

THE AMSTRAD CPC-464 ADVANCED USER GUIDE



MARK HARRISON

Σ
SIGMA
PRESS

The Amstrad CPC-464 Advanced User Guide

by Mark Harrison

Σ
SIGMA
PRESS

Copyright © 1984, Mark Harrison

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without the prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

ISBN 1 85058 014 6

Published by:

SIGMA PRESS,
5 Alton Road,
Wilmslow,
Cheshire,
U.K.

Distributors:

UK, Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester,
West Sussex, England.

Australia:
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia.

Acknowledgments

CPC-464 is a registered Trade Mark of Amstrad Consumer Electronics plc. We gratefully acknowledge the permission granted by Amsoft, the computer products division of Amstrad, to endorse this book as "Approved for Users of the CPC-464 by Amsoft".

Figure 1.2 is reproduced by kind permission of Spectron Artists Ltd.

Printed and bound in Great Britain by
J. W. Arrowsmith Ltd., Bristol

PREFACE

Any new microcomputer attracts a following of publishers and authors; a really interesting and feature packed machine provides fertile territory for the imaginative author, so we at Amsoft are particularly grateful to Mark Harrison for filling in most of the gaps that inevitably occurred in the original user handbook, as well as adding much to the general utility of the machine in the hands of the user.

It was a considerable relief to find an author who understook the machine so well from his brief acquaintance - so many early books on new machines are very thinly disguised reincarnations of similar works for other computers. And it was even more gratifying to find a publisher willing to continue the style we had established for reference books on the system - and thus hopefully reducing the scope for problems arising from misinterpretation to a minimum.

There's plenty more to be said about the CPC464 - it really is a machine for anyone from the outright beginner to the inveterate hacker. We hope that Mark Harrison finds time to continue his exploration, as books such as this can do nothing but good for both the product and its users.

William Poel

AMSOFT

CONTENTS

1.	Amstrad & Beyond: an introduction	1
	The Principle Components of the Amstrad CPC-464	1
	Z80 Processor	1
	Input/Output Devices	2
	Memory	2
	The Amstrad's Programming Language	4
	Amstrad BASIC Commands	6
	System commands	6
	Assignment statements	7
	Control statements	7
	Input/Output statements	9
	Graphic statements	10
	Other statements	11
	Amstrad BASIC Functions	11
	Simple numeric and mathematical functions	12
	String functions	13
	I/O functions	14
	System functions	14
2.	Strings and Character Manipulation	17
	The Amstrad Character Set	17
	String Operators	19
	String Manipulation Functions	18
	More String Functions	20
	Word Searches	21
3.	Simple Input/Output Techniques	25
	Screen Input	26
	Non-stop Input	27
	Redefinition of Keys	29
	Validity of Data	30
	Screen Output	30
	PRINT USING	31
	Control Characters	32
	Animated Effects	33
	Controlled Printing	34
	Screen Modes	35
	Colour	36
	Menu Selection	37
	Screen Requesters	39
	Flashing Colours	41
	Streams	41
	Windows	42

4.	Computers, Numbers and Mathematics	45
	Numbers on the Amstrad	46
	Simple Numeric Functions	47
	Mathematical Functions	52
	Trigonometric Functions	56
	Recursive Programming	60
	Simulation	61
5.	The Amstrad Memory Map	63
	Binary Numbers	63
	The Memory Map	65
6.	Time, Clocks and Interrupts	69
	Timing Facility	69
	Interrupts	70
	Error Traps	72
7.	Data Structures	75
	Arrays - static data dtructures	75
	Dynamic Data Structures	78
	Forward Linked List	79
	Circular List	82
	Double Linked Lists	82
	Stacks and Queues	83
	Graphs	85
	Trees	87
	Heuristic Programming	91
8.	Data Processing	95
	Techniques for Sorting	95
	Bubble Sort	96
	Insertion Sort	97
	Shell Sort	98
	Quick Sort	99
	Alphabetical Sorting	100
	Cassette Files	101
	Disk Files	106
9.	Amstrad Graphics	107
	User Defined Characters	107
	Transparent Printing	109
	High Resolution Control	110
	Three Dimensional Plotting	113

10. Sound and Synthesis	119
Characteristics of Sound Waves	119
SOUND and the Amstrad	120
Channel Status	121
Tone Period	122
Duration	122
Volume	123
Sound Synthesis	123
Attack	123
Decay	123
Sustain	124
Volume Envelope Control	124
Sound Interrupts	128
Appendix A: ASCII Character Set	131
Appendix B: Key Handler Codes	133
Appendix C: Colour Codes	134
Appendix D: Error Codes	135

LIST OF PROGRAMS

These programs illustrate useful CPC-464 programming techniques and are useful in their own right.

Number	Name	Page
1	The Amstrad Character Set	18
2	Mary Whitehouse	19
3	Word Search	21
4	Database	22
5	MANARAG! (anagrams)	23
6	Calendar	34
7	Cursor Selection	38
8	Screen Requester	40
9	Highest Common Factor	48
10	Dice Distribution	50
11	Enigma	51
12	Quadratic Equations	54
13	Prime Numbers	55
14	Triangle	59
15	Recursion	60
16	Rabbits	61
17	Reaction	69
18	Missile Attack	70
19	Clock Skeleton	71
20	F.A. Cup	76
21	Linked List	81
22	Shortest Routes	86
23	Animals	92
24	Bubble Sort	97
25	Insertion Sort	98
26	Shell Sort	98
27	Quick Sort	99
28	Alphabetical/Tree Sort	100
29	Address Book	103
30	Bubbles	111
31	Epicyclics	111
32	Sine/Cosine Waves	112
33	Three Dimensional Plots	113
34	Three Dimensional Histogram	114
35	Galaxy Explorer	115
36	Animation	116
37	Rotation	117
38	Simple Keyboard Instrument	122
39	OGONEK	127
40	Alien Attack	128

CHAPTER ONE

AMSTRAD & BEYOND

– an introduction

The Amstrad CPC 464 microcomputer was launched in mid 1984. It looks extremely elegant and unlike the majority of other micros within the same price range, the Amstrad is unique in being sold with its own display unit and built in cassette player. This gives it a distinct advantage over its competitors in that the complete system consists of just two units and a single power lead; gone are the large quantities of wire or 'spaghetti' found with many other micro systems.

In many respects the Amstrad resembles an iceberg with two thirds of its power hidden from view. It is the aim of this book to progress beyond an understanding of the basic functions in order to reveal the whole range of the Amstrad's capabilities. It is assumed that the reader has read the standard Amstrad users manual and is conversant with such terms as program, program flow, variables, BASIC, processing of data, software, hardware and so on. We won't waste your time with these things here! So, let's start with an overview of the Amstrad system.

The principle components of the Amstrad CPC 464

The Amstrad CPC 464, just like any other computer, has several components or items of 'hardware' in common, notably the memory, the I/O devices and the central processing unit.

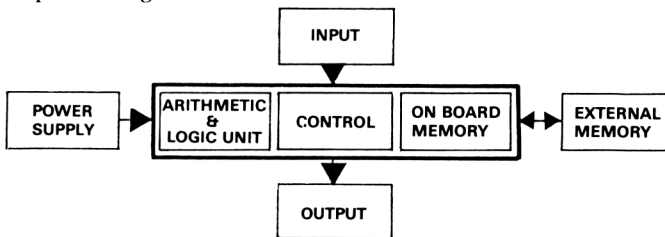


Fig 1.1 The principle components of the Amstrad CPC 464.

In detail, the components are:

Z80 Processor

The Amstrad's central processing unit (CPU) is a Z80A and is responsible for controlling all the operations of the computer. This is a relatively low cost

processor suitable for small systems and is used on many other popular machines i.e. Lynx, Sord, Sinclair to name just a few. One other function of the CPU is to evaluate any mathematical expressions involving addition, subtraction, multiplication and division (arithmetic) and also to test if numbers are positive, negative or zero (logic). Whilst the requirement of the arithmetic function is fairly obvious, the logic function is needed so that the computer can make decisions and know what actions to perform next.

Input/Output Devices

Data is entered into and accessed from the Amstrad by means of input and output (I/O) devices, examples of which include the keyboard and joysticks for input, the screen and loudspeaker for output, and a cassette unit or disk drive for both input and output. The Amstrad is available with either a colour medium resolution display or green monochromatic high resolution display. The moving-key keyboard is laid out in the standard 'qwerty' arrangement with an additional numerical keyboard. The supplied cassette recorder uses an advanced recording method for data transfer and is not compatible with other systems. At the rear of the machine are several expansion ports for interfacing to a printer, disk drive and other peripherals. A disk file system is currently being developed which will be compatible with CP/M.* Sound output is *via* a built-in loudspeaker or in stereo form from a Sony Walkman earphone socket; a volume control is also incorporated.

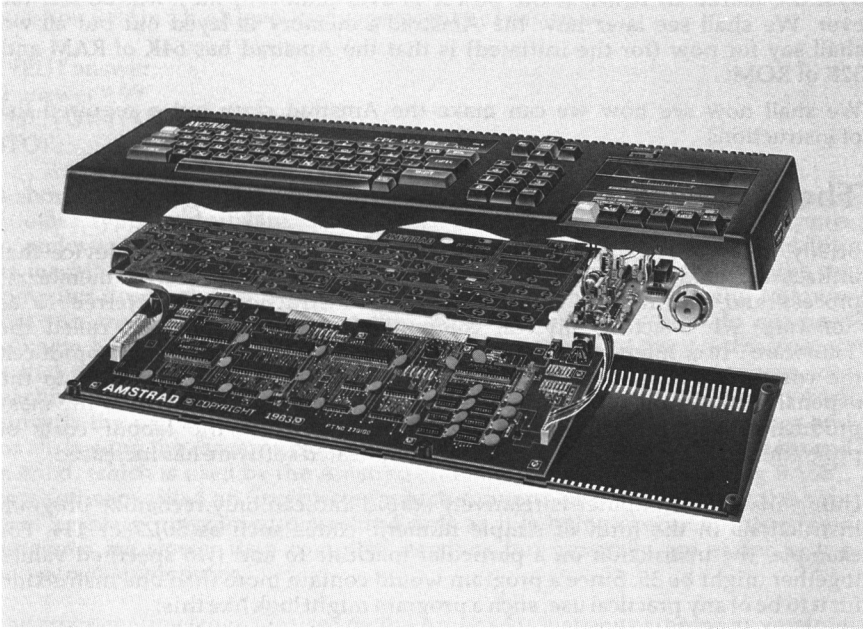
Memory

It is the function of the computer's memory to store all of the data that is held in the system. The Amstrad contains two different types of memory – these are read only memory (ROM) and random access memory (RAM), the difference being:-

ROM	RAM
Is memory that is preprogrammed when manufactured with permanent data. The contents cannot be changed by the user.	Is general purpose memory in which data may be stored and then retrieved when required. RAM loses its contents when the power is switched off.

In addition to storing a user's program and its associated data in RAM, a section of the Amstrad's memory is reserved for 'system software'. This is

* CP/M (Control Monitor and Program) is a disk based operating system for the control of a microcomputer and its peripherals. It was written by Digital Research and has since become an industry standard because it has been used so widely by over 80 computer manufacturers. The advantage of this standard operating system is that software written on a CP/M based machine is portable and can run on any other machine running under CP/M providing that there is sufficient memory available.



Some features of interest on the lower circuit board are:

The back row of chips is (left to right):

The sound generator; peripheral controller Z80A processor; Ferranti ULA. The chips on the right (in front of the ULA) are the RAM chips - 64K in all.

The Amstrad ROM is on the extreme left, in front of the sound generator chip.

Also clearly visible are, on the back of the board - left to right: Input/Output port; User ports; Printer interface; Disk interface; 5V power input (from monitor) and monitor socket.

The small circuit board at centre-right is for control of the cassette player.

The larger circuit board beneath the keyboard is the keyboard matrix.

Fig 1.2 The Amstrad CPC 464 microcomputer system.

a set of programs that is stored permanently in the computer and is used to control all the operations of the computer system; it is often referred to as the 'operating system'. Since this must not be altered either by a user's program or by switching the power off, a computer manufacturer always stores the system software in ROM. Remember that since a user's program and its data are stored in RAM, if the power is ever removed, they will be lost for ever. We shall see later how the Amstrad's memory is laid out but all we shall say for now (for the initiated) is that the Amstrad has 64K of RAM and 32K of ROM.

We shall now see how we can make the Amstrad carry out a required list of instructions.

The Amstrad's Programming Language

Briefly, the Amstrad may be described simply as an electronic device that utilizes a stored list of instructions called a program to receive, memorize, process and return data. Sometimes these instructions are referred to as 'software' while actual physical components of the system are called the 'hardware'. It is interesting to note that when computers were first produced the cost of commercial software was relatively cheap in comparison to the expensive hardware. Nowadays this trend has been reversed due to the mass production and miniaturization in electronics, while the labour costs of programmers who are developing more complicated software has increased.

Unfortunately a computer is relatively stupid and can only recognize program instructions in the form of simple numeric codes such as 50,27 or 114. For example, the instruction on a particular machine to add two specified values together might be 35. Since a program would contain more than one instruction for it to be of any practical use, such a program might look like this:

35,27,133,211,23,39,5,221,10,..... etc...

Obviously such a program is meaningless to anyone who does not know what instruction each code represents. This type of program is known as a machine code program and although it may seem both tedious and difficult to use, computer programmers in the days of the first computers had no alternative to this type of programming. It soon became obvious to the programmers that it would be easier for them to use codes that bore some resemblance to the instructions (such as "ADD", "LOAD") they referred to. At this stage it is important for the reader to remember that a computer can only ever understand programs in the machine code format; thus if a program is written with instructions referred to by different codes then it must at some stage be translated into the equivalent machine code program. This translation process is accomplished by yet another computer program – ultimately written in machine code.

A further advance in software was the development of several sophisticated programming languages known as high level languages. There are many different ones, the most common being Fortran, ALGOL, COBOL, Pascal, CORAL and BASIC. The choice of which language to use depends on the application. For example, a business application would probably use COBOL,

a scientific application might use Fortran and an application that required results to be produced within strict time limits could well use CORAL. Unlike the low level codes such as "ADD" and "LOAD", the codes in high level languages do not correspond one to one with the machine code values but, instead, allow the program to relate to the nature of the problem to be solved. For example, requiring a message "Correct" to be output if the answer to a question is 99 in a high level language might be written as:

```
INPUT answer;  
IF answer = 99  
THEN OUTPUT "Correct";  
STOP;
```

It should be evident at this stage that a high level language, when compared to other types of software, is far easier to read. As before, for a computer to understand the codes of a high level language it must be translated into its machine code equivalent; so long as the machine code version works the computer does not care where it came from. It might be reassuring to know that you are totally isolated from this translation process – you do not need to know how it works, just that it does. All that we shall say on this subject is that there are two methods used for translating high level languages into machine code versions. The first method uses software called a 'compiler' which converts the whole high level language program into machine code once and for all, leaving the machine code version to be used time after time. The second method, which is used by the Amstrad and many other micros running BASIC, uses software called an 'interpreter' which converts each small part of the high level language program into machine code as it is being executed. Interpreted programs are slower than compiled versions but generally are easier to use on a personal computer.

The programming language used by the Amstrad is just one of the many versions of the BASIC (Beginners All-purpose Symbolic Instruction Code) language that are now available. This simple language was designed originally for beginners in computing but has since become one of the most popular computer languages. Although efforts were initially made to standardize the statements in BASIC, there has recently been a tendency for computer manufacturers to produce versions of BASIC with statements that are unique to their machine. As a result there are some statements available on other computers which are absent from the Amstrad but this minor drawback can usually be overcome by the use of one or more of the available Amstrad statements. Similarly the Amstrad has several statements that are not found on other computers.

Although the Amstrad BASIC statements are listed below, they are mentioned for reference only – it is not the aim of this book to teach the fundamentals of BASIC programming since there are numerous books on the subject. You are recommended to use the rest of this chapter simply as a reference list. Have a quick look through it now, and refer back to it as necessary. It is not intended as "easy reading"!

For each BASIC keyword mentioned, the required syntax will be stated using the following notation:

AMSTRAD BASIC COMMANDS

BASIC keywords are written in capitals.

variable data is written within angle brackets '< >'

optional data is written within square brackets '['']

data items which may be repeated are followed by

SYSTEM COMMANDS

These are separated from other types of statements since they are not usually used within a program. They are used in the operation of the computer system and are accessed by typing the required statement without a preceding line number.

Command	Purpose	Syntax
AUTO	Automatically generates line numbers	AUTO [<line no>][,<increment>]
CAT	Displays files on tape	CAT
CLEAR	Clears all variables	CLEAR
DELETE	Deletes lines from program in a given range	DELETE [<line no>][-<line no>]
EDIT	Enters edit mode	EDIT <line no>
LIST	Lists program	LIST [<line no>][-<line no>] [,<stream>]
LOAD	Loads program into memory	LOAD <file name>[,<address>]
MEMORY	Resets BASIC memory pointers	MEMORY <address>
MERGE	Merges program from cassette into current program	MERGE <filename>
NEW	Erases contents of memory	NEW
RENUM	Renumbers part or all of current program	RENUM [<new line no>][,<old line no>][,<increment>]]
RUN	Executes program loaded from cassette or in current memory	RUN [<filename> or <line no>]
SAVE	Saves program or memory onto cassette	SAVE <filename>[,<filetype>]]][,<parameters>]]
TROFF	Turns off trace facility	TROFF
TRON	Turns on trace facility	TRON

ASSIGNMENT STATEMENTS

These are used to allocate either numeric values or a sequence of characters to specific sections or locations in a computer's memory.

Command	Purpose	Syntax
DATA	Specifies list of values to be used by the READ statement.	DATA <item>[,<item>].....
LET	Assigns either a numerical value or a sequence of characters to a variable.	[LET]<variable> = <expression>
MID\$	Replaces part of a string *Note that MID\$ can also be used as function (see string functions).	MID\$(<string>,<start pos>[,<length>]) = <string>
POKE	Assigns a value to a specific memory location.	POKE <address>,<expression>
READ	Takes as many items as required from the current position in the DATA statement and assigns each value in turn to the listed variables.	READ <variable>[,variable>]....
RESTORE	Repositions the READ pointer to the first item in a specified DATA statement or (if no line is specified) to the first item in all the DATA statements.	RESTORE [<line no>]

CONTROL STATEMENTS

These are some of the most vital statements in a computer's repertoire and allow us to control the order in which the program's instructions are carried out.

Command	Purpose	Syntax
AFTER	Invokes subroutine at a time interval	AFTER <integer expression>[,<integer expression>]GOSUB<line no>
CALL	Calls external subroutine	CALL <address>[,<parameter>]....
CHAIN	Loads and execute program (from line no if present)	CHAIN [MERGE]<filename>[,<line no>]
CONT	Continues from previously halted position	CONT

END	End of program	END
ERROR	Causes error action to be taken	ERROR <integer expression>
EVERY	Regularly invokes subroutine at time intervals	EVERY <integer expression>[, <integer expression>]GOSUB <line no>
FOR	Control loop – section of code is continually executed as a variable is incremented until it exceeds a terminating value.	FOR <variable> = <initial value> TO <terminating value> STEP <increment>
GOSUB	Calls internal subroutine	GOSUB <line no>
GOTO	Jumps to given line	GOTO <line no>
IF... THEN... ELSE	If condition is true control undertakes the following statements or jumps to the specified line.	IF <condition> THEN [<statement(s)>] > or <line no>][ELSE <statement(s)>]
NEXT	End of FOR control loop	NEXT [<variable>]
ON {GOSUB GOTO}	Control is passed to the line or subroutine whose position in the list is the integer component of the expression.	ON <expression> {GOSUB or GOTO} <line no>[, <line no>]....
ON BREAK GOTO	Enables a BREAK interrupt routine to be undertaken	ON BREAK GOTO <line no>
ON BREAK STOP	Disable BREAK interrupt routine	ON BREAK STOP
ON ERROR GOTO	Sets error trap	ON ERROR GOTO <line no>
ON SQ GOSUB	Enables sound queue interrupt	ON SQ <channel> GOSUB <line no>
RESUME	Resumes normal execution after error trap	RESUME[{NEXT or <line no> }]
RETURN	Returns to calling GOSUB statement	RETURN
WEND	End of WHILE loop	WEND
WHILE	Control loop – section of code is continually executed while a condition is true.	WHILE <condition>

INPUT/OUTPUT STATEMENTS

These are used to allow information to be entered into (input) and obtained from (output).

Command	Purpose	Syntax
CLOSEIN	Closes cassette input file	CLOSEIN
CLOSEOUT	Closes cassette output file	CLOSEOUT
ENT	Defines sound tone envelope	ENT <envelope no>[, <parameter>]..
ENV	Defines sound volume envelope	ENV <envelope no>[, <parameter>]..
INPUT	Input data items	INPUT[# <stream>,];["<prompt>";] <variable>[, <variable>]...
LINE INPUT	Input line	INPUT[# <stream>,];["<prompt>";] <string variable>
OPENIN	Opens cassette file for input	OPENIN <filename>
OPENOUT	Opens cassette file for output	OPENOUT <filename>
OUT	Output value to I/O port	OUT <port no>, <expression>
PRINT	Output data	PRINT[# <stream>,<item>][<separator> <item>].....
RELEASE	Releases sound channels	RELEASE <channels>
SOUND	Put sound onto sound queue	SOUND<status>, <tone period>[, < duration>[, <volume>[, <volume envelope>[, <tone envelope>[, <noise period>]]]]]
SPEED KEY	Defines keyboard repeat speed	SPEED KEY<start delay>, <repeat delay>
SPEED WRITE	Define cassette write speed	SPEED WRITE <expression>
WAIT	Waits on I/O port	WAIT <port no.>, <mask>[, < inversion>]
WIDTH	Sets width of line printer	WIDTH <expression>
WRITE	Output data (similar to PRINT)	WRITE[# <stream>,<item>][< separator> <item>].....

GRAPHIC STATEMENTS

These are used to control the excellent display features available on the Amstrad.

Command	Purpose	Syntax
BORDER	Changes colour of border	BORDER <colour 1>[, <colour 2>]
CLG	Clears graphics screen	CLG [<ink mask>]
CLS	Clears screen window	CLS [# <stream>]
DRAW	Draws line from current plot position to an absolute position.	DRAW <x coord>, <y coord>[, <ink mask>]
DRAWR	Draws line from current plot position to a relative position.	DRAWR <x offset>, <y offset>[, <ink mask>]
INK	Sets ink colour	INK <ink>, <colour 1>[, <colour 2>]
LOCATE	Moves current cursor position	LOCATE [# <stream>], <x coord>, <y coord>
MODE	Sets screen mode	MODE <expression>
MOVE	Moves current plot position to an absolute position.	MOVE <x coord>, <y coord>
MOVER	Moves current plot position to a relative position.	MOVE <x offset>, <y offset>
ORIGIN	Relocates screen origin (0,0)	ORIGIN <x>, <y>[, <left>, <right>, <top>, <bottom>]
PAPER	Sets background ink for characters	PAPER [# <stream>], <ink mask>
PEN	Sets foreground ink for characters	PEN [# <stream>], <ink mask>
PLOT	Plots a single point at an absolute position	PLOT <x coord>, <y coord>[, <ink mask>]
PLOTR	Plots a single point at a relative position to the current plot position	PLOTR <x offset>, <y offset>[, <ink mask>]
SPEEDINK	Sets flashing inks period	SPEEDINK <period 1>, <period 2>
SYMBOL	Defines character symbol by specifying bit matrix	SYMBOL <character no>[, <row data>]...

TAG	Enables text at current plot position	TAG [# <stream>]
TAGOFF	Cancels TAG for given stream	TAGOFF [# <stream>]
WINDOW	Sets up text window	WINDOW [# <stream>], <left>, <right>, <top>, <bottom>
ZONE	Sets print zone	ZONE <integer expression>

And finally,

Other Statements

Command	Purpose	Syntax
DEF FN	Defines user functions	DEF FN <name>(<parameter>[, parameter]...) = <expression>
DEFINT } DEFREAL } DEFSTR }	Sets default types (integer, real or string)	DEFINT <letter range> DEFREAL <letter range> DEFSTR <letter range>
DEG	Sets degrees mode	DEG
DI	Disable interrupts	DI
DIM	Declares arrays	DIM <variable>(<subscript>[, <subscript>]...), <variable>(<subscript>[, <subscript>]...)
EI	Enable interrupts	EI
ERASE	Clears arrays	ERASE <variable>[, <variable>]...
KEY	Sets function key	KEY <token no.>, <string expression>
KEY DEF	Redefines key	KEY DEF <key no>, <repeat> [, <normal>[, <shifted>[, <control>]]]
RAD	Sets radians mode	RAD
RANDOMIZE	Randomises current seed	RANDOMIZE [<numeric expression>]
REM	Remark	REM <comment>

AMSTRAD BASIC FUNCTIONS

Briefly, a function can be described as a rule which relates one set of values to another. The value in the first set is known as the argument and the value in the second set is the result.

SIMPLE NUMERIC & MATHEMATICAL FUNCTIONS

These are specialised functions which have numerous applications in mathematics, science and engineering. Also included is a random number generator.

Function	Purpose (Returns)	Syntax
ABS	Absolute value	ABS(<numeric expression>)
ATN	Arctangent	ATN(<numeric expression>)
CINT	Convert to integer	CINT(<numeric expression>)
COS	Cosine	COS(<numeric expression>)
CREAL	Convert to real	CREAL(<numeric expression>)
EXP	Exponential value	EXP(<numeric expression>)
FIX	Truncate to integer	FIX(<numeric expression>)
INT	Round down to integer	INT(<numeric expression>)
LOG	Natural logarithm value	LOG(<numeric expression>)
LOG10	Common logarithm value	LOG10(<numeric expression>)
MAX	Determines maximum value	MAX(<numeric expression>[,<numeric expression>].....)
MIN	Determines minimum value	MIN(<numeric expression>[,<numeric expression>].....)
PI	Constant 3.141592653	PI
RND	Random number generator	RND[(<numeric expression>)]
ROUND	Rounds value to a given number of decimal places	ROUND(<numeric expression>,[<places>])
SGN	Signum value	SGN(<numeric expression>)
SIN	Sine	SIN(<numeric expression>)
SQR	Square root	SQR(<numeric expression>)
TAN	Tangent	TAN(<numeric expression>)
UNT	Converts unsigned value to integer	UNT(<address>)

STRING FUNCTIONS

These are used for manipulating characters and text.

