATE 1,1:PRINT STRING$(39,241);
240 '            Top line
250 LOCATE 1,25:PRINT STRING$(39,240);
260 '           Bottom line
270 FOR row = 2 TO 24
280 '      Sides - Left, Right
290 LOCATE 1,row:PRINT CHR$(243);
300 LOCATE 39,row:PRINT CHR$(242);
310 NEXT
320 '
330 LOCATE 15,1:PRINT"Score = 0";
340 score = 0
350 '______ Set up variables etc _______
360 '
370 DEF FNr(n) = INT(RND(1)*n)+1
380 ' user function - generates random
390 ' numbers in the range 1 to n
400 ' and tail: ,1 is col ,2 is row
410 '
420 top = ASC("x"):body = 207
430 head$ = CHR$(top):tail$ = CHR$(body)
440 'snake characters
450 '
460 keyboard$ = "zx/;"
470 ' key controls - left, right, up, down
480 head = 1:tail = 3
490 'head & tail point to array snake%
500 '
510 snake%(head,1) = 20:snake%(head,2) = 12
520 ' starting location for head
530 '
540 snake%(2,1) = 20:snake%(2,2) = 13
550 ' middle section
560 '
```

```
570 snake%(tail,1) = 20:snake%(tail,2) = 14
580 ' last section - tail
590 '
600 oldcol = snake%(head,1)
610 oldrow = snake%(head,2)
620 ' oldcol/row are col/row coords
630 ' for snake's head
640 newcol = oldcol
650 newrow = oldrow
660 ' newcol/row are for updating
670 ' head coords - see main routine
680 '
690 '
700 lastkey$ = ";"
710 ' snake starts heading up the screen
720 '--------- End of Definitions -----
730 '
740 '            Draw the Snake
750 'print snake:-head,middle & tail
760 LOCATE snake%(head,1),snake%(head,2)
770 PRINT head$;
780 '
790 LOCATE snake%(2,1),snake%(2,2)
800 PRINT tail$;
810 '
820 LOCATE snake%(tail,1),snake%(tail,2)
830 PRINT tail$;
840 '
850 ' Now start the game itself
860 ' !!!!!!!!! Main Routine !!!!!!!!!
870 '
880 akey$ = INKEY$
890 IF akey$ = "" THEN akey$ = lastkey$
900 ' if no key pressed, carry on in
910 ' previous direction
920 '
930 LOCATE snake%(tail,1),snake%(tail,2)
940 PRINT" ";
950 ' erase tail
960 '
970 tail = tail-1
980 ' decrement tail pointer
990 '
1000 IF tail = 0 THEN tail = maxlen
1010 'force tail to far end of array
1020 'if tail points to zero
1030 '
1040 LOCATE snake%(head,1),snake%(head,2)
1050 PRINT tail$;
```

```
1060 '
1070 'replace head character with tail
1080 head = head-1
1090 ' decrement head pointer
1100 IF head = 0 THEN head = maxlen
1110 'force to far end if points to start
1120 'Update row and column values
1130 'ie location of head
1140 'Using Boolean logic
1150 newcol = oldcol + (akey$ = "z") - (akey$ = "x
")
1160 newrow = oldrow - (akey$= "/") + (akey$ = ";"
)
1170 '
1180 IF newcol<2 OR newcol>38 THEN 2400
1190 IF newrow<2 OR newrow>24 THEN 2400
1200 ' if off screen, end of game
1210 LOCATE newcol,newrow
1220 CALL 43800:check = PEEK(43799)
1230 IF check = top OR check = body THEN 2400
1240 'Run into self
1250 PRINT head$;
1260 ' redraw head in new location
1270 '
1280 snake%(head,1) = newcol
1290 snake%(head,2) = newrow
1300 ' update coords in snake array
1310 '
1320 oldcol = newcol
1330 oldrow = newrow
1340 '
1350 IF newcol = randcol AND newrow = randrow THEN
 GOSUB 2000
1360 ' hit the random number
1370 ' so do score routine
1380 '
1390 IF INSTR(keyboard$,akey$) = 0 THEN 1450
1400 ' if no valid key pressed,
1410 ' leave lastkey$ as is
1420 lastkey$ = akey$
1430 'set lastkey pressed to current key
1440 '
1450 IF randflag = 0 THEN GOSUB 1580
1460 ' randflag is zero if no random
1470 ' number is on the screen
1480 ' if it's zero - place a new one
1490 '
1500 IF randflag = 1 AND FNr(20)<3 THEN GOSUB 1820
1510 ' decrement any random number
```

```
1520 ' at random moments
1530 '
1540 GOTO 880
1550 'repeat main routine
1560 REM ------ Random numbers routine
1570 '
1580 randnum = FNr(9)
1590 ' randnum is 1 to 9
1600 randcol = FNr(36)+1
1610 ' randcol is 2 to 37
1620 randrow = FNr(22)+1
1630 ' randrow is 2 to 23
1640 '
1650 ' set cursor
1660 randnum$ = STR$(randnum)
1670 ' convert randnum to a string
1680 randnum$ = MID$(randnum$,2,1)
1690 ' strip leading and trailing spaces
1700 LOCATE randcol,randrow
1710 ' locate cursor
1720 CALL 43800:check = PEEK(43799)
1730 IF check = top OR check = body THEN 1600
1740 PRINT randnum$;
1750 ' display randnum
1760 randflag = 1
1770 ' set randflag - now have number on screen
1780          RETURN
1790 ' _____ End of Subroutine _____
1800 '
1810 'Decrement random number routine
1820 randnum = randnum-1
1830 ' take one from randnum
1840 IF randnum>0 THEN GOSUB 1660:RETURN
1850 ' if it's not zero, gosub display
1860 ' random number routine & return
1870 '
1880 LOCATE randcol, randrow
1890 PRINT" ";
1900 ' erase randnum
1910 randflag = 0
1920 ' set flag to zero (no randnum)
1930 randcol = 0:randrow = 0
1940 ' null these coords
1950          RETURN
1960 ' ____ End of Subroutine _____
1970 '
1980 REM increase length of snake
1990 '
2000 score = score + randnum
```

```
2010 ' update score
2020 LOCATE 22,1:PRINT score;
2030 ' and print it
2040 randflag = 0
2050 ' reset randflag (no randnum now)
2060 randcol = 0:randrow = 0
2070 ' reset coords
2080 FOR inc = 1 TO randnum
2090 ' for the value of the score
2100 ' do the following
2110 ' most of this is a repeat of
2120 ' the main routine
2130 'and allows the snake to grow
2140 ' to a maximum of maxlen units
2150 akey$ = INKEY$
2160 IF akey$ = "" THEN akey$ = lastkey$
2170 newcol = oldcol + (akey$ = "z") - (akey$ = "x
")
2180 newrow = oldrow - (akey$ = "/") + (akey$ = ";
")
2190 IF newcol<2 OR newcol>38 THEN 2400
2200 IF newrow<2 OR newrow>24 THEN 2400
2210 LOCATE snake%(head,1),snake%(head,2)
2220 PRINT tail$;
2230 LOCATE newcol,newrow
2240 CALL 43800:check = PEEK(43799)
2250 IF check = top OR check = body THEN 2400
2260 'Run into self
2270 PRINT head$;
2280 head = head-1
2290 IF head = 0 THEN head = maxlen
2300 snake%(head,1) = newcol
2310 snake%(head,2) = newrow
2320 oldcol = newcol
2330 oldrow = newrow
2340 IF INSTR(keyboard$,akey$) = 0 THEN 2360
2350 lastkey$ = akey$
2360 NEXT
2370                    RETURN
2380 _________END of Subroutine_______
2390 '
2400 GOTO 200
2410 ' run from start if snake's run
2420 ' off the screen
```

6
Numbers and Logic

Notation

To get the most out of your machine, from Basic and certainly from machine code, you ought to be familiar with three systems of counting; decimal, binary and hexadecimal. Since they operate by similar rules they're quite easy to understand.

All three systems use the mathematical convention of exponents. These are usually seen as superscripts following a number, e.g. 3^2. The superscript indicates how many times the number should be multiplied by itself, thus 3^2 (pronounced 'three squared') means $3 \times 3 = 9$. 4^3 (pronounced 'four cubed') means $4 \times 4 \times 4 = 64$. The Amstrad cannot use superscripts like this and uses the up-arrow symbol instead (unshifted pound sign, next to CLR on the top row). When printed, the symbol may appear as '^', depending on the type of printer. We'll use this throughout, so 4^2 means 'four squared', or 'four to the power of two', and so on.

Decimal

As with most systems of counting, the decimal system relies on the concept of position: the digit '3' has a different value in the two numbers 30 and 300. In the number 30, three means '3 times ten', while in the number 300, the three indicates 'three times 100'. So the value of '3' really depends on its position in a number. In the decimal system we call these positions 'units', 'tens', 'hundreds', 'thousands', and so on. If we label the digit positions from right to left, starting from zero, and raise 10 to the power of the position number, we get the sequence: 10^0, 10^1, 10^2, 10^3, ... This is the same as: 1, 10, 100, 1000, ..., which is units, tens, hundreds, thousands, etc. The rule is that the value of a digit in a number is that digit multiplied by 10 to the power of its position in the number. Once each digit has been evaluated in this way, they're all added together to get the overall value of the

number. For example, the number 952 represents 2×(10^0) + 5×(10^1) + 9×(10^2) = 2 + 50 + 900 = 952.

The reason why we use 10 in the these calculations is because the decimal system uses ten different digits (0 to 9): we count in base 10.

Binary

There are only two numbers in the binary system, zero and one. This system is used because these are the only two numbers a microprocessor can work with. Binary notation works very like decimal notation. Each position in a binary number has a value: two to the power of that position. So the binary number 10 means: 0 + 1×(2^1) = 0 + (1×2) = 0 + 2 = 2 (in decimal notation.) The number 10101 evaluates as 1×(2^0) + 0 + 1×(2^2) + 0 + 1×(2^4) = 1 + 0 + 4 + 0 + 16 = 21 (decimal). Note that since each digit can only be 0 or 1, all you really need to know to use base two are the position values. As each memory location can only hold 8 binary digits (one byte), all you need are the position values for positions 0 to 7. Here's a table for quick reference:

Bit values in a byte

Bit position	Decimal value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

You don't have to translate decimal numbers to binary by hand, as the Amstrad has the reserved word BIN$ which returns the binary representation of a decimal number. 'PRINT BIN$(9)' displays 1001. What's more, you can use two numbers, and of these the second specifies how many binary digits there are to be. Thus PRINT BIN$ (9,8) produces 00001001, making the binary pattern up to eight binary digits – a byte. If you omit one of the numbers, the Amstrad will take it as the number to convert, and will produce only as many binary digits as are necessary. The

largest number you can use is 65535, and the smallest is −32768. BIN$ can produce a maximum of 16 digits: 'PRINT BIN$(−1)' produces 16 ones, and if the number cannot be represented in as many digits as you specify, the effect will be as if you hadn't used one – the binary pattern will contain as many digits as is necessary. Remember, the result is a string, not a value, and has to be manipulated with string operators like MID$, LEFT$ and RIGHT$.

Hexadecimal

Assembly language programmers make much use of hexadecimal (base 16) notation, abbreviated to 'hex'. Working in base 16 involves using letters for numbers: A is 10, B is 11, F is 15 and the decimal number 16 is 10 in hex.

Just as in the decimal and binary systems, the position of a hex digit in a hex number is important. And just as in binary and decimal, the value of the position is calculated as the base number (16) to the power of that position.

Here's a table which shows how to count in hex:

Decimal	Hex	Decimal	Hex
0	0	17	11
1	1	18	12
2	2	19	13
3	3	20	14
4	4	21	15
5	5	22	16
6	6	23	17
7	7	24	18
8	8	25	19
9	9	26	1A
10	A	27	1B
11	B	28	1C
12	C	29	1D
13	D	30	1E
14	E	31	1F
15	F	32	20
16	10		

However, you don't have to work out hex-decimal equivalents by hand. The Amstrad can do the hard work for you. The function HEX$ returns the hexadecimal representation of a decimal number, so 'PRINT HEX$(249)' produces 'F9'. This is correct because '9' in hex is nine in decimal – position zero is the 'units' column in any base. 'F' is 15, it's in the first position, so evaluates as 15 x (16 ^ 1) = 15 x 16 = 240. 240 + 9 = 249, so HEX$ works. It cannot cope with decimal numbers over 65535, giving an 'Overflow' error message.

By convention, the ampersand symbol is used as a prefix to signify hex numbers. You can use it to convert from hex to decimal with statements like 'PRINT &FF' which produces 255.

Because HEX$ is a string function, it returns a string which is the representation of a decimal number, so 'PRINT HEX$(255)' will produce '&FF'. Remember that the result is a string, not a number, so it will have to be manipulated as such with string operators.

Unfortunately, the Amstrad only uses 16-bit integer maths for handling hex, so you'll get odd results (negative decimal numbers) with large decimal values. 'PRINT &7FFF' produces 32767, but 'PRINT &8000' (which is &7FFF + 1) produces –32768. This can prove awkward. For example, the loop which begins, 'FOR count = &FF TO &FFFF', will never be executed, because &FF is evaluated as positive, &FFFF as negative (–1), so the initial condition isn't satisfied and the loop is therefore bypassed. To get round this, you can define your own function to convert hexadecimal numbers to usable decimal values. 'DEF FN hx(n) = – (65536 * (n<0) – n)' does the trick, and 'PRINT FN hx(&FFFF)' will produce 65535, whereas 'PRINT &FFFF' produces –1.

Boolean logic

Boolean logic is a method of dealing with numbers which was first described by George Boole in the nineteenth century. Boolean logic embraces the reserved Basic words AND, OR and NOT, as well as more complex operations at the bit level.

AND is often used to test two conditons – if both are satisfied then some action may be taken. We use this word often enough in English, so it shouldn't be too hard to transfer its use to programming in Basic. For example, we might say, 'If the train's on time and I can catch a cab then I'll be there at 11 o'clock'. As we understand it, the last part (I'll be there at 11 o'clock) will

happen if, and only if, the first two parts are 'true'. In other words, if the train's late, or I can't get a cab, then I won't be there at 11 – and from that we see how OR is used. In English, NOT is used as in 'If the train's not on time or I miss it then I won't be there at 11 o'clock'. In Basic, NOT has the syntax, 'IF NOT (count = 3) THEN ...', which means the same thing as 'IF (count <> 3)'.

AND, OR and NOT can be used in Basic programs, usually with an IF...THEN construction, as in:

```
1010 IF lives <= 2 AND SCORE < 500 THEN PRINT
"You're not doing very well so far"
```

Often it's easier to bracket clauses together to make them more readable:

```
100 IF (lives <= 2) AND (SCORE < 500) THEN PRINT
"You're not doing very well so far"
```

You can draw up what are known as 'truth tables' to make Boolean operations clearer. For example, let's look at the simple clause, 'IF (count = 3) AND (value = 10) THEN GOTO 2000.

count = 3?	value = 10?	GOTO 2000?
YES	YES	YES
YES	NO	NO
NO	YES	NO
NO	NO	NO

Line 2000 will be executed if, and only if, both count = 3 and value = 10. If either of the variables' contents don't match the values given, the THEN section will be ignored, and the GOTO 2000 will not occur.

If we change the conditions to 'IF (count = 3) OR (value = 10) THEN GOTO 2000, this will affect the truth table as follows:

count = 3?	value = 10?	GOTO 2000?
YES	YES	YES
YES	NO	YES
NO	YES	YES
NO	NO	NO

If either condition is true, the GOTO will be obeyed. Only if neither of the conditions are true will it be ignored.

Truth and falsehood

Where we've used YES and NO, Boolean logic uses TRUE and FALSE, and the computer uses −1 and 0. If an expression like 'variable = 3' is TRUE, the computer flags this truth with a value of −1. Try entering 'value = 5:PRINT (value = 5)'. The number −1 appears because the variable 'value' has a value of five assigned to it. In effect, the left-hand side of the expression is the same as the right, so the expression is true. And, of course, 'PRINT (value = 3)' will produce 0 because the expression is false.

The TRUE and FALSE system allows the programmer to take some short cuts, albeit at the expense of readability. For example, the statement 'IF variable = 3 THEN flag = −1 ELSE flag = 0' becomes 'flag = (variable = 3)'. If 'variable' does have a value of three, the expression will produce −1 (TRUE) and this value will be assigned to 'flag'. If 'variable' isn't three, the expression will evaluate as zero (FALSE), and this value will be assigned to 'flag'.

The technique can be extended. The line 'IF value1 = 99 THEN variable3 = 5 ELSE variable3 = 0' can be condensed to 'variable3 = −5 * (value1 = 99)'. You can use the technique to replace multiple IF...THEN statements, so the following are equivalent:

```
999 REM IF...THEN version
1000 IF akey$ = "z" THEN xcoord = xcoord - 1
1010 IF akey$ = "x" THEN xcoord = xcoord + 1
1020 IF akey$ = ";" THEN ycoord = ycoord - 1
1030 IF akey$ = "." THEN ycoord = ycoord + 1
```

```
999 REM Boolean version
1000 xcoord = xcoord + (akey$ = "z") - (akey$ =
"x")
1010 ycoord = ycoord + (akey$ = ";") - (akey$ =
".")
```

Using Boolean logic like this produces virtually unreadable code, but does have the advantage of being executed faster than IF...THEN statements, and is therefore of particular value in games. The technique is used in the game described in Chapter 5.

Bit testing

Boolean operations can also be used to test the bit positions of a byte. To understand this we have to go back to binary notation. You might think that 'PRINT 2 AND 8' would produce 10, but it doesn't. AND isn't the same as '+', and you get zero. Try 'PRINT 8 AND 3', then 'PRINT 8 AND 10'. It doesn't seem to make sense, does it? However, there are very good reasons for the results you get. Let's look at each of these in turn. 8 is represented in binary by 00001000, and 2 is 00000010. When AND is used with numbers like this, it means 'the binary pattern of the result has bits set to zero if, and only if, corresponding bits in the two binary patterns are set (to one)'. This rule is shown in the following table:

AND

first bit	second bit	result
0	0	0
0	1	0
1	0	0
1	1	1

If we put the bit pattern for 8 over the bit pattern for 2 and apply the rule to corresponding bits, we get:

00001000 (8)
00000010 (2)
00000000 result = 0

If we do the same thing with '8 AND 3', we get

00001000 (8)
00000011 (3)
00000000 result = 0

However, if we compare '8 and 10' in the same way, we get:

00001000 (8)
00001010 (10)
00001000 result = 8

OR follows similar rules. Bits are set if either corresponding bit is set. '8 OR 10' produces:

```
00001000   (8)
00001010  (10)
00001010  result = (10)
```

The rules which govern OR are:

OR

first bit	second bit	result
0	0	0
0	1	1
1	0	1
1	1	1

The lesser-known operator XOR (exclusive OR) is very similar to OR, but excludes the case of both bits set. That is, if both bits are set, the result is 0, not 1 as with OR. XOR's rules are:

XOR

first bit	second bit	result
0	0	0
0	1	1
1	0	1
1	1	0

XOR is used in some screen handling operations (see Chapter 9).

Bit mapping

We can use binary logic to make use of a single byte to store a lot of information. Let's look at a practical example of this. Remember, it's only applicable to systems where information is of the 'yes' and 'no' type. Let's suppose that in a school there is a database and that teachers need to know which subjects a pupil is taking. Each student can take one each of pairs of subjects: French/German, Physics/Chemistry, Geography/History, Woodwork/Metalwork, Biology/Geology. Each pupil may be a boy or a girl, may be in an upper or lower stream and may be scheduled for 'O'

Bit position	7	6	5	4	3	2	1	0
Decimal value	128	64	32	16	8	4	2	1
'Subject' set	French	Physics	Geography	Woodwork	Biology	Female	Upper	O level
'Subject' zero	German	Chemistry	History	Metalwork	Geology	Male	lower	CSE

Example:

1	0	1	0	0	1	1	1	Binary pattern
French	Chemistry	Geography	Metalwork	Geology	Female	Upper	O level	'subjects'
128	0	32	0	0	4	2	1	Decimal values

Figure 6.1. How complex information may be encoded in a single byte using 'bit-mapping'. Here, the byte's value is 128+32+4+2+1=167

level or CSE exams. There are thus eight different pairs, and we can use one bit of a byte to indicate to which one of a pair the student belongs. For example, male students' code values will have a zero at bit 2, while for female students this bit will be set to one. Figure 6.1 should help explain how this works. If a pupil is down for 'O' level, then we want bit zero (the first bit) to be 'set', i.e. a one is placed at that bit position (giving a decimal value of 1). For a pupil in an upper group, bit one, the second bit will also be set (decimal value 2). If the student is taking Geology, we need a zero at bit 3 (decimal value ignored). If the student is taking Metalwork, Geography, Chemistry and French, the complete binary pattern is 10100111, or 128 + 32 + 4 + 2 + 1, making 167. The next problem to be faced is just how to code and decode information in this way.

Below we give a full listing of a short program to encode and decode values using such bit-mapping techniques. Subject names and other attributes are held in the two-dimensional string array 'attribute$(8,2)', there are eight attributes, and each attribute may have one of two values, e.g. male or female. Lines 360 to 400 encode the information by stepping through bits zero to seven, showing the attributes in turn, and adding '2 ^ bitnumber' to the code value if 'yes' is given. To decode this we simply have to AND the coded value with the decimal values 128, 64, 32, 16, 8, 4, 2 and 1 in turn. If the result of the AND is the decimal number, then there is a one (a bit set) at that bit position, which indicates which one of the attribute pair is relevant. Note especially the use of

brackets in line 510. Were the conditional statement written as 'IF byte AND dec.val = dec.val THEN ...' the program would not analyse codes correctly. This is because this statement is read by the Amstrad as 'IF byte AND (dec.val = dec.val) THEN ...', which has a completely different meaning as it will always AND the byte (which is the coded value) with −1.

Bit mapping like this can save enormous amounts of memory and can be used in many other applications, such as adventure games, for example. You could store information about locations, items and conditions of play in single bytes rather than large arrays.