Function	Purpose (Returns)	Syntax
ASC	ASCII Character code	ASC(<string>)
BIN\$	Converts a number into a string of binary digits	BIN\$(<unsigned integer>[,<field width>])
CHR\$	Character whose ASCII code is specified	CHR\$(<numeric expression>)
HEX\$	Converts a number into a string of hexadecimal digits	HEX\$(<unsigned integer>[,<field width>])
INSTR	Search for a substring	INSTR([<start pos>,<searched string>,<searched for string>)
LEFT\$	Extracts left hand part of a string	LEFT\$(<string>,<length>)
LEN	Obtains length of string	LEN(<string>)
LOWERS\$	Converts string to lower case	LOWERS\$(<string>)
MID\$	Extracts substring	MID\$(<string>,<start pos>,<length>)
RIGHT\$	Extracts right hand part of a string	RIGHT\$(<string>,<length>)
SPACE\$	String of spaces	SPACE\$(<length>)
STR\$	Converts a number into a string of digits	STR\$(<numeric expression>)
STRING\$	String of a specified character	STRING\$(<length>,<character>)
UPPER\$	Converts string to upper case	UPPER\$(<string>)
VAL	Converts a string of digits into a numeric value	VAL(<string>)

I/O functions

These are used with I/O operations.

Function	Purpose (Returns)	Syntax
INKEY	State of a key on the keyboard	INKEY (<key no.>)
INKEY\$	Input key from keyboard	INKEY\$
INP	Input from I/O port	INP (<port no.>)
JOY	State of joystick	JOY (<joystick no.>)
POS	Stream position	POS (# <stream>)
REMAIN	Remain count on delay timer	REMAIN (<timer no.>)
SQ	State of sound queue	SQ (<channel>)
VPOS	Vertical stream position	VPOS (# <stream>)

SYSTEM FUNCTIONS

These are used to enquire into the state of the system.

Function	Purpose (Returns)	Syntax
EOF	End of file test	EOF
ERR	Error number	ERR
ERL	Line number on which error occurred	ERL
FRE	Amount of free memory or 'Garbage collection' - frees unused space	FRE(0) FRE(" ")
HIMEM	Address of top of memory avail- able to BASIC	HIMEM
PEEK	Contents at a specified address in the memory	PEEK (<address>)
TEST	Enquires on ink at specified position	TEST (<x coord>, <y coord>)
TESTR	Enquires on ink at relative position to current plot position	TESTR (<x offset>, <y offset>)
TIME	Time (1/300 seconds) since switch on	TIME

XPOS	Horizontal coordinate of current plot position	XPOS
YPOS	Vertical coordinate of current plot position	YPOS

For more detailed information about the available statements, the reader is referred to the AMSTRAD user's manual.

Following on from this general information, the reader will now be taken on a programming course which will explore some of the most advanced techniques possible on the Amstrad. All the facilities demonstrated are illustrated with numerous programming examples – they are there to be used, altered, added to and subtracted from to meet your own requirements so please feel free to change them as much as you like!

CHAPTER TWO

STRINGS & CHARACTER MANIPULATION

A sequence of alpha numeric or graphic characters, that is characters which need not be of a numeric nature, are usually referred to as a "string". The Amstrad has a number of string facilities enabling us to use strings of characters in addition to pure numbers to greatly increase the versatility of our programs. We shall soon see that the Amstrad's string functions are fundamental to programs requiring the manipulation of characters.

THE AMSTRAD CHARACTER SET

The Amstrad has a character set of 256 items which can be visualised as its own unique alphabet. It includes alphabetic and numeric characters, Greek letters, pixel graphic symbols, punctuation marks, mathematical symbols, control characters, etc. The Amstrad character set is based upon the ASCII character set which stands for American Standard Code of Information Interchange, however, like so many other micro manufacturers, Amstrad have altered it to suit their own requirements - so it's not really standard at all! Each character is distinguished by its own character code which is a unique number between 0 and 255. Characters and their codes, which are listed in appendix A, can be converted between one and another by using the string functions `ASC` and `CHR$`.

For example:

Each of the expressions

```
ASC("A") and ASC("AMSTRAD")
```

 examines the first character and returns 65, the ASCII code of "A"

Similarly

```
CHR$(65)
```

returns the character "A"

Try confirming the ASCII codes of other characters on your Amstrad; of course you will have to use the `PRINT` command to display the results, i.e. `PRINT ASC("464")`

The following program displays all the Amstrad's character set:

```
10 FOR j = 0 TO 255
20 PRINT CHR$(j);
30 NEXT j
```

In addition to the expected characters (such as a, b, c...or 1, 2, 3...) there are a number of characters which represent control functions and cause some irregularities on the screen. These are standard ASCII characters and are located

in the first 32 character positions. The following amended program reduces our displayed character set to one of printable characters only.

PROGRAM1: THE AMSTRAD CHARACTER SET

```
10 FOR j = 32 to 255
20 PRINT CHR$(j);
30 NEXT j
```

STRING OPERATORS

Two strings may be compared with each other by using the = sign and will return a result of true if the strings are identical, i.e. all the characters are the same and the spaces are in the same position (including those at the end). The most obvious use of this is the incorporation within an IF...THEN statement.

e.g.

```
100 INPUT "CONTINUE";q$
110 IF q$ = "NO" OR q$ = "no" THEN STOP
120 'continuation of program           (The ' acts like a rem)
```

Sometimes spaces lurking at the end of a string can play havoc with such comparisons. Such spaces can be exposed by the LEN function which returns the number of characters in the string.

e.g. PRINT LEN(a\$)

Strings may also be compared with the '>' and '<' operators; one string is less than another if it precedes the other alphabetically, as in a telephone directory. We shall later use this to write an alphabetical sorting program.

Finally, strings may be joined together (concentrated) by using the '+' operator

e.g.

```
10 a$ = "AMSTRAD "
20 b$ = "CPC 464"
30 c$ = a$ + b$
40 PRINT c$
```

Note however that strings may not be subtracted, multiplied, divided or raised to a power.

STRING MANIPULATION FUNCTIONS

The Amstrad has three string functions which are invaluable for manipulating text and characters. These are LEFT\$, RIGHT\$ and MID\$ and are used for extracting a sub-string or slice from another string. LEFT\$(a\$,n) and

`RIGHT$(a$,n)` extract n characters from the left hand end or right hand end of string `a$` respectively.

e.g.
 if `a$ = "ABCDEFGHJIJ"`
 then `LEFT(a$,3)` would return `"ABC"`
 and `RIGHT(a$,4)` would return `"GHIJ"`

A more powerful statement is `MID$` which can be used to obtain slices from within the string, unlike `LEFT$` and `RIGHT$` which are limited to starting from one end of the string.

`MID$(a$,n,m)` will extract m characters starting at the n'th character from the string. If the last argument m is omitted then all succeeding characters from the n'th are obtained.

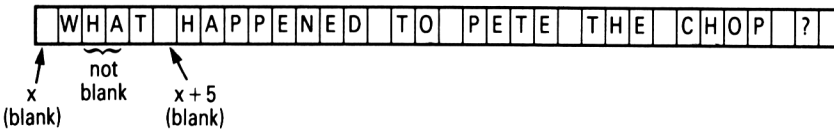
Amstrad BASIC has a further advantage over other versions in that `MID$` can also be used for assigning a string to a subsection of another. The slice that `MID$` specifies will be the destination of the string to be assigned.

e.g. `MID$(a$,n,m) = b$` will assign the first m characters of `b$` into `a$` starting at the nth character

Program 2 illustrates the use of slicing strings with the `MID$` function. A sentence of English can be entered and then re-displayed but with all the four letter words blotted out.

A string `a$` is entered so that the first and last characters are blank. A pointer `x` is set up to move along each character in turn. The computer first checks that character `x` is a space, then that the next four characters are all non spaces and, finally, that the next character is another space. If all these conditions are found to be true then a four letter word is present and so this subsection must be replaced by four stars. If it is found that not all the conditions are true then `x` is incremented by 1 and the next set of six characters is considered.

example



PROGRAM 2: MARY WHITEHOUSE

```

10 CLS
20 PRINT "THE SILICON CENSOR STRIKES AGAIN!!"
30 PRINT : PRINT "Enter your sentence"
40 PRINT : INPUT a$ : a$ = " "+a$+" "
50 FOR x = 1 TO LEN(a$)-5
60 IF MID$(a$,x,1)<>" " THEN 120

```

```

70 FOR y = 1 TO 4
80 IF MID$(a$,x+y,1) = " " THEN 120
90 NEXT y
100 IF MID$(a$,x+5,1)<>" " THEN 120
110 MID$(a$,x+1,4) = "****"
120 NEXT x
130 PRINT : PRINT a$
140 PRINT : INPUT "Continue";q$
150 IF LEFT$(q$,1) = "y" OR LEFT$(q$,1) = "Y"
THEN 10
160 END

```

MORE STRING FUNCTIONS

In some applications it is advisable to store numbers as strings of digit characters. For example you may wish to locate the decimal point in a number; this is much easier if the number is converted to a string and each character is then examined. It is also recommended practice to input strings and convert them into a numerical value. This avoids the problem of crashing a program by inadvertently entering non numeric characters when numeric variables requesting numeric data are specified in the INPUT statement. We will see more on how to avoid such problems in chapter 3 when we study validation of input data. Numbers are converted to and from character strings using **STR\$** and **VAL**.

STR\$ when applied to a numeric value, converts the number to a character string in the same format that the number would normally be printed.

e.g.

```

STR$(-12.01) would return "-12.01"
STR$(1000000000) would return "1E+09"

```

Similarly, **VAL** when applied to a string, returns the numerical value of the digits (if any) in the string. Thus we can safely enter a number by:

```
INPUT v$ : v$ = VAL(v$)
```

If any extra characters are entered they are ignored and the preceding number is returned. If the first character is unrecognised as a number then a value of 0 is returned.

It is also a common requirement to convert strings between upper and lower case; this can be done readily by the string functions **UPPER\$** and **LOWER\$**. When used, any non alphabetic characters present will remain unchanged.

e.g.

```

UPPER$("amstrad") would return "AMSTRAD"
LOWER$("COMPUTER") would return "computer"

```

Finally, two other functions **SPACE\$** and **STRING\$** return a string of spaces or a specified character of a required length.

e.g.
SPACES\$(5) would return 5 space characters
STRING\$(10, "*") would return "*****"

WORD SEARCHES

Amstrad BASIC has the string function `INSTR` which enables us to search for specific words or characters in a sentence; program 3 illustrates this. It allows a sentence and a keyword to be entered and then searches to see if the keyword is in the sentence; the sentence will then be re-displayed with all occurrences of the keyword highlighted.

PROGRAM 3: WORD SEARCH

```
10 CLS : PEN 1
20 PRINT "WORD SEARCH"
30 PRINT : PRINT "Enter keyword"
40 INPUT k$
50 PRINT : PRINT "Enter sentence"
60 INPUT s$
70 CLS
80 p=1 : v=1
90 WHILE v<>0
100 v = INSTR(p,s$,k$)
110 FOR j=p TO LEN (s$)
120 IF v=0 OR j<v THEN PEN 1 ELSE PEN 2
130 PRINT MID$(s$,j,1);
140 IF v<>0 AND j>v+LEN(k$)-2 THEN
p=v+LEN(k$):GOTO 160
150 NEXT j
160 WEND
170 PRINT : PRINT : INPUT "Hit ENTER to
continue";q$
180 IF q$ = "" THEN 10
190 END
```

One small point about this program is that `PEN 1` and `PEN 2` are used to alternate the colour of text being printed - we shall meet them again in more depth when we study colour in chapter 3.

The next example demonstrates the search technique applied to a series of as many items as you require, each contained in `DATA` statements. In addition to locating the keyword, program 4 also returns a word that is related to the keyword. So that you can understand the program, here is a little background information.

The following program has been devised for a hypothetical software house which has a large number of staff, each with various skills and experience of different machines and languages. The staff scheduler needed a means of keeping a record of each employee's skills and a method of obtaining the names of all those who were familiar with a specified subject item. So some sort of database was required.

The names of each employee are stored in **DATA** statements followed by their computer skills. To distinguish between a name and a skill, the name is preceded by the symbol "#". The program does a word search on each item; if a word is preceded by "#" then the employee's name is temporarily stored while his/her skills are examined. If a required skill is found, then the employee's name can be displayed. The program then continues to search through the remaining items until the termination symbol '%' is located.

This is probably one of the simplest forms of database programs in existence, but it can be very useful and is only limited by the memory size of the computer. It has the advantage that the staff scheduler can easily add to his records as the skills of his staff expand by amending the appropriate **DATA** statements. You can change the **DATA** statements to set up a database with something more relevant to your own needs, for example, a book or record library.

PROGRAM 4: DATABASE

```

10 CLS
20 PRINT : PRINT "SOFTWARE SKILLS LTD"
30 RESTORE : n$ = ""
40 PRINT : PRINT "Enter skill required"
50 INPUT s$ : s$ = UPPER(s$)
60 PRINT
70 READ w$
80 IF w$ = "%" THEN 120
90 IF LEFT$(w$,1) = "#" THEN n$ = MID$(w$,2)
100 IF w$ = s$ THEN PRINT n$,s$
110 GOTO 70
120 PRINT : INPUT "Hit ENTER to continue";q$
130 IF q$ = "" THEN 10
140 END

1000 'DATA ITEMS
1010 DATA #BROWN N, ADA, FORTRAN, VAX
1020 DATA #CLOUGH J, BASIC, COBOL, DEC, TAL
1030 DATA #FIELDS D, CORAL 66, COBOL, ARGUS,
TANDEM
1040 DATA #MACALISTER C, BASIC, COBOL, PDP, ARMY
SYSTEMS
1050 DATA #SINCLAIR M, ALGOL, COBOL, BURROUGHS
1060 DATA #STEVENS G, CAD/CAM, UNIX, PROLOG,
MICROCOBOL
1070 DATA #THOMPSON P, PASCAL, POLICE SYSTEMS
1080 DATA #YATES D, BASIC, Z80, 8088, CP/M,
MICROS
1090 DATA %

```

The final example of this chapter illustrates how we can manipulate text. A word is chosen at random from a list contained in **DATA** statements and then the letters are jumbled up to produce an anagram. The anagram is then displayed for a player to find the original word. The list of words can be expanded on if required by adding them in additional **DATA** statements.

PROGRAM 5: MANARAG!

```
10 CLS
20 PRINT "ANAGRAM"
30 PRINT : PRINT "Find the scrambled word"
40 a$ = "" : g$ = "" : RESTORE
50 n = 80 : 'number of words
60 FOR j = 1 TO RND*n+1
70 READ w$
80 NEXT j
90 x = LEN(w$) : ww$ = w$
100 FOR j = 1 TO x
110 n = INT(1+RND*x)
120 x$ = MID$(w$,n,1) : IF x$ = "*" THEN 110
130 MID$(w$,n,1) = "*"
140 a$ = a$ + x$
150 NEXT j
160 IF a$ = ww$ THEN 60
170 PRINT : PRINT a$
180 PRINT : INPUT "Enter guess";g$ : g$ =
UPPER$(g$)
190 IF g$ = "" THEN 180
200 IF LEFT(g$,1) = "?" THEN PRINT : PRINT
"Answer",ww$ : GOTO 230
210 IF g$<>ww$ THEN PRINT "try again" : GOTO 180
220 PRINT : PRINT "WELL DONE"
230 PRINT : INPUT "Hit ENTER to continue";q$
240 IF q$ = "" THEN 10
250 END
```

```
1000 ' word list
1010 DATA HYDROGEN, MONOPOLY, LOOPHOLE, CHIMNEY,
PACK, CHORAL, CHRONIC, DUET, ELLIPSE
1020 DATA FASCISM, MONEY, GRUEL, MALARIA,
RATIONAL, RATIO, SCOLD, CRY, COAX, LIBEL, QUIZ
1030 DATA EGOTISM, SAINT, BAPTIZE, ABACUS, AEON,
SKULL, TRIPOD, DELAY, FLINT, KNUCKLE
1040 DATA BERET, SLIT, MEMORY, SERVICE, OMEGA,
SOCIABLE, NEGLECT, HOBBY, RACQUET, SLASH
1050 DATA RACKET, WARRANT, COLUMN, CAVALIER,
GROSS, RECEIVE, TRAITOR, SCAB, SWINE, WOE
1060 DATA WHIRL, ACIDIC, POWER, AROUSE, INNINGS,
JEWEL, GRUDGE, CARNIVAL, GIGGLE, BIKINI
1070 DATA JUGGLE, FOXY, KHAKI, ACQUIT, JUBILEE,
ENIGMA, LOCH, POULTRY, ORACLE, POTTERY
1080 DATA ORTHODOX, INTERNAL, SQUEEZE, LACQUER,
IRONY, FLEXIBLE, HORRIFIC, COMPUTE, JUT, FLUID
1090 .....etc
```

If you get stuck whilst using this program, enter '?' and the answer will be revealed. One point about program 5 should be mentioned - RND is a pseudo

random number generator which returns a number 0 and 1. In chapter 4 we shall see what arguments can be passed to `RND` and see how the numbers are generated.

CHAPTER THREE

SIMPLE INPUT/OUTPUT TECHNIQUES

It probably comes as no great surprise to the reader that a computer, including the Amstrad, is useless unless it receives information that it can understand. The transfer of such information is done by means of input devices as discussed in chapter one. Similarly it is pointless in letting a computer process this information if we cannot access the results from the machine; this information may be transferred through output devices, also mentioned in chapter one. Devices that are capable of both input and output are called Input/Output (I/O) devices. In this chapter we shall discuss I/O techniques on the Amstrad and look at methods for improving the relationship between man and machine, sometimes called "the interface".

It is often the case that a programmer will spend much of his/her efforts improving the processing efficiency of a program whereas time spent on the ease of input for the user, eliminating the entry of invalid data, keeping the user fully aware of a program's state and the presentation of results would be far more beneficial. Information may be output directly to the user by three distinct methods; Shapes, Colours and Sounds, each of which is important in its own way.

Shapes can take the form of pictures or symbols, which can be recognised immediately or text, which although has to be deciphered should not pose too much of a problem. Whilst to the user text is less direct than symbols, it can often say much more in an efficient way. A further extension to shapes is animation in which the screen is updated rapidly to produce a dynamic display. Such displays are sensed quickly by the user; for example, a flashing error message has more effect than a static one. Dynamic displays can provide realistic images and are popular in arcade type games.

Colour is less powerful than shape but can be used to aid the identification of undistinguishable shapes. For example, a colour display with the top half in blue and the bottom half in green would resemble sky and grass far better than a monochromatic display can. Colour can also be used to highlight certain areas of the screen; for example, an error message displayed in a vivid red would soon be brought to the attention of the user. Finally, a colour display is nicer to look at if the colours are chosen carefully, than a two tone display.

Whilst the user must watch the screen to receive information in shape and colour form, sound output can reach the user even when he/she is not paying direct attention. Thus sound can be useful for attracting the attention of the user, if for example an operation is complete or an error has occurred. Such notes should be high and pleasant for correct operations and long low dirges

for the occurrence of errors. Sound can also add additional reality to a display. In a space invader type game, zapping and explosion sounds add realism as well as signalling to the player the results of his operations.

One major cause of user dissatisfaction is the long delays that occur in programs with long detailed numerical calculations. If programs are as efficient as possible then lengthy inactive screens with the computer showing no sign of life, should be reduced by showing screen messages such as "Program initiated", "Stage one complete" etc. This will not only reassure the user but also keep him/her informed of the program's position. At the end of a long process it is probably advisable to attract the user's attention with a bleep type sound.

The user may enter data directly into the computer by using either the keyboard or a games controller. The keyboard is by far the most flexible and powerful means of allowing entry of text, numerical values or single key responses. The Amstrad has over 75 keys, but each has an alternative meaning when pressed simultaneously with the [SHIFT] or [CONTROL] keys. Unfortunately it is common for a programmer who uses a keyboard frequently to forget that the users of their programs might not feel equally at ease. In this chapter we shall see how to make the entry of data via a keyboard as simple as possible. Whenever a user has to enter data, clear prompts should be given and any invalid data should be rejected with an error message explaining the problem and the opportunity given for the user to re-enter the data. Since keys have predefined labels it makes sense to try and use these labels as much as possible. For example, the best key to use for ending a program might be [CONTROL X] (eXit) or [CONTROL Q] (Quit) whereas to use something like [CONTROL 4] would be, to me, meaningless.

An alternative to the keyboard, is a game controller such as a joystick or paddle; these have severe limitations but can be preferable for some applications. A joystick has a set of five switches. Each of the first four are set when a knob is pushed left, right, up or down; diagonal movements cause two switches to be set. The fifth switch is set when a separate push button is pressed. A paddle is similar to the volume control on a radio and enters a single value, within a limited range, into the computer. However, the real advantage of a game controller is that the interface to the machine becomes second nature; a few minutes of moving the joystick to the left or right in order to move something on the screen left or right and one will start to do it without giving it any thought.

We shall now proceed and study some I/O techniques but remember the points we have covered in this introduction, notably:

- 1) ease and validation of input,
- 2) keeping the user fully aware, and
- 3) presentation of results.

SCREEN INPUT

The most obvious and commonly used method for entering data is with the INPUT statement which we have already used several times. There are a couple of points worth emphasising. It is possible to input several variables with one input statement; they are simply listed together separated by commas,

and when executed, the data is entered separated by commas. If the number of entered items does not match the requirement a request is made for the data to be re-entered. If you are entering a string that requires a comma to be entered, then it will be interpreted as the input separator between different items. This can be overcome by entering the string surrounded by quotes and then the string between the quotes will be accepted as a single input item.

As we have seen already, it is possible to include a prompt message within the input statement which is printed before the request for the entry of data. Following this prompt by using a semicolon produces a question mark after the prompt whilst a colon suppresses the question mark. Following the **INPUT** statement, but preceding the prompt, with a semicolon will cause the line feed character at the end of the entry of data to be suppressed i.e. printing will continue at the point of the last entered item.

A whole line of data, regardless of the comma separators, can be read by using **LINE INPUT** in which just a single string variable is specified

e.g. `100 LINE INPUT "prompt";a$`

We shall see later the use of these two statements with other devices by specifying a stream number.

NON-STOP INPUT

An excellent feature of the Amstrad is its ability to sense which key is being pressed during the execution of a program. The function **INKEY\$** is used but without any argument. If control comes across **INKEY\$** when a program is being executed, **INKEY\$** returns the character read from the keyboard; the pressing of [SHIFT] or [CTRL] along with another key is valid. If no key is pressed, then **INKEY\$** returns the empty string `""`; thus we have a method of creating a temporary halt, until any key is pressed.

e.g.
`100 PRINT "Press any key to continue"
110 IF INKEY$ = "" THEN 110
120 'continuation of program`

The input of information using **INKEY\$** has the advantage over **INPUT** in that it does not have to be followed by [ENTER] but remember, unlike **INPUT**, it will not wait for you to press a key. To input information using **INKEY\$**, a loop is set up so that if no key is being pressed the empty string is recognised and the program does not proceed. An important thing to notice is that **INKEY\$** could possibly change between lines and so it is advisable that as soon as **INKEY\$** is read it is assigned to a string variable and, from then onwards, that variable is used instead.

e.g.
`100 q$ = INKEY$`

```

110 IF q$ = "" THEN 100
120 'continuation of program

```

The following example uses concatenation with the INKEY\$ function to force the user to enter a date string of the correct format and is a good alternative to a simple INPUT statement. The complete string can then be checked for validity.

```

100 d$ = ""
110 PRINT "Enter : DD/MM/YY ";
120 FOR j = 1 TO 8
130 IF j = 3 OR j = 6 THEN c$ = "/" : GOTO 160
140 c$ = INKEY$
150 IF c$ = "" THEN 140
160 PRINT c$;
170 d$ = d$ + c$
180 NEXT j
190 PRINT
200 'validation and continuation

```

A slight extension to INKEY\$ is the function INKEY which interrogates the keyboard and returns a value which indicates the state of a specified key, analysing it whether it is pressed and, if so, along with the [SHIFT] or [CTRL] key. The values returned are listed in figure 3.1. The argument to INKEY is an integer value which identifies a key uniquely; a full list of key codes is given in appendix B.

This function is useful for detecting responses such as Y/N (yes/no) where the state of the SHIFT key is unimportant. From figure 3.1 it can be seen that the returned value is not -1 if the key is pressed.

e.g.

```

100 PRINT "Do you wish to continue (Y/N)?"
110 IF INKEY (43)<>-1 THEN GOTO 140
120 IF INKEY (46)<>-1 THEN STOP
130 GOTO 110
140 'continuation

```

			VALUE RETURNED
KEYS			
KEY	SHIFT	CTRL	
UP	?	?	-1
DOWN	UP	UP	0
DOWN	DOWN	UP	32
DOWN	UP	DOWN	128
DOWN	DOWN	DOWN	160

Figure 3.1 INKEY VALUES

REDEFINITION OF KEYS

There are often many situations when a string, perhaps in a command line, has to be continually re-entered. On the Amstrad we have the facility to program keys, so that when pressed they can return a sequence of characters; such keys are then referred to as function keys. There are 32 characters, ASCII values 128-159, which can each be expanded up to a maximum of 32 characters, however, the total number of expanded characters is 120.

Let us, for example, suppose we wanted to program a single key so that when it was pressed it would return

```
RUN "TAXPOINT" followed by the ['ENTER'] character.
```

First we must decide what key we want to use. For the time being, we are restricted to using keys that return an ASCII value in the range 128-159. It so happens that the keys on the numeric pad return values between 128 and 140, and that [CTRL] pressed simultaneously with the [ENTER] on the numeric pad will return 140. So to program this key the following command is used

```
KEY 140,"RUN" + CHR$(34) + "TAXPOINT" + CHR$(34)
+ CHR$(13)
```

Now try it.

By using **KEY DEF** we can amend the ASCII values returned by a certain key; the most obvious use of this is to amend keys so that they return ASCII values in the range that enables them to be programmed into function keys. Another use would be to disable the [ESC] (Break) key. The following parameters are specified in the given order:

<i>key number</i>	, see appendix B
<i>repeat switch</i>	, set to 1 to enable repeat, 0 to disable repeat
<i>normal</i>	, ASCII character value when key pressed alone
<i>shifted</i>	, ASCII character value when key pressed with [SHIFT] (not [CTRL])
<i>control</i>	, ASCII character value when key pressed with [CTRL]

When no new value is specified, the previous value remains. For example to program the key [CTRL E] to return EDIT:

```
KEY DEF 58, 0, 101, 69, 128
KEY 128 , "EDIT"
```

Having just mentioned the ability to enable/disable key repeat, it is worth mentioning that the auto repeat periods may be altered using **SPEED KEY** specifying the start delay, repeat period in 0.02 second units.

e.g. **SPEEDKEY 10, 10**

VALIDITY OF DATA

One method of preventing programs from crashing is to check that the data being entered is of the correct form. If the data is not correct then control should jump back to the same INPUT statement. The program should only continue if data of the correct form is entered. It is a good idea to display a message to indicate to the user what the problem with the input data was.

i.e.

INPUT data

I F data not of correct form **THEN GOTO**

continuation of program

Unlike many micros, the Amstrad checks that the entered data is of the correct type, i.e. string or numeric, to match the specified variables in the **INPUT** statement and so instead of crashing, requests that the data is re-entered.

We saw earlier when we studied **INKEY\$** that we can often force the user to enter data of the correct format, in our case a date string, but this doesn't guarantee that the entered values are valid, i.e. a month of 13 could have been given. It is, however, a simple task to then check that the month lies between 1 and 12 and then by referring to a list of days in each month that the date is valid - of course leap years must be taken into account. The numeric values are extracted from the date string using the **VAL** and **MID\$** string functions which we met in chapter 2.

Other methods for checking the validity of data include inspecting the lengths of strings using **LEN**, checking values are in a given range using the **'>'** and **'<'** operators, and checking certain characters are present by inspecting the ASCII character codes.

Another problem that can occur is when the user enters valid but incorrect input data. If this creates results that are difficult to rectify it is a good idea to re-display the entered data and ask the user to confirm it by pressing **Y** or **N** before further actions are undertaken.

Carelessness in checking input data can result in unnecessary time wasting which, in turn, can cause frustration for the user, so always try to make your programs as robust as possible.

SCREEN OUTPUT

You will be well aware that output is directed to the screen by using the command **PRINT**, we will see later that by specifying a stream number it is possible to output data to other devices. The statement is followed by the print item which can be a variable, arithmetic expression, strings, print control functions, control characters and if required a format mask. It is possible to include several print items within one **PRINT** statement by separating them with either a semicolon or comma.

Semicolon ;

A semicolon separating two print items causes the second item to be displayed immediately after the preceding item.

Comma ,

The television screen may be visualised as consisting of a sequence of print zones each, at switch on, of 13 characters. A comma separating two print items causes the second item to be displayed at the first position in the next print zone.

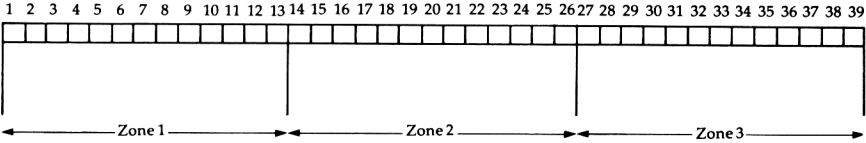


Figure 3.2

It is possible to alter the default print zone of 13 to your own requirements by the **ZONE** command

e.g. **ZONE 10**

PRINT USING

The Amstrad has a **PRINT USING** command that enables us to print the results in any format we want. The **USING** part specifies a format template consisting of the characters "# + - . * \$ ↑ , ! \ &". Each character in the template is inspected one by one and controls the image printed. Characters other than those format characters above, or format characters out of context or preceded by a '_' will also be printed. A full list of definitions is given in figure 3.3 (see next page) - note that exponential numbers are explained in chapter 4.

Any field overflow will be indicated by a '%' symbol.

examples

```
PRINT USING "####"; a
PRINT USING "Cost +$##,##.##"; c
PRINT USING "\ \"; a$
```

Fig. 3.3 PRINT USING format field specifiers – see next page

	<i>SPECIFIER</i>	<i>POSSIBLE NO. OF DIGITS</i>	<i>NO. OF CHARS</i>	<i>DEFINITION</i>
NUMERIC (max 20 chars excluding signs and exponent- ial digits)	#	1	1	Digit position
	.	0	1	Decimal point
	,	1	1	Digit position, puts comma every 3 digits to left of decimal point
	\$\$	1	2	Floating dollar sign which precedes leading digit
	**	2	2	Fills with leading asterisks
	**\$	2	\$	Floating dollar sign and leading asterisks
	+	0	1	Print either '+' or '-'. May appear at start or end of format template.
	-	0	1	Print '-' if negative. May only appear at end of format template.
↑↑↑↑	0	4	Exponential format	
STRING	!			First character only
	\<nspaces>\			First n characters
	&			Whole string

Figure 3.3 PRINT USING format field specifiers.

CONTROL CHARACTERS

In command mode it is possible to move the cursor about the screen using the cursor control keys. Amstrad BASIC enables such control functions to be programmed into PRINT statements, enabling cursor control during program mode. When programmed into a string, each control character is represented by a unique graphic symbol and is displayed in the program listing - another alternative is to access the character via the CHR\$ function.

← Move cursor one space to the left CHR\$ (8) or [CONTROL H]

→ Move cursor one space to the right CHR\$ (9) or [CONTROL I]

↓ Move cursor one space down CHR\$ (10) or [CONTROL J]

↑ Move cursor one space up CHR\$ (11) or [CONTROL K]

The next example produces five 'V's in a V formation

```
10 CLS
20 PRINT " → ↓ V → ↓ V → ↓ V → ↑ V → ↑ V ↓ ↓ ↓ ↓ ↓ "
```

By experimenting you will see that there are a number of other keys which

can be programmed into PRINT statements. Three useful ones to include in your repertoire are:

```
CHR$ (7) [CONTROL G] - sound bleeper
CHR$ (24) [CONTROL X] - exchange pen and paper inks
CHR$ (30) [CONTROL ↑] - re-position cursor to top left hand
                                corner of screen
```

In some cases it may be necessary to print the symbol associated with one of these control characters rather than use its control action. This can be done by first printing CHR\$ (1)

e.g. PRINT CHR\$ (1) + CHR\$ (8)

ANIMATED EFFECTS

With the aid of control characters and FOR . . . NEXT loops we can display a sequence of characters to produce simple animation. The idea is to construct a string which includes the characters to be displayed along with some cursor control characters to move the characters from their previous print position and some blank characters to overwrite the previous display. By controlling the program execution with a FOR . . . NEXT loop we can print the string several times in succession to produce some interesting effects.

```
10 'horizontal motion
20 CLS
30 PRINT : PRINT : PRINT
40 FOR j = 1 TO 35
50 PRINT CHR$(32) + CHR$(62) + CHR$(8);
60 FOR k = 1 TO 20 : NEXT k
70 NEXT j
80 FOR j = 1 TO 35
90 PRINT CHR$(32) + CHR$(8) + CHR$(8) + CHR$(60)
+ CHR$(8);
100 FOR k = 1 TO 20 : NEXT k
110 NEXT j
120 GOTO 40
```

We could use a similar technique to achieve vertical movement. The next example combines both horizontal and vertical movement to produce a diagonal motion.

```
10 'diagonal motion
20 CLS
30 FOR j = 1 TO 23
40 PRINT CHR$(32) + CHR$(10) + CHR$(214) +
CHR$(11) + CHR$(11 + CHR$(32) + CHR$(10));
50 PRINT CHR$(214) + CHR$(8) + CHR$(8) + CHR$(8)
+ CHR$(10);
60 NEXT j
70 GOTO 20
```

CONTROLLED PRINTING

The standard method for printing information on the screen is either along a line or down a column. This is perfectly acceptable for small quantities of data, but when printing has reached the 25th row, the screen will scroll upwards. Unfortunately, this means that the top of the display will vanish and you won't have time to read the results!

One way of avoiding this is to count the output lines on a clear screen and then freeze the display when the screen becomes full. You can carry on by pressing a specified key. In addition, titles could be displayed at the top of each fresh page.

A second problem that often occurs is when the value of a numeric variable increases (compared to a previous value) and its number of digits changes and all the following data is shifted over. This can happen partway down a screen which may slightly upset legibility or a layout, particularly in a table of results.

With Amstrad BASIC we can easily program our way out of this problem since there are two functions **TAB** and **SPC** that are used in conjunction with the **PRINT** command to direct output into specific screen columns.

The **SPC** is used to output a fixed number of spaces to the screen, or as we shall see later, to other peripherals whilst the **TAB** function is used to move the cursor a fixed number of column positions relative to the start of the current print line, starting on a new line if the column has already been passed. A semicolon is assumed to terminate both the **SPC** and **TAB** functions at all times.

These two functions are illustrated in program 6, **CALENDAR**.

This displays the calendar for any particular month and year. The original calendar was devised by Julius Caesar, but he fixed the year to be too long by eleven minutes. This was corrected with the introduction of the Gregorian calendar in Italy in 1582, although it was not introduced into England until 1752. The program will not give correct answers before this date.

The program uses a formula in a subroutine at line 1000 to calculate the day of the week on which a particular month commences. To use the program enter the month as a number from 1 to 12, followed by the year using all four digits.

PROGRAM 6: CALENDAR

```
10 CLS
20 PRINT "CALENDAR"
30 PRINT "Enter month MM/YYYY: ";
40 d$ = ""
50 FOR j = 1 TO 7
60 IF j = 3 THEN c$ = "/" : GOTO 80
70 c$ = INKEY$ : IF c$ = "" THEN 70
```

```

80 PRINT c$;
90 d$ = d$ + c$
100 NEXT j
110 m = VAL(LEFT$(d$,2)) : y = VAL(RIGHT$(d$,4))
120 IF m<1 OR m>12 THEN 10
130 CLS
140 PRINT " CALENDAR"
150 PRINT : PRINT "Month";m;SPC(5);"Year";y
160 PRINT : PRINT "SUN MON TUE WED THU FRI SAT"
170 GOSUB 1000
180 d = dd : m = m+1
190 IF m>12 THEN m = 1 : y = y+1
200 GOSUB 1000
210 IF dd<d THEN dd = dd+7
220 n = 28+dd-d
230 p = 4*d+1
240 FOR x = 1 to n
250 PRINT TAB(p);x;
260 p = p+4 : IF p>=26 THEN p = 1
270 NEXT x
280 PRINT : PRINT : INPUT "Hit ENTER to
continue",q$
290 IF q$ = "" THEN 10
300 END
1000 mm = m-2 : yy = y
1010 IF mm>0 THEN 1030
1020 mm = mm+12 : yy = yy-1
1030 c = INT(y/100) : yy = yy-100*c
1040 dd = 1 + INT(2.6*mm-0.19)+yy+INT(yy/
4)+INT(c/4)-2*c
1050 dd = dd MOD 7
1060 RETURN

```

Amstrad BASIC has yet another way of repositioning the current print position; this uses the command `LOCATE` where the new x and y coordinates are specified.

e.g. `LOCATE 10,1`

We shall meet this command on numerous occasions throughout the book.

SCREEN MODES

Up to now the screen has consisted of 25 rows of 40 character positions which, on the Amstrad, are set automatically at switch on. Each individual character position is, in fact, built up of an 8 x 8 matrix of dots or pixels which may or may not be illuminated. Thus we can describe the display as being made up of 320 x 200 pixels. Later we shall see how, by controlling the individual pixels, we can produce a high resolution display as opposed to a character display. When the screen is in this state it is said to be in default mode or

MODE 1. There are two other modes, called logically, MODE 0 and MODE 2 which take the attributes given in 3.4.

	<i>MODE 0</i>	<i>MODE 1</i>	<i>MODE 2</i>
<i>CHARACTERS</i>	25 lines of 20 characters	25 lines of 40 characters	25 lines of 80 characters
<i>PIXELS</i>	200 high by 160 wide	200 high by 320 wide	200 high by 640 wide

Figure 3.4: Screen Modes

Each of the screen modes may be selected by the command *MODE*, i.e. *MODE 0*, etc.

COLOUR

At last we have reached the exciting topic of colour, a subject whose importance has already been mentioned in the introduction to this chapter. The Amstrad is capable of displaying information in 27 colours although the number that can be displayed simultaneously is restricted, depending on the screen mode, to values shown in figure 3.5.

	<i>MODE 0</i>	<i>MODE 1</i>	<i>MODE 2</i>
<i>Max number of colours</i>	16	4	2

Figure 3.5: Colour maximums

Each of the 27 colours available is identified uniquely by a colour code number which are listed in appendix C. Each character position has two associated colours. The background colour is chosen by the command *PAPER* and the colour of the character itself is chosen by *PEN*. The number associated with both these commands is not the colour code, instead an ink number is specified which has a colour code associated with it, which in turn is set up by the *INK* command.

e.g

PEN 1 means set the foreground of future printed to the colour assigned to *INK 1*

PAPER 2 means set the background of future printed characters to the colour assigned to *INK 2*

while

INK 1, 0 means assign colour 0 to ink number 1

Thus it should be evident that we have two ink colours in use at any one time and if you want to see what you are printing it makes sense not to have the two inks set to the same colour!

The colour of the screen border is controlled independently by the **BORDER** command and this time the actual colour code is specified e.g. **BORDER 0** means set the border to the colour 0.

Try the following example to flick through the colours by pressing any key on the keyboard (except [ESC])

```
10 PEN 2 : PAPER 1
20 FOR j = 0 TO 26
30 CLS
40 BORDER j
50 INK 1,j : INK 2,(j+12)MOD 26
60 LOCATE 15,10 : PRINT "COLOUR";j
70 IF INKEY$ = "" THEN 70
80 NEXT j
```

MENU SELECTION

It is common practice for programmers to start with an initial idea and then continue to add extra refinements until the logic behind the program resembles a 'spaghetti like' structure with numerous jump statements sending control in all directions; a better approach is to use 'structured programming'. This is a methodology whereby a program is broken down into blocks or modules each of which has a single specific purpose. Structured programming also tries to avoid the **GOTO** statement in an effort to make understanding the logic in a program simple. Also, breaking down a program in this way means that each function can be tested in isolation so that any bugs present can be easily located.

An ideal method of producing a program that undertakes several functions is to write each in a separate block of code and then allow the user to select the required section by means of a menu displaying the available options.

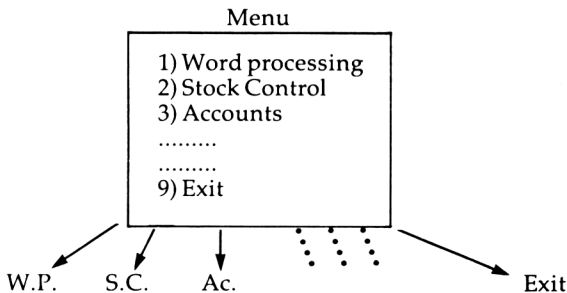
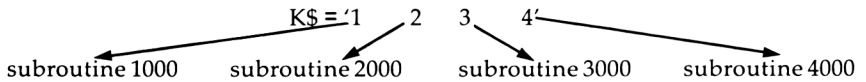


Figure 3.6: Menu Selection

Of course there is no reason why selecting a certain option could not lead to another menu displaying a further list of related options so that the user has to make an additional selection or return back to the main menu.

There are numerous ways in which a menu selection program could be written. The most common is the 'Chinese Take Away' method where the user selects a letter or number from a displayed list. Consider the example given below; in this case the user has four options to choose from by pressing 1, 2, 3 or 4. On selection, the program checks that the required key is contained in a string, say K\$ and then, if present, undertakes the required option as a subroutine, or if absent returns to re-display the main menu. By constructing a program in this fashion, it is easy to add extra functions to the program with only a minimal number of changes to the existing code.



Example Menu

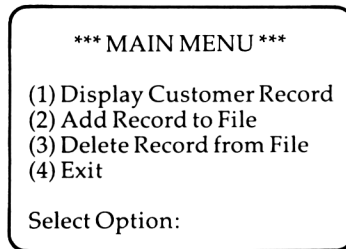


Figure 3.7

A slight variation on this method is to move the cursor up and down using the cursor control keys until it is level with the option required. Then by pressing a specified key the required option is undertaken. This is done by keeping a counter which is decremented or incremented as the cursor is moved; on selection the counter will correspond to the option required.

Program 7: CURSOR SELECTION

```

10 CLS
20 LOCATE 10,6 : PRINT "*** MAIN MENU ***"
30 RESTORE : READ n
40 FOR j = 1 TO n
50 READ a$
60 LOCATE 10,7+j : PRINT j;a$
70 NEXT j
80 p = 1
90 IF p>n THEN p = 1
100 IF p<1 THEN p = n
110 pp = p
120 LOCATE 9,7+p : PRINT ">"
130 FOR j = 1 TO 200 : NEXT j
140 IF INKEY(18) <>-1 THEN 190
150 IF INKEY(0) <>-1 THEN p = p-1
160 IF INKEY(2) <>-1 THEN p = p+1
170 IF p<>pp THEN LOCATE 9,7+pp : PRINT " " :
  
```

```

GOTO 90
180 GOTO 140
190 ON p GOSUB 1000,2000,3000,4000,5000
200 GOTO 10
210 DATA 5
220 DATA "Exit"
230 DATA "Display record"
240 DATA "Delete record"
250 DATA "Add record"
260 DATA "Help"
1000 END
2000 RETURN
3000 RETURN
4000 RETURN
5000 RETURN

```

The program has been written so that it is easy for you to tailor it to your own requirements. To do so amend the data statements to contain your own options, line 210 should contain the number of options and then add your subroutines, (ensure that it is called in line 190). The cursor control keys, up and down, will move the cursor and the option is selected by pressing [ENTER].

SCREEN REQUESTERS

In many programs it is common to find that the user is prompted line by line for information to be entered. For example,

```

Enter Input Device ? $TERM
Enter Output Device ? $LP
Enter Filename ? OUTPAY
Enter Access Mode ? READ ONLY
Enter Exclusion Mode ? SHARED

```

The entry of such data would be greatly improved if all the screen prompts could be displayed simultaneously and if the user could correct previously entered data. Such a routine that could handle this input is often referred to as a 'Screen Requester'.

We shall now see an example screen requester which handles the entry of a name, address and telephone number for use in such applications as a mailing list. It has been written in a form that can easily be amended to handle your own requirements. The list of prompts along with the maximum length of the input field is held in DATA statements at the end of the program.

Input data is entered on the screen at the position of the cursor. The cursor can be moved about the screen using the cursor control keys. When all the data has been entered the screen can be accepted by pressing the [COPY] key.

Two counters are used, LP and CP. LP is altered by the cursor up and cursor down keys; it points to the line being entered. Similarly, CP is altered by

the cursor left and cursor right keys; it points to the character in the line being entered. An array F\$ is used to record the entered data. As alphabetic and numeric keys are pressed, they are displayed on the screen, added to the array F\$ in the appropriate position and finally the cursor position is updated.

```

Name      JOE PUBLIC
Street    99, THE HIGH STREET
Town      NEWTOWN
County    SURREY
Post Code GU11 1AA
Phone     34567

```

Figure 3.8: Example Screen

Program 8: SCREEN REQUESTER

```

10 PAPER 0 : CLS
20 READ a$
30 LOCATE 14,4 : PRINT a$
40 READ n
50 DIM f$(n),x(n)
60 cp = 1 : lp = 1 : v = 0
70 FOR j = 1 TO n
80 READ a$,x(j)
90 LOCATE 2,5+j : PRINT a$
100 f$(j) = SPACE$(x(j))
110 NEXT j
120 IF lp>n THEN cp = 1 : lp = 1
130 IF cp<1 THEN cp = 1
140 IF cp>x(lp) THEN cp = 1 : IF v<>2 THEN lp =
lp+1
150 IF lp>n THEN cp = 1 : lp = 1
160 IF lp<1 THEN cp = 1 : lp = n
170 FOR j = 1 TO n
180 PAPER 3
190 LOCATE 14,5+j : PRINT f$(j)
200 NEXT j
210 PAPER 0
220 LOCATE cp+13,lp+5 : PRINT CHR$(143)
230 v = 0
240 c$ = INKEY$
250 IF c$ = "" THEN 240
260 c = ASC(c$)
270 IF c = 224 THEN 400
280 IF c = 13 THEN cp = 1 : lp = lp+1 : v = 2
290 IF c = 244 OR c = 240 THEN lp = lp-1 : v = 2
300 IF c = 241 OR c = 245 THEN lp = lp+1 : v = 2
310 IF c = 242 OR c = 246 THEN cp = cp-1 : v = 1
320 IF c = 247 OR c = 243 THEN cp = cp+1 : v = 1
330 IF v = 0 THEN MID$(f$(lp),cp,1) = c$ : cp =
cp+1

```

```

340 GOTO 120
400 CLS
410 FOR j = 1 TO n
420 PRINT f$(j)
430 NEXT j
440 END
1000 DATA "ADDRESS BOOK"
1010 DATA 6
1020 DATA "Name",20
1030 DATA "Street",25
1040 DATA "Town",15
1050 DATA "County",12
1060 DATA "Post Code",10
1070 DATA "Phone",15

```

FLASHING COLOURS

To make either the text, background or border flash we can specify a secondary colour in either the **INK** or **BORDER** statements.

e.g.
INK 1,10,20 any colour associated to ink 1 will alternate between colour 10 and colour 20
BORDER 10,20 similarly the border will alternate between colour 10 and colour 20

The frequency of the alternating colours can be set by the **SPEED INK** command, the first value specifies the period of the primary colour and the second value specifies the period of the secondary colour - all timed in periods of 0.02 seconds.

The use of colour adds a powerful facility to our programming skills and there will be numerous examples of its use in the rest of this book.

STREAMS

When we 'talk' to the Amstrad's I/O devices we do so by means of streams. The ideas behind streams are quite common in computing and are not a difficult concept to grasp. On the Amstrad there are ten input and ten output streams and each is associated with one of the I/O devices as given in figure 3.9. To communicate to a specific device we must specify the stream number after the I/O command. Up to now we have omitted the stream number from our program as the Amstrad's operating system defaults to stream 0.

<i>Stream Number</i>	<i>Device</i>
0	} window
1	
2	
3	
4	
5	
6	
7	
8	Printer
9	Cassette Unit

Figure 3.9

So, for example, to output a listing to the printer we would use `LIST,#8`. We shall meet I/O to the cassette unit and cassette files in chapter 8. What is interesting from figure 3.9 is that up to eight streams are linked to the screen and we shall see how this enables us to divide up the screen into separate areas, each of which may have different screen attributes or be used for distinct purposes. Such areas of screen are referred to as windows.

WINDOWS

To set up a screen window we must define its position on the screen and then use the `WINDOW` command which specifies a stream number followed by the left, right, top and bottom screen coordinates of the window. For example consider the screen window in figure 3.10.

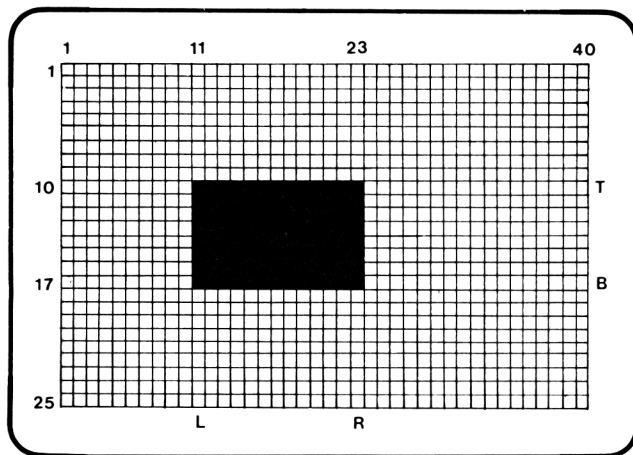


Figure 3.10

To define this window to stream 1 we would use
`WINDOW #1, 11, 23, 10, 17`
and from then onwards any I/O commands specifying stream 1 would be via the new window. When using the inner window we must remember that new screen coordinates apply with 1,1 as usual being the top left hand corner.

Consider the following example which can easily be customised and included in your own programs. The idea is that if an error occurs the subroutine at line 1000 is called up which displays a flashing error message on the bottom line. A sound bleep is also given and the user must acknowledge the error by hitting any key before continuing. Note that because window 0 overlaps the whole of the error text line, clearing window 0 automatically clears the error message. In chapter 6 we shall see how to cause error traps that automatically call an error subroutine.

```
10 e$ = "user message"  
20 WINDOW #1,1,40,25,25  
30 INK 3,3,26  
40 PAPER #1,3 : PEN #1,0  
50 CLS  
60 PRINT "Press 'e'"  
70 IF INKEY$ <>"e" THEN 70  
80 GOSUB 1000  
90 GOTO 50  
1000 LOCATE #1,1,1  
1010 PRINT #1,"Error : ";e$ : PRINT CHR$(7)  
1020 IF INKEY$ = "" THEN 1020  
1030 RETURN
```

This now concludes the chapter on simple I/O and hopefully has demonstrated the objectives set out in the introduction. The functions we have met so far, such as colour, screen modes, controlled printing, streams, windows, etc, will be encountered again and again as we lead into more advanced I/O including graphics and sound synthesis.

CHAPTER FOUR

COMPUTERS, NUMBERS AND MATHEMATICS

While everybody realises that a computer can handle very complicated arithmetic expressions, it is a common misconception for people to think that this is a simple task - in fact it is probably one of the hardest!! There are two main reasons for this; the first being that a computer is expected to be able to cope with a vast range of numbers and secondly that all the individual locations that make up a computer's memory are restricted to whole numbers within a specific range; on the Amstrad this is 0 to 255 inclusive. It is, however, comforting to know that when a high level language is used, a programmer need not have any knowledge of the complicated methods used to evaluate arithmetic expressions and mathematical functions as this is undertaken automatically by a section of software contained in the operating system. When programming in machine code it is still possible to call up these mathematical routines in the operating system but much more care is required with passing the operands and operators in a form that the routines comprehend, checking that no mathematical overflow occurs, accessing the results from the correct locations in memory and interpreting the results by converting them from the form returned to the required form.

Mathematics is a huge subject and its use within computing is very common. Computers controlling finance, navigation, defence, computer aided design & manufacture, office automation, stock control, arcade games, etc all require numerical evaluations whether it is for the complex task of tracking and flight planning aircraft which involves such factors as speed, weather conditions, the spherical shape of the world, or just simply keeping the score in a computer game. There are many levels of mathematics and there is no need to worry if you do not understand some of the more complex mathematical functions which are given on the Amstrad. While such mathematics as addition, multiplication, etc should be second nature, do not be alarmed if the latter section in this chapter seems double dutch; some of it is very specific to the requirements of engineers, scientists and technicians and so should not put you off in any way. Even if the reading does get difficult it is worth trying out the examples as they illustrate what your Amstrad is capable of. Mathematics can be fun and using a computer can be an excellent way of making the subject more enjoyable. Any reader who has an interest in such topics is referred to the following book:

"MATHS + COMPUTERS = FUN" by G T Childs
(published by Sigma Technical Press)

This book is useful for both children and many adults, since it shows that mathematics need not be a hard subject. It deals with many topics ranging from junior mathematics to those at an 'A' level standard and contains numerous aids, entertaining puzzles and over 50 BASIC programs.

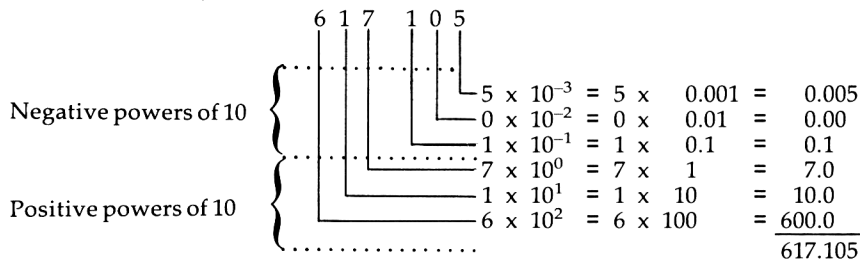
NUMBERS ON THE AMSTRAD

Those readers who have little experience in mathematics have probably never given a great deal of thought to the many notations in which numbers may be written. For example, the first three different numbers we ever learnt about were whole numbers, fractions and decimals. In a similar fashion, there are three particular notations that we shall consider, all of which are commonly used in computing. You will recognise the first two notations although the names given to them might be new.