```
10 'Bit Mapped data
20 '
30 DATA O level,Upper,Female,Biology,Woodwork,Geog
raphy,Physics,French
40 DATA CSE,Lower,Male,Geology,Metalwork,History,C
hemistry,German
50 '
60 'Load Arrays
70 DIM attribute$(8,2)
80 FOR attribute = 1 TO 8
90 READ attribute$(attribute,1)
100 NEXT
110 '
120 FOR attribute = 1 TO 8
130 READ attribute$(attribute,2)
140 NEXT
150 '
160 ZONE 20
170 'MAIN MENU
180 CLS:LOCATE 17,1:PRINT"SELECT"
190 LOCATE 10,7
200 PRINT"A...Assign a value"
210 LOCATE 10,9
220 PRINT"C...Check a value"
230 LOCATE 10,11
240 PRINT"E...End"
250 '
260 akey$ = UPPER$(INKEY$)
270 choice = -(akey$ = "A") - 2 * (akey$ = "C") -
3 * (akey$ = "E")
280 '
290 IF choice = 0 THEN 260
300 ON choice GOSUB 350,460
310 IF choice = 3 THEN CLS:END
320 GOTO 180
330 '
```

```
340 'Data Entry
350 CLS:byte = 0
360 FOR each.bit = 0 TO 7
370 PRINT attribute$(each.bit + 1,1);"...Y/N",
380 GOSUB 570
390 IF akey$ = "Y" THEN byte = byte + 2 ^ each.bit
:PRINT attribute$(each.bit + 1,1) ELSE PRINT attri
bute$(each.bit + 1,2)
400 NEXT
410 PRINT:PRINT"Value=";byte
420 GOSUB 620
430 RETURN
440 '
450 'Check a value
460 CLS:PRINT"Enter value"
470 INPUT byte:PRINT
480 FOR each.bit = 0 TO 7
490 LOCATE 15,each.bit + 7
500 dec.val = 2 ^ each.bit
510 IF (byte AND dec.val) = dec.val THEN PRINT att
ribute$(each.bit + 1,1) ELSE PRINT attribute$(each
.bit + 1,2)
520 NEXT
530 GOSUB 620
540 RETURN
550 '
560 ' Y/N Subroutine
570 akey$ = UPPER$(INKEY$)
580 IF akey$ <> "Y" AND akey$ <> "N" THEN 570
590 RETURN
600 '
610 'Press space subroutine
620 LOCATE 10,24
630 PRINT"Press space to continue"
640 IF INKEY(47) = -1 THEN 640
650 RETURN
```

Number-handling functions

The Amstrad has a number of specialised functions which make it easier to deal with numbers. Below we list some of the more useful of these.

MOD

MOD is used to find the modulus of one number with another. In a way it's like the 'remainder' in division sums. MOD returns what's left after a number has been divided by another as many times as possible with integer results. The following table may make this clearer:

operation	result
3 MOD 1	0
2 MOD 99	2
3 MOD 7	3
99 MOD 1	0
1 MOD 3	1
5 MOD 2	1
99 MOD 10	9
83 MOD 40	3

As you can see, any number MODded with 1 (n MOD 1) will produce zero, because any integer can be divided by 1 a whole number of times, with no remainder. 83 MOD 40 produces 3 because 40 goes into 80 twice, with 3 as a remainder.

If the Amstrad didn't have HEX$, MOD would be very useful for calculating the high and low bytes for assembly language 16-bit memory addressing. For example, to find the low byte of a 16-bit number (greater than 255) you use the formulae:

low byte = 256 * (number MOD 256)
high byte = INT (number/256)

Using HEX$ is much easier, however, since the low byte is the last two digits of the hex version of the number, and you can translate these into a decimal 8-bit number using PRINT. Thus the low byte of &BD7C is &7C, which you can convert to decimal using PRINT &7C to yield 124.

MOD is of particular value when you're dealing with different number bases. It's used in the real-time clock example in Chapter 11. There, we're dealing in 1/300ths of a second and have to convert this to seconds. This is achieved by dividing the elapsed time by 300, but then the seconds have to be converted

to minutes, while 'seconds MOD 60' gives the number of seconds left over. Similarly, INT(seconds / 3600)' gives hours and 'seconds MOD 3600' gives the remaining minutes. If we wanted to turn a seconds counter into a full calendar clock we'd have to use MOD 24 (for weeks and days), MOD 7 (months and weeks), and so on.

MIN and MAX

MIN and MAX return the smallest or largest number from a list of numbers, or expressions. Thus 'smallest = MIN (1,2,3,4)' assigns the value of 1 to the variable 'smallest'. Because you can also use variables, expressions like 'biggest = MAX (number1, value3,count)' are allowed, and can replace many IF...THEN statements.

ABS

ABS simply drops the negative sign from values or expressions. 'PRINT ABS (–20)' produces 20, and 'PRINT ABS(99-199)' produces 100.

SGN

SGN returns one of three values: –1, 0 or 1. This function evaluates the sign of a number, variable or expression and tests whether it is negative, zero or positive. In the statement 'expr.sign = SGN (33 – 39)', a value of –1 will be assigned to the variable 'expr.sign' because the expression 33–39 produces a negative result. If the result is zero, SGN returns zero. If it is positive, SGN returns 1.

PI

PI is a 'system constant'. It is rather like a variable, but has a constant value of 3.14159256, a closer approximation than the 22/7 we learn at school. PI is used most often in trigonometry, for calculating the surface areas of circles, spheres and so forth. The

formula for the surface area of a circle could be used in a function definition, such as:

```
10 DEF FN(area) = PI * radius ^ 2
```

LOG, EXP and LOG10

A logarithm is the power (exponential value) to which the base number must be raised to get that number. The logarithm (to base 10) of 100 is 2 because 100 is 10^2. Log10(1000) is 3, because 1000 is 10^3.

LOG(n) returns the natural logarithm of the number n. This is the log to base e, where 'e' is the number which is given by the formula 1 + 1/1 + 1/2 + 1/6 + 1/24 + ... + 1/n! and is approximately 2.7182818 (n! means 'n factorial' and is given by n x (n-1) x (n-2) x (n-3) x ... x 1).

EXP(n) is the converse of LOG, and returns e^n, i.e. e times itself n times, thus EXP(LOG(n)) returns n.

LOG10(n) returns the log to base 10 of n. Logarithms are used to make the manipulation of large numbers much simpler. Because logs represent exponential 'powers', they can be used to simplify some complex calculations. For example, a quick way to multiply two large numbers together is to look up the logs of the numbers, add the logs together, then look up the antilog of the result.

CINT and CREAL

These functions convert expressions to different data types. CINT converts to integer format, CREAL to real. 'CINT (9.999)' returns 10, so acts like ROUND. CREAL could be used as in:

```
10 x% = 9
20 x% = x% + 0.5
30 PRINT CREAL (x% + 0.5)
```

Note that while the '%' suffix to variables means they can only return integers (whole numbers), the internal representation of such integer variables is altered (internally at least) by mathematical operations. CREAL can be used to make such

changes apparent and extends the operations you can perform with integer variables.

UNT

There are two ways of dealing with sixteen-bit (two byte) representations of decimal numbers. You can use all 16 bits to represent decimal values, and this gives you a number range from 0 (all bits zero) to 65535 (all bits set to one). The standard method is that any bit's decimal value can be found by the expression '2^ (bit number)', where bits are counted from right to left, starting with zero. Hence the largest number, 65535, is found by 2^ 0 + 2^ 1 + 2^ 3 + ... +2^ 14 + 2^ 15 (or (2^ 16) – 1).

However, in this system you cannot represent negative numbers – a major drawback. The alternative is to use bit fifteen (the most significant bit) as a 'flag' which says 'this is a negative number' and this now means that the range of decimal numbers which can be represented in the sixteen-bit binary system is –32768 to +32767. Negative numbers are converted using the two's complement convention.

UNT will convert unsigned decimal numbers in the range 0 to 65535 to their signed, two's complement sixteen-bit equivalents. This is handy for some calculations needed in assembly language programming if you haven't got an assembler.

7
Machine Code

Machine code is the language your computer uses internally. Its instruction set can only act on half a dozen or so 'registers' (rather like variables in Basic). It is quite difficult to learn, but is very fast indeed. There are no commands like LOCATE or PRINT, and to write a routine to perform these instructions requires several lines of machine code. However, the speed of machine code makes it a very attractive alternative to Basic.

Registers

Machine code uses 'registers', which are represented by the letters A to H and L, and there are some commands which act on two registers at a time, the register pairs being AF, BC, DE and HL. The A register is most frequently used – 'A' stands for 'accumulator'.

Machine code instructions

Among the instruction set of the Z80 microprocessor are commands to load values into registers, put the contents of one register into another, store the contents of a register in a memory location, and so on. There are also the machine code equivalents of the Basic 'branching' words GOSUB and GOTO.

Op-codes

A machine code program is a group of bytes, some of which cause the Z80 to operate on data and registers, others of which are the data to be manipulated. Each instruction or command has a number associated with it and these numbers are known as 'op-codes'. For example, the two-byte sequence 00000110

10000000 (6 and 128 decimal) means 'load the value 128 into the B register'. The first byte is the 'load B' instruction, the second the data to be acted on.

Mnemonics

Machine code programming at the byte level is almost impossible because it's difficult to spot the difference between binary numbers like 00010100 and 00101000 at a glance, let alone remember that they 'mean' completely different things to the Z80. Consequently, an easier way of programming microprocessors has been developed. This is called 'assembly language' and uses abbreviations like 'LD B,n' which means the same as the example given above: load the next value (given by 'n') into the B register. These abbreviations are known as 'mnemonics' because they help the programmer remember what each instruction means. An assembly language program consisting of mnemonics and data can be typed into a program called an 'assembler', which converts the abbreviations into machine code and stores them in RAM. Arcade-type games are programmed in assembly language; they have to be because Basic is just too slow.

Mixing Basic and machine code

You don't have to write a whole program in assembly language. One of the nice features of Basic is that you can call up a machine code routine from within a Basic program, which you might do in order to perform some operation faster than is possible in Basic. You can call up machine code routines resident in ROM to do things not possible from the set of reserved words in Basic, and this is what we will concentrate on in this chapter.

Once a machine code program has been 'loaded' into memory, the computer can be told to run the program by the Basic word 'CALL', followed by an address. This tells the system to suspend whatever it's doing in Basic (remembering, of course, where it was and what it was doing), jump to the RAM address given and execute the machine code instructions it finds from there on. If the machine code routine ends with the op-code for the mnemonic 'RET', it will return control to the routine which 'called' it. This 'calling' routine may have been a Basic program, or a machine code program itself called from Basic. The process of calling up

machine code routines from Basic programs is very similar to the use of GOSUB...RETURN.

Machine code routines

Rather than dwell on the intricacies of assembly language programming, we'll give some simple routines which you can call up and use in your Basic programs. Studying the code and explanations should help you get started with assembly language programming. However, before we present you with the machine code routines there are several important points about Basic/machine code interfacing that need to be described.

Because Basic programs and their data take up storage space in RAM we have to tell the Amstrad to reserve some memory so that Basic can't use it. If we don't do this it's quite possible for the data stored in an array to overwrite and corrupt a machine code routine. This 'reservation' is done with the word MEMORY, followed by an address. The address forms a sort of 'fence' for a Basic program and its associated data – a program cannot make use of addresses higher than the one given. Therefore, machine code routines can start from that address plus one. If we want a machine code routine to start at address 43880, we use the command 'MEMORY 43879'. Then, once the machine code routine has been 'loaded' into memory, we can CALL 43880 to have the routine executed.

Scrolling the screen

The first routine is the simplest – it allows you to move the screen display up or down by one character line, and by calling it from a FOR...NEXT loop you can make the display scroll as far as you like.

The Amstrad ROM contains a number of machine code routines which are not directly available using the reserved words in Basic. One of these is called SCR HW ROLL and is responsible for moving the screen up during listings or when you've filled the screen with text using PRINT statements. This routine begins at address &BC4D (hexadecimal for 50395 decimal) and can also be used to move the screen down. The direction of the scroll depends on the contents of the B register. If the B register contains zero when the routine is executed the screen will scroll down, while

any non-zero value will produce an upward scroll. The colour for the new line at the top or bottom of the screen is taken from the A register, and normally this would be 0, for the default colour of dark blue.

To make the screen scroll, we first have to load the B register with either zero or a non-zero value. The mnemonic for 'load the B register with a value' is 'LD B,#n' – it's a two-byte instruction where n is the number to be stored in B. (The '#' symbol means that the register is to be loaded with the value given and is often used to distinguish this from instructions to load registers with the values held in addresses.) The op-code for LD B is 6, so this will be the first byte of our machine code routine. Since we'll be assembling the routine from 43880, we can use Basic to POKE the number 6 into this address: POKE 43880,6. The next byte is the number to be loaded into B and this determines the direction of the scroll. To scroll upwards we would use POKE 43881,255, while a downward scroll needs POKE 43881,0.

The next action is to CALL the ROM routine, and this requires rather more thought. The routine we want to CALL starts at &BC4D (50395 decimal), so the instruction will be CALL 50395. The op-code for CALL is C9 (205 decimal). However, we can't specify a number larger than 255 using a single byte, so the address has to be split across the two bytes following the CALL instruction. The Z80 allows numbers greater than 255 to be spread across two bytes in the order 'low byte, high byte'. The low byte of a number can be found using the formula, 'lowbyte = number – 256 * int(number / 256)' while the high byte is 'highbyte = int(number / 256)'. An easier way of obtaining these two numbers is to use the hex notation. The number 50395 is BC4D in hex, and the low byte is the last two numbers – 4D (77 decimal), while the high byte is BC (188 decimal). You can use 'PRINT &4D' and 'PRINT &BC' to check these. Finally, when the ROM routine has been executed and control has been returned to our special routine, we need to return control to our Basic program. This is achieved with the mnemonic 'RET', whose op-code is 201 (decimal). So far then we have:

```
Mnemonic/Data  Decimal equivalent

LD B,#0        (06,0)
CALL 40395     (205,77,188)
RET            (201)
```

The next step is to load the op-code numbers into memory using what's called a 'Basic loader'. We take the decimal numbers, put them in DATA statements, READ them and POKE them into contiguous RAM addresses. The Basic loader runs like this:

```
10 DATA 6,0,205,77,188,201
20 MEMORY 43879:address = 43879
30 FOR count = 1 TO 6
40 READ value
50 POKE address + count,value
60 NEXT count
```

Now, when you want the screen to scroll, POKE 43881 with 0 for down or 255 for up, then CALL 43880. The next program is a demonstration of the routine and assumes that the machine code routine has been loaded into memory.

```
70 CLS:FOR char = 65 TO 89
80 PRINT STRING$(38,char)
90 NEXT char
100 FOR up = 1 TO 10
110 POKE 43881,255
120 CALL 43880
130 NEXT up
140 FOR down = 1 TO 10
150 POKE 43881,0
160 CALL 43880
170 NEXT down
180 GOTO 100
```

A more complex version

The next routine does a very similar job, but is rather more flexible. In essence the program is the same as the one just described, but with this version you can specify the colour of the blank line created at the top or bottom of the screen by POKEing different values into location 43873. You can also dictate the number of lines the routine is to scroll the display by POKEing different values into location 43871.

First we list the addresses, mnemonics, hex codes and decimal equivalents:

Address	Mnemonic	Op-Code/Data	Decimal
43870	LD B,#A	06	6
43871		A	10
43872	LD A,#0	3E	62
43873		0	0
43874	PUSH BC	C5	197
43875	PUSH AF	F5	245
43876	LD B,#FF	06	6
43877		FF	255
43878	CALL &BC4D	CD	205
43879		4D	77
43880		BC	188
43881	POP AF	F1	241
43882	POP BC	C1	193
43883	DJNZ	10	16
43884		F5	245
43885	RET	C9	201

The first byte is the instruction to load the B register with the value in the next byte (held in address 43871). In this routine the B register is used in much the same way as the loop counter in a FOR...NEXT structure, so the value of the contents of 43871 dictates the number of lines by which the screen will be scrolled. The instruction in address 43872 loads the A register with the contents of address 43873, and this is the value of the colour code for the new line. Addresses 43874 and 43875 contain the instructions for PUSHing the BC and AF register pairs on to what is known as the stack – a temporary storage area. This has to be done partly because SCR HW ROLL corrupts all the registers, and if these register pairs aren't 'PUSHed' the program would lose track of how many times to scroll the screen and might change the ink colour, and also because we need to load the B register with the direction in which to scroll. The instruction at 43876 loads the B register with the direction to scroll. As given, the routine will produce an upward scroll. The CALL at 43878 is the same as in the first routine, it transfers control to the ROM routine at the address given by the next two bytes in the normal 'low byte, high byte' order. When the ROM routine returns, the BC and AF register pairs are restored to their previous values by POPping them off the stack in the opposite order to the PUSH instructions. The next instruction, DJNZ, in address 43883 means 'subtract one from the value of the contents of the B register, put that value into the B register and jump back to address 43874 if the B register contents are not zero'. This is the equivalent of the Basic

instructions: 'B = B — 1:IF B<> 0 THEN GOTO ...'.

The number following DJNZ is the 'displacement' for the backwards jump. This is calculated by working out how far back we need to go: —9 bytes (negative because we're going backwards). The value is calculated as follows: subtract 43874 from 43883 giving —9. This value has to be adjusted by subtracting two, because by the time the DJNZ instruction has been interpreted, the program counter (PC) is 'pointing' to an address two bytes beyond the DJNZ instruction. In this example, this gives a value of —11. Next we take what's called the two's complement of the result. This can be calculated by writing down 11 in binary (00001011) then reversing all the bits (changing zeroes to ones and vice versa) yielding 11110100. To this we add one to give 11110101 and finally translate it back into decimal to get 245. The Z80 will automatically realise that 245 means —11 and jump back that number of bytes, i.e. to address 43874. The beauty of an assembler program is that all such jumps are calculated for you and you can jump to 'labels', i.e. specific addresses, without having to work out the two's complement displacements by hand. The Basic loader is:

```
10 MEMORY 43869:address = 43869
20 scroll = 43870
30 DATA 6,10,62,0,197,245,6,255,205,77
40 DATA 188,241,193,16,245,201
50 FOR count = 1 TO 16
60 READ value
70 POKE address + count,value
80 NEXT
90  REM POKE 43871 with colour
100 REM POKE 43873 with number of lines
110 REM POKE 43877 for up/down
120 REM CALL 43870 to scroll
```

PEEKing text on the screen

The next machine code routine is one which will tell you the ASCII code of any character under the cursor. It's particularly useful in games based on the text screen, because you often need to know whether a moving character under user control has run into another character. An example of this would be the maze in a Pac-man type game, and the routine given is used in the 'Snake' game in Chapter 5.

The routine works by calling the ROM routine called TXT RD CHAR at address 47968 (BB60 hex). This attempts to match the character matrix under the cursor with the patterns for generating the default character set. If a match is found the ROM routine places the ASCII code of the character found in the A register. If the routine is unable to match the 'set' pixels in the matrix with those in ROM, the A register will contain zero.

To use the routine, move the cursor to the location you want to test with LOCATE, then CALL 43800. The ASCII code of any character under the cursor will be placed in address 43799 and can be PEEKed from Basic. If PEEK(43799) produces zero, then either the character space is blank, or you may have corrupted the character matrix at that location (e.g. by drawing a line through it using the graphics commands), or you've changed INK or PAPER colours. Here's the Basic loader for the routine:

```
10 MEMORY 43798
20 '43799 is used to store results
30 address = 43800
40 'Routine starts at 43800
50 '
60 'Decimal machine code
70 DATA 205,96,187
80 DATA 50,23,171,201
90 DATA 0
100 '
110 'Machine code loader
120 READ value
130 IF value = 0 THEN 260
140 POKE address,value
150 address = address + 1
160 GOTO 120
170 '
180 'all done
190 REM To use: place cursor with LOCATE
200 REM then CALL 43800
210 REM then PEEK(43799)
220 REM This returns ASCII value of
230 REM the character at the cursor position
240 REM or zero if no character
250 '
260 REM EXAMPLE
270 CLS
280 LOCATE 1,1
290 PRINT "ABCDEF"
300 FOR i = 1 TO 6
310 LOCATE i,1
```

```
320 CALL 43800
330 value = PEEK(43799)
340 LOCATE 1,10
350 PRINT "Character";i;" is ";CHR$(value)
360 PRINT "ASCII code is";value
370 LOCATE 10,20
380 INPUT "Press ENTER for next",a$
390 NEXT
```

This is the assembler code for the routine:

```
Address    Mnemonic        Hex         Decimal

43800      CALL 0BB60H     CD 60 BB    205 96 107
43803      LD (0AB17H),A   32 17 AB    50 23 171
48806      RET             C9          201
```

The ROM routine is CALLed in the first three bytes, then the contents of the A register are stored in address 43799, and that's it!

Filling boxes

While it's possible to use the WINDOW command to create blocks of colours, it's a clumsy technique. You have to define the window, define a colour for the paper, clear the window and so on. However, there's a ROM routine called SCR FILL BOX which begins at address &BC44 and fills character cells with the colour code held in the A register. The routine needs four other values to specify the left, right, top and bottom character positions of the box to be filled. These values must be placed in the H, D, L and E registers respectively.

The routine given here allows you to define the box colour and the corners of the boxes with five POKEs. The advantage of using your own machine code routine is that it acts independently of any windows on the screen, so you can define and use Basic windows for text, as well as filling rectangles with plain colour or even textured colours – which you cannot do easily from Basic.

```
10 '      Basic Loader
20 '    For box filling
30 MEMORY 43879
40 address = 43879
50 DATA 62,255,38,0,22,0,46,0,30,0,205
```

```
60 DATA 68,188,201
70 FOR count = 1 TO 14
80 READ value
90 POKE address + count,value
100 NEXT
110 '============= All Done ==========
120 '
130 'POKE 43881,colour
140 'POKE 43883,left column
150 'POKE 43885,right column
160 'POKE 43887,top row
170 'POKE 43889,bottom row
180 'CALL 43880 to fill box
190 '
200 '=========== Demonstration ========
210 MODE 1
220 colour = 43881
230 left = 43883:right = 43885
240 top = 43887:bottom = 43889
250 fill = 43880
260 '
270 texture = 255
280 TLHC = 0
290 'TLHC is Top Left-Hand Corner
300 dc = 1
310 'dc is TLHC increment
320 '
330 'Set up addresses to define box
340 POKE colour,texture
350 POKE left,TLHC
360 POKE top,TLHC
370 POKE bottom,24 - TLHC
380 POKE right,39 - TLHC
390 'Change texture
400 texture = texture - 10
410 IF texture < 0 THEN texture = 255
420 'Call box fill
430 CALL fill
440 TLHC = TLHC + dc
450 IF TLHC = 12 OR TLHC = 0 THEN dc = - dc
460 GOTO 340
```

Address	Mnemonic	Op-code/Data	Decimal
43880	LD A,#n	3E	62
43881		00	0
43882	LD H,#n	26	38
43883		00	0

```
43884   LD D,#n      16    22
43885                00     0
43886   LD L,#n      2E    46
43887                00     0
43888   LD E,#n      1E    30
43889                00     0
43890   CALL &BC44   CD   205
43891                44    68
43892                BC   188
43893   RET          C9   201
```

The routine itself is very simple. It loads the relevant registers with the appropriate values, then calls the ROM routine at &BC44. In the Basic listing, lines 30 to 100 are the Basic loader, lines 130 to 180 give the addresses to POKE to define the colour to fill the box and the addresses to POKE with the top, bottom, left and right text locations for the box. The limits on these are defined by the screen MODE. The demonstration (lines 210 to 460) assumes that the screen is in MODE 1. If you POKE a column value greater than the upper limit available (e.g. POKEing 43885 with a number greater than 20 in MODE 0) you'll get wrap-around (this means that items which should appear off-screen to the right appear at the left instead – try '10 CLS:PRINT "This string's too long to fit on one line, so it's wrapped around to the following line" '). The demonstration also shows the colour textures available. To find out which numbers give plain colours you'll have to experiment by POKEing 43881 with different values.

Stripy inks

The last machine code program in this chapter uses the ROM routine called SCR CHAR INVERT which XORs character ink colours. On entry it assumes that the B and C registers contain the two colours to use, while the H and L registers hold the screen location of the character in terms of rows and columns – H is used as the column, L as the row.

The assembler code for the routine gives the addresses, mnemonics, opcodes and decimal equivalents.

The Basic loader includes a demonstration which PRINTs strings of the characters 'A' to 'X' then applies the character invert routine to each character position in the row, according to two colours chosen at random. The two numbers displayed at the left of each row are the random numbers for the colour codes, so

when you see a combination you want to use, press ESC to pause the program and write down the values which you can then use in your own programs. As you'll see, the routine gives you access to unusual colour textures like stripy ink and paper.

Address	Op-code	Hex	Decimal
43890	LD B,#0	06	6
43891		00	0
43892	LD C,#0	0E	14
43893		00	0
43894	LD H,#0	26	38
43895		00	0
43896	LD L,#0	2E	46
43897		00	0
43898	CALL &BC4A	CD	205
43899		4A	74
48900		BC	188
48901	RET	C9	201

```
10 'Character Inverter
20 'Basic Loader
30 DATA 6,0,14,0,38,0,46,0,205,74,188
40 DATA 201
50 MEMORY 43889:address = 43889
60 FOR i = 1 TO 12
70 READ v
80 POKE address + i,v
90 NEXT
100 '
110 'POKE 43891 with 1st colour
120 'POKE 43893 with 2nd colour
130 'POKE 43895 with column
140 'POKE 43897 with row
150 'CALL 43890 to invert character
160 '
170 '========== DEMONSTRATION =========
180 DEF FNr(n) = INT(RND(1) * 255) + 1
190 MODE 0
200 char = 65
210 aline = 43897:position = 43895
220 colour1 = 43891:colour2 = 43893
230 value1 = 1:value2 = 128
240 FOR row = 1 TO 24
250 LOCATE 1,row
260 PRINT STRING$(19,char);
270 char = char + 1
280 NEXT
```

```
290 FOR row = 0 TO 23
300 LOCATE 1,row + 1
310 PRINT USING "###";value1;:PRINT " ";
320 PRINT USING "###";value2;:PRINT " ";
330 POKE colour1,value1
340 FOR column = 9 TO 18
350 POKE aline,row
360 POKE position,column
370 POKE colour2,value2
380 CALL 43890
390 NEXT
400 value1 = FNr(255):value2 = FNr(255)
410 NEXT
420 GOTO 290
```

You should POKE address 43891 with the colour of one ink and POKE 43893 with the other. The row and column values have to be POKEd into addresses 43895 and 43897 respectively, and here you must remember to take into account the screen MODE.

ROM calls

The following table lists some useful ROM routines. Those listed have no entry conditions, so can be called from Basic without passing parameters. For more detailed information, consult the firmware manual, available from Amsoft. This lists the major system routines, describing their action and entry and exit conditions. It is an invaluable aid to the assembly language programmer.