INTEGER Integers, commonly called whole numbers, are numbers which are a sequence of digits and which do not involve a decimal point or any fractional component. They may be either positive or negative. e.g. -156, -22, -1, 0, 3, 69, 999

REAL Real numbers are a sequence of decimal digits with a single decimal point either at the end, the beginning or between two digits. The digits to the left of the decimal point are weighted in positive powers of 10 and form the integer component of the number. Likewise the digits to the right of the decimal point are weighted in negative powers of 10 and form the fractional component of the number. Real numbers may be positive or negative. e.g. -2.3, 0.4, 31.0234, .125, -0.275

To illustrate this idea:



EXPONENTIAL Exponential numbers are written in a notation which is suitable for either very large or very small numbers. They are written using a real number followed by the symbol 'E' and an integer number. 'E' means "times 10 to the power of". Thus yEx means 'y times 10 to the power of x'.

e.g.

$$1.234E-2 = 1.234 \times 10^{-2} = 0.01234$$

$$1.234E0 = 1.234 \times 10^0 = 1.234$$

$$1.234E2 = 1.234 \times 10^2 = 123.4$$

The effect of the integer following the 'E' is to indicate how many places the decimal point has to be shifted. A positive value means a shift to the right and a negative value means a shift to the left.

e.g. 1E10 means 1 followed by 10 zeros,
i.e. 10,000,000,000

Using this scientific notation, the largest number (the furthest away from zero) which the Amstrad is capable of storing is approximately 1.7×10^{38} and the smallest number (the number nearest to zero) is approximately 2.9×10^{-39} . Numbers are stored to an accuracy of between 9 and 10 digits.

In chapter 5 we shall meet the binary and hexadecimal numbering systems which are forms of expressing numbers which bear more resemblance to the way that numbers are stored and operated on in the internal workings of a computer; until then, the above notations will suffice.

Variables

All readers should be fully aware that any items of user data stored in the Amstrad's memory are given symbolic names, referred to as variables. Variables may be used to locate integer, real or string data, and distinguished by a type marker i.e. %, ! or \$ respectively.

e.g.
a% integer
a! real
a\$ string

In fact the same name may be used for all three since they constitute separate variables. By omitting the type marker, a default type is assumed, at switch on it is set to real for all variables. It is possible to re-define the default by the commands **DEFINT**, **DEFREAL** and **DEFSTR**; these specify a range of letters for which all variables, whose names lie within the range, take the corresponding default type.

e.g. **DEFINT I - N** variables commencing with I, J, K, L, M, N refer to integer variables if the type marker is omitted.

So let us look at the Amstrad's functions.

SIMPLE NUMERIC FUNCTIONS

INT

The INT function returns the integer component of the argument by rounding down.

e.g. `PRINT INT(-5.6), INT(0), INT(5.6)` will cause -6, 0 and 5 to be displayed
`PRINT (X + 0.5)` will round X to the nearest integer

Program 9 demonstrates the use of the INT function by finding the highest common factor of two positive integers, i.e. it finds the largest number that can be divided into both numbers exactly. It uses a well known algorithm which can be easily understood by following the program on paper. Draw three columns labelled A, B and C and record the value of each variable in the corresponding column as it changes.

Program 9: HIGHEST COMMON FACTOR

```

10 CLS
20 PRINT "HIGHEST COMMON FACTOR"
30 PRINT
40 INPUT "Enter first number";a
50 INPUT "Enter second number";b
60 IF a<1 OR b<1 THEN 40
70 c = b : b = a-b*INT(a/b) : a = c
80 IF b<>0 THEN 70
90 PRINT : PRINT "H.C.F. is";c
100 IF INKEY$ = "" THEN GOTO 100
110 GOTO 10

```

CINT

The CINT function returns the integer component of the argument by rounding to the nearest integer in the range -32768 to 32767
e.g. `PRINT CINT(n)`

CREAL

The CREAL function returns the argument to a real number
e.g. `PRINT CREAL(n)`

FIX

The FIX function returns the integer component of the argument by truncating the fractional component
e.g. `PRINT FIX(n)`

ROUND

The ROUND function returns the argument rounded to a specified number of decimal digits or powers of ten
e.g. `PRINT ROUND(3419882, -4)` would display 342000
`PRINT ROUND(3.1415926, 4)` would display 3.1416

MIN

The MIN function when applied to a list of argument returns the minimum value
e.g. `PRINT MIN(10, 5, 15, 25, 10)` would display 5

MAX

The MAX function is similar to MIN but returns the maximum value
e.g. `PRINT MAX(-1, -2, -1, -4)` would display -1

RND

The RND function is a RaNDom number generator which can be extremely useful for games requiring an element of luck. It has random displays which produce random data, etc and its use is usually only limited by your imagination. The number generator is not truly random since it works by returning values sequentially from a very long list of numbers. Since the list is so long a random number generator of sorts can be achieved by starting at an unknown position which is more than adequate for the requirements of most users.

e.g. `LET a = RND` will assign a value to a that is greater than or equal to 0 and less than 1

RND is illustrated in program 10 which simulates a dice being rolled 100 times; this is achieved by obtaining a random integer between 1 and 6 inclusive by using:

```
INT(RND*6+1)
```

Since the randomness of RND is in question, the program continues to evaluate two mathematical values, average and variance. Averages are very common and you are probably well aware that an average is the sum of the value of items divided by the number of items; when throwing a "truly" random dice, an expected average would be 3.5. Variances are less common to non mathematicians and indicate the extent to which values tend to be spread about the average value, or simply, it is a measure of the "width" of distribution. So that we are not side-tracked from our topic of simple numeric functions, the mathematical formulae for evaluating averages and variances will just be stated and so may be meaningless to a few readers. However it may help to know that \sum means "sum of" and so

$\sum_{i=1}^n x_i$ would mean the sum of x_1, x_2, x_3, \dots up to x_n .

$$\text{Average: } \bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

$$\text{Variance: } \sigma = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}$$

$$= \frac{\sum_{i=1}^n x_i^2 - n\bar{x}^2}{n} \quad (\text{re-arranging mathematically})$$

To some readers it may not be apparent how variances work; an explanation has been omitted as it involves mathematics which are not relevant to the Amstrad user. It is, however, common practice in commercial programming for an algorithm or method to be devised by a different person to the one that actually codes the program; it is sufficient for the programmer just to know that the algorithm works. As you can see, you may now be in the position of many a commercial programmer; do not let it put you off!!

Returning to random numbers, there is a command **RANDOMIZE** which sets the position in the random number sequence to a specific position. The parameter to this command is a numerical value often referred to as the seed. Resetting the seed to a previous value guarantees that the same sequence of numbers is achieved. To really randomize the generator we should use **RANDOMIZE TIME** which sets the seed to a value based on the Amstrad's internal clock and as a result it would be difficult to repeat. Incidentally, if the command is executed with the seed omitted then the user will be prompted for a seed value. And so to our program:

Program 10: DICE DISTRIBUTION

```
10 RANDOMIZE TIME
20 CLS
30 PRINT "DICE DISTRIBUTION" : PRINT
40 s = 0 : ss = 0
50 FOR x = 1 TO 100
60 d = INT(RND*6+1)
70 s = s+d : ss = ss+d*d
80 PRINT d;
90 NEXT x
100 a = s/100
110 PRINT : PRINT : PRINT "AVERAGE";a
120 v = ss/100-a*a
130 PRINT : PRINT "VARIANCE";v
140 PRINT : PRINT "Press any key"
150 IF INKEY$ = "" THEN 150
160 GOTO 20
```

Of course you could change the input of the average and variance routines to something which is more appropriate for you; I'm sure your bank manager would be impressed if you quoted both the average and variance of your bank balance if you had to persuade him to let you off your bank charges!

There is a further extension to the function **RND** which is obtained by passing an argument which gives you the following results:

If positive	- returns random number (as if argument was absent).
If zero	- returns previous random number again.
If negative	- resets position in sequence and returns the first number from the new position.

The next program demonstrates how setting the **RANDOMIZE** seed to the same value always produces the same sequence of generated numbers. Program 11 is used to code and decode secret messages and works on a similar method to the German 'Enigma' coding device used in World War Two. This machine was dependent on an entered 'codeword' and would then code messages by replacing the characters in the message by other characters. To make things more difficult, when a character appeared several times in a message it would never be replaced by the same character each time.

After the program has been executed and a codeword has been entered press 'c' to code a message

'd' to decode a message
 'r' to reset the codeword
 'n' to enter a new codeword
 's' to return to the command mode.

The program uses the random number generator to choose the replacement characters. The basic idea lies in the fact that when we execute RANDOMIZE n we always commence at the same position in the sequence list. The seed used is dependent on the character codes of the characters in the codeword. Pressing 'r' will reset the sequence of random numbers to its initial position for the entered codeword. When sending the message it is important that the receiver knows the codeword so that he can set the random number generator to the correct starting position (on his Amstrad of course!!) Remember to keep resetting the codeword when necessary and make sure you inform your ally of the position of all spaces (if there is a space at the start of your input string then you should enter the message within quotes).

Program 11: ENIGMA

```

10 b$ = SPACES(1) +
"0123456789ABCDEFGHIJKLMNPOQRSTUVWXYZ"
20 b$ = b$ + b$
30 CLS : PRINT "ENIGMA CODER"
40 PRINT : INPUT "Enter Codeword";c$ :
c$=UPPER$(c$)
50 IF LEN(c$)<2 THEN 40
60 RANDOMIZE 10*ASC(LEFT$(c$,1)) +
ASC(RIGHT$(c$,1))
70 CLS : PRINT "ENIGMA CODER" : PRINT
80 PRINT "Enter:" : PRINT
90 PRINT "s)STOP" : PRINT "c)CODE" : PRINT
"d)DECODE"
100 PRINT "r)RESET CODEWORD" : PRINT "n)NEW
CODEWORD"
110 PRINT : INPUT a$ : a$=UPPER$(a$)
120 IF a$ = "S" THEN CLS:END
130 IF a$ = "R" THEN 60
140 IF a$ = "N" THEN 30
150 IF a$ = "C" THEN 250
160 IF a$<>"D" THEN PRINT CHR$(7) : GOTO 70
170 PRINT : PRINT "Enter message to decode"
180 PRINT : INPUT m$ : m$=UPPER$(m$) : PRINT
190 GOSUB 360
200 p = INT(RND*26+1)
210 PRINT MID$(b$,v+p,1);
220 m$ = MID$(m$,2)
230 IF m$ = "" THEN 330
240 GOTO 190
250 PRINT : PRINT "Enter message to code"
260 PRINT : INPUT m$ : m$=UPPER$(m$) : PRINT
270 GOSUB 360
280 p = INT(RND*26+1)

```

```

290 PRINT MID$(b$,37+v-p,1);
300 m$ = MID$(m$,2)
310 IF m$ = "" THEN 330
320 GOTO 270
330 PRINT : PRINT : PRINT "Press any key to
continue"
340 IF INKEY$ = "" THEN 340
350 GOTO 70
360 v = ASC(m$)
370 IF v<48 OR v>90 THEN v=47
380 IF v>64 AND v<91 THEN v=v-7
390 v = v-46
400 RETURN

```

MATHEMATICAL FUNCTIONS

We shall now take a look at a number of functions available on the Amstrad which are commonly used in mathematical and scientific applications; however, as previously warned, some readers may find them far too specific for their own needs.

EXP

EXP(x) is equal to a constant 'e' raised to the power of x where e is defined as 2.718281828. To explain how such a curious number was chosen, consider what happens to the expression $(1 + 1/n)^n$ as n gets very large (mathematicians would say "as n tends to infinity" and write it as $n \rightarrow \infty$). By running the program below we see that our expression tends to this number e, as defined above, as n tends to infinity.

i.e. written mathematically $\lim_{n \rightarrow \infty} (1 + 1/n)^n = e$

It turns out that a more general result is obtained from the expression $(1 + 1/n)^n$. This is found to tend to e^x as n tends to infinity.

i.e. $\lim_{n \rightarrow \infty} (1 + 1/n)^n = e^x$

So, we can now write a simple program to calculate e:

```

10 PRINT "exponential e"
20 FOR x = 0 TO 5
30 n = 10 ↑ x
40 e = (1 + 1/n) ↑ n
50 PRINT n;TAB(10);e
60 NEXT x

```

Note that we cannot fully demonstrate n tending to infinity for if we do, large inaccuracies occur due to us 'stretching' the restrictions contained in the Amstrad's O.S. numerical routines.

A graph of e^x plotted against x would look like this, such a curve is called an 'exponential curve'.

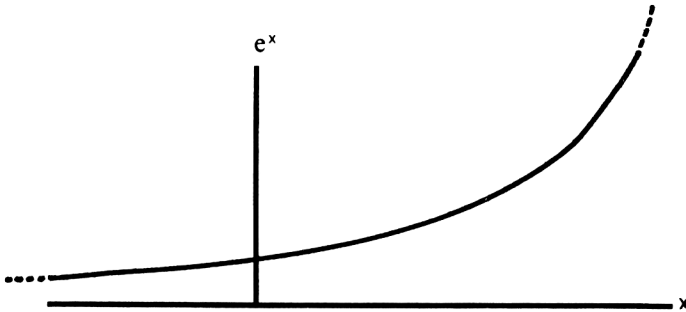


Figure 4.1 : Exponential curve

LOG

The LOG function produces the “natural logarithm” of a number. The logarithm of a number is the power needed to produce that number by raising the base to the power. Natural logarithms use a base of e .

e.g. $\text{LOG}(25) = 3.21887583$ because $e^{3.21887583} = 25$

From natural logarithms it is possible to find the logarithm in any base using the following relationship.

$$\text{Log}_s(x) = \frac{\text{LOG}(x)}{\text{LOG}(s)}$$

LOG10

If in the above equation the value of s was 10, then the resultant logarithms are called “common logarithms”. LOG10 will produce the common logarithm of a number directly.

SQR

The SQR function returns the square root of a number, i.e. if the argument is positive the function returns the number that when multiplied by itself is equal to the argument. Care must be taken since the square root of a negative number will result in an error.

$$\text{SQR}(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is greater than or equal to } 0 \\ \text{error} & \text{if } x \text{ is less than } 0 \end{cases}$$

ABS

The ABS function returns the absolute value of a number, this has the same magnitude as the argument but with a positive sign.

$$\text{ABS}(x) = \begin{cases} x & \text{if } x \text{ is greater than or equal to } 0 \\ -x & \text{if } x \text{ is less than } 0 \end{cases}$$

SGN

The **SGN** functions returns the signum of a number, this is +1 if the argument is positive, -1 if the argument is negative or 0 if the argument is zero.

$$\text{SGN}(x) = \begin{cases} +1 & \text{if } x \text{ is greater than } 0 \\ 0 & \text{if } x \text{ is equal to } 0 \\ -1 & \text{if } x \text{ is less than } 0 \end{cases}$$

So let's now look at a few examples where such functions are utilised.

Quadratic Equations

This program illustrates the use of the square root function, **SQR**. A quadratic equation is one of the form:

$$ax^2 + bx + c = 0$$

where a, b and c are known constants and x is required to be found so that the above equation is true. The following formula for the solutions to the equation can be proved correct mathematically but we will, once again, take the role of the commercial programmer and assume that the given method is correct. So, the solutions are

$$x = \frac{-b \pm \sqrt{b^2 - 4*a*c}}{2*a}$$

If $b^2 - 4*a*c$ is greater than zero then the expression within the square root can be evaluated and two solutions of x can be found. These are known as real solutions.

If $b^2 - 4*a*c$ is equal to zero then the square root part does not exist and so there is only one solution of x. This is also a real solution.

If $b^2 - 4*a*c$ is less than zero then the square root of the expression does not exist. Most readers would be content to say that no solutions exist - but do they? Well mathematicians, clever chaps, get around such problems by using a number notation called complex numbers. Complex numbers are written $x+iy$ where x is the real component, y is the imaginary component and i represents $\sqrt{-1}$. So returning to our problem, the solutions to our equation are complex with a real part equal to $-b/(2*a)$ and the imaginary part equal to $\sqrt{(4*a*c-b^2)/(2*a)}$ both of which are expressions which can be evaluated. Note that complex solutions cannot be evaluated directly with Amstrad BASIC and care must be taken not to use SQR on a negative number as an error will occur and stop execution. Fortunately for most of us, complex numbers are not used in everyday life.

Program 12: QUADRATIC EQUATIONS

```
10 CLS : PRINT "QUADRATIC EQUATIONS" : PRINT
20 INPUT "Enter first coefficient";a
```

```

30 INPUT "Enter second coefficient";b
40 INPUT "Enter third coefficient";c
50 PRINT
60 d = b*b-4*a*c
70 IF d<0 THEN 120
80 PRINT "Real roots"
90 PRINT "1. x = ";(-b+SQR(d))/(2*a)
100 PRINT "2. x = ";(-b-SQR(d))/(2*a)
110 GOTO 150
120 PRINT "Complex roots"
130 PRINT "real part = ";-b/(2*a)
140 PRINT "Imag part = ";SQR(-d)/(2*a)
150 PRINT : PRINT "Press any key to continue"
160 IF INKEY$ = "" THEN 160
170 GOTO 10

```

PRIME NUMBERS

A prime number is an integer that is only divisible exactly by 1 and itself. Program 13 prints out the first 100 prime numbers by assuming that the first two prime numbers are 2 and 3 and that all other prime numbers are odd.

The program tests to see if the odd number x is prime by dividing it by all the prime numbers less than \sqrt{x} that have already been found; if, in each case, the remainder is non-zero the number x is also prime. There is one point here that needs further explanation - why only divide by prime numbers previously obtained that are less than \sqrt{x} ? Non-prime numbers are not used since these would have at least two smaller prime numbers as factors. Numbers greater than \sqrt{x} are also not used since, if they were factors, then a factor less than \sqrt{x} must also exist.

As each prime number is obtained, it is stored in the array P(100) so that it can be used in testing future numbers. Because the Amstrad's arithmetic is only accurate to $9\frac{1}{2}$ significant figures, a number is assumed to be a factor if it has a remainder of less than $1E-10$.

Program 13: PRIME NUMBERS

```

10 CLS : PRINT "PRIME NUMBERS"
20 DIM p(100)
30 p(1)=2 : p(2)=3 : x=5 : n=3
40 PRINT "Prime number";1;TAB(18); "is" ; p(1)
50 PRINT "Prime number";2;TAB(18); "is" ; p(2)
60 FOR j = 2 TO n
70 IF x/p(j) - INT(x/p(j)) <1E-10 THEN 130
80 IF SQR(x) < p(j) THEN 100
90 NEXT j
100 p(n) = x
110 PRINT "Prime number";n;TAB(18); "is" ; p(n)
120 n = n+1
130 x = x+2
140 IF n<101 THEN 60

```

TRIGONOMETRIC FUNCTIONS

The Amstrad has four trigonometric functions available, **SIN** (sine), **COS** (cosine), **TAN** (tangent) and **ATN** (arctangent).

Trigonometric functions are all about angles and this is where we encounter our first problem. The majority of people measure angles in degrees; where a circle is divided into 360 degrees, a right angle is 90 degrees and a straight line is 180 degrees. Mathematicians, however, prefer to measure angles with a larger unit called the radian where one radian is the angle subtended by an arc of unit length on a circle of unit radius.

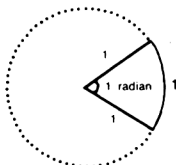


Figure 4.2

By definition there are two Pi radians in a circle where Pi is approximately 3.14159265358 and can be written as π . Thus 1 radian is $\frac{360}{2\pi}$ degrees which is about 57.3 degrees.

To convert radians into degrees, we multiply by $\frac{180}{\pi}$

e.g. $\frac{\pi}{3}$ radians = $\frac{\pi}{3} \star \frac{180}{\pi} = 60$ degrees

To convert degrees into radians multiply by $\frac{\pi}{180}$

e.g. 45 degrees = $45 \star \frac{\pi}{180} = \frac{\pi}{4}$ radians.

We can access π directly from the Amstrad by the function **PI**. We also have the option of selecting either radians or degrees with trigonometric operations by using the commands **DEG** and **RAD**; note that radians are used by default. Whilst the definitions of the trigonometric functions are relatively simple, an explanation of their use is not. A list of applications using them would be endless and so it is sufficient to say that they are invaluable; their importance to science and technology is probably equivalent to that of oxygen to life form.

We shall now look at the four functions. Consider the following right angled triangle with an angle of X radians and the sides labelled opposite, adjacent and hypotenuse.

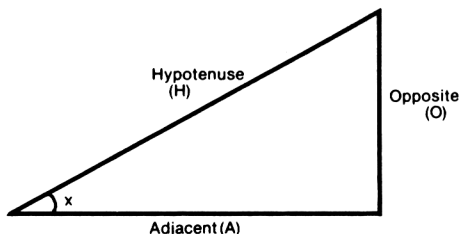


Figure 4.3

Sine, cosine and tangent are defined as the ratio of lengths of certain sides to othersides.

SIN
The sine of angle X is defined as the ratio of the length of the opposite side to the length of the hypotenuse in any right angled triangle with an angle of X radians.

$$\text{SIN}(X) = \frac{O}{H}$$

A graph of SIN(X) plotted against X would look like :

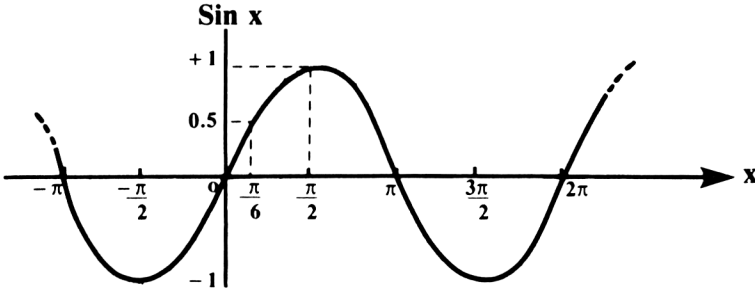


Figure 4.4: The Sine Wave

COS
The cosine of angle X is defined as the ratio of the length of the adjacent side to the length of the hypotenuse in any right angled triangle with an angle of X radians.

$$\text{COS}(X) = \frac{A}{H}$$

A graph of COS(X) plotted against X would look like:

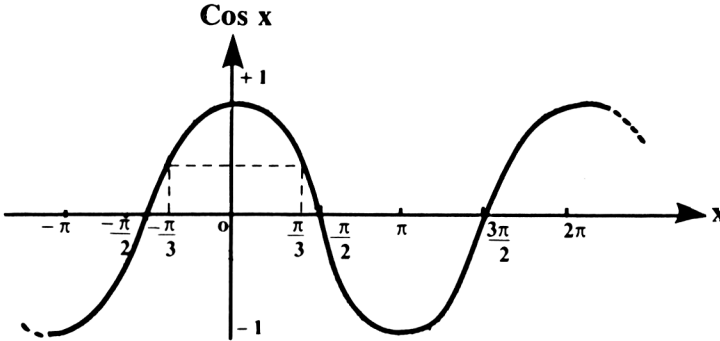


Figure 4.5: The Cosine Wave

TAN

The tangent of angle X is defined as the ratio of the length of the opposite side to the length of the adjacent side in any right angled triangle with an angle of X radians.

$$\text{TAN}(X) = \frac{O}{A}$$

A graph of TAN(X) plotted against X would look like:

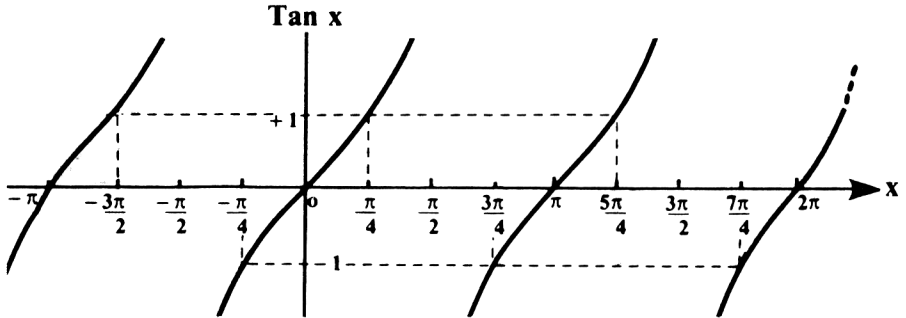


Figure 4.6: The Tangent Lines

The remaining trigonometric function is :-

ATN

The arctangent of y turns the angle whose tangent is y.

The arcsine and arccosine values, which are inverse sine and cosine functions respectively, can be found using the following formulae

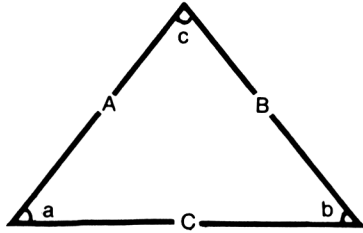
$$\text{Arcsine}(y) = \text{ATN} \left[\frac{y}{\text{SQR}(1-y^2)} \right]$$

$$\text{Arccosine}(y) = \text{ATN} \left[\frac{y}{\text{SQR}(1-y^2)} \right] + \frac{\pi}{2}$$

Note that in both cases y is always in the range $-1 \leq y \leq +1$

Whilst it is unfortunate that the Amstrad lacks the functions to obtain arcsine and arccosine values, the following program shows that their evaluation using ARCTAN is relatively simple.

Program 14 calculates the angles of triangles where the lengths of all three sides are known. Consider the triangle on the following page where lengths A, B and C are known.



The following relationship is known to exist (remember the commercial programmer!)

$$C^2 = A^2 + B^2 - 2 * A * B * \text{Cosine}(c)$$

Rearranging the equation mathematically gives: $c = \arccos \frac{(A^2 + B^2 - C^2)}{(2 * A * B)}$

Similar relationships exist for the remaining two angles. Program 14 allows the lengths of all three sides to be entered and then calculates the angles in degrees. Note that the sum of all three angles in a triangle is always 180°, thus we have a little test to illustrate the Amstrad's accuracy.

Program 14: TRIANGLE

```

10 CLS : PRINT "TRIANGLE" : PRINT
20 DEF FN t(x,y,z) = (y*y+z*z-x*x)/(2*y*z)
30 DEF FN a(x) = 90-ATN(x/SQR(1-x*x))
40 INPUT "Enter side one";a
50 INPUT "Enter side two";b
60 INPUT "Enter side three";c
70 PRINT : PRINT
80 DEG
90 x = FN t(a,b,c)
100 IF ABS(x)>1 THEN PRINT "Invalid!!" ; GOTO 230
110 aa = FN a(x)
120 x = FN t(b,a,c)
130 IF ABS(x)>1 THEN PRINT "Invalid!!" ; GOTO 230
140 bb = FN a(x)
150 x = FN t(c,a,b)
160 IF ABS(x)>1 THEN PRINT "Invalid!!" ; GOTO 230
170 cc = FN a(x)
180 PRINT "Angle opposite side one" ; ROUND
(aa,2)
190 PRINT "Angle opposite side two" ; ROUND
(bb,2)
200 PRINT "Angle opposite side three" ; ROUND
(cc,2)
210 PRINT
220 PRINT "Sum of angles is" ; aa + bb + cc

```

```

230 PRINT
240 PRINT "Press any key to continue"
250 IF INKEY$ = "" THEN 250
260 GOTO 10

```

RECURSIVE PROGRAMMING

We shall now take a brief look at a software technique called recursion which can often be useful especially for evaluating mathematical functions which involve a repetitive processing loop. In BASIC, a recursive subroutine is one that calls itself and can often result in neat and elegant solutions.

For example, consider the mathematical expression factorial n (or $n!$) which expands to:

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

where n is a positive integer (i.e. $6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$)

which is equivalent to

$$n! = \begin{cases} n * (n-1)! & n > 1 \\ 1 & n = 1 \end{cases}$$

Thus a subroutine to evaluate $n!$ could be written as n multiplied by the result of calling itself to evaluate $(n-1)!$ Care has to be taken to stop at $1!$ i.e. to stop the recursive calls.

At this stage it is necessary to see how the operating system keeps control of nested subroutine calls by using what is called the **GOSUB** stack. We shall meet stacks in more detail in chapter 7; until then it is sufficient to know that when a subroutine is called the line number of the calling **GOSUB** statement is stored until a **RETURN** statement is encountered; control then returns to the line following the most recent stored line number which is now removed from the stack. If more **RETURNS** are executed than calling **GOSUB** statements then the stack becomes empty and an error occurs.

Program 15: RECURSION

```

10 CLS
20 INPUT "Enter value";x
30 GOSUB 1000
40 PRINT : PRINT "Factorial value";f
50 END
1000 f = 1
1010 IF x>1 THEN f = f*x : x = x-1 : GOSUB 1010
1020 RETURN

```

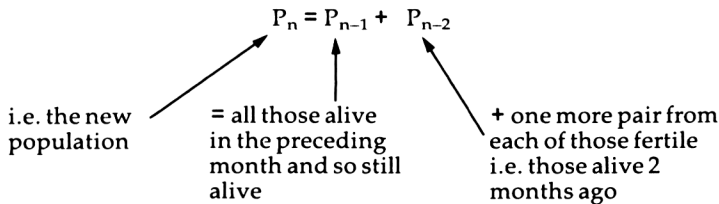

SIMULATION

Many processes or systems for example, chemical reactions, population models, queues, traffic flow, battles, electronic circuits, etc may be examined on a computer by simulating the items that are liable to change. What is required is for the programmer to build a mathematical model which finds relationships between the variables and takes into account the effect of time and external events. Simulations on computers have many advantages. Some processes can take a very long time to complete, a simulation can present the result in seconds. It is also a lot less expensive for a trainee pilot to crash a flight simulator than the real thing!

Mathematical models relying on simulations can involve extremely complicated models which are far beyond the scope of this book. However, let us examine a simple model that simulates a simple population model.

In the thirteenth century, an Italian mathematician Leonardo Fibonacci studied the vast explosion in the rabbit population. He considered how the population grew, starting with just one pair of rabbits and assumed that it took a pair of newborn rabbits one month before they became fertile and could reproduce. From then on they would breed an additional pair each month. It is assumed that no deaths or migration occurred in the timespan under consideration.

Analysing the problem it can be seen that if the population of month n is P_n then



It is also known that $P_1 = 1$ and $P_2 = 1$

The values of P_n are found from program 16, the sequence of numbers obtained are known as the Fibonacci sequence.

Program 16: RABBITS

```
10 CLS : PRINT "FIBONACCI SEQUENCE" : PRINT
20 p1 = 1 : p2 = 1 : m = 2
30 PRINT "Month 1 :";p1;"Rabbit"
40 PRINT "Month 2 :";p2;"Rabbit"
```

```
50 p = p1 + p2 : m = m + 1
60 PRINT "Month";m;": ";p;"Rabbits"
70 p1 = p2 : p2 = p
80 IF m<36 THEN 50
```

CHAPTER FIVE

THE AMSTRAD MEMORY MAP

Up to now we have regarded the Amstrad's memory as just a hardware component whose function it is to store all the data held in the ROM and RAM system described in chapter one. The best way of describing the internal structure of the computer's memory is to visualise a long sequence of storage boxes or cells each identified by a unique numerical label commencing with zero and incremented by one each time, just like lockers at a railway station. The label of each cell in the memory is called an address. In each box or cell, a number may be stored, which in the case of the Amstrad and most other personal computers, is restricted to the range of 0 to 255. In addition to numbers, it is possible to store characters and program instructions represented by numerical values; these values are known as character codes and instruction codes. When referring to a size of memory, the abbreviation K is frequently chosen to represent 1024 (or 2 raised to the power 10) cells of memory.

BINARY NUMBERS

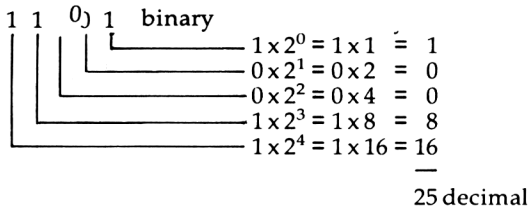
We shall now see the simple form in which the Amstrad stores its data. The nature of electronics restricts components to being either switched on or off and this means that the computer can only recognise two states; these are usually written for convenience as '0' or '1'. Just as words and sentences are built up by using more than one letter and numbers consist of several digits, so computer expressions are represented by sequences of '0's and '1's. Such patterns of '0's and '1's are called 'binary numbers'.

When numbers are written in the normal decimal or base 10 representation the digit furthest to the right gives the number of units, the digit to its left gives the number of tens (the base of our number system), the next digit to the left gives the number of hundreds (the base squared), and so on. Binary numbers use a base of 2; the digit furthest to the right gives the number of units, the digit to its left gives the number of twos (the base), the next digit gives the number of fours (the base squared), the following digit gives the number of eights (the base cubed) and so on.

Conventionally, the number 345 in the decimal numbering system that we use means:

3	4	5	decimal	
			$5 \times 10^0 = 5 \times 1 = 5$	= 5
			$4 \times 10^1 = 4 \times 10 = 40$	= 40
			$3 \times 10^2 = 3 \times 100 = 300$	= 300
				345 decimal

Likewise a binary number such as 11001 is equivalent to:



When counting, a decimal '1' is carried over into the next column, wherever a '1' is reached i.e. after 0 comes 1, after 1 comes 10, after 1 comes 11, after 11 comes 100, and so on. This sequence can be illustrated by the following routine which uses the string function **BIN\$** which converts a numerical argument in the range -32768 to 65535 to a string containing its equivalent binary form.

```

10 CLS
20 FOR x = 0 TO 255
30 PRINT "Decimal";x;TAB(13);"Binary ";BIN$(x)
40 NEXT x

```

It can be seen that the binary numbering system works on the same principle as decimal but since more digits are required to represent a number in binary than in decimal, it would be more cumbersome for us humans to use. If we want to use binary numbers on the Amstrad we must precede the digits with **&X** so as to distinguish them from decimal.

It is now possible to explain why a number stored at a particular address in the Amstrad's memory has to be restricted to the range 0 to 255. This is because a storage cell contains exactly eight binary digits; thus the range is from 0000 0000 (0 decimal) to 1111 1111 (255 decimal). When referring to binary numbers in computing, an individual digit is called a bit and a group of eight bits at an address is called a byte. The bits are often numbered 0 to 7 with bit 0 being the least significant bit and bit 7 the most significant bit.

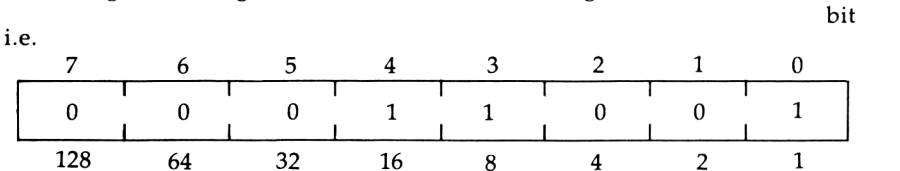
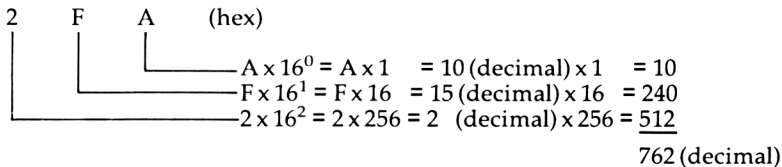


Figure 5.1

Numbers other than integers between 0 and 255 are stored by combining several bytes.

Since long strings of '0's and '1's are difficult to memorize and awkward to work with, some simpler notations have been developed. One such system uses a numbering system with a base of sixteen and is known as the hexadecimal system. This requires 16 digits but instead of designing six new

symbols, the first letters of the alphabet are used. The sixteen digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. It is important in this context not to interpret the new symbols as letters but as digits capable of the mathematical operations of addition, subtraction etc. As an example consider the hexadecimal number 2FA:

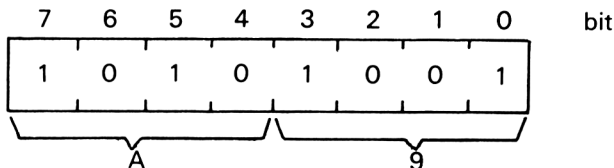


HEX\$ returns its argument in hexadecimal form

```
e.g.
10 CLS
20 FOR x = 0 TO 255
30 PRINT "Decimal";x;TAB(13);"Hex.";HEX$(x)
```

With a byte containing eight bits, its value may be expressed with just two hexadecimal digits; thus with the possible exception of decimal numbers the hexadecimal numbering system is often used in computing in preference to any other numbering system

e.g.



The Amstrad interprets numbers as hexadecimal if preceded by **&** e.g. **&AB**

THE MEMORY MAP

The Amstrad's memory is subdivided into several distinct sections each of which has a specific purpose. The visual structure of the memory is given in a diagram called a memory map, as shown in Figure 5.2 – see next page.

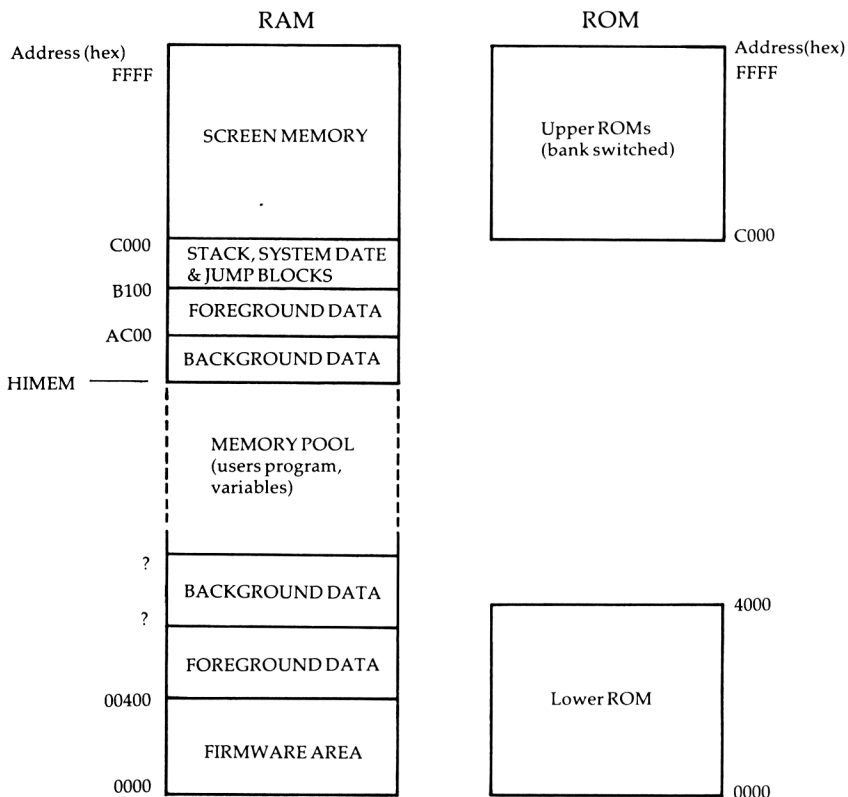


Figure 5.2 : AMSTRAD MEMORY LAYOUT

The memory map is complicated because although there are only 64K addressable locations available, the Amstrad system contains 64K of RAM and 32K of ROM, in fact there is provision for ROM expansion of up to 252*16K (nearly 4 Mega) bytes.

The standard 32K of ROM is split into two groups of 16K; the top 16K, addresses C000-FFF contains the BASIC interpreter ROM, and the bottom 16K, addresses 0000-3FFF contains the Amstrad's firmware which is software to control the hardware and routines which may be called by a user's machine code program. It can be seen from figure 5.1 that the BASIC interpreter overlays a section of RAM which contains the contents of the screen memory (i.e. the data displayed). An electronic component called a gate array switches in either ROM or RAM depending on the access requirement of the firmware. The display controller and C.P.U. are timed so that they never need to access these addresses simultaneously.

As stated previously, up to 252 of the 16K ROMS may be overlaid on the top 16K and are defined to be either foreground or background (max 7) ROMS.

A foreground ROM contains single programs (e.g. other languages, operating systems, business applications, games) executed one at a time whereas a background ROM contains routines (e.g. expansion peripheral routines) that may be called up.

At switch on, the firmware searches through these 'banked' ROMs one by one (or page by page) and executes a start-up on the first one encountered. On an unexpanded Amstrad, the first (and only) ROM present is BASIC which is obtained at switch on. BASIC lies in page 1; if any other program, such as a game on a cartridge, has to be obtained at switch on then it must lie in page 0. Any ROM present may be called by using a vertical bar 'I' followed by a ROM name. Thus we can call a start up on the BASIC interpreter ROM by the command **I BASIC**.

Fortunately, the Amstrad has an excellent BASIC which makes occasions for delving directly into the Amstrad's memory very rare. However machine code programmers will need to know the memory layout in detail and the position of the firmware's routines. Detailed information on this is available from Amstrad in their firmware manual. Since the position of these routines could well change if the firmware ROM is updated, calling these routines from your program would mean that it would only run on the original version. To overcome this problem, a jump block at locations BB00 to BD39 is used; a program calls a routine at a specific address in the jump block which in turn calls up the routine required. Future updates to the firmware would mean the contents of the jump block would have to be amended but a user's machine code program would be unaffected. It is also possible to amend the contents of the jump block to point to your own routines thus altering the effect of the firmware - note that all the registers and values returned must be compatible with the original system.

A BASIC program and its variables are stored in the memory pool shown in figure 5.2. The lowest address is set by the firmware when BASIC is entered. The highest address is located by a pointer called HIMEM and can be accessed directly by the BASIC function **HIMEM**. We can 'trick' the Amstrad into thinking it has less memory by changing HIMEM, using the command **MEMORY**, to point to a lower address and so releasing a section of memory that is safe from the system for such uses as machine code programs, graphics data etc.

e.g. to release 10 bytes from BASIC: **MEMORY HIMEM-10**

The function **FRE(0)** returns the number of bytes free to BASIC; try executing **PRINT FRE(0)** before and after the above **MEMORY** command to prove that 10 bytes were released.

We can write and read directly in to the Amstrad's RAM by using the two commands **POKE** and **PEEK** respectively.

i.e.

POKE A, V

will write the value V into the address A
(V is restricted to integers in the range 0 to 255)

V = PEEK(A)

will assign the contents at address A to variable V

Whilst many other micros require these functions to get below the surface of their machines, the equivalent control on the Amstrad can usually be obtained by using its powerful BASIC.

CHAPTER SIX

TIME, CLOCKS AND INTERRUPTS

Timing Facility

The Amstrad has an internal counter which comes into effect the moment the machine is switched on. The counter is incremented by 1 every $\frac{1}{300}$ th of a second and it is extremely accurate. We now have the facility to add timing to our programs and the counter can be used, for example, as a clock, stop-watch or for timing moves in games. We can also time the lengths of processes to compare the efficiency of different programming methods.

To access this counter we use the command TIME and to find a time length in seconds we need to know initial and final values of TIME and then divide the difference by 300. For example, program 17 illustrates the TIME function by testing the speed of your reactions. Note that it is necessary to subtract $\frac{1}{100}$ th of a second from the time evaluated to allow for processing time - try running the program with line 120 absent.

PROGRAM 17: REACTION

```
10 CLS
20 LOCATE 10,10 : PRINT "Press ENTER to start"
30 IF INKEY(18) = -1 THEN 30
40 CLS
50 FOR x = 0 TO 3000*RND
60 IF INKEY(47)<>-1 THEN CLS : PRINT "CHEAT!!" :
END
70 NEXT x
80 LOCATE 10,10 : PRINT "Press SPACE BAR!!"
90 BORDER 7,16
100 t1 = TIME
110 PRINT CHR$(7)
120 IF INKEY(47) = -1 THEN 120
130 t2 = TIME
140 BORDER 1
150 LOCATE 8,13
160 PRINT "Time taken" ; ROUND ((t2-t1-3)/
300,2);"seconds"
170 FOR x = 0 TO 3000 : NEXT x
180 GOTO 10
```

It is difficult to describe precisely what makes a computer game addictive, but including a 'highest score obtained' feature is likely to increase addiction!

This gives the players a tendency to keep having 'just one more go' to be top - and of course paying you their 10p pieces!!

Program 18 demonstrates such a facility with a simple arcade type game. The player has to dodge a missile attack by moving a little man to the left or right under the control of the cursor keys. The object of the game is to survive as long as possible and a record of the best time is kept.

PROGRAM 18: MISSILE ATTACK

```
10 DEFINT p : DIM m(25) : RANDOMIZE TIME
20 CLS : BORDER 2+RND*25
30 n = 20 : t = 100 : z1 = TIME
40 p = 40*RND+1
50 LOCATE p,25 : PEN 1
60 x = x+1 : IF x>25 THEN x = 1
70 m(x) = p : v = 0
80 PRINT CHR$(239) : PRINT
90 IF m((x+3)MOD 25) = n+1 THEN 170
100 IF INKEY(8) <>-1 AND n>1 THEN n = n-1 : v = 3
110 IF INKEY(1) <>-1 AND n<38 THEN n = n+1 : v =
2
120 LOCATE n,1 : PEN 2
130 PRINT SPACE$(1) + CHR$(248+v) + SPACE$(1)
140 FOR j = 1 TO t : NEXT j
150 IF t = 1 THEN t = t-1
160 IF m((x+2) MOD 25) <> n+1 THEN 40
170 z2 = ROUND ((TIMEz1)/300,2)
180 PRINT CHR$(7)
190 LOCATE n+1,1 : PRINT CHR$(238)
200 FOR j = 1 TO 300 : NEXT j
210 IF z2>h OR h = 0 THEN h = z2
220 CLS : PEN 2
230 LOCATE 14,10 : PRINT "Time";z2
240 LOCATE 14,12 : PRINT "Highest";h
250 LOCATE 14,15 : PRINT "Play again Y/N"
260 IF INKEY(43)<>-1 THEN ERASE m : GOTO 10
270 IF INKEY(46)<>-1 THEN CLS : BORDER 1 : END
280 GOTO 260
```

INTERRUPTS

The occurrence of interrupts is a common feature in computing. An interrupt occurs, when under certain conditions, the processor stops whatever it was doing, carries out another bit of processing, and then on completion, returns to complete the previous task. Interrupt handling is usually written in machine code; however the Amstrad has some unique BASIC commands that can handle time interrupts, that is after specified times a subroutine may be evoked and then when the routine is completed, control returns to the position in the program where the interrupt occurred. Two commands may be used, AFTER

and EVERY; the first sets up a time interrupt to occur after a specified time whilst the second command sets up time interrupts to occur regularly at a specified frequency period. There are four timers available for such interrupts and therefore, up to four time interrupt subroutines may be used. Each interrupt timer has a priority - timer 3 has the highest and timer 0 the lowest. If an interrupt occurs during the processing of another interrupt subroutine, the interrupt will occur instantly if the timer is of a higher priority, otherwise the interrupt will be queued until the first interrupt has been completed. When setting an interrupt one must state the time period to elapse in 1/50ths of a second followed by the interrupt timer number to be used.

e.g.
 AFTER 100,3 GOSUB 1000 - execute the subroutine at line 1000
 once every two seconds

EVERY 50 GOSUB 1000 - execute the subroutine at line 1000
 every second. Note that since no timer
 number is specified, 0 is used
 by default.

Care must be taken when using these commands since the subroutines can be executed at any point in the program.

The following example of time interrupts displays a clock in the top left hand corner which chimes every hour. Since the clock is processed in an interrupt subroutine it may be incorporated with your own programs. A window #1 has been set up to display the clock and so your program should avoid that stream.

PROGRAM 19: CLOCK SKELETON

```

10 WINDOW #1,1,9,1,1 : WINDOW #0,1,40,2,25
20 CLG : PRINT "Enter time:" : PRINT
30 INPUT "Enter hours";h
40 IF h<0 OR h>23 THEN 20
50 INPUT "Enter minutes";m
60 IF m<0 OR m>59 THEN 20
70 INPUT "Press ENTER on time signal";q$
80 CLS
90 EVERY 50 GOSUB 1000
100 'continuation of program
110 GOTO 110

1000 s = s+1
1010 IF s>59 THEN s = s MOD 60 : m = m+1
1020 IF m>59 THEN m = m MOD 60 : h = h+1 : PRINT
CHR$(7)
1030 IF h>23 THEN h = h MOD 24
1040 IF s MOD 2 = 0 THEN c$ = ":" ELSE c$ =
SPACE$(1)
1050 LOCATE #1,1,1

```

```

1060 PRINT #1, USING "##!";h;c$;
1070 PRINT #1, USING "##!";m;c$;
1080 PRINT #1, USING "##";s
1090 RETURN

```

There may be cases where it is important that no interrupt occurs - this may be done by disabling interrupts at the start with the command **DI** and then, when interrupts are allowed again, they can be enabled with the command **EI**. Finally, it is possible to disable a single timer with the function **REMAIN** which also returns the remaining time count.

BREAK INTERRUPT

The break interrupt has a higher priority than anything else. [BREAK] involves hitting the [ESC] key twice - pressing [ESC] once will temporarily halt execution until any other key is pressed. Whilst [BREAK] normally returns control to command mode, we have the option of directing execution to a 'BREAK' subroutine using **ON BREAK GOSUB**.

```

e.g. 10 CLS
20 ON BREAK GOSUB 1000
30 'continuation of program

```

```

1000 CLS
1010 PRINT "Do you wish to save results (y/n)"
1020 IF INKEY(46)<>-1 THEN END
1030 IF INKEY(43) = -1 THEN 1020
1040 'save results

```

A dangerous, or perhaps useful, trick is to put a single **RETURN** statement at the **BREAK** subroutine which means that the user cannot [BREAK] out of a program (except at an **INPUT** statement).

Break interrupts may be disabled by: **ON BREAK STOP**
The commands **DI** and **EI** do not affect **BREAK** interrupts.

ERROR TRAPS

Normally when an error occurs in our BASIC programs, execution ceases and control returns to command mode signalling that an error has occurred. However, the Amstrad has the powerful facility to set up error traps. This means that when an error occurs, instead of the program returning to command mode, an error section is undertaken which could either rectify the problem or give the user more information about what has happened.

To set up an error trap we use the statement **ON ERROR GOTO** followed by the line number to which control is to be transferred.

e.g. **ON ERROR GOTO 1000.**

Similarly, the error trap may be disabled by specifying a line number of 0
e.g. `ON ERROR GOTO 0`

Two functions which give additional information on the error are `ERL` which returns the line number of the statement where the error occurred and `ERR` the error number (see appendix D).

e.g. `PRINT "Error";ERR;"occurred on line";ERL`

At this stage the reader is reminded of how we set up a window in chapter 3 for user error messages in which, when the line was displayed at the base of the screen, a beep was sounded and the user had to press any key to continue - an ideal use for error traps.

Having entered an error trap we may use the command `RESUME` to continue program execution at either the line where the error occurred, the following line after the error line or at a specified line.

e.g.
`RESUME` continue at line ERL
`RESUME NEXT` continue at line following ERL
`RESUME 500` continue at line 500

Finally, it is possible to program user errors with the command `ERROR` followed by an error number which need not be in the range of BASIC error numbers. Whilst program errors are picked up by the BASIC interpreter; invalid data, which has to be trapped by the program, can now be handled in a similar fashion.

e.g.
`100 INPUT "Enter account number";an$`
`110 IF LEN(an$)<>7 THEN ERROR 100`

CHAPTER SEVEN

DATA STRUCTURES

We shall now take a look at the advanced topic of data structures which is a means of manipulating collections of similar data efficiently. To begin with we shall refresh our memories on the subject of arrays.

ARRAYS - STATIC DATA STRUCTURES

We often have a requirement to reserve a block of cells in the Amstrad's memory that can be referred to by a single name (referring to the block) and a number (referring to the particular cell in the block); such a structure is called an array. Control is instructed to reserve blocks of a particular length by the DIMension statement.

e.g. `DIM x (15)` instructs the computer to reserve an array called 'x' consisting of sixteen elements

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----

Fig. 7.1

Each cell can store one number. When the block is declared, each cell is initialised to zero. We may refer to any cell by the variable name followed by the cell number in parenthesis. For example, cell 5 is referred to as `x(5)`. The term in brackets is called the subscript. The same cell could be referred to with a variable subscript for example `x(a)` if the variable `a` had been set to a value of 5. Subscripts may also be variable expressions but their results must lie in the range specified by the DIM statement. If an array is referred without ever being DIMensioned, it is assumed to have a length of eleven elements.

It is also possible to set up arrays with more than one subscript
e.g. `DIM y(3,4)`

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

y

Fig. 7.2

Such a structure is called a two dimensional array; remember however that the cells are physically stored sequentially in the computer's memory. The same idea works for an `n` dimensional array. If the array variable is followed by either a % or \$ sign, then, as usual, the variables are either integers or strings respectively.

Program 20 demonstrates arrays by simulating a knock-out football competition consisting of six rounds with 64 competing clubs. Each club is stored in a DATA statement along with a value representing the quality of the team. If your favourite club is absent from the list then exchange it with one that is present. An array r(7,64) is used to store the team numbers of the clubs that reach each round. Matches are drawn at random; when a club has been selected its team number in the array is negated so that it is not selected again. The number of goals scored is dependent on the value of the team and home advantage. If a match is drawn then a replay takes place at the previous visitors' home ground - the semi finals and final, of course, are played on neutral grounds. Have fun!!