```
        Keyboard

&BB00 Initialise Key Management System
&BB03 Reset Key Management System
&BB18 Wait for key-press
&BB06 Get next key press in A register

        Display

&BB4E Initialise text VDU system
&BB51 Reset text VDU system
&BBBA Initialise graphics VDU system
&BBBD Reset graphics VDU system
&BB6C Clear current window
&BD19 Wait for TV frame flyback
```

Cassette

&BC65 Initialise cassette system
&BC6E Start cassette motor
&BC71 Stop cassette motor

Sound

&BCA7 Reset Sound Manager System
&BCB6 Stop all sounds
&BCB9 Restart sounds

8
Introduction to Graphics

Colours

As mentioned in Chapter 1, the Amstrad has three screen MODES which determine the screen resolution and the colours available. MODE 0 is also called the multi-colour mode and gives the greatest range of colours – up to 16 out of a possible 27 may be made to appear on the screen at the same time.

MODE 1 is the default mode, the one the Amstrad adopts when it's switched on, and this offers a maximum of four different colours on show together.

MODE 2 gives the highest resolution, but only two colours can be displayed at a time. The colour codes are given in the following table:

Code	Colour	Code	Colour
0	Black	14	Pastel Blue
1	Blue	15	Orange
2	Bright Blue	16	Pink
3	Red	17	Pastel Magenta
4	Magenta	18	Bright Green
5	Mauve	19	Sea Green
6	Bright Red	20	Bright Cyan
7	Purple	21	Lime Green
8	Bright Magenta	22	Pastel Green
9	Green	23	Pastel Cyan
10	Cyan	24	Bright Yellow
11	Sky Blue	25	Pastel Yellow
12	Yellow	26	Bright White
13	White		

Resolution

In MODE 0, there are 20 columns per line, MODE 1 gives a 40 column display, while in MODE 2 there are 80 columns. Each character is made up from an 8 by 8 matrix, but the size of the dots varies according to the mode. The smallest point of colour on any screen is called a pixel (short for picture element), and the size of these varies from mode to mode. You would use MODE 2 for detailed graphs, for example, where colour is not important but resolution is, MODE 2 could also be used for word-processing because the 80 column format is one of the easiest to work with (most printers produce 80 column text, but the text on the screen at this resolution is not too clear, at least on the colour monitor version of the machine). MODE 0 may allow the most colours, but the detail is rather coarse and this mode is best used for display purposes, educational programs for younger users and so on. The most flexible mode is MODE 1, with its reasonable character size and four-colour display.

Although the screen is 640 units wide by 400 units deep, the number of pixels available in each mode varies. In MODE 2 you can set (illuminate) any of 640 pixels across the screen in any of 200 rows, MODE 1 gives a 320 by 200 pixel display, while the pixel resolution of MODE 0 is 160 by 200. If you look carefully at the size of characters in the different screen modes you'll see that they are all the same depth, but as we move from MODE 0 to MODE 2 the characters are compressed horizontally. In effect, a pixel in MODE 2 is a single horizontal unit; in MODE 1 each pixel occupies two horizontal units; while in MODE 0 this doubles again to four. In all cases, the vertical height of pixels is the same, two rows of the screen.

This method of mapping the screen isn't as confusing as it may sound at first. What it means in effect is that you can use the same co-ordinates in any of the screen modes to refer to a given point. This means that a graphics program can be used in any of the modes with very little modification. But, because of the way the different modes operate, the size of the pixels set on the display and the range of colours available will change from mode to mode.

Border colour

Control over the Amstrad's display is easy and very flexible. You can change the colour of the screen border and the colours of the background or foreground. You can even define two colours between which the border should switch, and the rate at which this is to happen. The border colour is set by the Basic command BORDER n, where n is the colour code. BORDER 3 will produce a red colour, BORDER 1 gives a dark blue – the default colour.

Giving the BORDER command two numbers, such as 'BORDER 2,3', sets two colours for the border, and it will alternate between the two colours. To define the rate at which they should switch you need to use the command SPEED INK. This needs two numbers to define the length of time for which each ink should appear. The time intervals are measured in 1/50ths of a second. So 'SPEED INK 1,1' means that each colour will appear for 1/50th of a second before being replaced by the other. 'SPEED INK 50,50' gives a flashing rate of one colour per second. The upper limit is 255 (approximately 5 seconds). The two numbers do not have to be the same. If you want the first ink on for half a second and the other for two seconds you would use 'SPEED INK 25,200' or 'SPEED INK 200,25'.

Background and foreground colours

Setting the background and foreground colours is not as simple a process as setting the colour for the border.

INK

To write on paper you must select a pen and fill it with ink, and this is how the Amstrad's colour system works. The background colour is the PAPER, PEN is the pen you're writing with, and it has to be filled with ink to get results. The reserved word INK allows you to assign colour values to the INKs you want to use. For example, if you wanted to set INK 0 to light blue, you'd use INK 0,14. To make INK 2 green you'd use INK 2,9.

When the Amstrad is turned on, each INK (numbered 0 to 15 – there are only 16 possible INKs) is assigned a colour code (see table below). This varies from mode to mode, but can be changed at will. Each PEN (and these have the same numbers as the INKs) should be thought of as being 'filled' with the colour assigned to its INK number. Thus, when we assigned the colour code for green to INK 2 above, we were 'filling' PEN 2 with green ink. Similarly, making INK 0 light blue 'fills' PEN 0 with light blue ink.

Default PEN/INK colour assignments in MODE 0

PEN	Default Colour Code	Colour
0	1	Blue
1	24	Bright Yellow
2	20	Bright Cyan
3	6	Bright Red
4	26	Bright White
5	0	Black
6	2	Bright Blue
7	8	Bright Magenta
8	10	Cyan
9	12	Yellow
10	14	Pastel Blue
11	16	Pink
12	18	Bright Green
13	22	Pastel Green
14	1,24	Flashing Blue, Bright Yellow
15	16,11	Flashing Pink, Sky Blue

Default PEN/INK colour assignments in MODE 1

PEN	Default Colour Code	Colour
0	1	Blue
1	24	Bright Yellow
2	20	Bright Cyan
3	6	Bright Red
4	1	Blue
5	24	Bright Yellow
6	20	Bright Cyan
7	6	Bright Red
8	1	Blue
9	24	Bright Yellow

10	20	Bright Cyan
11	6	Bright Red
12	1	Blue
13	24	Bright Yellow
14	20	Bright Cyan
15	6	Bright Red

Default PEN/INK colour assignments in MODE 2

PEN	Default Colour Code	Colour
0	1	Blue
1	24	Bright Yellow
2	1	Blue
3	24	Bright Yellow
4	1	Blue
5	24	Bright Yellow
6	1	Blue
7	24	Bright Yellow
8	1	Blue
9	24	Bright Yellow
10	1	Blue
11	24	Bright Yellow
12	1	Blue
13	24	Bright Yellow
14	1	Blue
15	24	Bright Yellow

PAPER

To set the background colour of the screen you use the command 'PAPER n', where n is an INK number, not the actual colour code. Thus PAPER 3 means 'set the background to whatever colour is assigned to INK 3'. When the Amstrad is turned on, the background, or PAPER colour, is set to INK or PEN number 0. The foreground, i.e. the PEN being used, is PEN number 1.

PEN

The command PEN decides in which colour characters will be written. If you enter 'PEN 0', or use the instruction in a program,

all characters which subsequently appear on the screen will be light blue. If you make the PEN colour the same as the PAPER colour, naturally you won't see anything until you change one or the other: the characters will be written in the same colour as the background.

As well as being able to make the border alternate between two colours, you can also do this with the foreground and background colours. The method is to give three numbers to the reserved word INK, as in INK 1,0,26. The first number is the INK number you are defining, the second and third numbers are the actual colour codes of the colours to be used – in this case black and white. As with BORDER, SPEED INK can be used to alter the flashing rate. Indeed, a SPEED INK command used to flash the border will affect any INK set to two colour codes.

Graphics

The Amstrad can use the screen to produce what are generally called 'graphics': basically, coloured line and point images. As with text characters, you can set the colours for the foreground and background. There are a number of specialised graphics commands available, including DRAW and DRAWR for producing lines, and MOVE and MOVER for moving the graphics cursor.

Co-ordinates

The graphics screen is laid out rather like graph paper – the bottom left-hand corner has the co-ordinates (0,0), i.e. column zero, row zero, and this point is known as the 'origin'. This is very different from the text screen layout, in which the top left-hand corner has the co-ordinates (1,1). However, it makes it easy to adapt mathematical graphing routines which use this notation for axes and origins. In any mode, all co-ordinates are given as column (left-right) first, then row (up-down).

POS and VPOS

POS is used to establish the current position of the cursor across a stream. It can therefore be used to find out how far the print head

has crossed the platen of a printer. PRINT "A";POS(#0) produces 'A 2', because after printing the 'A', the cursor is at the second print position on the line. The space between the 'A' and the number 2 is due to the leading space given to positive numbers by PRINT. POS(#8) will tell you how far the print head has gone, #8 is the (reserved) printer stream. POS(#9) will tell you how many characters have been sent to the cassette since the last CHR$(13) or carriage return. POS logs all printing characters since the last carriage return sent to any streams – note that non-printing characters (those with ASCII values of 31 or less) are not counted.

VPOS returns the vertical position of the cursor, and because this would be a nonsense on any stream other than one going to the screen, must only be passed a number between 0 and 7 as in 'ycoord = VPOS(#1)'.

MOVE

The graphics cursor can be moved around the screen without setting any pixels by using the reserved words MOVE and MOVER. MOVE requires two numbers, the column and row position to move to. 'MOVE 100,100' moves the graphics cursor to the point which is 100 pixels along the x-axis (columns) and 100 pixels up (along the y-axis) as measured from the origin. The top right-hand corner of the screen is given by the co-ordinates (639,399).

MOVER

MOVER means 'move relative', and moves the cursor relative to wherever it happens to be. The command adds the numbers supplied to the existing values of the row and column graphics cursor location and moves the cursor to the new location. That is, if the cursor is at (100,100) and your program has the command 'MOVER 10,–20', the cursor will be moved to (110,80).

You can move the cursor way off the screen without getting an error message – try 'MOVE 1000,1000', for example. The only error message you'll get here is if you use numbers out of the integer range of –32768 to +32767, and then the error message will be 'Overflow'.

PLOT

To set or illuminate a pixel at a given screen location you use the word PLOT. This needs two numbers (at least) and illuminates the pixel at the co-ordinate referenced by the first two numbers. The colour of the point may be given by a third number, which is the code of the ink to be used – not the actual colour you want. So 'PLOT 50,100,3' illuminates the pixel at (50,100) in whatever colour has been assigned to INK number 3. PLOT not only sets the point indicated, but also moves the cursor to that point.

PLOTR

As you might expect, PLOTR is the 'relative' version of PLOT and, like MOVER and DRAWR, sets a pixel (and moves the cursor to that point) whose co-ordinates are calculated by adding the numbers following the command to the current co-ordinates of the graphics cursor. Just like PLOT, you can add a third number to define the colour in which the pixel is to be set.

DRAW

The cursor can be made to produce a line when it is moved, and this is done with the words DRAW and DRAWR. DRAW moves the cursor to the position given by the first two numbers which follow the command, just like the commands described above, but DRAW also produces a line which links the new point to the last point. Again, you can add a third number to specify an INK to be used for the colour of the line. 'DRAW 20,20,1' will draw a line between wherever the cursor may be to the pixel at (20,20), in whatever colour is produced by INK number 1.

DRAWR

DRAWR produces a line relative to the current cursor position. Thus DRAWR 20,20 means 'draw a line to a point 20 pixels along

the screen and 20 pixels up from wherever the cursor is at the moment'. Because the origin (0,0) is at the bottom left of the screen, if the first number is positive, the cursor will be moved to the right, a positive second number will move the cursor upwards. If the first number is negative, the cursor will move to the left and if the second number is negative, the cursor will be moved down the screen.

TEST and TESTR

Sometimes it's useful to be able to tell what colour a particular pixel is in. There are two commands, TEST and TESTR, that let you do this. 'TEST column,row' produces or returns a number which gives the colour code of the pixel at that point. TESTR does the same, but relatively. If TEST or TESTR produce zero, then the pixel at the defined point is in the background colour. The commands are used as follows:

check = TEST 100,100

or

check = TESTR 10,–10

Note that both TEST and TESTR move the cursor to the point to be tested.

Erasing pixels and lines

Points and lines can be erased by PLOTting or DRAWing in the background colour, which is set to INK 0 when the Amstrad is turned on. For example:

```
10 MODE 0
20 MOVE 0,0
30 DRAW 639,399,4
40 GOSUB 1000:'pause
50 DRAW 0,0,0
60 GOSUB 1000:'pause
70 MOVE 0,399:DRAWR 639,-399,7
80 GOSUB 1000
```

```
90 DRAW 0,399,0
100 GOTO 20
1000 FOR pause = 1 TO 100:NEXT:RETURN
```

This draws a line to the top right-hand corner in light blue ink, then draws another back to the origin in the background colour, rubbing out the first line. It then draws a line from top left to bottom right, in purple, and erases it. The subroutine at line 1000 pauses for 100 cycles of a FOR...NEXT loop. Note that when a 'MODE' command is issued, the cursor is placed at the origin (0,0), and also that any DRAW or DRAWR command without a third argument uses the last INK colour specified.

XPOS and YPOS

The reserved words XPOS and YPOS return the column and row co-ordinates respectively of the graphics cursor. This is especially useful for resetting the cursor to where it was before some pattern is drawn. These words are used not as commands, because they don't move the cursor, but to assign the cursor co-ordination to variables, as in 'xcoord = XPOS:ycoord = YPOS'. The use of this technique is illustrated in the examples given below.

Examples

Before looking at more advanced graphics, it's worthwhile getting some practice with the commands described so far in this chapter. We'll produce some simple shapes like squares and circles.

Squares

To draw a line round the edge of the screen we can use two methods. The slowest is to plot each and every point, starting from the origin at the bottom left of the screen, working up, then right, down the right-hand side and finally along the bottom back to the origin. Each of the four lines can be produced by a FOR...NEXT loop. For example, to travel upwards we need to begin at the origin (MOVE 0,0), then plot all the y (vertical) values from 0 to 399, keeping x (the horizontal co-ordinate) at zero. We use the value 399 as the end point for y because there are 400

vertical points, but we start at zero. This suggests a fragment of code like this:

```
10 MODE 2:MOVE 0,0
20 x = 0
30 FOR y = 0 TO 399
40 PLOT x,y
50 NEXT
```

To move across the top we need to set the y value to 399 and adjust the x values from 0 to 639, plotting each point as we go:

```
60 y = 399
70 FOR x = 0 TO 639
80 PLOT x,y
90 NEXT
```

To move downwards from the top right-hand corner, we first set x to 639, then step down the y values from 399:

```
100 x = 639
110 FOR y = 399 TO 0 STEP - 1
120 PLOT x,y
130 NEXT
```

Finally, we set y to zero and step back along the horizontal values from 639 to zero:

```
140 y = 0
150 FOR x = 639 TO 0 STEP - 1
160 PLOT x,y
170 NEXT
```

This works very well, but it's rather slow. A much faster method is to define the points we want linked by lines, and then draw them. We've worked out that the corners of the screen are given by the co-ordinates (0,0) – bottom left, (0,399) – top left, (639,399) – top right, and (639,0) – bottom right. All we need to do is to move the cursor to each point and draw a line to the next, and so on. You'll be rather surprised at the difference in speed between this program and the last:

```
10 MODE 1
20 PLOT 0,0
30 DRAW 0,399
40 DRAW 639,399
50 DRAW 639,0
60 DRAW 0,0
```

There are other ways of producing the same effect: we could use DRAWR, and you might like to work out a routine to do this using that command. As with all aspects of programming, the best way to learn is to set yourself a problem, then try to solve it.

Graphics subroutines

Subroutines can make programming easier. For example, if we have a program which needs to draw a number of squares, it would be very tedious to have to write the code for every individual square. Instead, we can use a general-purpose square-drawing routine which we can call up at any time to draw a square at any position on the screen. Ideally, the cursor should be returned to the point it was at before the routine was called, and we can arrange for this using XPOS and YPOS to log the cursor co-ordinates on entry to the subroutine, then reset them just before exiting (lines 1000 and 1060 in the routine given below). Because a square has sides of the same length, all we need to tell the subroutine are the top left-hand co-ordinates of the square and the length of the sides. The co-ordinates for the square's top left-hand corner should be set to 'leftcol' and 'toprow' before calling the routine, while the length of the sides is assumed to be held in the variable 'sidelength'. The subroutine goes like this:

```
1000 xcoord = XPOS:ycoord = YPOS
1010 PLOT leftcol,toprow
1020 DRAWR sidelength,0:'top line
1030 DRAWR 0,-sidelength:'right line
1040 DRAWR -sidelength,0:'bottom line
1050 DRAWR 0,sidelength:'left line
1060 MOVE xcoord,ycoord
1070 RETURN
```

Lines 1020 to 1050 add or subtract the length of each side from the column or row co-ordinates of the current cursor to draw the edges.

Circles

The Amstrad is unusual among home computers in that it has no built-in command for drawing circles. However, it's very easy to write a subroutine which adds this facility to your programs. The procedure for drawing a circle is fairly straightforward and can be performed within a FOR...NEXT loop stepping through the range 0 to 360 degrees. If we imagine a circle to be the outer rim of a clock face, 0 degrees is '12 o'clock', 90 degrees corresponds to '3 o'clock', 180 is '6 o'clock' and 270 is '9 o'clock'. 360 degrees is a full circle and brings us back to where we started. Naturally we have to specify the centre of the circle and its radius before calling the routine. Each x co-ordinate on the periphery of the circle is calculated by adding the sine of the number of degrees times the radius to the x co-ordinate of the centre. The y co-ordinate of each point of the circle is given by multiplying the cosine of the number of degrees by the radius and adding that to the y co-ordinate of the centre (see Figure 8.1).

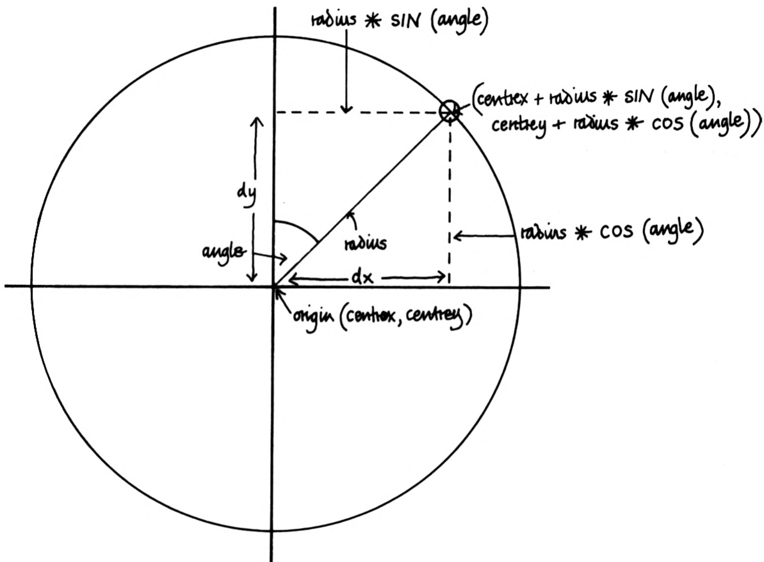


Figure 8.1. The trigonometric functions involved in calculating the co-ordinates of a point on the periphery of a circle. The x co-ordinate is found by adding radius * SIN (angle) to centrex. The y co-ordinate is given by centrey + COS (angle) * radius

The trigonometric functions COS and SIN are built into the Amstrad's Basic, and the theory of their operation is beyond the scope of this book. The only point to note is that you can switch your Amstrad between operating in degrees and radians (the latter is more usual on home computers), by using the reserved words DEG and RAD. As most of us are more familiar with degrees, we'll use that convention here, hence the use of DEG at the start of the subroutine, which runs as follows:

```
1999 REM CIRCLE subroutine
2000 DEG
2010 xcoord = XPOS:ycoord = YPOS
2020 PLOT centrex,centrey
2030 FOR degrees = 1 TO 360
2040 xpoint = centrex + radius * SIN(degrees)
2050 ypoint = centrey + radius * COS(degrees)
2060 PLOT xpoint,ypoint
2070 NEXT
2080 MOVE xcoord,ycoord
2090 RETURN
```

Filling shapes

We can easily arrange to have the square and circle routines fill the shapes they draw. The first thing to do is to set up two variables in the main program – 'yes' and 'no' to −1 and 0 respectively. We can also make use of another variable, 'fillit', which we set to 'yes' or 'no' before calling the routines. In the routines themselves we'll have a few lines of code which do the painting in of the shape, and we'll use a GOTO to skip them if 'fillit = no'. To get this effect, just add the following lines to the squares subroutine:

```
1061 IF fillit = no THEN 1070
1062 FOR row = toprow TO toprow-sidelength STEP -1
1063 MOVE leftcol,row
1063 DRAWR sidelength,0
1064 NEXT
```

This simply draws lines across the box from top to bottom. Don't forget to define 'yes' and 'no' somewhere before the subroutine is called:

```
15 yes = -1:no = 0
```

And you'll have set 'fillit' to 'yes' or 'no' before calling the routine:

```
85 fillit = yes
```

Here are the extra lines for the circle routine:

```
1061 IF fillit = no THEN 1070
1062 DRAW centrex,centrey
```

These lines of code fill circles by drawing lines from each of the points on the periphery to the centre. As above, you must set 'yes' to −1 and 'no' to 0 in an early part of the program, and set 'fillit' to 'yes' if you want a circle filled.

Speeding up graphics

There are faster ways of drawing circles. To begin with we can use integer variables for the degrees, the radius, and so on.

ORIGIN

We can also make use of the fact that we can alter the origin to any column and row values. Normally, the origin for graphics operations is set to (0,0) – the bottom left-hand corner. However, the word ORIGIN allows us to place the origin wherever we want on (or off) the screen. Then we can use DRAWR or MOVER to make relative moves from that point.

If we set the origin as the centre of the circle, then the calculations and operations for plotting the periphery become much simpler and therefore faster. The following subroutine draws circles using 'r%' as the radius, with 'xc%' and 'yc%' as the origin (centre):

```
1000 DEG
1010 FOR d% = 1 TO 360
1020 ORIGIN xc%,yc%
1030 PLOTR r% * SIN(d%),r% * COS(d%)
1040 NEXT
1050 RETURN
```

This routine doesn't reset the cursor to where it was before you called it, but that's easy to rectify just by splicing in the methods shown above.

To modify the routine to fill the circle, change the PLOTR in line 1030 to DRAWR, and if you want to alter the colours of the circle, add a third argument (for the INK or PEN code) to the PLOTR or DRAWR commands.

The next program draws filled circles of all the default colours in MODE 0:

```
10 MODE 0
20 i% = 1
30 r% = 100
40 xc% = 320:yc% = 200
50 GOSUB 1000
60 i% = i% + 1
70 IF i%>15 THEN i% = 0
80 GOTO 50
999 REM Circle routine starts here
1000 DEG
1010 FOR d% = 1 TO 360
1020 ORIGIN xc%,yc%
1030 DRAWR r%*SIN(d%),r%*COS(d%),i%
1040 NEXT
1050 RETURN
```

There are even faster methods of producing circles. Although the routine that follows may seem very long just to produce a simple circle, it's fairly fast. Unfortunately, the mathematical principles behind it are beyond the scope of this book, and we present it as a useful subroutine when speed is important but memory is not. Readers might also get pleasure out of trying to work out just how it works: a clue is that it's based on the fact that the symmetry of the circle means that the 'dx' and 'dy' values in any 45 degree quadrant (see Figure 8.1 on p. 139) are the same as in any other, though some transposition of values may be needed.