PROGRAM 20: F.A. CUP

```

10 RANDOMIZE TIME
20 CLS : PEN 1
30 PRINT "F.A. CUP" : PRINT "=====" : PRINT
40 PRINT "Will your football team win the F.A.
   Cup" : PRINT
50 PRINT "Find out by watching the
   Grandstand","teleprinter": PRINT
60 PRINT "The semi-finals and final match
   are","played on neutral grounds" : PRINT
70 PRINT "Press ENTER to continue"
80 IF INKEY(18) = -1 THEN 80
90 CLS : PEN 2
100 CLEAR
110 DIM t$(64),r(7,64),g(64)
120 FOR x = 1 TO 64
130 READ t$(x),g(x)
140 r(1,x) = x
150 NEXT x
160 FOR y = 1 TO 6
170 PEN 1 : PRINT "Round";y
180 IF y = 5 THEN PRINT "Semi-"; : IF y>5 THEN
   PRINT "Final"
190 PEN 2
200 n = 2^(7-y)
210 FOR x = 1 TO n/2
220 d1 = INT(1+RND*n)
230 IF r(y,d1)<0 THEN 220
240 r(y,d1) = -r(y,d1)
250 d2 = INT(1+RND*n)
260 IF r(y,d2)<0 THEN 250
270 r(y,d2) = -r(y,d2)
280 d1h = INT(RND*g(d1)*RND*(3+(y>4)))
290 d2a = INT(RND*g(d2)*RND*2)
300 PRINT t$(-r(y,d1));TAB(13);d1h:TAB(18);t$(-
   r(y,d2)); TAB(30);d2a
310 IF d1h<>d2a THEN 350
320 PEN 3 : PRINT "Replay" : PEN 2
330 t = d1 : d1 = d2 : d2 = t

```



```

340 GOTO 290
350 IF d1h>d2a THEN r(y+1,x) = -r(y,d1)
360 IF d1h<d2a THEN r(y+1,x) = -r(y,d2)
370 PRINT
380 FOR z = 1 TO 500 : NEXT z
390 NEXT x
400 NEXT y
410 PEN 1
420 PRINT : PRINT "CHAMPIONS";t$(r(7,1)) : PRINT
430 PRINT "PRESS SPACE BAR to play again"
440 IF INKEY(47) = -1 THEN 440
450 GOTO 10
460 DATA "Torquay",2, "Reading",3, "Blackburn
R",4, "Norwich City",4
470 DATA "Portsmouth",3, "Barnsley",4, "High
Wycombe",2, "Bolton Wand",3
480 DATA "Fulham",3, "Shrewsbury",3, "Charlton
A",4, "Brighton",4
490 DATA "Leicester C",4, "Southampton",5, "Crewe
Alex",2, "Aldershot",3
500 DATA "Sheffield U",3, "Stockport C",2,
"Swansea C",3, "Wigan Ath",2
510 DATA "Orient",4, "C Palace",4, "Oxford U",3,
"Carlisle",3
520 DATA "Grimsby T",4, "Oldham Ath",4,
"Hereford",2, "Bristol Rov",3
530 DATA "Darlington",3, "Doncaster R",3,
"Cardiff City",4, "Wrexham",3
540 DATA "Sunderland",4, "Tottenham H",5, "Aston
Villa",5, "Blackpool",3
550 DATA "West Brom A",5, "Birmingham C",4,
"Cambridge U",3, "Manchester U",5
560 DATA "Everton",4, "Coventry",5, "Derby Co",3,
"Gillingham",3
570 DATA "Halifax T",2, "Wolves",3, "Notts F",5,
"Notts Co",4
580 DATA "Arsenal",4, "Leeds Utd",2, "Bristol
City",2, "Watford",6
590 DATA "Liverpool",6, "Chelsea",4, "West
Ham",4, "Q.P.R.",4
600 DATA "Newcastle U",4, "Ipswich T",4,
"Brentford",3, "Sheffield W",4
610 DATA "Barnet",2, "Luton Town",3, "Manchester
C",4, "Stoke City",4

```

There are, however, occasions when we require a more flexible system of ordering and so we turn to several different methods which can be covered by the general title '*dynamic data structures*'. While many of these techniques can be handled more suitably by other languages - for example ALGOL or Pascal - they can all be undertaken with Amstrad BASIC.

DYNAMIC DATA STRUCTURES

As an example of what dynamic data structures are all about, consider the following table which contains a list of customers orders with a car dealer. All the records are kept in the order of the customer's surname as in Fig. 7.3 which enables the information referring to a particular customer to be found easily. Suppose further that a record for the customer HARRISON has to be added to the list.

NAME	INITIAL	MODEL	COLOUR	DATE DUE
BANKS	M	240 DL	Green	04.5.83
BEECH	G	760 GLE	Silver	30.6.83
GRAHAM	R	360 GLT	Red	25.5.83
HART	W	240 GL	White	25.5.83
HOPTON	J	360 GLS	Blue	17.6.83
TROWSDALE	D	260 GLE	Black	19.6.83
TURNER	M	340 DL	Red	17.6.83
HARRISON	M	340 GL	Silver	30.6.83

Fig. 7.3

So that the table remains in order, the new record must be placed between HART and HOPTON. If these records were kept using normal office index cards any new record could be added by slotting the new card in at the appropriate position. Likewise, when the customer's order had been completed the corresponding record could be deleted from the system simply by removing the card

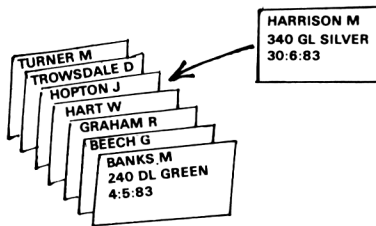


Fig. 7.4

Our problem is to simulate such a system where we can add and delete records and, by re-using any vacated space, we keep the occupied memory down to a minimum. One data structure that can be used for this application is called a *linked list*. When using linked lists for small quantities of data the advantages of flexibility are offset by the disadvantages of complexity. However, it is useful to learn the techniques since general and more complicated structures can then be handled in a similar fashion.

THE FORWARD LINKED LIST

Unlike static data structures, the ordering of a linked list is contained alongside the data. Each element in the structure will contain two pieces of information; the first item will be the data itself and the second item will be a pointer which relates the element with another element. The data could contain several fields, e.g. name, address, telephone number, etc; but for simplicity, our explanations will consider just a single data field.

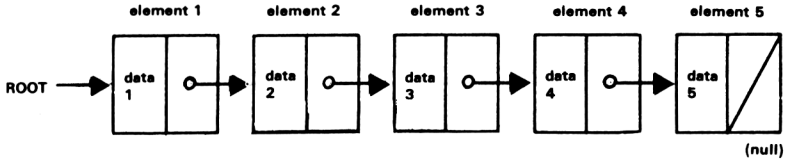


Fig. 7.5

Since element 1 is linked to element 2 the pointer in element 1 will have the value 2; similarly, pointers 2, 3 and 4 will have values 3, 4 and 5 respectively. As element 5 is the last element in the list its pointer is 'null' and so this could be indicated by a negative value, say -1.

Another method would involve setting the pointer to the address in the memory where the element is situated. This is a form of "indirect addressing" that we need not discuss further.

The structure shown in Fig. 7.5 could be stored on the Amstrad by using a two dimensional array where x(e,1)$ would contain the data of element e and x(e,2)$ would contain the pointer to the next element. A variable ROOT can be used to point to the first element i.e.

```
ROOT = 1
x$(1,1) = "data 1"      x$(1,2) = "2"
x$(2,1) = "data 2"      x$(2,2) = "3"
x$(3,1) = "data 3"      x$(3,2) = "4"
x$(4,1) = "data 4"      x$(4,2) = "5"
x$(5,1) = "data 5"      x$(5,2) = "-1"
```

To retrieve the data we access all the elements in the list; this is done by starting from ROOT and then moving along the structure, accessing each item of data, until the null pointer is reached.

An item can easily be deleted from a linked list by amending the pointer from the previous element so that it points instead to the next item. The pointer of the element that has been deleted should then be changed to a value to signal that it is empty - the only point of this is so that we can re-use that element in the array.

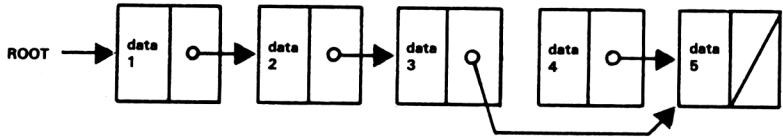


Fig. 7.6

If instead we want to insert an item into our list there are two stages to undertake; firstly, we must find where in the list the item should be placed, and secondly, amend the pointers to include the new item. The position where the new item is placed depends on what the user wants; it could be simply to put the new item immediately after the last item or perhaps, if the list is ordered, in the correct position as in our example below:

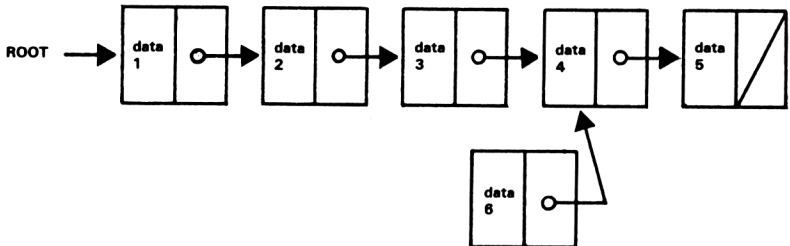


Fig. 7.7

First the pointer from element 6 should be changed to point to element 4 and then the pointer of element 3 should be made to point at element 6:

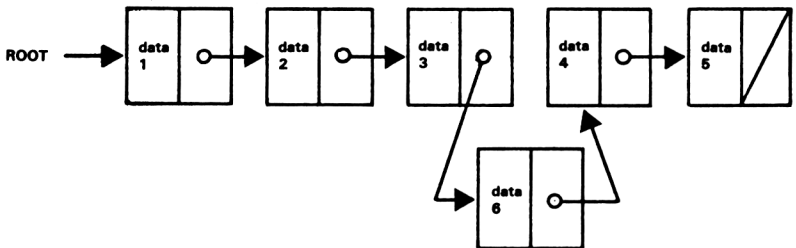


Fig.7.8

i.e.
 $x\$(6,1) = \text{"data 6"}$
 $x\$(6,2) = \text{"4"}$
 $x\$(3,2) = \text{"6"}$

We shall now see some routines that can handle linked lists; note that special considerations have to be made when the data is to be placed at the beginning or end of the list.

PROGRAM 21: LINKED LIST

```

10 CLS
20 null = -1
30 n = 100 : DIM x$(n,2) : GOSUB 1000
40 CLS
50 LOCATE 10,10 : PRINT "LINKED LISTS"
60 LOCATE 10,12 : PRINT "F1...Initialise"
70 LOCATE 10,13 : PRINT "F2...List"
80 LOCATE 10,14 : PRINT "F3...Delete Item"
90 LOCATE 10,15 : PRINT "F4...Add Item"
100 k$ = INKEY$ : IF k$ = "" THEN 100
110 k = ASC(k$)-48
120 IF k<1 OR k>4 THEN 100
130 ON k GOSUB 1000, 2000, 3000, 4000
140 GOTO 40

1000 root = null
1010 FOR x = 1 TO n
1020 x$(x,2) = ""
1030 NEXT x
1040 LOCATE 10,17 : PRINT "Initialisation
Complete"
1050 FOR j = 1 TO 500 : NEXT j
1060 RETURN

2000 CLS
2010 p = root
2020 WHILE p<>null
2030 PRINT x$(p,1)
2040 p = VAL(x$(p,2))
2050 WEND
2060 PRINT : PRINT "End of list"
2070 IF INKEY$ = "" THEN 2070
2080 RETURN

3000 CLS
3010 LOCATE 1,10
3020 PRINT "Enter data item to be deleted" :
INPUT d$
3030 IF d$ = "" THEN 3000
3040 IF root = null THEN 3120
3050 p = root
3060 IF x$(p,1) = d$ THEN root = VAL(x$(p,2)) :
x$(p,2) = "" : GOTO 3100
3070 pp = p : p = VAL(x$(p,2)) : IF p = null THEN
3120
3080 IF x$(p,1)<>d$ THEN 3070
3090 x$(pp,2) = x$(p,2) : x$(p,2) = ""
3100 PRINT "Item deleted"
3110 GOTO 3130
3120 PRINT "Item not in list"

```

```

3130 FOR j = 1 TO 500 : NEXT j
3140 RETURN

4000 CLS
4010 LOCATE 1,10
4020 PRINT "Enter data item to be inserted" :
INPUT d$
4030 FOR x = 1 TO n
4040 IF x$(x,2) = "" THEN p = x : GOTO 4070
4050 NEXT x
4060 STOP
4070 x$(p,1) = d$
4080 IF root = null THEN x$(p,2) = STR$(root) :
root = p : GOTO 4150
4090 IF d$ < x$(root,1) THEN x$(p,2) = STR$(root) :
root = p : GOTO 4150
4100 q = root
4110 qq = q : q = VAL(x$(q,2))
4120 IF q = null THEN x$(qq,2) = STR$(p) :
x$(p,2) = STR$(null) : GOTO 4150
4130 IF x$(p,1) > x$(q,1) THEN 4110
4140 x$(qq,2) = STR$(p) : x$(p,2) = STR$(q)
4150 PRINT "Inserted"
4160 FOR j = 1 TO 500 : NEXT j
4170 RETURN

```

MORE ADVANCED LISTS

Although most applications can be handled by forward linked lists there are a couple of amendments that can be made which enhance the power of the structures we have seen.

CIRCULAR LIST

In a *circular list* or *ring* the last element is made to point back to the first element. The main advantage of such a structure is that an item which precedes an element identified, can still be accessed without having to restart at the ROOT.

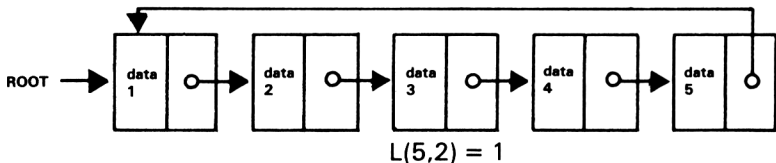


Fig. 7.9

DOUBLE LINKED LISTS

Even greater power can be added to linked lists by including a backward pointer which links up an element with its predecessor in the structure.

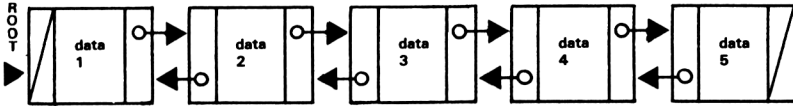


Fig. 7.10

This enables the structure to be searched in either direction. By using more than one pointer it is possible to order the list in more than one type of order.

In the small applications which we are likely to meet, the advantage of such a structure will be offset by an increase in the memory required for additional pointer storage.

STACKS AND QUEUES

There are two very useful linear data structures that are commonly used in computing called *stacks* and *queues*.

A stack is a method of storing and retrieving data in the computer with the basic principle that the most recent item of data stored will be the first retrieved. Storing information is known as *pushing* onto the stack and retrieving information is known as *poping* the stack. In our example, the top of the structure is indicated by a pointer called TOP.

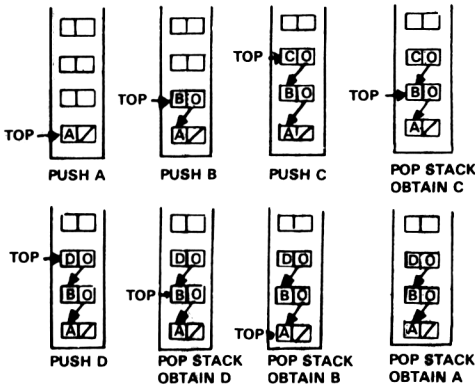


Fig. 7.11

Although it is never obvious to the programmer, the operating system uses a stack structure when dealing with subroutines. When control executes `GOSUB`, the line and statement position numbers are pushed onto a stack and then control continues from the line specified in the statement. When control executes `RETURN`, it continues from the position that is popped from the stack. This technique enables subroutines to be nested several levels deep although careful management is required by the programmer. If a `RETURN` is executed without

a corresponding **GOSUB** statement an empty stack will be present and, when popped, an error occurs.

A queue structure differs in its rule that the earliest item stored will be the first retrieved. Two pointers are required, **HEAD** and **TAIL** which point in the structure to the first and last items respectively.

A queue structure is used with the keyboard buffer where keys are stored as they are pressed. Since it is a queue, the earliest keys pressed are accessed before the more recent keys. Any released space is then available to store future key presses.

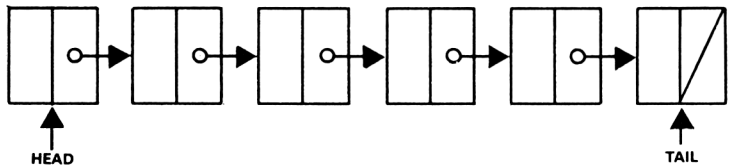


Fig. 7.12

When an item is retrieved, **HEAD** is made to point to the next element in the structure, i.e. to the element that is pointed to by the pointer of the element that is being retrieved.

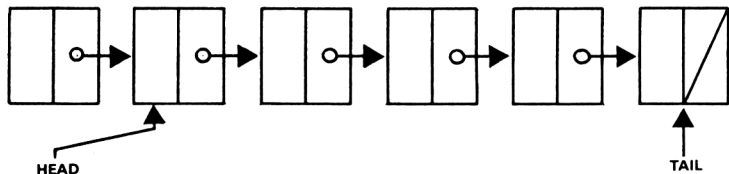


Fig. 7.13

When an item is added both **TAIL** and the last element are made to point at the new item.

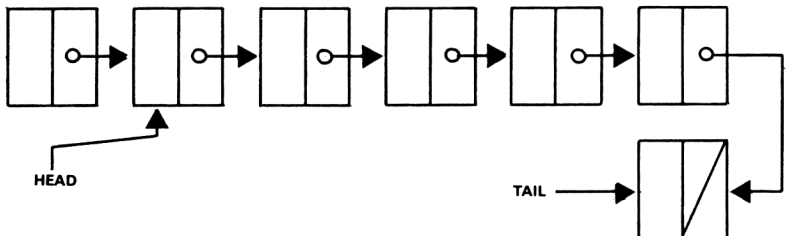


Fig. 7.14

The main problem is that a queue gradually drifts through the memory as retrievals and additions occur. One solution is to use a circular list so that, if all the memory at the end of a structure runs out, then items can be added at the beginning. If **TAIL** ever reaches **HEAD** then we have run out of space.

GRAPHS

Although the concepts of the linked lists are very useful as an insight into data structures, their use is severely limited as they can only function in one dimension, either forwards or backwards. In order to utilize these techniques with any practical ideas we often need to handle data structures in more than one dimension; such structures are called *graphs*.

Consider for example the following which represents the air routes that a certain air company undertakes.

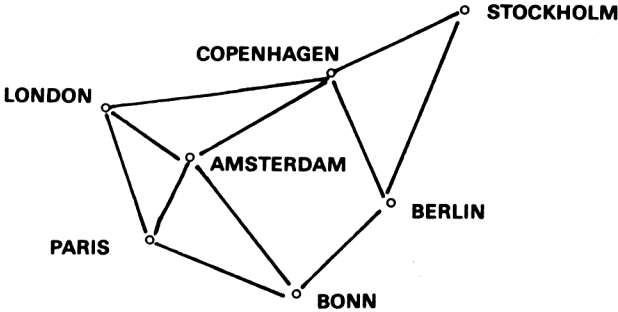


Fig. 7.15

This could be represented by the following structure.

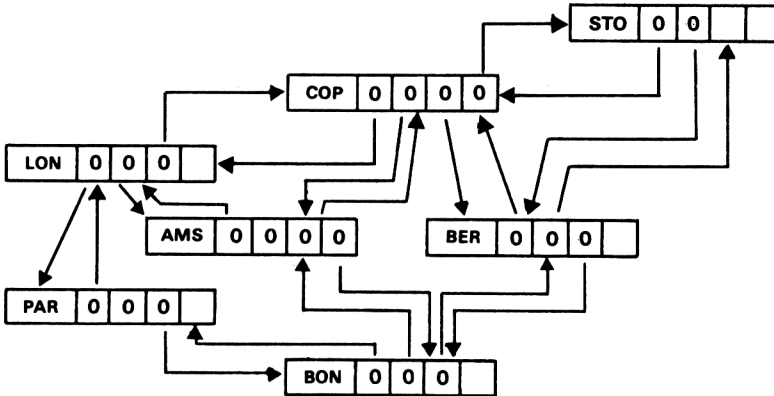


Fig. 7.16

Additional data items could be included which, for example, could give the cost, distance, etc. between two cities.

When referring to graphs; the intersecting points are called *nodes* and the links that join the nodes are called *edges*. Edges can be either directed or undirected and may or may not contain a value (for example, in our above

example cost, distance etc). Nodes can represent many things; such as towns or positions, production output, steps in a process, etc.

Graphs have many applications - as an example we shall take a look at the problem of finding the shortest path between two nodes, e.g. referring to Fig. 7.15, what is the shortest route from London to Berlin?

The algorithm that we shall use calculates the shortest paths between all pairs of nodes in the given graph and can also print out the optimum route.

The graph of n nodes may be represented in the computer by a $n \times n$ array (or matrix), say $x(n,n)$ where element $x(s,d)$ would be the distance from a source town s to a destination town d .

e.g.

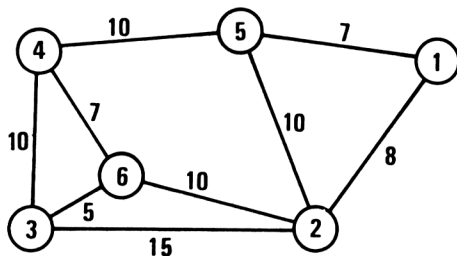
$$\begin{matrix}
 & \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \\
 \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} & \begin{pmatrix} 0 & 8 & \infty & \infty & 7 & \infty \\ 8 & 0 & 15 & \infty & 10 & 10 \\ \infty & 15 & 0 & 10 & \infty & 5 \\ \infty & \infty & 10 & 0 & 10 & 7 \\ 7 & 10 & \infty & 10 & 0 & \infty \\ \infty & 10 & 5 & 7 & \infty & 0 \end{pmatrix}
 \end{matrix}$$


Fig.7.17

Any two towns that are not directly connected take a distance value of infinity - on the Amstrad we have to make do with a very large value, say $1E+20$.

The procedure for calculating the shortest routes is:

The distance of each route is considered separately with the distance of the same route but going through each other town (if possible). e.g. Distance $1 \rightarrow 6$ is compared to the distances $1 \rightarrow 2 \rightarrow 6$, $1 \rightarrow 3 \rightarrow 6$, $1 \rightarrow 4 \rightarrow 6$ and $1 \rightarrow 5 \rightarrow 6$, and the shortest value is placed into our array at element $x(1,6)$.

A separate array is used to record the optimal route.

Program 22 will allow you to enter a graph, edge by edge by specifying a source town number, a destination town number and a distance. When you have entered all the information, you simply input a set of invalid data, such as 0,0,0. The program will then work out the shortest distances between each pair of towns. Then by specifying two town numbers the program will display the shortest route and its distance. Press the [SPACE BAR] to enter another pair or key 's' to start again.

PROGRAM 22: SHORTEST ROUTES

```

10 CLS : CLEAR
20 ON ERROR GOTO 10
30 INPUT "Enter maximum number of towns";n
40 DIM x(n,n),y(n,n)
50 CLS

```

```

60 FOR p = 1 TO n : FOR q = 1 TO n
70 IF p<>q THEN x(p,q) = 1E+20
80 NEXT q : NEXT p
90 PRINT "Enter town s, town d, distance"
100 INPUT s,d,v
110 IF s = d OR s<1 OR s>n OR d<1 OR d>n THEN 130
120 x(s,d) = v : x(d,s) = v : y(d,s) = s : y(s,d)
= d : GOTO 100
130 PRINT "Please wait"
140 FOR p = 1 TO n
150 FOR q = 1 TO n
160 FOR r = 1 TO n
170 d = x(q,p) + x(p,r)
180 IF x(q,r) <= d THEN 200
190 x(q,r) = d : y(q,r) = y(q,p)
200 NEXT r : NEXT q : NEXT p
210 CLS
220 PRINT "Lowest cost between two towns" : PRINT
230 PRINT "Enter two towns"
240 INPUT s,d
250 IF s<1 OR s>n OR d<1 OR d>n THEN 240
260 IF x(s,d) = 1E+20 OR x(s,d) = 0 THEN PRINT
"Not connected" : GOTO 380
270 PRINT
280 PRINT "Cost from";s; "to";d; "is"; x(s,d)
290 PRINT : PRINT "Via Towns" : PRINT
300 IF y(s,d)<>d THEN 340
310 PRINT "Direct"
320 GOTO 380
330 PRINT "Passing through";
340 IF y(s,d) = d THEN 380
350 PRINT y(s,d);
360 s = y(s,d)
370 GOTO 340
380 PRINT : PRINT : PRINT "Press SPACE BAR to
continue"
390 k$ = INKEY$
400 IF k$ = "s" OR k$ = "S" THEN 10
410 IF k$ = SPACE$(1) THEN 210
420 GOTO 390

```

TREES

There is one special type of graph commonly used in computing called a tree. A tree is a graph that contains no isolated nodes and no cycles, that is, there is one and only one route for getting from one node to another. We are usually concerned with trees in which all nodes are directed away from one specific node called the ROOT; every node except for the root has exactly one edge entering it.

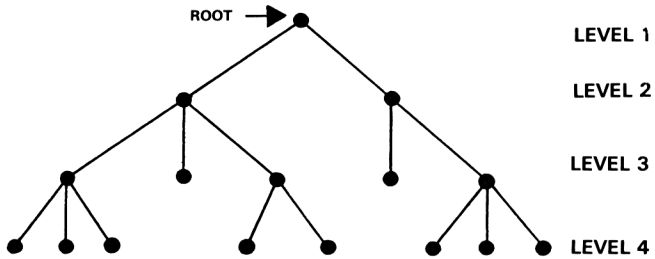


Fig. 7.18

The number of nodes away from the ROOT is often called the level where the ROOT is situated at level 1. The depth of a tree is its maximum level. Often a preceding node is called a parent node and a descending node is called a sibling.

In order to keep things simple we shall restrict ourselves to binary trees: these have a maximum of two edges leading from each node.

Binary trees may be stored on the Amstrad using arrays in a similar fashion to linked lists except that three fields are required; one is used to contain the data and the remaining two to contain the pointers.

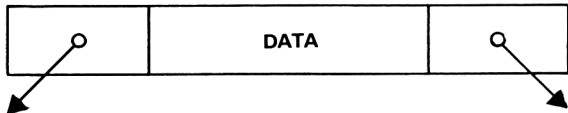


Fig. 7.19

As with linked lists we require the data to be set up in some logical order. For example, let us see how we would store the following sequence of car names in alphabetical order in a binary tree.

MERCEDES, FERRARI, PORSCHE, LOTUS, VOLVO, BMW, SAAB.

We commence with the first name, MERCEDES, as the root of our tree, and at this stage the node has no descendants.

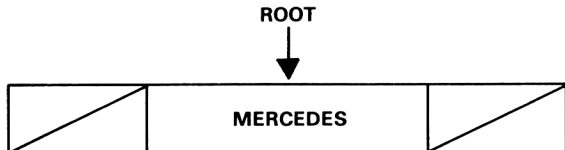


Fig. 7.20

Next we introduce FERRARI and since it precedes MERCEDES alphabetically it will become a descendant on the left.

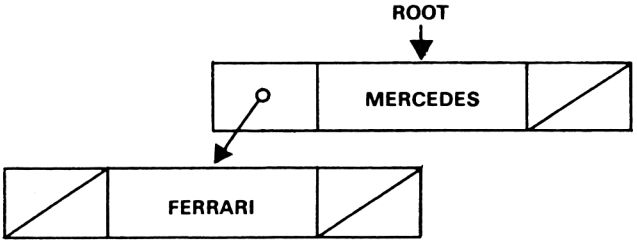


Fig. 7.21

The next item, PORSCHE follows MERCEDES and so becomes a right descendant of it.

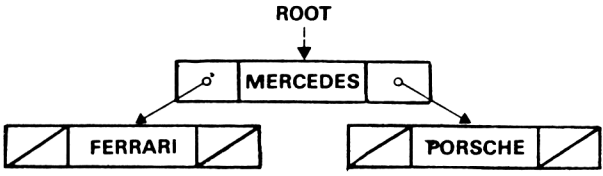


Fig. 7.22

After examining all the information and arranging it alphabetically the tree will resemble

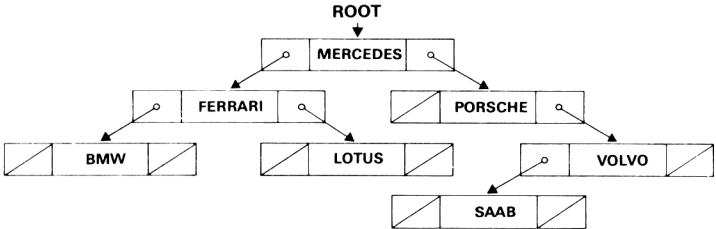


Fig. 7.23

With Amstrad BASIC we could use the arrays n\$ to store the data items and l and r to store the left and right pointers. Other data items such as performance details about the cars can be added to the structure by storing them in arrays and using the corresponding subscripts.

1	n\$(1) = "MERCEDES	l(1) = 2	r(1) = 3
2	n\$(2) = "FERRARI	l(2) = 6	r(2) = 4
3	n\$(3) = "PORSCHE	l(3) = -1	r(3) = 5
4	n\$(4) = "LOTUS	l(4) = -1	r(4) = -1
5	n\$(5) = "VOLVO	l(5) = 7	r(5) = -1
6	n\$(6) = "BMW	l(6) = -1	r(6) = -1
7	n\$(7) = "SAAB	l(7) = -1	r(6) = -1

The next obvious task is to be able to select a name and to request all the data that is associated with it. Having entered a name, the program searches down the tree starting at the ROOT and then travelling either left or right depending on the alphabetical order of the entered name and the one stored at the present node. If the null pointer is reached before the entered name is located then the name is not present in the tree. This method of searching is a form of what is called a binary chop and can be very fast, even with vast quantities of data. For example, by making seven comparisons we can get down to level 8 of the tree which could mean that up to 255 elements have been searched through.

To output all the data in the tree in the order set up, we require a systematic method of travelling to each node once and in the order of the leftmost nodes before those to their right.

If we consider any node in the binary tree, providing its pointers are not null, it is linked to two binary subtrees. Similarly, all the nodes in each of these subtrees are joined to further subtrees. Thus a binary subtree may be abbreviated to

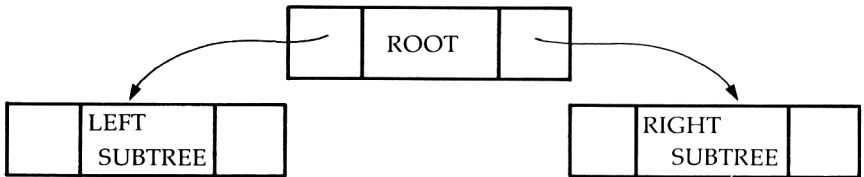


Fig. 7.24

If we travel to each of these three components in a fixed order it is called a *traversal*.

There are three traversals that we can make.

- 1) Left Subtree
- 2) Node
- 3) Right Subtree

(LNR)

- 1) Left Subtree
- or 2) Right Subtree
- 3) Node

(LRN)

- 1) Node
- or 2) Left Subtree
- 3) Right Subtree

(NLR)

Further traversals using the same fixed order should be made at each subtree. If we use the order LNR we can obtain our data in the order that the structure was set up with.

Consider the following tree under a LNR traversal

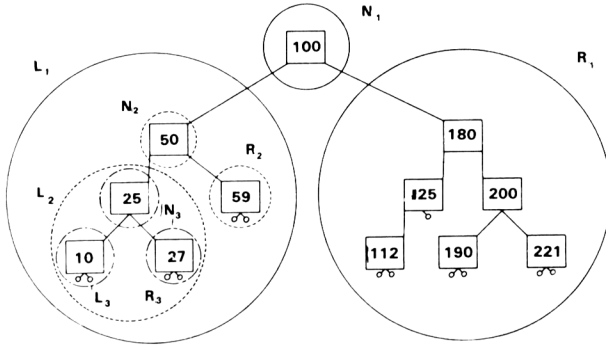


Fig. 7.25

Thus our order would be $L_1, 100, R_1$
 Subtree L_1 can be traversed LNR to give $L_2, 50, 59$
 and L_2 can be traversed LNR to give 10, 25, 27.
 Thus our order so far is 10, 25, 27, 50, 59, 100, R_1 .
 When R_1 has traversed LNR will have accessed the whole tree in order.

In chapter 8, when we study ‘methods of sorting data’ we shall see how to sort a sequence of numbers by creating a binary tree and then undertaking an LNR traversal - you may like to have a quick browse through chapter 8 now while the topic is fresh in your mind.

HEURISTIC PROGRAMMING

By now you should have some ideas of the flexibility that dynamic data structures can provide. Trees are the most common form found in computing and are used in a varied range of unusual subjects. Many games can be studied using trees, where different branches provide the various options available; by using some optimizing algorithms a computer can examine all the possible states that could occur for any feasible move it next makes.

A similar idea is used when a program learns from previous operations and remembers what results occur for certain moves by building onto the tree. The program commences with a tree consisting of a single node, the ROOT, and at that stage knows nothing; with time the tree will have expanded and soon can become “intelligent”. Programs that learn with experience are called *heuristic*.

Heuristic programming is demonstrated in program 23 which starts off with a small number of animals and facts in its memory. It asks you to think of an animal and then it tries to guess your animal by asking a series of questions to which you reply ‘yes’ or ‘no’. If the computer is unable to guess the animal it will ask you for a question and answer that it can use in the future.

The program builds up a binary tree with animals stored at the bottom nodes in the tree and questions in the other nodes. By asking questions and analysing

the answers the program can decide on which path to travel down the tree. When an animal is reached the computer asks if it is correct; if not, it asks for a question that distinguishes between the animal in the tree and the animal chosen.

PROGRAM23: ANIMALS

```

10 GOSUB 380
20 CLS : PRINT "ANIMALS" : PRINT
30 PRINT "Think of an animal" : PRINT
40 FOR j = 1 TO 2500 : NEXT j
50 PRINT "Answer Y/N to the following questions"
: PRINT
60 p = 1
70 IF l(p) = 0 AND r(p) = 0 THEN 140
80 FOR j = 1 TO 1000 : NEXT j
90 PRINT q$(p); " ? ";
100 k$ = INKEY$ : IF k$ = "" THEN 100 ELSE k$ =
LOWER$(k$)
110 IF k$ = "n" THEN PRINT "n" : p = r(p) : GOTO
70
120 IF k$ = "y" THEN PRINT "y" : p = l(p) : GOTO
70
130 GOTO 100
140 FOR j = 1 TO 1000 : NEXT j
150 PRINT "IS THE ANIMAL A ";q$(p); " ? "
160 k$ = INKEY$ : IF k$ = "" THEN 160 ELSE k$ =
LOWER$(k$)
170 IF k$ = "y" THEN PRINT "I thought so" : GOTO
330
180 IF k$ <> "n" THEN 160
190 nf = nf+2 : IF nf > n THEN PRINT "Increase
array sizes" : STOP
200 PRINT
210 INPUT "What had you chosen";q$(nf+1) :
q$(nf+1) = UPPER$(q$(nf+1))
220 q$(nf) = q$(p)
230 PRINT
240 PRINT "Enter a question that distinguishes"
250 PRINT "a ";q$(nf); " from a ";q$(nf+1)
260 PRINT : INPUT q$(p) : q$(p) = UPPER$(q$(p))
270 PRINT : PRINT "Would the answer be Y or N for
a "
280 PRINT q$(nf); " ? "
290 k$ = INKEY$ : IF k$ = "" THEN 290 ELSE k$ =
LOWER$(k$)
300 IF k$ = "y" THEN l(p) = nf : r(p) = nf+1 :
GOTO 330
310 IF k$ = "n" THEN r(p) = nf : l(p) = nf+1 :
GOTO 330
320 GOTO 290
330 PRINT : PRINT "Would you like to play again?"

```



```

340 k$ = INKEY$ : IF k$ = "" THEN 340 ELSE k$ =
LOWER$(k$)
350 IF k$ = "n" THEN END
360 IF k$ = "y" THEN 20
370 GOTO 340
380 REM initialise
390 n = 150 : nf = 12
400 DIM q$(n),l(n),r(n)
410 FOR j = 1 TO 13
420 READ q$(j),l(j),r(j)
430 NEXT j
440 RETURN
450 DATA "IS IT A BIRD",3,2
460 DATA "IS IT A MAMMAL",4,5
470 DATA "DOES IT FLY",6,7
480 DATA "DOES IT HAVE A TRUNK",8,9
490 DATA "FROG",0,0
500 DATA "ROBIN",0,0
510 DATA "OSTRICH",0,0
520 DATA "ELEPHANT",0,0
530 DATA "DOES IT HIBERNATE",10,11
540 DATA "SQUIRREL",0,0
550 DATA "DOES IT SWIM",12,13
560 DATA "SEAL",0,0
570 DATA "DOG",0,0

```

One small snag: when you turn off your 464, it will forget all that it has ever learned about frogs, elephants and so on. You can rectify this by replacing the DATA statements by a file of records which is updated before switch-off.

Since the topic of data structures is both large and complicated anyone interested in further study should obtain one of the numerous books specializing in the subject. A recommended book is "Successful Software for Small Computers" by G Beech, published by Sigma Technical Press. This contains numerous BASIC programs relevant to various data structures.

CHAPTER EIGHT

DATA PROCESSING

TECHNIQUES FOR SORTING

One of the few advantages a computer has over the human brain is its ability to undertake long and laborious, but relatively simple tasks, in a short time. One such task we shall examine is the mundane job of sorting data comprised of numeric or alphabetic items into some sequence – usually into increasing or alphabetic order. There have been many algorithms devised for doing such operations although their efficiency is usually inversely proportional to their complexity.

We will restrict ourselves to looking at the following four algorithms.

METHOD	COMPLEXITY	EFFICIENCY	
		<i>Small quantities of data</i>	<i>Large quantities of data</i>
Bubble sort	Very simple	Excellent	Poor
Insertion sort	Very simple	Excellent	Poor
Shell sort	Average	Good	Good
Quick sort	Complex	Good	Very Good

Fig. 8.1

For each method it is usual to store the data in arrays - the size of which need only be limited by the availability of RAM. Thus one thing which should always be considered when deciding on an algorithm is the storage requirement; some methods require a great deal of memory in addition to the array containing the items of data.

Each of the algorithms mentioned above will now be explained in detail with a program routine to illustrate it.

The main code below can be used to set up random data and to call each sort program. In order that the programs can be compared, the code will also give the processing time involved.

MAIN CODE

```
10 CLS : PRINT "SORTING TECHNIQUES" : PRINT
20 ZONE 8
30 INPUT "Enter number of items";n
40 IF n<5 OR n>1000 THEN 10
```

```

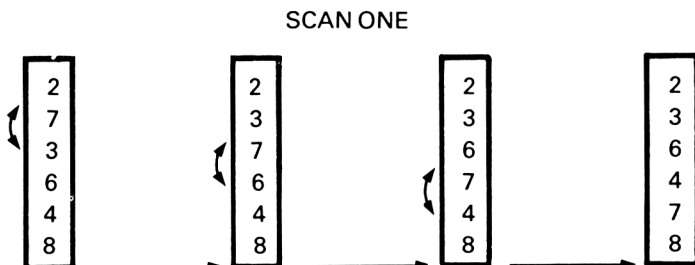
50 DIM d(n),s(40,2)
60 FOR j = 1 TO n
70 d(j) = INT(RND*2500)
80 NEXT j
90 CLS : PRINT "SORTING TECHNIQUES" : PRINT
100 PRINT "Enter" : PRINT
110 PRINT "F1...Bubble sort"
120 PRINT "F2...Insertion sort"
130 PRINT "F3...Shell sort"
140 PRINT "F4...Quick sort"
150 k$ = INKEY$ : IF k$ = "" THEN 150 ELSE k =
ASC(k$)-48
160 IF k<1 OR k>4 THEN 150
170 t1 = TIME
180 ON k GOSUB 1000,2000,3000,4000
190 t = ROUND ((TIME-t1)/300,2)
200 PRINT : PRINT "Data sorted"
210 PRINT : PRINT "Time taken ";t;"seconds"
220 PRINT "Do you wish to see the data? (y/n/s)"
230 k$ = INKEY$ : IF k$ = "" THEN 230 ELSE k$ =
LOWER$(k$)
240 IF k$ = "n" THEN 60
250 IF k$ = "s" THEN END
260 IF k$<>"y" THEN 230
270 CLS : PRINT "Sorted data" : PRINT
280 FOR j = 1 TO n
290 PRINT d(j),
300 NEXT j
310 FOR j = 1 TO 5000 : NEXT j
320 GOTO 60

```

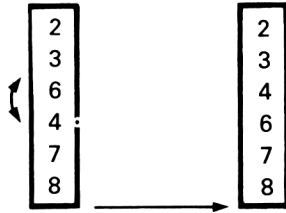
Bubble Sort

The first method of sorting is called a 'bubble sort' because the lowest values can be thought of as floating upwards to one end of the array and the highest values sink to the bottom. The array is continually scanned and two neighbouring items are swapped if the first has a higher value than the other; if no exchanges are made, the list is in order.

Example



SCAN TWO



SCAN THREE

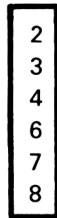


Fig. 8.1

No exchanges are made on scan 3 which implies that the list is ordered.

PROGRAM 24: BUBBLE SORT

```
1000 REM bubble sort
1010 FOR j = 1 TO n-1
1020 f = 0
1030 FOR i = 1 TO n-j
1040 IF d(i) <= d(i+1) THEN 1070
1050 f = 1
1060 t = d(i+1) : d(i+1) = d(i) : d(i) = t
1070 NEXT i
1080 IF f = 0 THEN 1100
1090 NEXT j
1100 RETURN
```

Insertion Sort

The 'insertion sort' method enables a list of items to be ordered with a single scan of the array. When an item is found to be out of order, its correct position is found and then by moving the other items one place along the array, it can be correctly positioned.

Example:

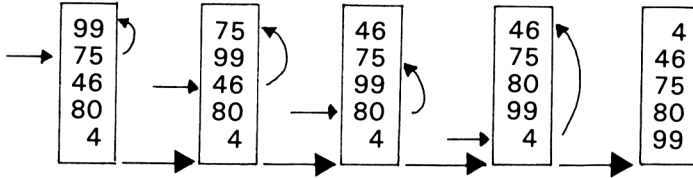


Fig. 8.3

PROGRAM 25: INSERTION SORT

```

2000 REM insertion sort
2010 FOR j = 2 TO n
2020 t = d(j)
2030 FOR i = j-1 TO 1 STEP -1
2040 IF d(i) <= t THEN 2070
2050 d(i+1) = d(i)
2060 NEXT i
2070 d(i+1) = t
2080 NEXT j
2090 RETURN

```

Shell Sort

Whereas the 'bubble sort' only compares adjacent data items in the array, a shell sort makes initial comparisons between items that are far apart, on the assumption that if elements are far apart and have to be swapped it is more efficient to do it as soon as possible. The separation between data elements is called the 'sort interval'. Initially, the sort interval is set to the number of elements and then on each scan the interval is reduced by one half until the final scan is equivalent to the 'bubble sort'.

PROGRAM 26: SHELL SORT

```

3000 REM shell sort
3010 si = n
3020 IF si < 1 THEN 3120
3030 si = INT(si/2)
3040 f = 0
3050 FOR i = 1 TO n-si
3060 IF d(i) <= d(i+si) THEN 3090
3070 f = 1
3080 t = d(i+si) : d(i+si) = d(i) : d(i) = t
3090 NEXT i
3100 IF f = 0 THEN 3020
3110 GOTO 3040
3120 RETURN

```

Quick Sort

Finally, we have the 'quick sort' algorithm. It is one of the more complex sorting algorithms but usually gives the fastest results. The general idea is that the items are split up (partitioned) into sub-groups which, in turn, are partitioned further until the sub-groups are small enough so that a form of 'bubble sort' can be applied on the group efficiently. However, while the sub-groups are being sorted the program must remember the position of the remaining unsorted data items. This is done by storing the 'start' and 'end' position of the array of the items that remain unsorted. These positions are stored in a second array so that the first positions stored are the first positions retrieved; such a structure is called a stack. The sort is complete when the stack becomes empty.

Two pointers are used in the quick sort, one indicates the start and the other the end item in the array. One of the two pointers will always point to the element in the array which was initially at the first position in the array; this pointer is called the pivot. The two items that are currently pointed to are swapped if the one nearer the start of the array is greater than the other. The second pointer is then moved one position along the array towards the pivot and the process is repeated. This continues until the two pointers meet, in which case the array is partitioned about the pivot into sub-groups and the process is then repeated on each new sub-group. The positions are stored in the array $s(n,2)$. If, because of vast quantities of data a 'stack overflow' message appears, then the dimension of the array must be increased. Example:

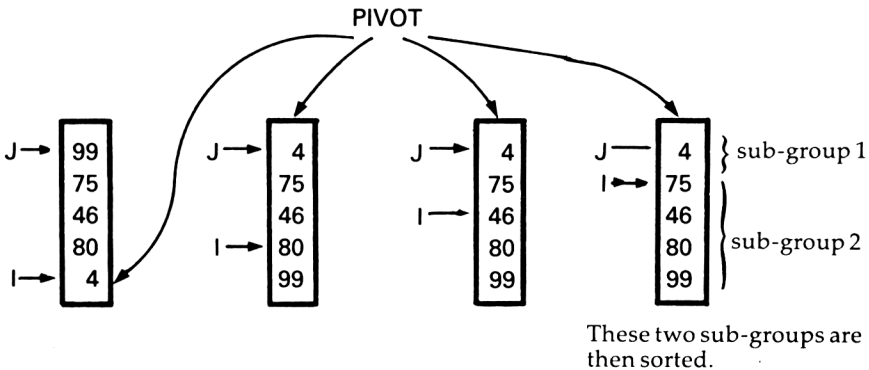


Fig. 8.4

PROGRAM 27: QUICK SORT

```

4000 REM quick sort
4010 ps = 1
4020 s(1,1) = 1
4030 s(1,2) = n
4040 IF ps = 0 THEN 4200

```

```

4050 ii = s(ps,1) : jj = s(ps,2) : ps = ps-1
4060 p = 0 : i = ii : j = jj
4070 IF d(i) <= d(j) THEN 4100
4080 p = 1-p
4090 t = d(i) : d(i) = d(j) : d(j) = t
4100 IF p = 0 THEN i = i+1
4110 IF p = 1 THEN j = j-1
4120 IF i<j THEN 4070
4130 IF i >= jj THEN 4170
4140 ps = ps+1
4150 IF ps>40 THEN PRINT "STACK OVERFLOW" : STOP
4160 s(ps,1) = i+1 : s(ps,2) = jj
4170 jj = i-1
4180 IF jj>ii THEN 4060
4190 GOTO 4040
4200 RETURN

```

ALPHABETICAL SORTING

One final sorting program demonstrates how to sort a sequence of strings alphabetically. This is done by creating a binary tree and then undertaking an LNR traversal (see chapter 7). The traversal is done by starting at the ROOT and continually travelling left, popping the nodes onto a stack. Wherever a null pointer is reached we pop the stack and print out the data value of the node. Having gone as far as possible we then backtrack to the parent node by again popping the stack and printing its data value. Then if there is a right-hand edge, we undertake the same process on its subtree. This process is repeated for the whole tree. Remember that when comparing strings, a string is found to be less than another if it comes first in alphabetical order (for example, as in a dictionary).

PROGRAM 28: ALPHABETICAL/TREE SORT

```

10 CLS
20 null = -1 : m = 150
30 DIM n$(m),r(m),l(m),s(m)
40 GOSUB 250
50 CLS : PRINT "ALPHABETICAL SORTING" : PRINT
60 PRINT "Enter" : PRINT
70 PRINT "F1...Initialise"
80 PRINT "F2...Add to tree"
90 PRINT "F3...List tree"
100 k$ = INKEY$ : IF k$ = "" THEN 100 ELSE k =
ASC(k$)-48
110 IF k<1 OR k>3 THEN 100
120 ON k GOSUB 200,400,600
130 GOTO 50

200 REM initialise
210 PRINT : PRINT "Initialise"
220 PRINT : PRINT "Are you sure?"

```



```

230 k$ = INKEY$ : IF k$ = "" THEN 230
240 IF k$ = "n" THEN 310
250 FOR j = 1 TO m
260 n$(j) = "" : r(j) = 0 : l(j) = 0
270 NEXT j
280 root = null : v = 0
290 PRINT : PRINT "Initialisation complete"
300 FOR j = 1 TO 1000 : NEXT j
310 RETURN

400 REM add
410 CLS : PRINT "Add to tree" : PRINT
420 INPUT "Enter data";d$
430 v = v+1 : IF v>m THEN PRINT "Increase array
size" : STOP
440 IF d$ = "" THEN 420
450 IF root = null THEN n$(v) = d$ : r(v) = null
: l(v) = null : root = v : GOTO 530
460 vv = ROOT
470 IF n$(vv)<d$ THEN 500
480 IF l(vv) = null THEN l(vv) = v : GOTO 520
490 vv = l(vv) : GOTO 470
500 IF r(vv) = null THEN r(vv) = v : GOTO 520
510 vv = r(vv) : GOTO 470
520 n$(v) = d$ : r(v) = null : l(v) = null
530 PRINT : PRINT "Item added"
540 PRINT : PRINT "Exit y/n"
550 k$ = INKEY$ : IF k$ = "" THEN 550
560 IF k$<>"y" THEN 400
570 RETURN

600 REM list
610 CLS : PRINT "Press SPACE BAR to list" : PRINT
620 IF INKEY$ <> SPACE$(1) THEN 620
630 IF root = null THEN 730
640 vv = root : p = 1
650 IF vv = null THEN 690
660 s(p) = vv : p = p+1
670 vv = l(vv)
680 GOTO 650
690 p = p-1 : vv = s(p)
700 PRINT n$(vv)
710 vv = r(vv)
720 IF p>0 THEN 650
730 PRINT : PRINT "End of list"
740 IF INKEY$ = "" THEN 740
750 RETURN

```

CASSETTE FILES

By now you should be well aware (maybe due to a previous mistake!) that when the power is disconnected from the Amstrad, the stored program and

its data are lost for ever. Programs are easily stored on normal domestic cassette tape for later retrieval as explained in the Amstrad users manual. Information may be stored permanently in a similar fashion by communicating with stream number 9 - however, this technique does have some severe limitations. The standard system contains just a single cassette unit, so to handle cassette input and output one has to keep swapping tapes when appropriate. Also, there is a long time delay if the section of tape containing the information required is a long way away from the present tape position. Since all the information is stored and accessed one item after another, the cassette files are said to be organised sequentially. Data is input and output to cassette files in blocks of 2K - these are small enough to be stored in the main memory and are stored on the tape with small intervals between them which enables the cassette unit to have time to stop and start, see figure 8.5. When data is loaded from the tape, one block is loaded into the cassette buffer and the data can then be read from it. Similarly, when data is being stored on tape it is written to the cassette buffer which is transferred to the tape when it becomes full or the end of the file is signalled. This technique avoids having to continually stop and start the tape for each data item, and so enables the data to be more closely packed onto the tape.

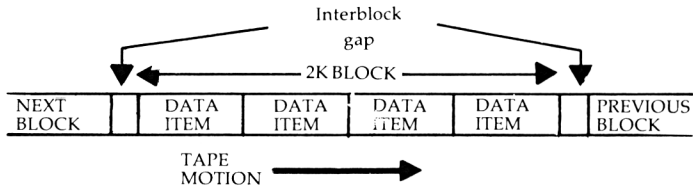


Figure 8.5: Sequential Storage on Tape

In large commercial installations, the slowness of sequential files on magnetic tape means that most amendments are made to large data files by using 'transaction files'. Consider, for example, a bank with all the current accounts of its customers stored on magnetic tape. During the day the value of some of the accounts will be altered as cheques are received and cashed. It would be very inefficient, if not impossible, to change the values of the accounts as each alteration occurs. Instead, a transaction file is built up during the day which contains each amendment to an account as it takes place. At the end of the day, the transaction file is sorted using some unique key (probably the account number) until the records are in the same order as the records appear on the master tape file. The two tapes can then be matched to update the accounts and produce a new master file.

The one difference between communicating with cassette files and other I/O streams is that the files have to be opened using the commands **OPENIN** and **OPENOUT** for input and output respectively. In both cases the file name should be specified - note that if the first character is ! then the cassette operational prompts are omitted.

e.g. **OPENIN " ! ACCOUNTS "**

Information may now be passed to and from the cassette by using any of the I/O statements specifying a stream number of 9.

Finally, when all I/O is complete, the files should be closed, using the **CLOSEIN** and **CLOSEOUT**. Note that with output this is vital since it writes the remaining contents of the buffer to tape and appends an end-of-file (EOF) marker to the file. This marker may be tested for during **INPUT** to signal that all the data has been read using the function **EOF**.

The following program demonstrates cassette files by keeping a file of names and addresses. The file is read into memory and then the user may elect to add, delete or access names and addresses to the file. If the file is amended the user may rewrite the updated file to tape. In practice, it is recommended that you use three tapes; one for the program, another for the master file and the remaining as a back-up containing the second most recent version.

The maximum number of names in this program is 75 but this may be increased by amending the variable in line 70.