```
10 REM fast circle
20 DEG:radius% = 190:DIM point(90,1)
30 ORIGIN 0,0
40 centrex = 320:centrey = 200
50 'Fast circle
60 GOSUB 5000:PRINT delayf;"seconds"
70 '
80 PRINT"Press space to continue"
90 IF INKEY(47) = -1 THEN 90
```

```
100 '
110 'Normal circle
120 GOSUB 6000:PRINT "Fast was";delayf;"seconds":P
RINT"This took";delayn;"seconds"
130 PRINT"Fast to slow ratio =";delayf/delayn
140 '
150 END
160 '
4999 'Fast circle
5000 CLS:const = TIME:PRINT"Calculating"
5010 'Calculate quadrant points
5020 FOR degree% = 0 TO 90
5030 PRINT".";
5040 point(degree%,0) = radius% * SIN(degree%)
5050 point(degree%,1) = radius% * COS(degree%)
5060 NEXT
5070 '
5080 'Plot all points
5090 CLS
5100 FOR degree% = 0 TO 90
5110 PLOT centrex + point(degree%,0),centrey + poi
nt(degree%,1)
5120 PLOT centrex + point(degree%,0),centrey - poi
nt(degree%,1)
5130 PLOT centrex - point(degree%,0),centrey - poi
nt(degree%,1)
5140 PLOT centrex - point(degree%,0),centrey + poi
nt(degree%,1)
5150 NEXT
5160 delayf = (TIME - const) / 300
5170 RETURN
5180 '
5999 'Normal circle
6000 CLS:const = TIME
6010 FOR i%=0 TO 360
6020 ORIGIN centrex,centrey
6030 PLOTR radius% * SIN(i%),radius% *COS(i%)
6040 NEXT
6050 delayn = (TIME - const) / 300
6060 RETURN
```

We also include an even faster method, based on the same principles:

```
10 'Really fast circles
20 'Using eight segments
30 DEG:CLS
40 radius% = 150
```

```
50 xorigin% = 320:yorigin% = 200
60 no.steps% = 16
70 step.angle = 90 / no.steps%
80 chord = (radius% * 2 * PI)
90 chord = chord / (360 / step.angle)
100 int.angle = (180 - step.angle) / 2
110 no.units = no.steps% / 2
120 DIM dx(no.units),dy(no.units)
130 FOR count% = 0 TO no.units
140 angle = 90 - count% * step.angle
150 angle = int.angle - angle
160 dx(count%) = chord * COS(angle)
170 dy(count%) = -(chord * SIN(angle))
180 NEXT
190 xpoint = xorigin%
200 ypoint = yorigin% + radius%
210 PLOT xpoint,ypoint
220 ' 0 to 45 degrees
230 FOR count% = 0 TO no.units
240 DRAWR dx(count%),dy(count%)
250 NEXT
260 ' 45 to 90 degrees
270 FOR count% = no.units TO 0 STEP -1
280 DRAWR -dy(count%),-dx(count%)
290 NEXT
300 '90 to 135 degrees
310 FOR count% = 0 TO no.units
320 DRAWR dy(count%),-dx(count%)
330 NEXT
340 '135 to 180 degrees
350 FOR count% = no.units TO 0 STEP -1
360 DRAWR -dx(count%),dy(count%)
370 NEXT
380 '180 to 225 degrees
390 FOR count% = 0 TO no.units
400 DRAWR -dx(count%),-dy(count%)
410 NEXT
420 '225 to 270 degrees
430 FOR count% = no.units TO 0 STEP -1
440 DRAWR dy(count%),dx(count%)
450 NEXT
460 '270 to 315 degrees
470 FOR count% = 0 TO no.units
480 DRAWR -dy(count%),dx(count%)
490 NEXT
500 '315 to 360 degrees
510 FOR count% = no.units TO 0 STEP -1
520 DRAWR dx(count%),-dy(count%)
530 NEXT
540 END
```

Ellipses

The formula for drawing the outside points of an ellipse is almost the same as that for drawing a circle. The only difference is that you have to use two radii, one for the vertical measure (y co-ordinates), the other for the horizontal (x co-ordinates). Ellipses are described in terms of their height to width ratio, i.e. the vertical radius divided by the horizontal. For an ellipse which is wider than it is tall, the ratio will be less than 1. For ellipses which are taller than they are wide, the ratio will be greater than one. An ellipse with a height to width ratio of one is a circle. The modified circle-drawing routine is now:

```
1000 DEG
1010 xcoord% = XPOS:ycoord% = YPOS
1020 FOR degrees% = 1 TO 360
1030 ORIGIN xcentre%,ycentre%
1040 PLOTR xradius * SIN(degrees%),yradius *
COS(degrees%)
1050 NEXT
1060 MOVE xcoord%,ycoord%
1070 RETURN
```

Here, 'xradius' defines the width, 'yradius' sets the height.

Spirals

Spirals are slightly harder to produce. First, a spiral is more than one turn, and secondly, the radius (distance from the centre of each point) increases with the number of degrees turned. However, these problems are easily overcome. One turn is 360 degrees, so two revolutions are 720 degrees and so on. This simply means altering the main FOR...NEXT loop in the circle routine to 'FOR degrees% = 0 TO 1440', or whatever value suits your needs. This could be done with a FOR...NEXT loop, like 'FOR degrees% = 1 TO 360 * no.turns'. The radius needs to be increased in proportion to the number of degrees turned.

Here's a short spiral subroutine which you can experiment with to see how varying the various values and formulae affects the shape of the spiral. To call the routine you must first define several

variables, 'centrex' and 'centrey' for the centre, starting 'radius' and number of turns (in the variable 'no.turns'). Lines 10 to 30 show this. Line 1060 is the one to alter to change the density of the shape.

```
10 centrex = 320:centrey = 200
20 radius = 1
30 no.turns = 4
40 GOSUB 1000:REM Draw Spiral
50 END
999 REM Spiral-drawing subroutine
1000 CLS:DEG
1010 FOR degrees% = 0 TO 360 * no.turns
1020 ORIGIN centrex,centrey
1030 xpoint% = radius * SIN(degrees%)
1040 ypoint% = radius * COS(degrees%)
1050 PLOTR xpoint%,ypoint%
1060 radius = radius + degrees%/1000
1070 NEXT
1080 RETURN
```

As a final example, here's a very similar listing which extends the spiral routine and shows how to define the colour of points set on the screen with a third parameter for PLOTR (line 1020). Note how the function (FN altrad) which alters the size of the radius is redefined on each pass through the loop.

```
10 CLS:DEG
20 centrex = 320:centrey = 200
30 for deginc = 10 TO 1 STEP -1
40 no.turns = 3
50 for radalt = 1000 TO 300 STEP -50
60 MOVE centrex,centrey
70 DEF FN altrad(n) = radius + degrees%/radalt
80 radius = 1
90 GOSUB 1000
100 NEXT:NEXT
110 END
999' Spiral Routine
1000 FOR degrees% = 1 TO 360 STEP deginc
1010 ORIGIN centrex,centrey
1020 PLOTR radius * SIN(degrees%),radius *
COS(degrees%),RND(1)* 14 + 1
1030 radius = radius + FN altrad(n)
1040 NEXT:RETURN
```

9
Advanced Text and Graphics

Linking text and graphics

While it's always possible to LOCATE the cursor and PRINT on the screen, it's often useful to be able to display text at a particular graphics location. The graphics resolution is much finer than that of the text and is therefore better suited to such tasks as labelling diagrams. The graphics co-ordinate system can also be used for making the movement of characters much smoother.

TAG and TAGOFF

This reserved word has no immediate, apparent effect, but it 'links' the text and graphics cursors, so that the top left-hand pixel of the cursor matrix lies over the graphics cursor.

The example we'll use is that of drawing a clock face. The routine is very similar to those given above for drawing circles, the only addition is the word TAG. This puts the text cursor at the graphics cursor location, while TAGOFF turns off the linkage, leaving the text cursor where it was. We have to step through a FOR...NEXT loop in steps of 30 degrees, because this STEP value gives us degree points of 0, 30, 60, 90, 120, etc., which correspond to the position of the figures on a clock at 12, 1, 2, 3 and 4 o'clock, and so on. A simple clock face routine looks like this:

```
10 MODE 1
20 xcentre% = 320:ycentre% = 200
30 radius% = 100
40 GOSUB 1000
50 END
1000 xcoord = XPOS:ycoord = YPOS
1010 DEG
1020 FOR degrees% = 0 TO 360 STEP 30
1030 ORIGIN xcentre%,ycentre%
1040 PLOTR radius% * SIN(degrees%),radius% *
```

```
COS(degrees%)
1050 TAG
1060 PRINT mid$(str$(hours),2,2);
1070 TAGOFF
1080 hours = hours + 1
1090 NEXT
1100 MOVE xcoord,ycoord
1110 RETURN
```

The TAG instruction means that if the graphics cursor moves, the text cursor has to be moved as well, and this can slow things down. Therefore it's always a good idea to use TAG and TAGOFF to 'sandwich' PRINT statements, as in this listing (lines 1050 to 1070).

In line 1060, STR$ is used to convert the value of the numeric variable 'hours' to its string representation. This is done to avoid the printing of a leading space which would occur if 'PRINT hours' were used.

Note in particular the semicolon at the end of the PRINT statement in line 1060. If you omit this you'll see two strange symbols displayed after each number. These are an arrow pointing down and left followed by a down arrow. These are the symbols for carriage return and line feed which, as described earlier, are produced after every printed item to force the cursor to the start of the next line down. In graphics mode we don't want these characters (CHR$(13) and CHR$(10)) to be visible, so we must use the semicolon to suppress them.

The routine is far from ideal, as you'll note if you PLOT the centre, and draw radii to each 30 degree point. If you want to tidy it up to produce an analogue real-time clock (one with hands), you'll have to adjust the character placings using MOVER.

To illustrate further the use of TAG, and the mixed text/graphics modes the word allows, here's another program. This is more than just an example, but less than a finished product. The listing gives you control over a 'cross-hair', which you can move in the four compass headings using the keys 'z', 'x', '/' and ';'. Pressing space draws converging lines from the bottom right and bottom left-hand sides of the screen to the centre of the cursor. The listing serves to demonstrate a number of points and you should find it fairly easy to splice in your own routines to turn it into a full game.

```
10 ON BREAK GOSUB 340:ON ERROR GOTO 340
20 SPEED KEY 1,1
30 SYMBOL AFTER 249
40 SYMBOL 250,24,24,24,231,24,24,24
```

```
50 ax = 7:ay = 6
60 INK 1,3:INK 3,26
70 MODE 1:curcol = 320:currow = 175
80 cursor$ = CHR$(250)
90 'Move cursor
100 PLOT 0,400,3
110 MOVE curcol,currow:TAG
120 CALL &BD19:PRINT cursor$;:TAGOFF
130 a$ = LOWER$(INKEY$)
140 IF a$ = CHR$(32) THEN GOSUB 250
150 MOVE curcol,currow:TAG
160 CALL &BD19:PRINT CHR$(32);:TAGOFF
170 curcol = curcol - 8 * (a$="x") + 8 * (a$="z")
180 currow = currow - 8 * (a$=";") + 8 * (a="/")
190 if curcol <1 THEN curcol = 639
200 if curcol >639 THEN curcl = 1
220 GOTO100
230 'Fire
240 x = curcol + ax:y= currow -ay
250 MOVE 0,0
260 DRAW x,y,1
270 MOVE 39,0
280 DRAW x,y,1
290 MOVE 0,0
300 DRAW x,y,1
310 MOVE 639,0
320 DRAW x,y,1
330 RETURN
340 CALL &BB00
```

Because TAG and a subsequent PRINT produce a character whose top left-hand corner lies at the current graphics cursor co-ordinates, graphics/text adjustments often have to be made, e.g. line 240 above. Here we want to ensure that the lines converge in the centre of the cross-hair. You should experiment with the values in 'ax' and 'ay' (which stand for 'adjust x' and 'adjust y') to see how the text characters tie in with the graphics co-ordinates. When the cursor is moved, we move it 8 pixels at a time (see lines 170 and 180). Lower values will slow movement considerably. In this example, we use 'curcol' and 'currow' to reference the graphics cursor co-ordinates, then add 7 to 'curcol' and subtract 6 from 'currow' for the cursor's centre. Different modes may require different constants here.

As the routine stands, the cursor 'wraps round' from each side. That is, it will disappear from the right or left-hand sides of the screen to reappear on the other side. But you're free to send it as far up or down as error messages will permit (see above). 'CALL

&BD19' in line 120 waits until the next frame flyback (scan of the TV) occurs before moving the cursor, so making apparent movement smoother. Line 100 plots a point off the screen, using INK 3. This sometimes seems to be necessary to alter the colours used when printing on the graphics screen like this. Line 10 sets up a trap for the escape key being pressed twice, redirecting control to line 340. This is necessary because of line 20, which sets the key delay and repeat to values which are unusable in direct mode. Line 340 itself restores the 'Key Manager' system, resetting SPEED KEY and all key-associated controls back to their default values when the program is stopped by an error or a user interrupt.

Screen write options

Normally, when a character is 'written' to the display, it is 'forced', i.e. the character matrix erases any pixels under it. However, it's possible to switch the screen-handling into 'transparent' mode so that characters are placed on the screen without erasing what's already at that location. This means that you can superimpose one character on another, i.e. the one underneath will show through the one on top. The transparent option allows you to do things like label diagrams more accurately than the normal 'force' mode because you don't have to worry about erasing part of the diagram with part of a character's matrix.

To 'enable' (switch on) the transparent option, you have to PRINT two CHR$:

```
PRINT CHR$(22);CHR$(1);
```

Restoring normal mode needs:

```
PRINT CHR$(22);CHR$(0);
```

Here's an example which displays the message 'HELLO' with 'GOODBYE' on top:

```
10 MODE 1
20 PRINT CHR$(22);CHR$(1);
30 LOCATE 10,10:PRINT "HELLO";
40 LOCATE 10,10:PRINT "GOODBYE";
```

This is a trivial example, but shows how the process works. You can combine selected characters to give many different effects and, as we'll show later, you can use other 'control codes' to make up extra characters.

Printing CHR$s with values below 32, which is where the ASCII character set begins, produces special effects on the screen handling, but does not necessarily produce anything on the display. These codes are called 'non-printing control codes' and are explained below.

The Amstrad provides three other ways of writing pixels or characters to the screen which give the programmer great control over the display.

Pixels or characters can be produced using the three Boolean operators AND, OR and XOR (Boolean operations are dealt with in Chapter 6). Essentially, what happens when these options are selected is that the relevant Boolean operator will be applied between the existing contents of the pixel being written to and the colour specified for writing to it, the result being placed in the relevant screen memory address.

To set any of the different ways of writing to the screen you first have to send one of the non-printing control codes. This is done by printing CHR$(23) followed by CHR$(n), where 'n' is a number between 0 and 3. The default option is 'forced', as described above, and this has a value of 0. To XOR pixels, use 'PRINT CHR$(23);CHR$1;'. AND requires 'PRINT CHR$(23); CHR$(2);' and an OR effect is given with PRINT CHR$(23); CHR$(3);'.

Curvestitch program

This is a computer version of the 'curvestitch' effect (when apparent curves are produced by straight lines). The program makes use of these screen writing options to demonstrate their differences. We won't explain the entire program, since drawing any one side is essentially the same process as drawing any other, and you'll notice that there's just one subroutine to handle drawing the lines. Working from the top left-hand corner of the screen, what the program does is to draw lines from co-ordinates with decreasing y values (with the x co-ordinate held at zero) to meet increasing values of x on the x-axis (y=0).

```
10 REM Curve Stitch - colour version
20 inkcode=2
30 screencode=0
40 inkmask=0
50 xinc=10:yinc=-7
60 CLS
70 '
80 INK 0,inkcode
90 screencode=screencode+1
100 screencode=screencode MOD 4
110 PRINT CHR$(23);CHR$(screencode);
120 '
130 REM bottom left
140 xEND=1:yEND=1
150 xstart=1:ystart=400
160 GOSUB 380
170 '
180 REM top right
190 xstart=640:ystart=1
200 yend=400:xend=640
210 xinc=-xinc:yinc=-yinc
220 GOSUB 380
230 '
240 REM bottom right
250 xstart=640:ystart=400
260 xend=640:yend=1
270 yinc=-yinc
280 GOSUB 380
290 '
300 REM top left
310 xstart=1:ystart=1
320 xend=1:yend=400
330 xinc=-xinc:yinc=-yinc
340 GOSUB 380
350 '
360 GOTO 80
370 '
380  PLOT xstart,ystart
390 DRAW xEND,yEND,inkmask
400 ystart = ystart+yinc
410 xEND=xEND+xinc
420 '
430 IF xstart<1 OR xstart>640 THEN RETURN
440 IF xend <1 OR xend>640 THEN RETURN
450 IF ystart<1 OR ystart>400 THEN RETURN
460 IF yend<1 OR yend>400 THEN RETURN
470 '
480 GOSUB 530
490 '
```

```
500 GOTO 380
510 '
520 ' change colours
530 inkcode=inkcode+1
540 IF inkcode>24 THEN inkcode=0
550 inkmask=inkmask+1
560 IF inkmask>6 THEN inkmask=0
570 IF inkcode=inkmask THEN 550
580 RETURN
```

Moire program

Interference patterns can be produced quite easily on the Amstrad, and we include an example here. Try using different screen write options in different MODEs to explore the possibilities.

```
10 REM Interference patterns
20 MODE 1
30 GOSUB 170
40 PRINT CHR$(23);CHR$(scrmode);
50 y% = 400
60 FOR x% = 640 TO 0 STEP -4
70 ORIGIN 0,0
80 DRAWR x%,y%,colour%
90 NEXT
100 GOSUB 170
110 FOR x% = 0 TO 640 STEP 4
120 MOVE 640,0
130 DRAW x%,y%,colour%
140 NEXT
150 GOSUB 170
160 GOTO 60
170 LOCATE 7,24:PRINT SPACE$(30);
180 LOCATE 7,25:PRINT SPACE$(30);
190 LOCATE 7,24
200 INPUT"Mode 0 - 3";scrmode
210 RESTORE:FOR i = 1 TO scrmode
220 READ scrmode$:NEXT
230 LOCATE 20,24
240 PRINT"  ";scrmode$;
250 LOCATE 7,25
260 INPUT"Colour 0 - 4";colour%
270 RETURN
280 DATA Normal (forced),XOR,AND,OR
```

Non-printing control codes

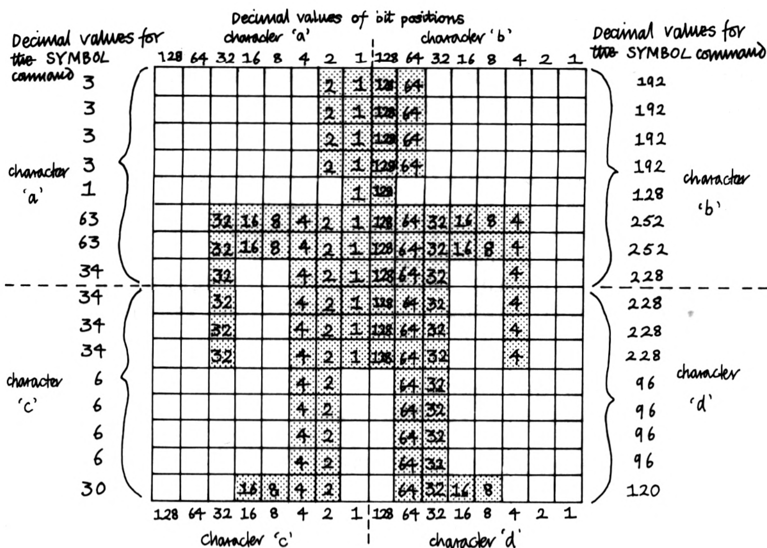
These are the set of screen-handling commands effected by 'PRINT CHR$(n)', where n is a value between 0 and 31. They do not produce characters on the screen unless they are immediately preceded by CHR$(1), as in 'PRINT CHR$(1);CHR$(12)'. This makes an extra 32 characters available, and their default symbols are shown by this simple routine:

```
10 MODE 1
20 FOR char = 0 to 31
30  PRINT CHR$(1);CHR$(char)
40   FOR pause = 1 TO 100
50   NEXT pause
60 NEXT char
```

Normally CHR$(12) clears the screen – it's the screen version of 'form feed' on a printer. CHR$(8) is a backspace (i.e. 'PRINT CHR$(8)' moves the cursor one character position to the left), 'PRINT CHR$(9)' moves the cursor to the right. Some codes are of particular value – CHR$(20) clears the screen from the cursor to the bottom right-hand corner, while CHR$(18) clears the display from the cursor to the end of the current line. This can be useful for making sure that an item of text appears on its own on a line: 'PRINT CHR$(18);prompt$;' erases any other items on the current line, then displays the characters in 'prompt$'.

Extra-large characters

Because the non-printing control codes are accessed by CHR$ you can treat them just as if they were normal characters. This means that they can be incorporated into longer strings by concatenation (using '+'), or used between your own defined characters. In MODEs 1 and 2, the characters are rather small and it can be useful to have larger shapes than the normal character size. One way of doing this is to link normal characters together with control codes so that they appear as one when printed. For example, you could define the four characters to form the quadrants of a square, let's call them 'a', 'b', 'c', and 'd' (see Figure



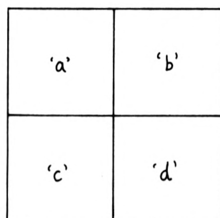


Figure 9.1. Creating 'extra-large' characters by joining four normal-sized characters

9.1). You could call the characters a$, b$, c$ and d$, and join them like this:

```
square$ = a$ + b$ + CHR$(10) + CHR$(8) + CHR$(8)
+ c$ + d$
```

Now, whenever you 'PRINT square$', the four characters will appear together. The technique is straightforward: a$ and b$ are displayed first, CHR$(10) is the control code for 'line feed' and tells the Amstrad to move the cursor down one line. CHR$(8) is a

backspace, and as there are two of these the cursor will be moved to the character position under the 'a' quadrant (quadrant 'c') before printing the shapes represented by the string variables 'c$' and 'd$'. As an example of this, try:

```
10 MODE 1
20 large.char$ = CHR$(214) + CHR$(215) + CHR$(10)
+ STRING$(2,8) + CHR$(213) + CHR$(212)
30 era.char$ = STRING$(2,32) + CHR$(10) +
STRING$(2,8) + STRING$(2,32)
40 LOCATE 1,1:GOSUB 1000:LOCATE 1,1:GOSUB 2000
50 LOCATE 10,10:GOSUB 1000:LOCATE 10,10:GOSUB
2000
60 GOTO 40
1000 PRINT large.char$;
1010 FOR pause = 1 TO 500:next
1020 RETURN
2000 PRINT era.char$;:RETURN
```

Note the use of STRING$ in lines 20 and 30 to generate two backspaces (CHR$(8)+CHR$(8)) and two spaces (CHR$(32)+ CHR$(32)).

To make further use of this technique, you could draw up a 16 × 16 grid on which to design a shape, then split it into four squares of 8 × 8 which you could then use to calculate the numbers required for the four SYMBOL commands (see Figure 9.1). The character squares could be linked as above, and other, larger shapes can be constructed and used in this way.

Another useful control code is CHR$(31). This operates very like the Basic reserved word 'LOCATE', and like it has to be followed by two numbers, giving the column and row (text) co-ordinates to which the cursor is to be moved. It's used as in 'PRINT CHR$(32);CHR$(5);CHR$(10);' which would move the cursor to the fifth column of the 10th row. The reason it's useful is because it can be used to bypass a curious feature of the Amstrad's screen-handling. What happens is that the Amstrad sometimes formats strings oddly, and long numbers printed without the 'USING' format command may also suffer from this problem. To see the difficulty, RUN the following:

```
10 MODE 1:row = 1
20 FOR col = 1 TO 20
30 LOCATE col,row
40 PRINT "Is this too long?"
50 NEXT
```

You should find that instead of wrapping the message round from one side of the screen to the next when it would be too long to fit on a line, the Amstrad prints a carriage return and line feed before displaying the message, causing it to appear at the start of the line. CHR$(31) allows you to avoid this, and combining it with DEF FN adds a useful facility, a revised version of LOCATE:

```
DEF FN place$(col,row) = CHR$(31) + CHR$(col) +
CHR$(row)
```

Now, instead of using a line like '100 LOCATE 13,24:PRINT "Press space" ', you can use 'PRINT FN place$(13,34) + "Press space" '.

Bouncing ball routine

A routine that can be used and modified for many games is the bouncing ball. We'll give the bare bones of the process here so that you can develop it as you like.

To move a character on the screen we must first be able to control its position, which we do using variables for its column and row co-ordinates. Then we can LOCATE the cursor at any screen position and display the character with PRINT. To make the character appear to move we must erase it by locating the cursor at its co-ordinates and printing CHR$(32), a blank space. The next step is to update the character's co-ordinates, and so the cycle repeats.

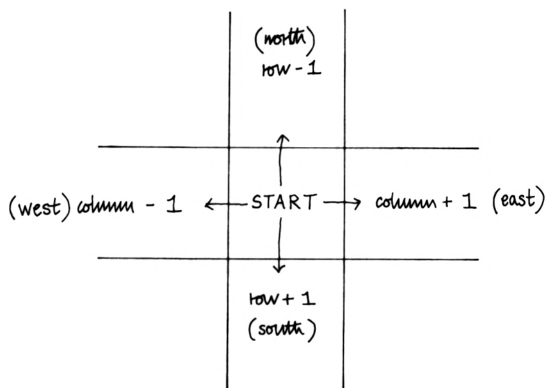


Figure 9.2. Moving in the basic compass directions

To move a character upwards on the screen means subtracting from the character's row co-ordinate, and moving down involves adding to this variable. Movements left and right mean altering the column co-ordinate: add to move right, subtract to move left (see Figure 9.2). Combining these movements to calculate column and row alterations for 'up and right' (i.e. north east), it's clear that this means adding one to the character's column and subtracting one from its row. Applying the same principles we can work out the column and row alterations for north west, south east, or south west (see Figure 9.3 and the table below).

Direction	column (x)	row (y)
north east	+	—
north west	—	—
south east	+	+
south west	—	+

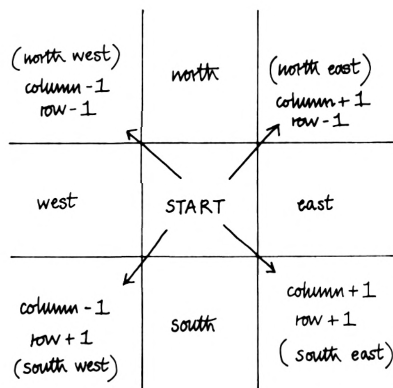


Figure 9.3. Combining compass movements to calculate row and column differences for diagonal movement

Note that the value added to the column and row will determine the speed at which the character appears to move.

What will happen if the character reaches the edge of the screen? If we don't change its direction it will either disapear off the screen, or an error message will be produced, as would occur if either the row or column variables became less than one (try 'LOCATE 0,0'). To prevent this we need to test the values that the column and row co-ordinates would be on the next move. If they're outside the screen limits then the character's direction of

movement needs to be changed. If the character is moving upwards, it must start moving down and vice versa, and the same principle applies to left and right movement. We can use two variables, 'xdir' and 'ydir' to control the direction of movement of the character, and these may have values of +1 or −1. They are added to the character's column or row co-ordinates to update the character's location on the screen. If 'xdir' is −1 we know that the character is moving to the left, because when this is added to the column co-ordinate the latter is reduced by one. A value of + 1 in 'xdir' indicates movement to the right. To reverse the horizontal direction of movement all we need is an instruction to reverse the sign of 'xdir', such as 'xdir = − xdir' (remember that −(−1) = +1). The same principles for logging and controlling up and down movements apply to 'ydir'.

A similar technique is used in the game described in Chapter 5, but here we use the pixel resolution of the graphics facility of the Amstrad and TAG to make movement smoother. The routine looks like this:

```
10 xcoord = 320:ycoord = 200
15 'Directions - south west at the start
20 xdir = -1:ydir = 1
30 MODE 1
35 'Erase character first
40 MOVE xcoord,ycoord
50 TAG:PRINT CHR$(32);:TAGOFF
55 'Update x
60 xcoord = xcoord + xdir
65 'Update y
70 ycoord = ycoord + ydir
75 'Hit side?
80 IF xcoord < 1 OR xcoord > 631 THEN THEN xdir =
- xdir:xcoord = xcoord + xdir
85 'Hit top/bottom?
90 IF ycoord < 1 OR ycoord > 391 THEN ydir = -
ydir:ycoord = ycoord + ydir
95 'Move graphics/text cursors & display
character
100 MOVE xcoord,ycoord
110 TAG:PRINT "O";:TAGOFF
115 'and repeat
120 GOTO 40
```

Windows

A window is a rectangular area of the screen which you can treat almost as a small screen: you can clear it to a given colour or display messages in it. The Amstrad allows you to define up to eight such windows on the screen.

Windows are set up with the WINDOW instruction. This takes five arguments: the reference number for the window, its left and right column limits, and its top and bottom rows. Windows can overlap and you can send messages, etc. to any window using 'PRINT #n', where 'n' is a window number between 0 and 7. It's important to realise that these are not real windows, in the sense that when information in one window overlaps information in another, the latter is erased and will not reappear when the upper window is cleared.

#0 is the default window. When the Amstrad is turned on it is defined as the size of the screen display and error messages always appear in window #0. Windows are sometimes referred to as streams or channels: #8 is the printer stream, so 'PRINT #8' will hang up the machine unless a printer is connected and turned on (but ESC, ESC will return you to direct mode). #9 is the cassette channel: 'PRINT #9' sends data to the cassette, but a file must have been opened first (cassette handling is explained in Chapter 11).

Once several windows have been defined you can print in them with PRINT #n, and if you fill them they'll scroll within their limits. Windows which overlap are not independent: information scrolls within a given area of the real screen, regardless of which window's contents may be affected.

Windows can be exchanged with 'WINDOW SWAP w1,w2', where w1 and w2 are the window numbers to be exchanged. This allows you to divert printed messages to any of your defined windows very easily.

Setting up windows

The general form of the WINDOW instruction is 'WINDOW #n,left,right,top,bottom', but you don't have to give the arguments in that precise order. The stream or window number must come

first, but then whichever of the next two values is the greater will be taken as the right-hand column of the window, the smaller as the left-hand column limit. The same applies to setting the top and bottom rows of the window.

Windows and colours

Many of the text output commands such as CLS, PRINT, PAPER, PEN and LIST can be followed by a number to indicate to which window the command relates. PAPER #2,3 sets the paper or background colour for window number two to whatever colour has been assigned to INK number three. PEN #2,5 means 'use pen number five in window number two', so text sent to that window will appear in whatever colour ink has been assigned to that pen.

So much for the theory. It's quite easy once you get the hang of it, but even then it's a very good idea to make notes when you're programming of which inks are set to which colours, and so forth. Going back a stage, it's good practice to make a sketch of where each window is to appear on the real screen (preferably on squared paper), well before sitting down to enter the program.

Here's a simple example of using windows in MODE 0. It should give you some idea of how to tackle windowing and shows some of the problems you may encounter. The screen is split into four sections, each of which is given a different background and foreground colour, then different letters are displayed in each of the windows. Note that the LOCATE instruction can be used to format messages in windows, but that its arguments always refer to the real screen (see below). Note too that colour settings are not restored at the end of a program, nor when switching from MODE to MODE.

```
10 'Demonstration of windows
20 MODE 0
30 WINDOW #1,1,10,1,12
40 WINDOW #2,11,40,1,12
50 WINDOW #3,1,10,12,25
60 WINDOW #4,11,40,12,25
70 PAPER #1,1:PEN#1,0
80 INK 0,0:'set INK 0 to Black
90 CLS #1
100 PAPER #2,3:PEN #2,4
110 CLS#2
120 PAPER #3,4:PEN #3,2
```

```
130 CLS#3
140 PAPER #4,7:PEN #4,13
150 CLS #4
160 FOR count = 1 TO 26
170 FOR channel= 1 TO 4
180 PRINT#channel,STRING$(10,channel+count+64);
190 NEXT channel,count
200 LOCATE 5,12:PRINT"THAT'S ALL"
210 GOTO 210
```

Using windows

A window might be defined as 'WINDOW #1,10,30,6,18' and 'PRINT #1,"Message";' will send the string to the window. The main thing to remember is that if you want to use LOCATE, the numbers you use refer to the real screen, so it's a good idea to use variables to store the values of the left/right and top/bottom limits of each window. The window definition above becomes:

```
10 w1left = 10:w1right = 30
20 w1top = 6:w1bottom = 18
30 WINDOW #1,w1left,w1right,w1top,w1bottom
```

This makes printing to a window much easier. For example, to place the letter 'A' in the first character location of the window, you'd use 'LOCATE w1left,w1top:PRINT #1,"A" '. To have strings 'centred' on their print line you could use a function to find the starting column for the string (provided the string's length is less than the width of the window):

```
10 DEF FN centre(left,right) = ROUND(((right -
left) / 2) - (LEN(a$) /2))
```

Now set 'a$' to the string of characters you want centred, 'left' and 'right' to 'wnleft' and 'wnright' (the left and right limits of window 'n') and LOCATE the cursor with 'LOCATE FN centre (left,right),row:PRINT #n,a$' (where 'row' is the screen row of the window in which you want the message displayed).

With all this messing about with colours and windows, you're bound to get a bit lost at some point. Inadvertently setting the foreground to the same colour as the background, or defining tiny windows, are common causes. However, there is a ROM routine which may help. To reset all the colours back to their default values, use 'CALL &BB80' which resets the entire VDU system.

Window overlaps

Completely separate windows are fairly easy to manage, but you may encounter some difficulties if you make windows overlap. To see this, alter the window definition line in the previous example as shown below, then RUN the program again.

```
20 WINDOW #1,1,10,1,12
30 WINDOW #2,10,20,1,12
40 WINDOW #3,1,10,12,25
50 WINDOW #4,10,20,12,25
```

Now, because the windows overlap at column 10 and row 12, you'll see that when the first quadrant scrolls it picks up the top line of the quadrant beneath. And the top right quadrant does the same with the bottom right-hand section. The result is that at the end of the program the screen is filled with colours from the bottom two sections. This is yet another reason for careful program design – scrolling windows can play havoc with the display.

Contours

The final example in this chapter is quite a complex graphics program. Given the co-ordinates for two polygons, one inside the other, it draws the 'contour' lines between each 'corner', point or vertex of the outer shape to the nearest point on the inner shape. You can alter the DATA statements to have your own shapes drawn, these must be given in clockwise order, x then y co-ordinate of each point, and the outer shape comes first. If you change the number of points for either shape, you'll have to alter the values of 'n1' or 'n2'.

The program stores the data for each shape in two two-dimensional arrays, 'x' and 'y'. Subscripts (n,1) give the vertices for the outer shape, (n,2) refers to the points of the inner shape. The program calculates which of the inner points is nearest each outer point, storing the results in the array 'd'. The calculation adds the absolute values of the differences between the x and y co-ordinates of the inner and outer points. For each outer point these differences are ranked and the smallest

difference gives the nearest inner point. Then it's just a question of drawing the contours. The variable 's' dictates the density of the detail, i.e. how closely lines are drawn, and high values can be used in MODE 2 for intricate patterns.

```
10 'Contour Drawing
20 'Set up your own MODE and colours
30 MODE 2:INK 0,0:INK 1,26
40 '
50 'n1 = outer points, n2 inner
60 n1 = 8:n2 = 4
70 ' Set up dimensions for outer, inner
80 'd() is for differences
90 '(nearest points)
100 DIM x(2,n1),y(2,n1),d(n1,n2)
110 '
120 'Read in DATA - outer shape
130 FOR k = 1 TO n1 - 1
140 READ x(1,k),y(1,k)
150 NEXT
160 'Move to start of outer shape
170 PLOT x(1,1),y(1,1)
180 'Draw lines, point to point
190 FOR k = 1 TO n1 - 1
200 x1 = x(1,k):x2 = x(1,k + 1)
210 y1 = y(1,k):y2 = y(1,k + 1)
220 a = x2 - x1:b = y2 - y1
230 DRAWR a,b
240 NEXT
250 'Save coords of last point
260 a = XPOS:b = YPOS
270 DRAWR x(1,1) - a,y(1,1) - b
280 '
290 'Read in DATA for inner shape
300 FOR k = 1 TO n2
310 READ x(2,k),y(2,k)
320 NEXT
330 MOVE x(2,1),y(2,1)
340 'and draw it
350 FOR k = 1 TO n2 - 1
360 x1 = x(2,k):x2 = x(2,k + 1)
370 y1 = y(2,k):y2 = y(2,k + 1)
380 a = x2 - x1:b = y2 - y1
390 DRAWR a,b
400 NEXT
410 a = XPOS:b = YPOS
420 DRAWR x(2,1) - a,y(2,1) - b
430 '
```

```
440 'Now calculate differences
450 ' between outer & inner points...
460 FOR d = 1 TO n1:FOR d2 = 1 TO n2
470 d(d,d2) = ABS(x(1,d) - x(2,d2)) + ABS(y(1,d) -
 y(2,d2))
480 NEXT:NEXT
490 '...and rank them
500 FOR r = 1 TO n1
510 FOR b = 1 TO n2
520 bn = 0:FOR c = 1 TO n2
530 IF d(r,c) > bn THEN bn = d(r,c):cn = c
540 NEXT
550 d(r,cn) = n2 - b + 1
560 NEXT:NEXT
570 a = x(1,1):b = y(1,1):ox = a:oy = b
580 MOVE a,b
590 'Set detail - high=fine, low=coarse
600 s = 50
610 FOR i = 1 TO s - 1
620 FOR f = 0 TO i - 1:st = st + 1 / s
630 NEXT f
640 FOR ep = 1 TO n1
650 FOR ss = 1 TO n2
660 IF d(ep,ss) = 1 THEN j = ss
670 NEXT
680 '
690 'Now draw the 'in-betweens'
700 a = x(1,ep) + (st * (x(2,j) - x(1,ep)))
710 a = ROUND(a)
720 b = y(1,ep) + (st * (y(2,j) - y(1,ep)))
730 b = ROUND(b)
740 DRAWR a - ox,b - oy
750 ox = a:oy = b:NEXT:st = 0
760 NEXT
770 '===========ALL DONE==============
780 'DATA is x,y
790 'Outer shape coordinates
800 DATA 10,390,320,350,520,250,600,125,450,10,220
,10,160,50,80,200
810 'Inner shape coords
820 DATA 300,300,380,250,350,200,340,200
```

10
Sound

In this chapter we'll detail the Amstrad's sound-producing commands. There are listings to help you experiment with what is one of the most complex of the Amstrad's features, and we'll explain how to use the Amstrad to produce musical and other sounds.

Sound is produced via the Amstrad's internal speaker, but you can also use portable stereo cassette player headphones to listen to the sounds and this allows you to exploit the stereo effects. You can connect your Amstrad to your hi-fi system to amplify the sound – the internal speaker is too small to do justice to the sound quality produced. Headphones or hi-fi should be connected to the Amstrad via the socket at the very right-hand side of the rear of the case (when viewed from the rear).

Basic sound

The basic sound-producing command is SOUND. You can pass this command up to 7 numbers, but you can get away with only two. 'SOUND 1,284' will produce a note – International 'A', the A above middle C – for 1/50th of a second (0.02s). The first number is the 'channel' through which the sound is to be made, the second number is the 'period' of the sound and is related to the frequency. The sound lasts for 1/50th of a second – the default sound length or 'duration'. There are three channels through which sound may be produced, and we'll follow Amstrad's convention of designating these by the letters A, B and C. The value 284 produces the note 'International A' which has a frequency of 440 cycles per second, 'middle C' has a frequency of 478 cycles per second. Your Amstrad manual, Appendix 7, pp. 1-3, gives details of the numbers to use for the different notes and octaves. Note that the numbers given under the heading 'period' are the numbers to use with SOUND, those given under 'frequency' are the cycles per second of the sound produced.

Channels and duration

SOUND can be passed a minimum of two arguments, but can deal with far more complex details. For example, you may specify the duration of a sound with a third argument, which dictates the length of the sound in 1/100ths of a second. 'SOUND 2,284,100' means 'play International A through sound channel B for one second'. The longest duration you can supply is 255, giving a note of about 2.5 seconds long.

You might expect that to play sound through channel C would need a command beginning 'SOUND 3,...', but you'd be wrong. To use the third channel you use 'SOUND 4,...'; other possible values include 8, 16, 32, 64, 128 and combinations of these. Channel arguments are bit-mapped (see Chapter 6).

Having three channels allows you to do things like play a basic tone through one channel and add harmonics from other channels. Moreover, if you connect stereo headphones or a stereo system to the Amstrad, you'll notice that sound from channel A is directed to the left, channel B to the right, with channel C in the middle. You could make use of this to produce some interesting stereo effects, e.g. by making sounds apear to move by cycling them through channels A, B and C.

So far we have 'SOUND channel,period,duration', which is enough to produce the rudiments of music. Here is a program which sets up a numeric array (note%) to hold the period values for the notes in the various octaves:

```
10 REM Music pitches
20 '
30 CLS
40 '
50 'Data for notes and their values
60 'First (upper) octave
70 DATA F#,F,E,D#,D,C#,C,B,A#,A,G#,G
80 '
90 'Read note names
100 DIM note$(12),note%(9,12)
110 FOR note% = 1 TO 12
120 READ note$(note%)
130 NEXT
140 '
150 PRINT:PRINT"Calculating note values"
```

```
160 'Calculate note values
170 FOR octave% = 4 TO -4 STEP -1
180 realoctave% = octave% + 5
190 FOR note% = 1 TO 12
200 frequency = 440 * (2 ^ (octave% + (10 - note%)
 / 12))
210 period=ROUND(125000/frequency)
220 PRINT".";
230 IF period > 4095 THEN 250
240 note%(realoctave%,note%) = period
250 NEXT:NEXT
260 '
270 CLS
280 '
290 GOSUB 350
300 CLS
310 GOSUB 510
320 END
330 '
340 'Play notes
350 LOCATE 1,1:PRINT"Octave":
360 LOCATE 1,2:PRINT"Note"
370 LOCATE 1,3:PRINT"Period"
380 '
390 FOR octave% = 5 TO 8
400 FOR note% = 12 TO 1 STEP -1
410 LOCATE 7,1:PRINT octave%
420 LOCATE 6,2:PRINT note$(note%);"  ";
430 LOCATE 8,3:PRINT note%(octave%,note%);
440 IF note%(octave%,note%) = 0 THEN 470
450 SOUND 1,note%(octave%,note%),100
460 FOR i = 1 TO 1000:NEXT
470 NEXT:NEXT
480 RETURN
490 '
500 'Scale of C
510 PRINT"Scale of C"
520 FOR octave = 5 TO 7
530 FOR note = 12 TO 1 STEP -1
540 IF RIGHT$(note$(note),1) = "#" THEN 560
550 SOUND 1,note%(octave,note)
560 NEXT:NEXT
570 RETURN
```

The formula used in lines 200 and 210 is that given in the Amstrad manual, Appendix 7, p. 3. You can use the basic routine (lines 170 to 250) at the start of your own programs, then use the array to pass values to SOUND with:

```
SOUND channel,note%(octave,note),duration
```

The variable 'octave' should contain a number between 1 and 8 and 'note' a value between 1 and 12. The string array 'note$()' holds the names of the notes, and is used in the second demonstration (lines 520 to 560) in the listing to play the scale of C (which contains no sharps or flats). The method used is to skip note names found in the 'note$' array if the last character of the note name is '#'. A much better method of generating scales is given below.

Sound in full

You can use seven arguments with the SOUND command:

Argument	Range	
channel	1 to	128
period	0 to	4095
duration	−32768 to	+32767
start volume	0 to	7
volume envelope	0 to	15
tone envelope	0 to	15
noise period	0 to	31

Not all these arguments have to be used. The last can be omitted, as it adds white noise (a sort of hiss, generated by playing random frequencies) to tones (see below). According to the Amstrad manual, the value should be a number in the range 0 to 15, though you can use numbers in the 16 to 31 range and get predictable effects without error messages. You don't have to specify volume or tone envelopes; you can use a zero at these positions in the argument list, or two commas. Omitting the tone or volume envelope numbers makes the Amstrad default to the predefined zero envelopes. It's always a good idea to use the full list, with variables, as in a subroutine like this:

```
10000 SOUND channel, period, duration, svolume,
volenv, tonenv, noiseper
10010 RETURN
```

Remember, it's easier to alter the values of variables than all the SOUND commands in a program.

Frequency

A pure tone can be represented by a sine wave when we plot it on a graph like Figure 10.1. The crest-to-crest time distance is a measure of the pitch, or frequency of the sound. The closer the peaks, the higher the note. When we say that International A has

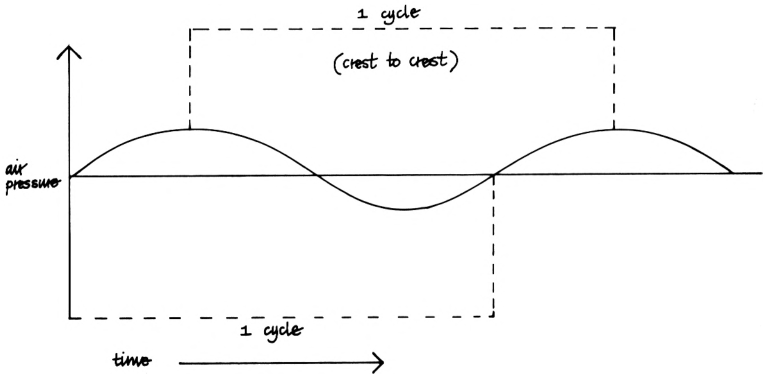


Figure 10.1. Sine wave

a frequency of 440, what we really mean is that there are 440 cycles per second. A cycle can be measured as the time taken for the vibration to pass the 'x' axis (no volume) and return to it, as shown in Figure 10.1. This is the same as the crest-to-crest distance. 'Cycles per second' is often abbreviated to 'cps' or 'Hz' (for Hertz), thus 478Hz is middle C.

The human ear responds to frequencies in the approximate range 50Hz to 15000Hz. The lower and upper frequencies of this range sound more like rumbles and high-pitched whistles than musical notes, which are generally in the range 100 to 4000.

Notes

Musical notes are given the letters A to G, some of which can have superscripts # (sharp) or b (flat). The distance between one note and its neighbours is called a semi-tone, two semi-tones

make a tone and the sequence which forms the basis of Western music runs:

A, $A^{\#}$, B, C, $C^{\#}$, D, $D^{\#}$, E, F, $F^{\#}$, G, $G^{\#}$

Note that there is no note $B^{\#}$, and no $E^{\#}$. Flats are equivalent to sharps, i.e. $A^{\#}$ is the same as B^{b} and $G^{\#}$ is the same as A^{b}. That is, the above sequence is the same as:

A, B^{b}, B, C, D^{b}, D, E^{b}, E, F, G^{b}, G, A^{b}

Much of music is based on the concept of the 'scale'. To find the notes in the scale of a particular note or 'key' you must follow the sequence Tone, Tone, Semitone, Tone, Tone, Tone, Semitone, which we'll abbreviate to TTSTTTS, giving eight notes or an

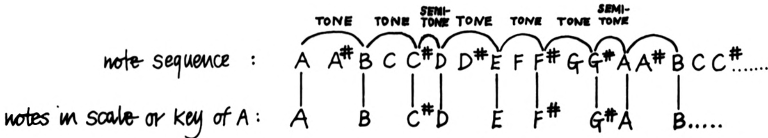


Figure 10.2. Using the TTSTTTS formula to work out the scale of A

'octave'. Figure 10.2 shows how to work out which notes form the scale of A. We begin at A and count Tone (giving B), Tone ($C^{\#}$), Semitone (D), Tone (E), Tone ($F^{\#}$), Tone ($G^{\#}$) and Semitone (A of the next octave), giving the sequence A B $C^{\#}$ D E $F^{\#}$ $G^{\#}$ A. The scale of C is a sequence of notes with no sharps or flats – C D E F G A B C.

Using the TTSTTTS formula, we can devise a routine to play any scale beginning with any note in the array. First we set up a string array to hold the sequence 'TTSTTTS' and an integer array to hold the number of notes to count from one note to the next in the scale:

```
10 scale$ = "TTSTTTS"
20 DIM scale%(7)
```

Then we step through 'scale$', assigning values to 'scale%' using Boolean logic (see Chapter 6):

```
30 FOR note% = 1 TO 7
40 astep$ = MID$(scale$,note%,1)
50 scale%(note%) = -2 * (astep$ = "T") - (astep$
= "S")
60 NEXT
```

This is the same as:

```
30 FOR note% = 1 TO 7
40 IF MID$(scale$,note%,1) = "T" THEN
scale%(note%) = 2
50 IF MID$(scale$,note%,1) = "S" THEN
scale%(note%) = 1
60 NEXT
```

This sets up values in scale% like this: '2212221' which gives a numeric version of 'TTSTTTS' and provides a look-up table for playing notes in a scale using the array of period values set up in the previous listing. For example, having set up the array, you could use a routine like this to play a scale:

```
100 INPUT "octave",oct
110 INPUT "note",anote
120 INPUT "duration",dur
130 REM Trap illegal values...
140 SOUND 1,note%(oct,anote),dur
150 FOR count = 1 TO 7
160 anote = anote - scale%(count)
170 IF anote < 1 THEN oct = oct - 1
180 IF anote = 0 THEN anote = 12
190 IF anote = -1 THEN anote = 11
200 SOUND 1,note%(oct,anote),dur
210 NEXT
```

Chords

There are many types of chord, from major to minor and augmented seventh. We've only space here to give you a brief outline of the simplest.

Major and minor chords consist of three notes played simultaneously – so it's useful that the Amstrad has three channels and that sounds can be made to coincide! First you must work out which notes form the scale of the note from which you wish to start, as shown in Figure 10.2. Then you must calculate the notes which make up the major chord. These are given by the

starting note itself, the note two notes above it and the note two notes above that. The sequence is thus 1, 3, 5. The C major chord is taken from the scale of C (C D E F G A B C), and the notes used are C, E and G, i.e. the 1st, 3rd and 5th notes of the scale. Using octave 2 (Amstrad manual, Appendix 7, p.3), this gives:

```
SOUND 1,119,100
SOUND 2,95,100
SOUND 4,80,100
```

Minor chords require that the second note be flattened, i.e. taken down a semitone, not from the scale itself, but from the absolute twelve note sequence, so the C minor chord is C, E^b, G.

```
SOUND 1,119,100
SOUND 2,100,100
SOUND 4,80,100
```

Tone envelopes

ENT stands for tone envelope. It allows you to specify how the frequency of a note should change as it's played. The easiest way to think of this is to imagine a Swanee whistle or trombone – as each note is sounding the player can make the note 'slide' up or down. Similarly, a singer or violinist may use 'vibrato' to make a note 'wobble' – what's happening is that the basic frequency of each note is raised and lowered rapidly as it sounds.

The Amstrad cannot alter the pitch of a tone as smoothly as a musical instrument. To define tone envelopes you have to compromise by using a staircase pattern. For example, if you want to raise the tone period of a note by 100 units (see Figure 10.3) you must first decide how long the note is going to last. Let's say 1 second. This we can divide into 100 steps, giving a step length of 1, i.e. 0.01 seconds (step lengths are always multiples of 0.01 seconds). As there are 100 steps and we want to raise the period by 100 units, each step will have a value of —1, because lower period values produce higher pitches. These values are used in the ENT instruction like this:

```
1000 ENT 1,100,-1,1
```

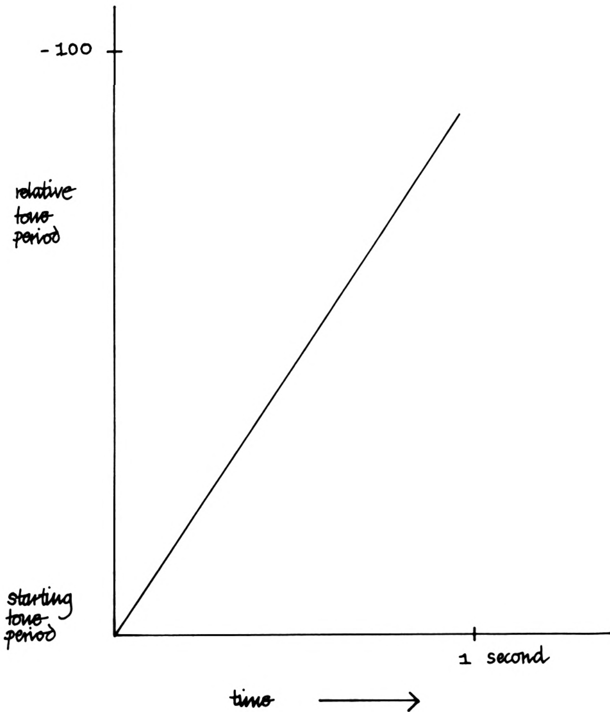


Figure 10.3. Raising the pitch of a tone by 100 units over one second. Note that the tone period is reduced to raise the frequency

The first number is the envelope number, the second is the number of steps, the third the change in tone period for each step and the fourth the length of time each step is to last. The tone period increment is negative because you have to reduce the tone period to raise the pitch. Figure 10.4 shows how the numbers relate to the sound. To hear how this envelope alters sounds, try:

```
10 ENT 1,100,-1,1
20 SOUND 1,300,100,7,0,1
```

It's worth changing some of these values to hear how it affects the sound. There need not be 100 steps, for example. All that's required is that the tone period be raised by a certain amount over a given period of time. If we chose to have 50 steps, we would

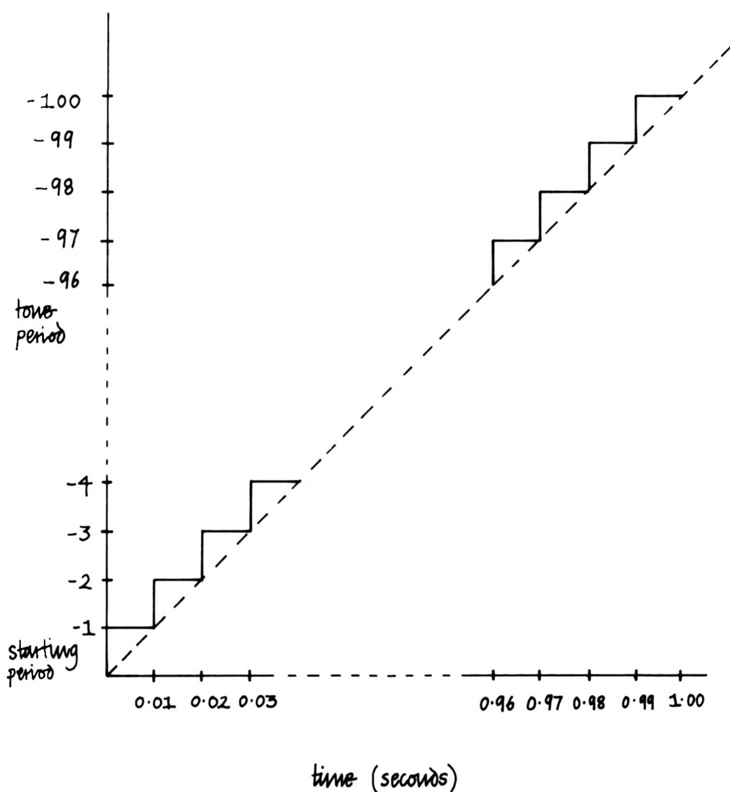


Figure 10.4. Converting the straight-line graph shown in Figure 10.3 to a staircase pattern for the ENT instruction. The graph shows part of the tone envelope defined by the instruction 'ENT 1, 100, −1, 2.' When a tone is sounded using this envelope, its tone period (specified by the SOUND command) will be reduced by 1 unit every 1/100th second. The envelope assumes that the tone duration is one second, so there are 100 steps

have to double the step length to 0.02 seconds and the instruction would be:

```
100 ENT 1,50,-1,2
```

There are limits to the values you can use with ENT. There are only 15 envelopes you can define (1–15); envelope number zero is the default envelope and cannot be changed. The number of

steps must be in the range 0 to 239, the step size must not be less than —128 nor greater than 127. The duration of each step must be at least 0.01 seconds and not more than 255 (0 is treated as 256).

When defining tone envelopes, the first number (the envelope number) may be negative. If it is, and if the tone envelope finishes before the note has finished playing (which time is defined by the duration of the note in the SOUND command), then the tone envelope will be repeated until the note ends. To hear how this affects notes, try altering the first values of the ENT commands above to —1, and increasing the duration of the note (the third argument of SOUND). If the envelope number is positive, then when the tone envelope has been 'used up' when a note is playing, the default envelope will be used. As another example of the difference between repeating and non-repeating tone envelopes, try the following:

```
10 ENT -1,100,5,1
20 PRINT "Negative envelope"
30 SOUND 1,284,400,7,0,1
40 INPUT "Press ENTER to continue"
50 INPUT a$
60 ENT 1,100,5,1
70 PRINT "Positive envelope"
80 SOUND 1,284,400,7,0,1
```

Designing a tone envelope

You can specify up to five sets of tone-changing steps for each tone envelope number. The best way to design a tone envelope is to sketch its outline on graph paper, then draw in a staircase that fits as closely as possible. This can then be divided into up to five sections, and the more sections you have the greater the amount of detail you can get into a sound. Then each section has to be defined as three values, the number of steps in length, the period change for each step, and step duration, as described above. For example, simple vibrato can be produced by raising and then lowering the initial tone period by, say, ten units, so it always returns to its starting frequency. This is shown in Figure 10.5, and can be produced with the tone envelope defined by:

```
ENT 1,10,-1,1,10,1,1
```

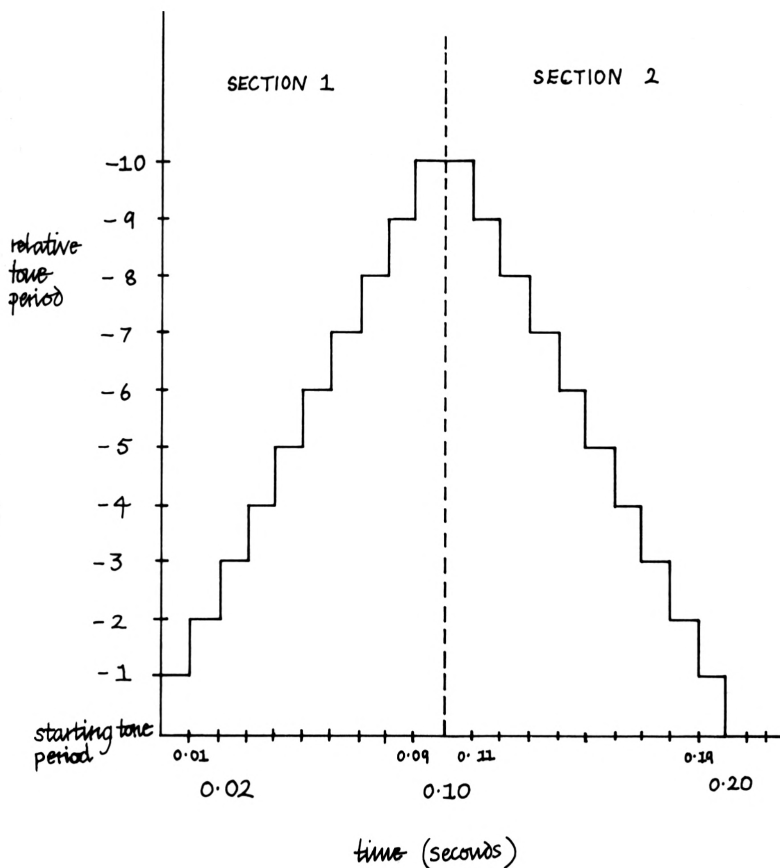


Figure 10.5. Using two sections in a tone envelope. The first section raises the pitch of the tone by lowering its period value by 10 units in 10 steps. The second section lowers the pitch to the starting tone period. The envelope shown is defined by the instruction 'ENT 1,10,–1,1,10,1,1'

If you want to use tone envelopes which repeat, you'll have to exercise particular care, because the effects of addition or subtraction from the starting frequency are cumulative. That is, if an envelope raises the frequency of a tone by a certain amount, and the envelope has a negative number, then the tone will continue to rise as the envelope is repeated across the duration of the note. As an example of this, try the following:

```
10 ENT -1,50,-1,1,40,1,1
20 SOUND 1,200,255
```

You'll hear that as the sound progresses the pitch gradually rises. This is because the tone envelope used makes the pitch rise by 50 steps, then drop by 40 – a net gain of 10 on every repetition. The pitch rises and falls, but the overall effect is an increase in frequency.

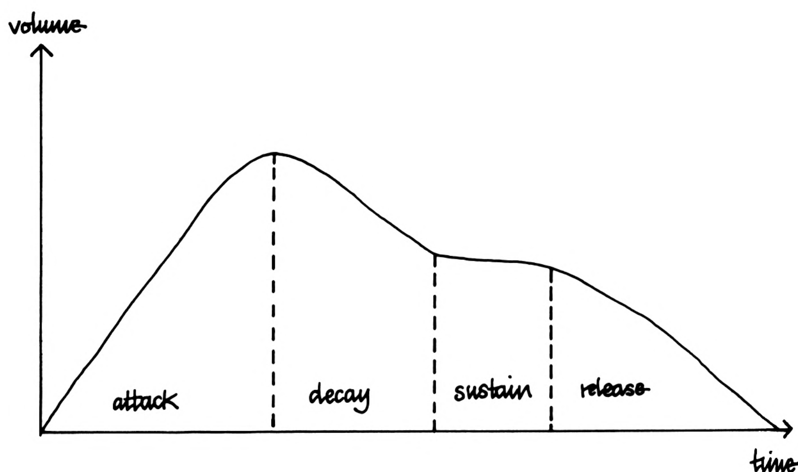


Figure 10.6. Idealised volume envelope of a musical note

Volume envelopes

Volume envelopes are defined in much the same way as tone envelopes and add character to notes. It is the volume envelope which dictates those features described in musical terms as 'attack', 'decay', 'sustain' and 'release'.

The volume envelope allows you to alter the volume or amplitude of a sound as it is being played. 'Attack' is how quickly the volume rises at the beginning of a note. After the note has reached a maximum volume, it dies away a little and this part of its volume envelope is known as 'decay'. The 'sustain' section is usually the bulk of the note and gives the volume at which the note is held before being 'released', and this last phase defines how the note dies away (see Figure 10.6).

These distinctions are somewhat arbitrary, and any given note's overall envelope may be hard to define in these terms. None the less, they are useful when we want to be able to define different types of tone on the computer. Figure 10.7 shows sketches of the volume envelopes of some musical instruments. The sketches are only intended to give a rough indication of the volume envelope for an instrument.

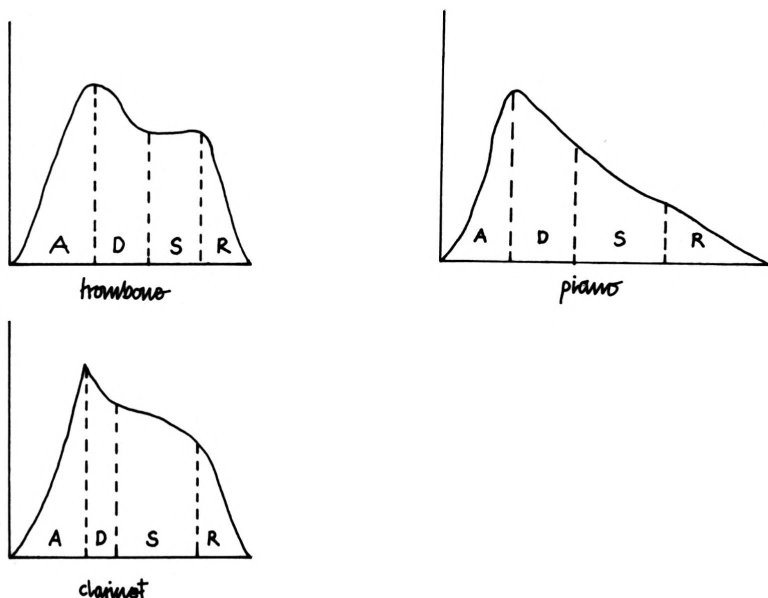


Figure 10.7. Typical volume envelopes of three instruments

Experimenting with volume envelopes

The best way to become familiar with the effects you can produce with volume envelopes is to experiment. Entering and altering ENV parameters can be tedious, and the listing which follows makes the task simpler.

The four phases of attack, decay, sustain and release each have three parameters which describe the change in volume over time of each section. These are the number of steps in the phase, the size of the volume increment for each step and the step length in

1/50th seconds. In the listing, the letters a, d, s and r are used as prefixes, so 'adur' holds the duration for each step of the attack phase, 'rinc' is the increase in volume for each step of the release phase, and so on (to decrease the volume use a negative increment).

```
10 'ENV generator
20 note = 300
30 ' Define volume as high & low
40 hivol = 15:lovol = 8
50 vol = hivol - lovol
60 'Define number of steps for each section
70 asteps = 3
80 dsteps = 3
90 ssteps = 4
100 rsteps = 4
110 'Calculate volume increases for each phase
120 ainc = ROUND(hivol/asteps)
130 dinc = -ROUND(vol/dsteps)
140 sinc = 0
150 rinc = -ROUND(lovol/rsteps)
160 'Define ADSR durations
170 adur = 2
180 ddur = 1
190 sdur = 10
200 rdur = 1
210 MODE 2
220 LOCATE 1,3:PRINT"ATTACK:"
230 LOCATE 1,4:PRINT"DECAY:"
240 LOCATE 1,5:PRINT"SUSTAIN:"
250 LOCATE 1,6:PRINT"RELEASE:"
260 LOCATE 10,1:PRINT"Steps","Change","Duration","
Total change","Length"
270 LOCATE 12,3:PRINT asteps,ainc,adur,asteps*ainc
,asteps*adur
280 LOCATE 12,4:PRINT dsteps,dinc,ddur,dsteps*dinc
,dsteps*ddur
290 LOCATE 12,5:PRINT ssteps,sinc,sdur,ssteps*sinc
,ssteps*sdur
300 LOCATE 12,6:PRINT rsteps,rinc,rdur,rsteps*rinc
,rsteps*rdur
310 '
320 'Define ENV 1
330 ENV 1,asteps,ainc,adur,dsteps,dinc,ddur,ssteps
,sinc,sdur,rsteps,rinc,rdur
340 LOCATE 1,8:PRINT"Envelope is: channel,";asteps
;",";ainc;",";adur;",";dsteps;",";dinc;",";ddur;",
";ssteps;",";sinc;",";sdur;",";rsteps;",";rinc;","
;rdur
```

```
350 'Calculate note length
360 length = asteps*adur+dsteps*ddur+ssteps*sdur+r
steps*rdur
370 LOCATE 1,10:PRINT"Total length is";length
380 fv = asteps*ainc+dsteps*dinc+steps*sinc+rsteps
*rinc
390 PRINT"Final volume is";fv
400 IF fv < 0 THEN length = length - fv
410 'Play note
420 SOUND 1,note,length,0,1
430 END
```

Line 20 defines the note value, its pitch or period. Line 40 assigns values to 'hivol' and 'lovol'. 'Hivol' is the highest volume, the peak reached at the end of the attack phase; 'lovol' is the note volume at the end of the decay phase. 'Vol' is defined as the difference between these and is used in the calculation of the volume increment for each step of the decay phase.

Lines 70 to 100 assign the number of steps to each phase. Lines 120 to 150 work out the volume increase for each step in each section. This is calculated as the number of steps in the total volume change of the section divided by its number of steps. The duration of each step in each section is assigned to the relevant variables in lines 170 to 200.

The table of values shown just before the note is played is drawn up in lines 210 to 300. Line 330 defines the volume envelope, line 340 displays its format. The total length of the note is calculated as the sum of the number of steps in each section multiplied by the duration of each step. The final volume ('fv') is calculated in line 380. Note that due to the fact that we have to use integers with ENV, this may not be zero, as it ought to be. This may require you to make minor adjustments to get a perfect envelope.

Of course, there's no reason why you should stick to the ADSR approach, but practice with it may help you to master ENV more quickly.

Noise period

This is the last argument of the SOUND command and is optional. A positive value indicates that noise should be added to the tone produced by the other arguments. The value supplied (in the range 1–31) specifies the noise period. The effect of increasing the

number is difficult to describe, but the 'hiss' produced varies in quality. Try the following:

```
10 FOR n = 0 TO 31
20 PRINT n
30 SOUND 1,200,100,7,,,n
40 NEXT
```

Note how the first few values of n are displayed long before they're produced as sound. This is because these values are sent to A's sound queue, and it's only when the queue is full and the program is forced to wait for a free space that it seems to behave properly. To correlate the printed values of n with those being used in the SOUND command, add:

```
35 FOR i = 1 to 1000:NEXT
```

Note the use of commas in the SOUND command, which sets intervening arguments to zero.

Channels, rendezvous and holds

The channel argument (the very first value passed to SOUND) isn't just for specifying whether channel A, B or C should produce the sound. It may also be used to ensure that notes begin at the same time, that a note should be 'held' and so on. The technique used for this detail is bit-mapping (see Chapter 6). The 'meanings' of the bit positions are as follows:

Bit	Decimal value	Meaning
0	1	Use channel A
1	2	Use channel B
2	4	Use channel C
3	8	Rendezvous with A
4	16	Rendezvous with B
5	32	Rendezvous with C
6	64	Hold
7	128	Flush

To use this table, first decide how the sound is to be used, then add up the relevant decimal values and use the result as the first argument in the SOUND command. For example, suppose that we

have two notes that we wish to play starting at the same time ('rendezvous'). One is to go to channel A, the other to channel B. The channel value for one note will be 17, which we get by adding 1 and 16; 1 means 'use channel A' and 16 is 'rendezvous with Channel B'. The other note will use a value of 10, which we calculate as 2 + 8, 2 for channel B plus 8 for rendezvous with A.

Bit 6, decimal value 64, is used to 'hold' the sound. This means that the sound will be placed in the queue as normal, but when it reaches the head of the queue, it will stay there until a RELEASE command (or a sound with bit 7 set) is issued for that channel. While it's sitting waiting other sounds may be added to the queue, which can create a bottleneck, so using bit 6 requires some careful thought.

If bit 7 of the channel argument is set (decimal value 128), it will move the sound to the head of the channel's sound queue, force the sound to be played at once (so other sounds are ended) and leave the channel's queue empty – pretty drastic!

Sound queue

Each sound channel has a queue of sounds waiting to be played. The SOUND command places a tone's parameters in the queue for the channel specified. There can be up to four sounds in a queue and the Amstrad has some special sound queue handling commands. SQ takes 1, 2, or 4 as its argument (for the channels A, B or C) and returns a number which reflects the state of the channel. Using a number with SQ which is not 1, 2 or 4 gives an 'Improper argument' error message.

SQ returns an integer whose bit positions map out the state of the queue tested. The following table shows how the information is mapped in the byte.

Bits		Meaning if returned set
0, 1, 2	–	number of free spaces in queue (returns 0 to 4)
3	–	rendezvous with A at head
4	–	rendezvous with B at head
5	–	rendezvous with C at head
6	–	Hold at head
7	–	Channel is active

To use the information in the byte returned by SQ, you have to perform AND operations (see Chapter 6) to extract the relevant details. For example, to find out how many free spaces there are in the channel's queue, first you have to get the byte, say for channel A, then AND the result with 7 (bits 0, 1 and 2 set). The result will give the number of free spaces in A's queue:

```
100 channelA.stat = SQ (1)
110 free.entries = channelA.stat AND 7
```

To check if the sound at the head of the queue is set to rendezvous with another channel, you would AND the byte returned by SQ with 56. The number 56 is the decimal value given by the binary pattern 00111000 – i.e. bits 3, 4 and 5 set. If the result is not zero, a rendezvous is set, and to find out to which channel the synchronisation is geared, you would AND the result with 32 (channel C), 16 (channel B) or 8 (channel A). If the result is the number you used in the AND operation, then the rendezvous is set for the channel given. For example:

```
100 channel.Astat = SQ(1)
110 rvous = channel.Astat AND 56
120 rvousA = rvous AND 8
130 rvousB = rvous AND 16
140 rvousC = rvous AND 32
150 IF rvousA = 8 THEN ....
160 IF rvousB = 16 THEN ....
170 IF rvousC = 32 THEN ....
180 'etc
```

To test if the sound at the head of the queue is set to hold, you would AND the result of SQ with 64. If the result is 64, then the sound at the head is holding, and will need a RELEASE command to have it played.

ANDing the result of SQ with 128 will tell you if the channel is currently active, i.e. if a sound is being played through it.

Note that the use of SQ disables any active ON SQ ... GOSUB instruction for that channel (see below).

RELEASE

This command is used to release any holds set for notes in the channel number given. You can use any integer between 1 and 7

with the command, which allows you selectively to release single channels or any combination of channels. RELEASE 1 releases holds in A's sound queue, RELEASE 7 releases holds in all channel queues, RELEASE 6 releases holds in channel B and C.

The main purpose of setting sounds to hold, then issuing RELEASE, is to 'prime' channels (i.e. fill their sound queues with notes, but not play them) and then release them. This gives great control over the sequencing and synchronisation of notes, and therefore musical fragments such as chords, harmonies, tunes, and so on.

ON SQ...GOSUB

ON SQ is used rather like the 'EVERY' interrupt command (see Chapter 12). It detects when a free space in the indicated channel's sound queue becomes available. As with other commands, you have to use the numbers 1, 2 or 4 for the channels. When a free space is found, control passes to the line number following the GOSUB, allowing you to place new entries in the relevant channel's queue. Note that both SQ and SOUND disable ON SQ.

Ring modulation

This occurs when two specifically related frequencies are played together. To calculate the values for the ring modulation of two frequencies, first select your pitches in Hertz, then calculate their sum, and the absolute value of their difference (this means subtracting one from the other and ignoring the sign of the result). Then look up the nearest period numbers in the period/frequency table in the Amstrad manual and pass the values to SOUND command for different channels. The result, at least with higher frequencies, is a 'bell' or 'chiming' sound.

The following listing shows how you can write routines to do the calculations for you.

```
10 REM Ring modulation
20 DEF FN freq(note) = 440 * (2 ^ (octave + (10 -
note) / 12))
30 DEF FN period(freq) = ROUND(125000 / freq)
40 FOR count = 1 TO 10
```

```
50 octave = INT(RND(1) * 2) -1
60 note1 = INT(RND(1) * 12) + 1
70 note2 = INT(RND(1) * 12) + 1
80 IF note1 = note2 THEN 60
90 freq1 = FNfreq(note1)
100 freq2 = FNfreq(note2)
110 sumfreq = freq1 + freq2
120 diffreq = ABS(freq1 - freq2)
130 period1 = FNperiod(sumfreq)
140 period2 = FNperiod(diffreq)
150 IF period1 > 3822 OR period2 > 3822 THEN 60
160 IF period1 < 12 OR period2 < 12 THEN 60
170 SOUND 10,period1,100,7
180 SOUND 17,period2,100,7
190 NEXT
```

Line 20 sets up a function which calculates the frequency of a note from the arguments supplied for the note's number and octave. FN period (Line 30) returns the Amstrad's period value for a given frequency.

The main loop begins at line 40 and runs to 190. Lines 50 to 70 assign random numbers to the three variables 'octave', 'note1' and 'note2'. The sum of the two frequencies and the absolute value of the difference between them are calculated in lines 110 and 120. These values are then passed to FN period to obtain the Amstrad's codes for the notes to be played. Lines 150 and 160 check that the values produced are 'legal', i.e. won't generate an error message. If the values are not acceptable, program flow is diverted back to line 60 to generate two other notes.

The two notes produced by lines 170 and 180 are arranged to rendezvous by their first parameters, and line 190 repeats the loop.

11
The Cassette System

The Amstrad's built-in cassette recorder must be the envy of many other home micro owners. It's reliable and very convenient, and you can make good use of it in a variety of ways from Basic.

Loading programs

One of the first things you'll want to do with your new machine is run a commercial program. This has to be loaded into memory from cassette. The operation is very simple: type RUN " ", then press ‹ENTER› and the Amstrad will prompt you to press the PLAY key on the recorder and to press another key. Once you do this the Amstrad will turn on the cassette motor, read the program from tape into its memory, and then run it. A quick way to load and run a program is to press CTRL and the small ENTER key on the numeric keypad, which will automatically produce RUN", then all you have to do is press PLAY and a key on the keyboard, and the next program on tape will be loaded and run. LOAD" " or LOAD" will load the next program found on cassette into memory, but won't run the program. RUN and LOAD would normally be followed by the name of a program enclosed in quotes.

Saving programs

To save a program you've entered to tape, make sure you won't overwrite another program, place a cassette in the tape recorder, then type SAVE "programname". Your machine will prompt you to press the PLAY and RECORD keys of the recorder, and as soon as you have done this (and have pressed another key), will record the program on the tape, along with its name. The program name can be up to sixteen characters long, and it's a good idea to use the date as part of the name so you can easily find out when a program was saved, and hence which is the latest version of one

of your programs. If you use more than sixteen characters, the seventeenth onwards will be ignored. All filenames are converted to upper case when they're saved.

If you use the command SAVE " ", i.e. you don't give the program a name, it will be recorded with the name 'Unnamed file'.

SPEED WRITE

If you're saving very long programs, but don't want to have to wait ages to save or load the program, you can make the Amstrad record the information on to tape at twice the normal speed. The command for this is SPEED WRITE 1, and to reset the recording speed you use SPEED WRITE 0. One nice thing about this command is that you don't have to remember whether you used the fast or slow speed to record a program – the Amstrad will work that out for itself (unlike other home computers). Note that the tape recorder doesn't go faster or slower – SPEED WRITE only affects how quickly information is sent to the tape recorder.

Saving options

There are a number of options you can use when saving programs. For example, if you prefix the name of the program with an exclamation mark, as in SAVE "!CIRCLES", then the messages which would normally be displayed are suppressed, e.g. you won't be prompted to press play and another key. Note that the ! is removed from the program name when it's recorded. Similarly, if in a program you want to use the tape system to load information, but would rather use your own prompts than those supplied by the Amstrad, just prefix the file or program name with ! and write your own messages.

Protecting programs: the ,P option

Sometimes you'll want to 'protect' your programs from prying eyes, and the Amstrad allows you to save a program so that it has to be loaded into memory using RUN or CHAIN, and cannot just be loaded, then listed. To protect a Basic program, just add ',P' to the SAVE command, as in SAVE "game",P. However, you must be

sure that when you save a program with the ,P option, you really have perfected it, because no one (not even you) will be able to list it if there are any problems with it. If you try to load a protected program, it will appear to load without problems, but when you try to list it, you'll find it has vanished. Similarly, if you try to break into a protected program using ESC, ESC, the program will disappear. You can only RUN or CHAIN (see below) a protected program. If you protect a program which contains an error of syntax, or which would generate an error message, then that program is a write-off – you simply can't access it because when it stops due to the error, the Amstrad's protection system will erase the program from memory.

Files

Strictly speaking, all the chunks of information you record on tape, be they programs, string arrays, screen pictures or machine code routines, should be called files.

Chapter 4 included a listing of a simple database program which makes use of the tape recorder to store data from a string array. Files created in this way are often called 'sequential text files' because due to the medium, information has to be stored in a linear fashion and the data is encoded in ASCII format. Sequential files are slow and clumsy for information storage and retrieval. You can't re-record part of a file and the whole file has to be processed before you can get to information at the end of the file. Random access files are much more efficient, but are only found on fast tape or disk systems.

ASCII format: the ,A option

Basic programs are stored in a special way in the Amstrad. The computer 'tokenises' all the Basic keywords, i.e. uses numbers to represent reserved words like PRINT, GOTO and so on. Programs are stored from address 438 and the following listing displays how it itself is stored in RAM.

```
10 REM line 10
20 'Basic program revealer
30 MODE 2
40 address = 438
```

```
45 PRINT "Address  Line"
46 PRINT "      Bytes"
50 bytes = PEEK(address)
60 line.no = PEEK(address) + 2
70 PRINT address;bytes;line.no;
80 FOR count = bytes to bytes - 2
90 conts = PEEK(address) + count
100 IF conts > 31 AND conts < 128 THEN PRINT
CHR$(conts); ELSE PRINT conts;
110 NEXT
120 PRINT
130 address = address + bytes
140 GOTO 50
```

You'll see that line 10 is 14 bytes long (including spaces), and that the code for 'REM' is 197. Notice the difference between REM and its abbreviation the apostrophe (line 20). PRINT is given the value 191 (lines 45 and 46), and so on. Note that the last character of a variable has 128 added to its ASCII code ('s' becomes 143). (You could use this knowledge to write a program that would change itself with POKE.) The tokenising method is used because of the saving in space; the number 197 occupies a single byte, REM takes three. However, because programs are also stored on tape in this tokenised format, you would not be able to read a program from cassette into a program like a wordprocessor.

The ,A option saves programs to tape in ASCII format. Programs saved in this way take longer to record because the ASCII format takes more space, but they will load more reliably and can be treated as normal text files.

Recording (saving) data

To open a file on tape you use the word 'OPENOUT', followed by a filename in quotation marks. Then, to send data to the cassette for recording, you use yet another variant on the PRINT statement: PRINT #9. You'll remember that stream or channel number 9 is reserved for the cassette system. PRINT #9 must be followed by a comma, but from there on the format is the same as that for printing on the screen. Once you've finished sending data to the cassette you must close the file with 'CLOSEOUT', because the Amstrad needs to record a special 'end of file' code so that it can keep track of information store on tape.

Because you can use a variable with PRINT #, you can test out a

file-handling program on-screen (perhaps in a window), before committing data to tape. This is useful because it means that you don't have to wait while data is written to tape, and allows you to debug your program easily and quickly.

The way to do this is to set up a variable for the PRINT channel at the start of the program under development, e.g. device = 0. Then use PRINT #device in any tape writing routine, and when you're satisfied that the routines are working correctly, just alter the value of 'device' to 9, so data will be sent to the tape, not the screen.

When saving data to tape, don't worry if nothing seems to happen when your program should have sent a small amount of data to the cassette. The Amstrad uses a 'buffer' (a spare section of memory) to write information to before sending it to tape. Only when the buffer is full, or a CLOSEOUT command is issued, will the information be sent to the cassette unit.

WRITE

This command is unusual among Basic dialects. It is very like PRINT in that it can be followed by a stream expression, as in WRITE #9, or WRITE #0, the latter being the screen default window (i.e. #0 is assumed if a stream expression is omitted). It is then followed by a list of items to be 'printed' (e.g. 'WRITE #0,"this",value,that$'). In the output from WRITE, numbers are separated by commas, strings are enclosed in double quotes. For example: 'WRITE #9, value1, astring$, value2' sends the three items to the cassette, each item separated from its neighbour by a comma. This has the effect of sending a CHR$(13) after each and is the equivalent of the three separate PRINT statements:

```
PRINT #9,value1
PRINT #9,astring$
PRINT #9,value2
```

Note that 'PRINT #9, value1, value2, value3' would separate the items with TAB zones. Using WRITE #9 value1, astring$, value2 allows you to use a single INPUT #9 statement to read the data back in, as in: INPUT #9, number1, astring$, number3

Alternatively, you could use three separate INPUT #9 commands:

```
INPUT #9,number1
INPUT #9,astring$
INPUT #9,number2
```

The value of WRITE is that you can send data lists to the tape stream with simpler statements than if you use PRINT.

Retrieving data

Just as a tape file has to be opened for output before you can send data to it, so it must be opened for input before you can read data from it. The command for this is 'OPENIN', usually followed by a file name in quotes. Attempt to OPENIN a program will produce the 'File type error' message, unless it has been saved with the ,A option (see above). And of course you can't run an ASCII file, unless it's a Basic program.

It's a good idea to use the first few bytes of a data file to describe itself. The database program in Chapter 4 sends the number of row entries in the array to be saved before sending the data in the array. When the file is opened, the number of row entries is read in as the first piece of information and is used as the upper limit of the loop counter which reads in each row of the array. In the program the number of columns is fixed, but there's no reason why you shouldn't alter the program so that it also records the number of columns in the array.

EOF

This is short for 'End Of File' and is used when you don't know how long a file is. If you try to INPUT #9 when all the data in a file has been read, you'll get an 'EOF met' error message and EOF allows you to test for the end of a file to avoid this. The function is often used as in 'IF EOF THEN GOTO XXXX'. For example, the following fragment will read all the data from a file and print it on the screen. When the file has been exhausted (i.e. EOF returns a value 'true'), the file is closed and the program ends.

```
10 OPENIN""
20 IF EOF THEN 60
30 INPUT #9,a$
40 PRINT a$
```

```
50 GOTO 20
60 CLOSEIN
```

LINE INPUT

If you save string data which contains commas (such as an address), the commas will be treated as separators by INPUT #9. A string such as 'Mary Jones, 191 The Avenue, Durham' would be treated as three separate items. To see this, run the following program:

```
10 CLS:OPENOUT"TEST"
20 PRINT #9,"abc,def,ghi"
30 CLOSEOUT
```

Now rewind the tape, NEW the progran and run this data retrieval program:

```
10 CLS:OPENIN "TEST"
20 IF EOF THEN 70
30 INPUT #9,a$
40 count = count + 1
50 PRINT count,a$
60 GOTO 20
70 CLOSEIN
```

This shows how INPUT treats commas and how the data has been split up. However, if you alter line 30 to 'LINE INPUT #9,a$', rewind the tape and run the program again, you'll see that LINE INPUT avoids this potential problem.

Saving blocks of memory

The SAVE command has a number of options which allow you to save more than just a program or data to tape. For example, you may want to save a picture you have created on the screen, and rather than save the program which created the image, you can save the picture itself. The format for this is:

```
SAVE "picture",B,&C000,16384,&C000
```

The B means that a block of memory is to be saved, &C000 is the starting address of the video memory in RAM, 16384 refers to the length of the block of memory to be saved (all screen modes use 16K of RAM) and the last parameter indicates the starting address for reloading the information. You can get some interesting results using this process, for example, for loading patterns created in one MODE into another. Here's a program which draws an interference pattern and saves it to tape:

```
10 MODE 2
20 y = 400
30 FOR x = 639 TO 0 STEP -3
40 ORIGIN 0,0
50 DRAWR x,y
60 NEXT
70 FOR x = 0 TO 639 STEP 3
80 MOVE 639,0
90 DRAW x,y
100 NEXT
110 SPEEDWRITE 1
120 SAVE"!interference", B, &C000, 16384, &C000
```

When the program is running, press RECORD and PLAY on the tape recorder, and when the image has been drawn it will be saved to tape as what is called a binary file. Now type NEW, rewind the tape and type in the following program:

```
10 MODE 0
20 LOAD"!interference"
```

When you run this you'll see that the screen isn't mapped out simply from start to end. As the data is read in it appears on the screen in lines. The first is pixel line 0, followed by 8, 16, and so on. When the top line of each character row has been read in, the next pixel lines (1, 9, 17, etc.) are loaded until the display is filled. Notice that the screen image alters as the program is loading data from tape – when it's reading data, no colours flash, but between reading blocks and once the file has been loaded, many of the colours flash.

Understanding how the screen is mapped out allows you to create new effects from old patterns, and it's even possible to arrange to have the top and bottom halves of a screen display interchanged, mirror imaged vertically, and so on. All you need to know is that the screen start address is (usually) &C000 (49152 decimal), and that each of the 200 lines occupies 80 bytes. Take

into account the way the screen is mapped and it should be fairly easy to PEEK each byte and PRINT the contents to tape.

Of course, any information on the display, including cassette prompts, will also be recorded, so don't forget to use the ! filename prefix with SAVE and LOAD.

Saving characters

Having redesigned some or all of the Amstrad's characters, you may want to use them in other programs, and it would be tedious to have to enter all the commands for each program. However, because character definitions are stored in RAM, you can save the character set, or just part of it, to tape, and load it into other programs.

The character set is stored in RAM from address &A500 (42240 decimal). Each character is defined by 8 bytes, so the last byte of the first symbol – CHR$(32), space – is stored in address 42247. There are 224 printing characters (32 to 255), so the set ends at 42247 + (223 * 8), i.e. 44031, and is therefore 1792 bytes long. To save the entire set you would use 'SAVE "CHAR.SET",B,42240,1792,42240'.

You can save any part of the character set. If you redefine ASCII characters from 128 to 170, you must first calculate the start address (42240 + (128 – 32) * 8), then the length – the number of bytes to be saved. As 43 characters (128 to 170) are involved, the length of block of RAM to be saved is 344 bytes. The SAVE command will therefore be:

```
SAVE "chars",B,43008,344,43008
```

Saving machine code

Saving a machine code routine (such as the text screen PEEK program in Chapter 7) can save you time because you don't then need to type in (and possibly make errors in) the DATA statements needed to construct the routine at the beginning of each program that uses it. Instead, you could write a short routine like this:

```
10 REM Create and Save machine code
20 DATA 205,96,187,50,23,171,201
30 MEMORY 43798
```

```
40 FOR count = 1 TO 7
50 READ value
60 POKE 43799 + count,value
70 NEXT
80 SAVE "PEEK.TXTSCRN",B,43799,8,43799
```

Cataloguing files

Although you should write down the name of each and every program you save to tape, there will be times when you don't and that's when CAT comes in handy. When you type CAT, the Amstrad will respond with the same prompt as if you'd typed LOAD" or RUN". Once you've pressed PLAY and a key, the computer will read information about each file on the tape, and display the name of each file followed by a symbol according to the file type. The symbols are:

```
$          - Normal Basic program
%          - Protected Basic program
*          - ASCII file
&          - Binary file
```

(A binary file is a block of memory such as a machine code routine, character set or screen image.)

If you want to stop the process you may have to press ESCape a number of times, because the Amstrad will not allow you to interrupt it while reading from or writing to tape – short of your pressing STOP or turning the machine off.

If you turn up the volume control on the right-hand side of the machine while the Amstrad's cataloguing the cassette, you'll hear how information is recorded. Each file is recorded in blocks, not as one long unit. For each file there's what's called a 'header' – this is a tone which tells the Amstrad that a file is about to appear and includes information such as the file name and its type. The header is followed by the data which constitute the file itself, in blocks. Each block has its own header, which is how the Amstrad can tell you which block number is being loaded. The block system is used for security – it reduces the scope for errors. The information on the tape sounds like a high-pitched shriek because it is recorded at a very fast rate.

Chaining programs

You can use RUN to load and run a second program from within a program in RAM. However, when you do this you lose the existing program, which is completely overwritten by the new program. Not only is the first program lost, but all its data (variables, arrays, etc.) is erased as well. Sometimes, if you want to use a very large program, or if you want to use a number of separate, but related programs, this can be very inconvenient. The Amstrad offers facilities for 'chaining' or 'merging' programs together so that a second program can use the data from a first, or a program in memory can be extended with one on tape.

CHAIN

CHAIN can be used within a program to load and run another program which has been recorded on tape. Its main value is that you can RUN a second program, without losing the values of variables set up by the first program. This is useful – if a program is too large for memory it can be split into two parts which run consecutively, each part being saved as a separate file. The facility could also be used over a series of games or educational programs without losing track of a cumulative score. CHAIN is used as in:

```
10000 CHAIN "NEXT.PROG"
```

You can also specify a line number from which the new program is to run:

```
CHAIN "PART.TWO",3000
```

This will load the program called PART.TWO, and begin running it from line 3000. If there is no line 3000, an error message will result.

CHAIN MERGE

This is a more complex version of the CHAIN command. It allows you to merge a program on tape with one in memory, with the option of deleting some or all of the program. Any lines in the program in memory with the same line number as lines in the program on tape are lost – they are overwritten by the incoming lines. The syntax for CHAIN MERGE is similar to that for CHAIN, but you can also add ',DELETE' and a range of line numbers. For example:

```
5000 CHAIN MERGE "PART.THREE", 9000, DELETE
3000-6000
```

This will merge the program called PART.THREE with the current program, and begin execution of the resultant program from line 9000. Lines 3000 to 6000 of the program in memory will be deleted and the DELETE command is used just as it is in direct mode. Here are some more examples:

```
1000 CHAIN MERGE "PART.FOUR",,DELETE -1000

2000 CHAIN MERGE "PART.FIVE",6000,DELETE 9000-
```

The first example merges the program called PART.FOUR, deletes all lines up to line number 1000, and runs the resulting program from its first line. The second example merges the program called PART.FIVE, deleting all line numbers from 9000, and runs the program from line 6000.

CHAIN and its variants must be used with care. All user-defined functions set up with DEF FN are forgotten, so the new program must redefine them. Any ON ERROR GOTO condition is turned off and all FOR...NEXT, WHILE...WEND or GOSUB constructions are abandoned, DATA statements are RESTORED and any open cassette data files are abandoned. DEFSTR, DEFINT and DEFREAL instructions are also forgotten. As with LOAD and SAVE, if you use ! as the first character of the filename, cassette prompt messages will be suppressed, so you will have to use your own to avoid confusion.

ROM calls

There are number of ROM routines which you can CALL from your programs that can help file handling. For example, 'CALL &BC65' resets the cassette manager system by closing any open files, turning on prompt messages and setting SPEED WRITE to zero – the default conditions. CALL &BC6B enables or disables the prompt messages. If the messages are to be disabled, the A register must contain zero; a non-zero value enables the messages. You therefore have to use a very simple, six-byte machine code routine which loads the A register with the relevant value, then calls the ROM routine. Here is one way of doing just that:

```
10 MEMORY 43879:address = 43879
20 POKE 43880,62: REM LD A,n
30 POKE 43881,255:REM data
40 POKE 43882,205:REM CALL
50 POKE 43883,107:REM low byte of &BC6B
60 POKE 43884,188:REM high byte of &BC6B
70 POKE 43885,201:REM RET - 'return'
```

Having loaded the machine code you can disable the cassette handling prompts with 'POKE 43881,0:CALL43880'. To enable the messages use 'POKE 43881,255:CALL 43880'.

To turn the cassette motor on, use 'CALL &BC6E' and to turn it off, use 'CALL &BC71'.

Cassette error messages

These can be divided into two sections: read errors and write errors.

Read errors are not very common, as the Amstrad's tape system seems pretty reliable. They indicate a fault when the system is trying to interpret data loaded from tape, such as occurs if you press STOP while a file is being read, or if a tape contains corrupt data. This may occasionally happen if a commercial game supplier uses an unreliable tape-duplication system, if the tape has been stored near a strong magnetic field, such as found

around television sets, or if the tape has flaws or has been subjected to a static charge.

There are three types of read error: a, b and c. 'Read error b' is the most likely to be recoverable – it simply means that data was read incorrectly, so rewind the tape and try again. 'Read error a' or 'Read error c' may indicate more serious problems.

The easiest way to get tape errors is to be careless in your handling of tapes. Use only reasonably high quality cassette tapes, not longer than C90s, but don't be taken in by extravagant claims for 'computer quality' tapes. Treat tapes with care, and don't re-use them too often. If you do want to record over material written to tape, erase the whole tape first with a hi-fi or portable cassette deck.

Don't interrupt the Amstrad when it's writing to tape, either by pressing ESC twice, by using any of the cassette recorder keys or by turning off the computer before you've switched all the cassette keys off.

If you do get one of the read error messages, rewind the tape and try again. Read errors may arise if you're trying to load a program or data which was recorded using SPEED WRITE 1 on another machine. If you're going to give people copies of your programs, use SPEED WRITE 0. The faster speed should prove fairly reliable, providing you always save and load on one machine only.

The 'Rewind tape' prompt means that for some reason a data block has been encountered out of sequence. It is most often seen following a read error in a previous block.

There is only one write error message, and it is uncommon because it can only occur when the cassette system fails to send information to tape fast enough. This can only happen if you've been tinkering with the routine at &BC68.

12
Interrupts

The Amstrad's version of Basic has a set of commands found on no other home micro. They allow you to have subroutines executed at specified intervals and provide a great deal of flexibility for the programmer. For example, it is very easy to arrange to have a 'real-time clock' displayed on the screen, to have music play while a program is running or to flash up reminders at regular intervals. The ability to do all these things and more relies on the concept of 'interrupts' – normally the province of the assembly language programmer. Some of the Amstrad's interrupt system is used specifically for sound generation, and that has been dealt with in Chapter 10.

Timers

There are five timers in the Amstrad, and you have met one of them as TIME, but that has nothing to do with the sort of interrupts we will discuss here. The timers you can make use of are numbered from zero to three and they count upwards from zero to 255, in steps of 1/50th of a second. This means that you cannot have a subroutine processed more than 50 times a second, or after more than about every five seconds.

An interrupt is a 'request' to the system from some timer for a subroutine to be processed. When an interrupt request is made, the system will suspend whatever it is doing to 'service' the request. In Basic this means that a program may be interrupted in the middle of a FOR...NEXT loop, then a subroutine will be executed because of an interrupt, then the loop will be continued from where it left off. The process is rather like GOSUB...RETURN; the computer 'remembers' where it was when it was interrupted, and returns to that point once the subroutine has been executed. Of course, this is not without problems – what happens if an interrupt's subroutine takes longer to process than the interval at which it is processed? What happens if an interrupt subroutine

alters variables being used by some part of the main program which has been suspended while the interrupt is serviced?

EVERY

The most useful interrupt handling command is EVERY. It allows you to define an interval, a timer to use and the starting line number of the subroutine you want executed. EVERY has to be followed by two numbers (separated by a comma), a GOSUB and a line number, as in 'EVERY 50,1 GOSUB 1000'. This means, 'after every 50 counts on timer number 1, start to process the subroutine which begins at line 1000'. The first number is the interval: this specifies how often the interrupt is to be generated. As the timers count at 50 times a second, in this example the subroutine will be processed every second. The second number specifies the timer to be used, in this case timer number one. Interrupts generated by EVERY can only be followed by GOSUB, you can't use GOTO, and of course the subroutine must end with RETURN, so that the program can jump back to the point in the main routine at which it was interrupted.

Here's a simple example to show how the process works:

```
10 MODE 1
20 EVERY 50,1 GOSUB 40
30 GOTO 30
35 REM End of main routine
40 n = n + 1
50 LOCATE 10,10
60 PRINT n;"seconds";
70 RETURN
```

Here, the subroutine which is processed every second according to timer number one adds one to the variable n, and displays the variable's current value. This gives a simple second timer. Later, we'll show how to use this sort of technique to code a real-time clock into your programs. Note line 30, which effectively prevents the program from ending, but the interrupt continues and will do so until the program is stopped.

Here's another fairly simple example which shows how the timing of interrupts can be altered within a program:

```
10 MODE 1
20 n = 1
30 EVERY n,3 GOSUB 1000
40 GOTO 40
1000 PRINT n
1010 n= n + 1
1020 IF n > 20 THEN n = 1
1030 EVERY n,3 GOSUB 1000
1040 RETURN
```

In this example you can see that the subroutine increases its own interrupt interval by 0.02 seconds every time it is called.

Disabling and enabling interrupts

Our third example brings out some of the problems of using interrupts in Basic. It produces three numbers 'bouncing' around the display. Each number represents the timer being used to generate interrupts and each number has its own 'bounce routine'.

```
10 'Bouncing numbers
20 CLS
30 x0 = 10:y0 = 10
40 x1 = x0:y1 = y0
50 x2 = x0:y2 = y0
60 b0$ = "0":b1$ = "1":b2$ = "2"
70 dx0 = 1:dy0 = 1
80 dx1 = -1:dy1 = 1
90 dx2 = 1:dy2 = -1
100 '
110 EVERY 4,0 GOSUB 170
120 EVERY 3,1 GOSUB 270
130 EVERY 5,2 GOSUB 370
140 GOTO 140
150 '
160 ' Number 0
170 DI:LOCATE x0,y0
180 PRINT " ";
190 x0 = x0 + dx0
200 y0 = y0 + dy0
210 IF x0 > 39 OR x0 < 2 THEN dx0 = -dx0
220 IF y0 > 22 OR y0 < 2 THEN dy0 = -dy0
230 LOCATE x0,y0:PRINT b0$;
240 EI:RETURN
250 '
```

```
260 'Number 1
270 DI:LOCATE x1,y1
280 PRINT" ";
290 x1 = x1 + dx1
300 y1 = y1 + dy1
310 IF x1 > 39 OR x1 < 2 THEN dx1 = -dx1
320 IF y1 > 22 OR y1 < 2 THEN dy1 = -dy1
330 LOCATE x1,y1:PRINT b1$;
340 EI:RETURN
350 '
360 'Number 2
370 DI:LOCATE x2,y2:PRINT " ";
380 x2 =x2 + dx2
390 y2 = y2 + dy2
400 IF x2 > 39 OR x2 < 2 THEN dx2 = -dx2
410 IF y2 > 22 OR y2 < 2 THEN dy2 = -dy2
420 LOCATE x2,y2:PRINT b2$;
430 EI:RETURN
```

The listing shows two important reserved words in the interrupt vocabulary: DI and EI. These stand for Disable Interrupts and Enable Interrupts. DI is used when you have a section of program which you do not want to be interrupted, and is often best placed as the first instruction of the subroutine to be 'protected' from interruption. EI is used once the section has been completed and allows interrupts to proceed as normal. If you remove each DI from the subroutines in the listing and then RUN the program, you'll see why it can be important to 'protect' sections of a program from being interrupted. If none of the interrupt subroutines are protected from each other, unpredictable effects may occur – some of the characters are not erased, others appear in the wrong place, and so on. The reason for this is fairly simple. Consider a program line like:

```
1000 LOCATE row,col:PRINT attrib$(10);
```

What happens if an interrupt request occurs between LOCATE and PRINT and the interrupt subroutine moves the cursor to a different location from that specified by the first LOCATE? Clearly, when control returns from the interrupt routine, any text in the PRINT instruction will appear in the wrong place. Matters would be further confused were the two statements to be interrupted by more than just one cursor-moving routine. This explains why, when you remove DI from the example above, some of the numbers are not erased and characters appear at incorrect locations.

Timer priority

There are a number of other problems which may arise as a result of the interaction of interrupt requests from the four timers, and to understand these it's important to have some idea of the technique the system uses to log such requests and how it decides which ones to service. In effect, there is an interrupt request 'queue', to which all requests are added as they are made. This means that requests to have a subroutine processed won't just be ignored if the system is doing something else, or is too busy to service the request. When time is allocated to assessing the queue, not all requests are given the same priority. Timer number three is the most important, so if interrupt requests arrive simultaneously from timer three and any other timer, then that of timer three will be processed first. This hierarchy also applies to requests in the interrupt queue – the system will deal with these in order of priority, with timer zero being the least important. Therefore it's a good idea to work out which subroutines are the most important before assigning them as interrupt-driven to a particular timer, and whether or not DI should be used on entry to a subroutine.

The priority system can cause other difficulties. Try altering the interval rates in the bouncing numbers routine. If you set the interval rate for timers 1 or 2 to low values (e.g. 'EVERY 1,1 GOSUB ...'), you'll find that interrupt requests from timer zero are ignored. Indeed, if you set timer two to interrupt every 1/50th second ('EVERY 1,2 GOSUB 260') then not even interrupt requests from timer one are serviced. What's happening here is that the interrupt request queue is rapidly filled up with requests from the higher order timer, so any requests placed there by lower order timers are either over-ridden because of the priority system, or aren't added to the queue because there's no room (once the queue is full, any incoming requests are ignored).

This queueing can be demonstrated using the ESCape key. Pressing ESCape produces an imperative interruption; the program is 'suspended' until another key is pressed. If this second key is another ESCape, the program is aborted, and control switched to direct mode. If any other key than ESCape is pressed (apart from SHIFT or CTRL), the program continues from where it left off. However, the timers are independent of this interruption,

so interrupt requests are still being made, even though the program has apparently been stopped.

Enter and run the following program. You'll see that timer one is used to count in one-second intervals, timer two in two-second intervals. Pressing ESC suspends the program, and pressing space a few seconds later shows timer two rushing to catch up, followed by timer one, and the synchronisation is fine. However, if the program is suspended for some time (a few minutes) you'll find that the timers are out of step – timer number two's priority takes over.

```
10 CLS
20 EVERY 50,1 GOSUB 1000
30 EVERY 100,2 GOSUB 2000
40 GOTO 40
1000 count1 = count1 + 1
1010 LOCATE 1,1:PRINT "Timer 1";count1;
1020 RETURN
2000 count2 = count2 + 2
2010 LOCATE 1,2:PRINT "Timer 2";count2
2020 RETURN
```

For another demonstration of this queuing effect, run the bouncing number program and press ESC once. Wait a few seconds, then press any other key. You'll see the highest order timer rushing to catch up – the number 2 zaps about the screen – then the next timer in priority catches up, and so on until things are back to normal.

There is one other consideration when using EVERY. You must be careful to check how long an interrupt-driven routine takes to be processed. As you can imagine, there will be problems if an interrupt routine takes longer to process than the interval specified for its processing. While it is being processed, and assuming that it is 'protected' by DI, other requests may be piling up in the queue, possibly from its own timer!

There are two ways of using the timers to show time – as time elapsed since a certain point, or as 'clock' time. The latter requires the user to enter the time at some point in the program. Here's a 'time-elapsed' routine:

```
10 const = TIME
20 CLS
30 EVERY 50,3 GOSUB 10000
40 GOTO 40
```

```
50 '
9999 'Clock subroutine
10000 counts = TIME - const
10010 seconds = ROUND(counts / 300)
10020 minutes = ROUND(seconds / 60)
10030 hours = ROUND(seconds / 3600)
10040 seconds = seconds MOD 60
10050 minutes = minutes MOD 60
10060 hours = hours MOD 24
10070 second$ = STR$(seconds)
10080 minute$ = STR$(minutes)
10090 hours$ = STR$(hours)
10100 second$ = RIGHT$(second$,2)
10110 minute$ = RIGHT$(second$,2)
10120 hour$ = RIGHT$(hour$,2)
10130 LOCATE 1,1
10140 PRINT hour$;":";minute$;":";second$;
10150 RETURN
```

And here's a digital clock:

```
10 DEF FN strip$(anyvar) = RIGHT$(STR$(anyvar),2)
20 CLS
30 INPUT "Hours=",hrs
40 INPUT"Minutes=",mins
50 INPUT"Seconds=",secs
60 EVERY 50,3 GOSUB 10010
70 CLS
80 GOTO 80
90 '
10000 'Time Update
10010 DI:secs = secs + 1
10020 IF secs > 59 THEN secs = 0:mins = mins + 1
10030 IF mins > 59 THEN mins = 0:hrs = hrs +1
10040 IF hrs > 23 THEN hrs = 0
10050 LOCATE 1,1
10060 PRINT FN strip$(hrs);":";
10070 PRINT FN strip$(mins);":";
10080 PRINT FN strip$(secs);
10090 EI:RETURN
```

AFTER

AFTER is another interrupt command, but it's less useful than EVERY. AFTER simply dictates that some subroutine is to be processed after a given number of counts on a specified timer.

The timers are the same as those used with EVERY, and count up from zero, when the command is encountered, to a maximum of 255, at the rate of 50 counts per second. AFTER is normally a once-only command – when it is encountered the relevant timer is set to zero, any other interrupts relating to that timer are cancelled, and when the interval specified is reached the designated subroutine is processed. This means that AFTER and EVERY are mutually exclusive for any given timer. EVERY cancels any other interrupt command for a timer, just like AFTER. However, because you can reset interrupt assignments at any point in a program, you can design routines which make the most of this word by reassigning values to AFTER in the subroutines themselves. Here's one example which shows how AFTER can be made to behave like EVERY:

```
10 REM AFTER Demo
20 CLS
30 REM Subroutines which reset
40 'themselves
50 AFTER 50,1 GOSUB 1000
60 AFTER 100,2 GOSUB 2000
70 GOTO 70
80 '
1000 PRINT"Subroutine 1"
1010 AFTER 50,1 GOSUB 1000
1020 RETURN
1030 '
2000 PRINT TAB(5);"Subroutine 2"
2010 AFTER 100,2 GOSUB 2000
2020 RETURN
```

ON BREAK GOSUB

As mentioned above, the escape key acts as a high priority interrupt, and you can test for two presses of this key. Two presses of ESC normally mean that a program will stop whatever it's doing and pass control back to the user in direct mode. However, ON BREAK GOSUB allows you to divert control to your own subroutine if a user tries to break into the program. You can use the instruction to prevent anyone stopping your programs and listing or altering them, but it could also be used as an instant 'help' facility, accessible at any point in a program. It's also a valuable program development tool. Here's the basic principle of the 'unbreakable' program:

```
10 ON BREAK GOSUB 10000
15 ON ERROR GOTO 10000
20 REM Rest of Program
30 REM
9000 GOTO 20
10000 RETURN
```

Puzzled? Switch off the machine and re-enter the program, but before running it type TRON to trace the execution of line numbers. Every time the program is interrupted by ESC, control passes to the subroutine at line 10000 which returns control back to where it was interrupted. It's an inescapable loop.

Another method would be to use the word RUN at line 10000, which would cause the program to start from the beginning at any attempt to stop it.

To provide 'help' screens (instant information about what the user should do), all you have to do is to divert control to a 'subsidiary' set of routines when ESC is pressed twice. This could begin with a 'main menu' from which the user could select screens of information. Better still, it could be made 'context-sensitive', i.e. present relevant information, perhaps with the option of accessing the main help menu. To do this you'd have to know where the interrupt had occurred, and therefore would have to keep track of operations in a variable – preferably a string variable for clarity. For example:

```
10 ON BREAK GOSUB 10000
20 REM Main program
30 REM Menu for operations selection
1000 REM End of main section
3000 REM Invoice section
3010 place$ = "invoice"
3020 REM Rest of invoicing routine
3990 RETURN
3999 REM End of invoices
10000 REM HELP
10010 IF place$ = "invoice" THEN GOSUB 1100
10020 IF place$ = "receipt" THEN GOSUB 1200
10030 REM Rest of Help screens
10040 RETURN
11000 MODE 1:LOCATE 1,1:PRINT "Help on Invoicing"
11010 REM Rest of help info
11090 RETURN
```

For program development you can use ON BREAK to jump to a routine which displays the values of variables:

```
10 ON BREAK GOSUB 10000
20 REM Rest of program
9999 END
10000 PRINT "Length of string="|LEN(words$)
10010 PRINT "Count="|count
10020 PRINT "Press space to continue"
10025 akey$ = ""
10030 WHILE akey$ <> CHR$(32)
10040 akey$ = INKEY$
10050 WEND
10060 RETURN
```

Similarly, you can use ON BREAK to get out of awkward situations. For example, if you set the keys to repeat rapidly after a very short period of time with the command SPEED KEY 1,1 (which you might do in a game in order to get a fast keyboard response), you'll find that if you break into the program with two presses of ESC, the keyboard is unusable. The keys repeat so fast that it's impossible to type a command to get things back to normal. To make sure the key delay and repeat values are set to the default values, we divert ESC, ESC to a subroutine which resets the values via a ROM call and which lacks a RETURN – so the program 'falls through' and control is returned to direct mode:

```
10 ON BREAK GOSUB 10000
20 SPEED KEY 1,1
30 REM Rest of program
9999 REM
10000 CALL &BB00
```

You can even make sure that the machine has to be turned off to end a program, i.e. you can disable the CTRL SHIFT ESC sequence which normally reboots the machine (as if it has just been turned on). This is achieved using POKE 48622,201 (disable reboot) while POKE 48622,195 enables reboot. The escape key can be disabled by CALL 47947.

ON ERROR GOTO

ON ERROR is very similar to ON BREAK. It's used to make the program jump to a line number if an error message would be

generated. In the example above, you should also add '15 ON ERROR GOTO 10000', because if you've made a syntax error, or if your program generates an error like division by zero, then the keyboard handling will be reset to normal.

RESUME

ON ERROR GOTO has a 'paired' word, RESUME. This can be followed by the word NEXT or a line number. If you use an ON ERROR GOTO command early in your program, when an error is encountered the program will jump to the line number specified after the GOTO. Here you could have a set of specific error-handling operations and at the end a RESUME NEXT command. This passes control back to the statement after the one which produced the error. Alternatively, you can pass control to any specified line number, as in 'RESUME 5500'.

ERR and ERL

Closely allied to ON ERROR are the keywords ERR and ERL. These are system variables, so you can't use them as variable names. ERR returns the error number (a list of these is given in Appendix 8 of the Amstrad manual). ERL returns the line number in which the error occurred. ERR and ERL are numeric variables, so you can write specific error handling routines to rectify problems, without having your program stop. For example, you can have the error type and its code reported, without your program necessarily ending:

```
10 ON ERROR GOTO 10000
20 REM Rest of program
9999 END
10000 PRINT "ERROR";ERR;"In line";ERL
10010 RESUME NEXT
```

If you were reading from a cassette file and an 'EOF met' (end of file error message) was encountered, the following would help:

```
10000 IF ERR = 24 THEN CLOSE:PRINT "Unexpected
end of file in line";ERL
10010 CLOSE:RESUME NEXT
```

You could even use ON to divert control to a number of error-handling routines, e.g. 'ON ERR GOSUB 1000,2000,2500'.

REMAIN

REMAIN is used to return timer counts, and to disable timer interrupt requests. It's used as in 'dummy = REMAIN (n)'. The number in brackets must be an integer and refers to one of the four timers. REMAIN returns the time left on the given timer, but also disables that timer and resets it to zero. You can use it simply to cancel an interrupt assignment, i.e. turn off interrupt requests from a timer. Or, which is rather more difficult, you could use it to test how far a given timer had got, and then reset the interrupt interval for that timer or pass a new value to it. If the timer is not enabled (i.e. no interrupt assignment has been made to that timer) then REMAIN will return zero.

DUCKWORTH HOME COMPUTING

EXPLORING ADVENTURES ON THE AMSTRAD

by Peter Gerrard *£6.95*

This is a complete look at the fabulous world of Adventure Games for the Amstrad Computer. Starting with an introduction to adventures, and their early history, it takes you gently through the basic programming necessary on the Amstrad before you can start writing your own games.

Inputting information, room mapping, movement, vocabulary – everything required to write an adventure game is explored in detail. There follow a number of adventure scenarios, just to get you started, and finally three complete listings written specially for the Amstrad, which will send you off into wonderful worlds where almost anything can happen.

The three games listed in this book are available on one cassette at £7.95

COMPUTER CHALLENGES FOR THE AMSTRAD

by Richard Hurley & David Virgo *£6.95*

With the aid of ten superb programs, this book demonstrates the use of artificial intelligence on the Amstrad CPC464. The first two chapters introduce you to the principles of artificial intelligence and the more advanced features of Locomotive Basic which are used in this book. The rest of the book is divided into two parts: the first contains puzzles for you to solve; and the second a collection of stimulating games in which you will find the computer a worthy adversary.

The puzzles include Crossword Puzzler, which will provide you with an endless supply of crosswords, and The Cube, which is a graphical representation of Rubik's Cube. The games include Cribbage, which will tax your card-playing skills to the limit, Backgammon, complete with on-screen prompts, and Draughts, which is designed to play the best possible game while keeping the time taken for each move down to well below one minute.

Richard Hurley is Head of Computer Studies at Hurstpierpoint College in Sussex, and has written several books on computing. David Virgo also teaches computing at Hurstpierpoint.

Write in for a catalogue.

DUCKWORTH

The Old Piano Factory, 43 Gloucester Crescent, London NW1 7DY

Tel: 01-485 3484

Duckworth Home Computing

THE AMSTRAD PROGRAMMER'S GUIDE
Bryan Skinner

Whether you've just started programming or simply want to be able to get more from your Amstrad, you'll find this book a mine of useful information and programming ideas.

Basic programming is introduced in the first four chapters, chapter five shows how to design and code a game from scratch, and there are further chapters on machine code, sound, graphics, the cassette system and interrupts. The Amstrad's functions and facilities are clearly explained throughout and each chapter contains example programs. The book contains many useful programming techniques, such as data compression by bit-mapping, the complex use of arrays and ROM calls. Many of the chapters have longer listings which you can develop for use in your own programs.

Bryan Skinner is a computer journalist with a special interest in software. He is software editor of *Personal Computer News* and the author, with Mike Gerrard, of *Mr Chips Comes Home,* also published by Duckworth.

Duckworth
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1984 5

IN UK ONLY £6.95 NET

THE AMSTRAD PROGRAMMER'S GUIDE

BRYAN SKINNER

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.