PROGRAM 29: ADDRESS BOOK

```
10 CLS
20 ON BREAK GOSUB 200
30 WINDOW #1,1,40,25,25 : PEN #1,3 : INK 3,3,24
40 e$ = CHR$(164)+ " Mark R Harrison"
50 GOSUB 7000
60 name=1 : street=2 : town=3 : county=4 :
postcode=5 : phone=6
70 m = 75 : DIM d$(m,6) : ptr = 1 : f$ = "DATA"
80 CLS : PRINT "ADDRESS BOOK" : PRINT
90 PRINT "Select:" : PRINT
100 PRINT "F1...Retrieve file"
110 PRINT "F2...Save file"
120 PRINT "F3...Add record"
130 PRINT "F4...Delete record"
140 PRINT "F5...Display record"
150 PRINT "F6...Display names"
160 k$ = INKEY$ : IF k$ = "" THEN 160 ELSE k =
ASC(k$)-48
170 IF k<1 OR k>6 THEN e$ = "Invalid option" :
GOSUB 7000 : GOTO 160
180 ON k GOSUB 1000,2000,3000,4000,5000,6000
190 GOTO 80
200 CLS : PRINT "BREAK"
210 PRINT "Do you want to save data?"
220 k$ = INKEY$ : IF k$ = "" THEN 220 ELSE k$ =
LOWER$(k$)
230 IF k$ <> "n" THEN GOSUB 2000
240 END

1000 CLS : PRINT "Retrieve file"
1010 PRINT
1020 OPENIN f$
1030 FOR ptr = 1 TO m
1040 FOR j = 1 TO 6
```

```

1050 INPUT #9,d$(ptr,j)
1060 NEXT j
1070 IF EOF THEN 1100
1080 NEXT ptr
1090 e$ = "Structures full" : GOSUB 7000
1100 CLOSEIN
1110 RETURN

2000 CLS : PRINT "Save file"
2010 PRINT : PRINT "Enter cassette speed"
2020 PRINT "0...Supersafe"
2030 PRINT "1...Speedload"
2040 k$ = INKEY$ : IF k$ = "" THEN 2040
2050 IF k$ = "0" THEN SPEED WRITE 0 ELSE SPEED
WRITE 1
2060 PRINT
2070 OPENOUT f$
2080 FOR ptr = 1 TO m
2090 IF d$(ptr,name) = "" THEN 2130
2100 FOR j = 1 TO 6
2110 WRITE #9,d$(ptr,j)
2120 NEXT j
2130 NEXT ptr
2140 CLOSEOUT
2150 RETURN

3000 CLS : ptr = 0
3010 PRINT "Add record" : PRINT
3020 PRINT "Enter:" : PRINT
3030 INPUT "NAME";n$
3040 IF n$ = "" THEN e$ = "No name specified" :
GOSUB 7000 : GOTO 3000
3050 FOR j = m TO 1 step -1
3060 IF d$(j,name) = "" THEN ptr = j
3070 IF d$(j,name) = n$ THEN e$ = "Name not
unique" : GOSUB 7000 : GOTO 3190
3080 NEXT j
3090 IF ptr = 0 THEN e$ = "File full" : GOSUB
7000 : GOTO 3190
3100 d$(ptr,name) = n$
3110 INPUT "STREET";d$(ptr,street)
3120 INPUT "TOWN";d$(ptr,town)
3130 INPUT "COUNTY";d$(ptr,county)
3140 INPUT "POST CODE";d$(ptr,postcode)
3150 INPUT "PHONE";d$(ptr,phone)
3160 PRINT : PRINT "Correct?"
3170 k$ = INKEY$ : IF k$ = "" THEN 3170 ELSE k$ =
LOWER$(k$)
3180 IF k$ = "n" THEN PRINT : GOTO 3110
3190 RETURN

4000 CLS : PRINT "Delete record" : PRINT

```

```

4010 INPUT "Enter name";n$ : PRINT
4020 FOR ptr = 1 TO m
4030 IF d$(ptr,name) = n$ THEN d$(ptr,name) = ""
: GOTO 4060
4040 NEXT ptr
4050 e$ = "Name not in list" : GOSUB 7000 : GOTO
4080
4060 PRINT : PRINT "Name deleted"
4070 FOR j = 1 TO 2500 : NEXT j
4080 RETURN

```

```

5000 CLS : PRINT "Display record" : PRINT
5010 INPUT "Enter name";n$ : PRINT
5020 FOR ptr = 1 TO m
5030 IF d$(ptr,name) = n$ THEN 5060
5040 NEXT ptr
5050 e$ = "Record not present" : GOSUB 7000 :
GOTO 5120
5060 PRINT "STREET",d$(ptr,street)
5070 PRINT "TOWN",d$(ptr,town)
5080 PRINT "COUNTY",d$(ptr,county)
5090 PRINT "POST CODE",d$(ptr,postcode)
5100 PRINT "PHONE",d$(ptr,phone)
5110 IF INKEY$ = "" THEN 5110
5120 RETURN

```

```

6000 CLS
6010 FOR ptr = 1 TO m
6020 IF d$(ptr,name) <> "" THEN WRITE
d$(ptr,name)
6030 NEXT ptr
6040 PRINT : PRINT "EOF"
6050 FOR j = 1 TO 2500 : NEXT j
6060 RETURN

```

```

7000 PRINT CHR$(7)
7010 PRINT #1,e$
7020 IF INKEY$ = "" THEN 7020
7030 PRINT #1,SPACE$(20)
7040 RETURN

```

It is very important to make sure that the tape heads of the cassette unit are kept clean at all times. When cassette tape passes over the tape head, it leaves a small deposit; after a time these deposits will collect to such an extent that the recordings will be muffled and the Amstrad will not be able to decipher the signals. It is also worth using high quality tapes since some poor quality ones may not give a large enough signal for the system to interpret. If the signal is too low the Amstrad will search indefinitely for a program or data file. Another fault with poor quality tapes is that they shed their magnetic coating more easily. This can cause a more rapid build up of metal oxide on the heads but, more seriously, the tape may lose an item of vital information.

Disk Files

At the time of writing it is envisaged that CP/M, a disk operating system, will soon be available on the Amstrad. The disk drive is now becoming the most common item of hardware for storing information used by a microcomputer. Their advantages include good reliability and fast access times. Unfortunately their cost has led them to be regarded by some hobbyists as expensive luxuries.

CP/M is discussed in more depth in **"CP/M - The Software Bus"** (published by Sigma Press) which is an ideal companion for both novice and experienced programmers alike.

CHAPTER NINE

AMSTRAD GRAPHICS

Every reader of this book will probably have a friend who, at first sight of your machine, will invite you to switch it on and expect to see graphics like those found in amusement arcades. We shall now take our first step towards achieving such displays.

There are two methods of amending the information displayed on the screen; these are:

- 1) Clearing the screen with the command `CLS` and then rebuilding the display; this is done line by line, each of which has to be stored and so uses up valuable memory space. Another disadvantage is that the screen rebuilding can take a long time.
- 2) Amending the required section of the screen by repositioning the cursor, either with cursor control characters or the command `LOCATE`, and then printing over the original section.

Some readers may be accustomed to a technique called `PEEK/POKE` graphics; where information is displayed on the screen by `POKE`ing the section of memory which contains the screen information and then by `PEEK`ing the same area we can work out the position of the characters. Unfortunately, because of the layout of the Amstrad's screen memory, the `PEEK/POKE` graphics technique is not really feasible. However, we can always get round this problem by using the `LOCATE` command and storing the position of characters in variables.

USER DEFINED CHARACTERS

Whilst the standard character set has an excellent range of symbols there will always be a time when the user requires a symbol that is not available. We shall now see how to define our own characters and create graphic symbols such as mathematic symbols, animals, rockets, explosions, space invaders, etc, - the list is endless.

All characters are defined by eight bytes in memory which represent the 8 x 8 grid of pixel dots that make up the character. Each bit that is set to a '1' in a byte represents a pixel illuminated in the current `INK` colour, see figure 9.1.

e.g. <i>DECIMAL</i>	<i>BINARY</i>	<i>CHARACTER</i>
24	0 0 0 1 1 0 0 0	
60	0 0 1 1 1 1 0 0	
102	0 1 1 0 0 1 1 0	
102	0 1 1 0 0 1 1 0	
126	0 1 1 1 1 1 1 0	
102	0 1 1 0 0 1 1 0	
102	0 1 1 0 0 1 1 0	
102	0 1 1 0 0 1 1 0	
0	0 0 0 0 0 0 0 0	

Fig. 9.1

All character formation data originates from the Amstrad's ROM at a section known as the character generator. So that the user can amend some of the formation data values to create new characters, the data has to be placed into RAM at a pre-determined position; however, if all 256 characters were stored in RAM, a mammoth 256 x 8 or 2K bytes would be required. So, as a compromise, at switch on the Amstrad gets the formation data for the first 240 characters from ROM and for the remaining 16 characters from a copy of the ROM located in RAM. This may be altered by **SYMBOL AFTER** which specifies the ASCII code of the first character to be located in RAM e.g. **SYMBOL AFTER 128** - characters 128 to 255 can be redefined. Note that when **SYMBOL AFTER** is executed all previously redefined characters are returned to their original values.

A character is redefined by the command **SYMBOL** which specifies the character code followed by up to eight values defining the pixel data. If less than eight values are specified, then zero is assumed for the latter absent values.

As an example let's define CHR\$(255) to be a friendly space invader.

<i>DECIMAL</i>	<i>BINARY</i>	<i>CHARACTER</i>
66	0 1 0 0 0 0 1 0	
36	0 0 1 0 0 1 0 0	
189	1 0 1 1 1 1 0 1	
189	1 0 1 1 1 1 0 1	
255	1 1 1 1 1 1 1 1	
60	0 0 1 1 1 1 0 0	
36	0 0 1 0 0 1 0 0	
231	1 1 1 0 0 1 1 1	

Fig. 9.2

And so, all we need to tell the Amstrad is:

SYMBOL 255,66,36,189,189,255,60,36,231

Now by printing the character **CHR\$(255)** our invader appears.


```

10 CLS
20 SYMBOL 255,66,36,189,189,255,60,36,231
30 INK 1,21,17 : INK 0,0
40 PEN 1 : PAPER 0 : BORDER 0
50 x = 20 : y = 12
60 x = x + INT(RND*3-1) : y = y + INT(RND*3-1)
70 IF x<1 OR x>40 OR y<1 OR y>25 THEN END
80 LOCATE x,y : PRINT CHR$(255)
90 FOR j = 1 TO 1000 : NEXT j
100 LOCATE x,y : PRINT SPACE$(1)
110 GOTO 60

```

Notice how, by making the invader flash slowly, the effect is improved. Of course larger characters could be obtained by printing four more characters next to each other.

Animation effects can be achieved by designing several characters to make up a series of frames, and then printing them rapidly over each other. The following example shows a little stick man dancing:

```

10 CLS
20 v = INT(RND*4+248)
30 LOCATE 20,12
40 PRINT CHR$(v)
50 FOR k = 1 TO 200 : NEXT k
60 GOTO 20

```

TRANSPARENT PRINTING

One control character we have not seen yet is `CHR$(22)` which is used to enable/disable the transparent printing option. A second character is also used which acts as the switch.

i.e.
`PRINT CHR$(22) + CHR$(0)` disables option
`PRINT CHR$(22) + CHR$(1)` enables option

Normally, when a character is printed, the character that was originally present at the print position is completely removed. If the transparent mode is enabled then the character being printed is superimposed on top of the first character, and the background pixels take the previous colours that were present. This enables several colours to be displayed at one character position. This is illustrated in the following example which uses the Sigma Press logo.

```

10 CLS
20 PRINT CHR$(22) + CHR$(1)
30 LOCATE 25,1
40 INK 3,20 : PEN 3
50 PRINT STRING$(13,CHR$(143))
60 LOCATE 25,1
70 INK 2,0 : PEN 2

```

```

80 PRINT CHR$(190) + CHR$(8) +
STRING$(13,CHR$(210))
90 LOCATE 26,1
100 INK 1,15 : PEN 1
110 PRINT "Sigma Press"
120 PRINT CHR$(22) + CHR$(0)

```

HIGH RESOLUTION CONTROL

Initially we considered the Amstrad's display as being divided up, in MODE 1, into 25 lines of 40 characters; we then went on to see how each character position was further divided up into an 8 x 8 grid of smaller pixel dots that could be either illuminated or not. We shall now see how to control the individual pixel dots. Just as with character positions, the number of pixels present depends on the screen mode as shown in figure 9.3.

<i>MODE</i>	<i>No. of pixels</i>
0	160x200
1	320x200
2	640x200

Figure 9.3

A specific pixel is referred to by a two number coordinate (x,y) where x is the horizontal position and y is the vertical position that the pixel lies in.

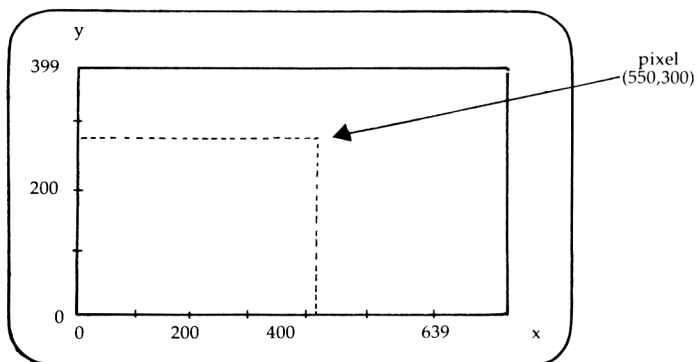


Figure 9.4: The Amstrad's pixel layout

It is important to remember that the coordinates in figure 9.4 are the same for all screen modes; thus in some modes certain coordinates will refer to

the same pixel. The ratio of the vertical/horizontal mapping is arranged to have a ratio of 1; (so that if we plot a circle it won't appear squashed). It is possible to ink in a particular pixel by the statement **PLOT** x,y or **PLOT** x,y,i where i is an ink number. A further extension to this is the **PLOTR** statement whereby the x and y values specified are offsets from the previous plot position.

Similarly, **DRAW** causes a line to be drawn from the present position to a new point (x,y) relative to the origin $(0,0)$; alternatively **DRAWR** causes the line to be drawn from the present position to a new point specified by offsets relative to the previous plot position. We may enquire at any time on the x and y coordinates of the current plot position with the functions **XPOS** and **YPOS**.

So let's look at a few simple examples:

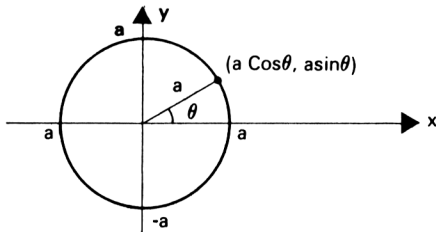
PROGRAM30: BUBBLES

```

10 CLS : PRINT "BUBBLES"
20 DEG : RANDOMIZE TIME
30 FOR j = 1 TO 25
40 x = INT(RND*640)
50 y = INT(RND*350)
60 r = INT(RND*75+25)
70 i = INT(RND*3+1)
80 PLOT x+r,y
90 FOR q = 0 TO 360 STEP 10
100 DRAW x+r*cos(q),y+r*sin(q),i
110 NEXT q
120 NEXT j

```

Try replacing line 90 with: **FOR** $q = 0$ **TO** 7200 **STEP** 123



This illustrates how we can draw circles by plotting all the points which are given by $(a \cos \theta, a \sin \theta)$ where a is the radius

Fig. 9.5 Circle plot

PROGRAM31: EPICYCLICS

```

10 CLS : PRINT "EPICYCLICS"
20 v = 4/3
30 MOVE 400,200
40 FOR j = 0 TO 6*PI STEP PI/40

```

```

50 x = 300+100*COS(j*v)*COS(j)
60 y = 200+100*COS(j*v)*SIN(j)
70 DRAW x,y
80 NEXT j

```

Try replacing the value of v to other values such as $\frac{\pi}{3}$; in some cases it may be necessary to increase the terminating value of the control loop in line 40.

PROGRAM 32: SINE/COSINE WAVES

```

10 CLS
20 p = 0
30 PRINT CHR$(23) + CHR$(p)
40 FOR j = 0 TO 10*PI STEP PI/25
50 x = j*50/PI
60 y1 = 100*SIN(j)
70 y2 = 100*COS(j)
80 PLOT x,150 : DRAWR 0,y1,2
90 PLOT x,150 : DRAWR 0,y2,1
100 NEXT j

```

*This plots a cosine wave
(yellow) on top of a sine
wave (bright blue)*

Try replacing the value of p in line 20 to 3 - what do you notice when you rerun the program? Areas of the cosine wave covering the sine wave should now appear to be coloured red. Whilst control character `CHR$(23)` followed by `CHR$(0)` sets the ink to normal, `CHR$(23)` followed by `CHR$(3)` causes the displayed ink to take a value dependent on both the ink that is already present and the ink that was specified. The new ink value is found by finding the 'logical OR' of the two inks; that is, inspect the two ink values in binary form and obtain a new ink value consisting of bits set, if and only if, a corresponding bit is set in either ink.

e.g. the two inks are 1, and 2 -	<i>DEC</i>	<i>BINARY</i>
	1	01
	2	10
	3	11

thus the ink where the areas are superimposed will take the colour of ink 3.

A full list of values that can follow `CHR$(23)` are summarised in fig. 9.6.

<code>CHR\$(23) +</code>		<i>New ink binary value consists of bits set if and only if:-</i>
--------------------------	--	---

<code>CHR\$(0)</code>	Normal	
<code>CHR\$(1)</code>	XOR	Corresponding bits are not equal
<code>CHR\$(2)</code>	AND	Both of the corresponding bits are set
<code>CHR\$(3)</code>	OR	Either of the corresponding bits are set

Fig. 9.6 INK MODES

Program 33 demonstrates that it is possible to draw a function of two variables as three dimensional solid surfaces. To see how the program works, consider figure 9.7. The picture is built up by plotting parallel curved lines at positions on the screen which relate to the value at the plotted position. If you refer to figure 9.7, you will see that we are generating one slice at a time, from the front to the back of the 'solid' surface.

To produce the three dimensional effect, the program considers the line ST and all lines parallel to it, and calculates the value of the function at the points where the straight lines meet the curved lines. The point is plotted on the screen at a position which takes into account the value of the function and also which curved line is being considered. However, to give the impression of a solid surface, the point would only be plotted if there was no surface in front of it, thus a point is only plotted if the y coordinate of the plot position is higher than any previous value found on the line being considered.

Try replacing the function line with your own ideas, however they must be scaled so that the plotting does not leave the screen.

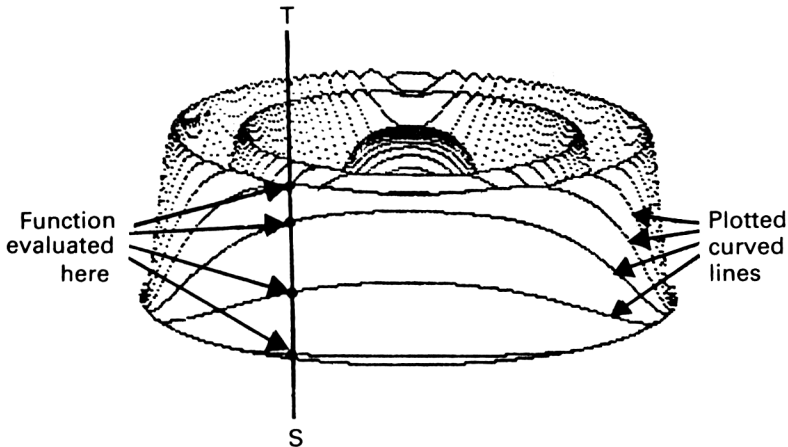


Figure 9.7: THREE DIMENSIONAL PLOT

PROGRAM 33: THREE DIMENSIONAL PLOTS

```

10 DEF FN a(j,k) = 50*(SIN((j*j+k*k)/1000))
20 CLS
30 q = 175
40 FOR x = 0 TO q STEP 2
50 y = SQR(q*q-x*x)
60 m = -1E+09
70 FOR z = -y TO y STEP 4
80 v = FN a(x,z)
90 p = v + z/3 + 100
100 IF p <= m THEN 120 ELSE m = p

```

```

110 PLOT 320+x,p : PLOT 320-x,p
120 NEXT z
130 NEXT x

```

Try experimenting with your two variable functions in line 10, but remember that they should be scaled so that the PLOTting does not leave the screen.

Program 34 will plot a three dimensional histogram for up to 18 values. The main plotting is undertaken in the subroutine situated at line 200 and is written in a form that should be easy to incorporate into your own programs. Two values need to be set up before it is called; q is a number between 1 and 18 which corresponds to the position on the screen where the block is to be displayed and y is the vertical length of the block. The three dimensional effect is achieved by colouring the three visible sides of the block in different shades of mauve.

A new statement, **ORIGIN**, has been introduced which allows us to reposition the graphics screen coordinate grid. Coordinate (0,0) is moved to the specified position. It is also possible to specify a further four values which set up a new graphics screen window, as shown below:

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

In our program, the y coordinate has been shifted vertically upwards which makes plotting simple with a line of text underneath the histogram.

The main problem is for the program to choose a suitable scale for the data. In this case, by inspecting line 120, you can see a block of 300 units drawn for the largest item of data and that all the other blocks have been drawn in proportion.

PROGRAM 34: THREE DIMENSIONAL HISTOGRAM

```

10 CLS
20 ORIGIN 0,25
30 DIM d(20)
40 READ v$
50 LOCATE 3,25 : PRINT v$
60 FOR j = 1 TO 18
70 READ v : IF v = -1 THEN 110
80 n = j : d(j) = v
90 IF v>m THEN m = v
100 NEXT j
110 FOR q = 1 TO n
120 y = d(q)/m*300
130 GOSUB 200
140 NEXT q
150 PEN 2
160 LOCATE 1,1 : END

200 FOR p = 0 TO y
210 PLOT 32*q,p

```

```

220 INK 2,8 : DRAWR 30,0,2
230 INK 3,7 : DRAWR 10,10,3
240 NEXT p
250 FOR p = 1 TO 10
260 PLOT 32*q+p,y+p
270 INK 1,4 : DRAWR 32,0,1
280 NEXT p
290 RETURN

```

```

500 DATA "J F M A M J J A S O N D SALES 1984"
510 DATA 100,77,155,77,23,56,24,100,78,145,166,66
520 DATA -1

```

We can check which colours are placed at a particular position by using the functions TEST and TESTR; the first requires absolute coordinates to be specified and the second requires relative offsets from the current plot position. This is demonstrated in program 35 in which you are in charge of a spaceship under the control of the onboard computer - an Amstrad and its cursor control keys!! The objective is to navigate the ship and avoid stars, your trail and the edge of the universe. The best time achieved is recorded and displayed continually - good luck!

PROGRAM 35: GALAXY EXPLORER

```

10 MODE 1 : RANDOMIZE TIME
20 CLS : PRINT "GALAXY EXPLORER" : PRINT
30 INPUT "Enter skill level 1 - 99";sf
40 IF sf<1 OR sf>99 THEN 30
50 sf = sf/100
60 h = 0
70 INK 0,0 : INK 1,24 : INK 2,23 : INK 3,26
80 BORDER 0 : PAPER 0 : CLS
90 PRINT "Best time";h,
100 PLOT 0,0
110 DRAW 639,0,1 : DRAW 639,380,1 : DRAW 0,380,1
: DRAW 0,0,1
120 x = 300 : y = 190 : k = INT(RND*4) : kk = 3-k
130 t1 = TIME
140 k$ = INKEY$ : IF k$<>" " THEN kk = ASC(k$)-240
150 IF kk >= 0 AND k<4 THEN k = kk
160 IF k = 0 THEN y = y+2
170 IF k = 1 THEN y = y-2
180 IF k = 2 THEN x = x-2
190 IF k = 3 THEN x = x+2
200 IF TEST(x,y)<>0 THEN PRINT "CRASH!!!" : GOTO
250
210 PLOT x,y,2
220 p = INT(RND*638+1) : q = INT(RND*378+1)
230 IF TEST(p,q) = 0 AND RND<sf THEN PLOT p,q,3
240 GOTO 140

```

```

250 t = ROUND((TIME-t1)/300,2)
260 IF t>h THEN h = t
270 IF INKEY$ <> "" THEN 270
280 IF INKEY$ = "" THEN 280
290 GOTO 80

```

Up to now we have considered the screen with two separate types of windows - character and pixel graphics. However, there may be times when we want to integrate the two. With the command **TAG** we are able to redirect the output of a stream so that it can be written at the current plot position allowing text and symbols to be combined with graphics. Since the resolution of the graphics screen is greater, using **TAG** enables us to obtain smoother scrolling.

e.g.
TAG #0 - tags the top left of the character output to stream 0 to the current graphics position.

This mode may be disabled by **TAGOFF**

e.g.
TAGOFF #0

When printing characters in **TAG** mode they should be followed by a semi-colon to suppress control characters such as line feed and carriage return. The next example illustrates the smooth animation effects that can be achieved by using **TAG**.

PROGRAM 36: ANIMATION

```

10 CLS
20 p = 0 : s = 10
30 TAG #0
40 PLOT 0,183 : DRAWR 639,0,1
50 RESTORE
60 FOR j = 1 TO 4
70 READ v
80 p = p+1 : IF p = 285 THEN 160
90 MOVE p,200
100 PRINT SPACES$(1);CHR$(v);
110 MOVE 600-p,200
120 PRINT SPACES$(1);CHR$(v);
130 FOR q = 1 TO s : NEXT q
140 NEXT j
150 GOTO 50
160 PLOT 315,211,3 : PRINT CHR$(228);
170 DATA 248,250,249,251

```

One excellent effect of movement can be achieved by drawing an object with pixels of various colours and then rotating the colours through it. This is illustrated in program 37 which shows a rotating sphere.

PROGRAM 37: ROTATION

```
10 DEG
20 INK 0,1 : INK 1,1 : INK 2,1 : INK 3,1
30 PAPER 3 : BORDER 1 : CLS
40 i(0) = 13 : i(1) = 24 : i(2) = 0
50 b = 100
60 i = 0
70 ORIGIN 320,200
80 FOR a = -b TO b STEP 5
90 MOVE 0,b
100 FOR t = 0 TO 180 STEP 18
110 DRAW a*SIN(t),b*COS(t),i
120 NEXT t
130 i = (i+1)MOD 3
140 NEXT a
150 FOR a = 0 TO 2
160 INK 0,i(a)
170 INK 1,i((a+1)MOD 3)
180 INK 2,i((a+2)MOD 3)
190 FOR b = 0 TO 200 : NEXT b
200 NEXT a
210 GOTO 150
```

This, then, ends our look at graphics on the Amstrad. You have been given an insight into the vast range of graphics control possible on the Amstrad. I leave you to carry on making discoveries that could last a programming lifetime!

CHAPTER TEN

SOUND AND SYNTHESIS

Many Amstrad owners will be extremely proud of the excellent sound facilities of their machines. Our next topic is to investigate these features and see what sounds can be produced. There is only room in this book to scratch the surface of the subject. Anyone who has ever learned to play a musical instrument will know that it takes years of practice to achieve proficiency and the same is true of playing music on the Amstrad. Once you've read this chapter, the only way to master these skills is to experiment and learn from the results obtained.

Before we enter the realms of sound synthesis on your Amstrad let us take a brief look at what sound is.

Characteristics of Sound Waves

We all have plenty of experience with sound in everyday life and know that different sounds can be distinguished from one another - but what makes one sound different from another? Each sound is made by a transfer of energy caused by the disturbance of air from one point (called the source) to the ear. The difference in the sounds are effected by three factors: i) pitch ii) quality and iii) volume.

If the source oscillates f times a second, the sound emitted is said to have a frequency of f Hz (Hertz). Also, if the energy travels a distance of λ from the source in one second then the sound produced has a wavelength of λ . This can be represented diagrammatically as in Figure 10.1.

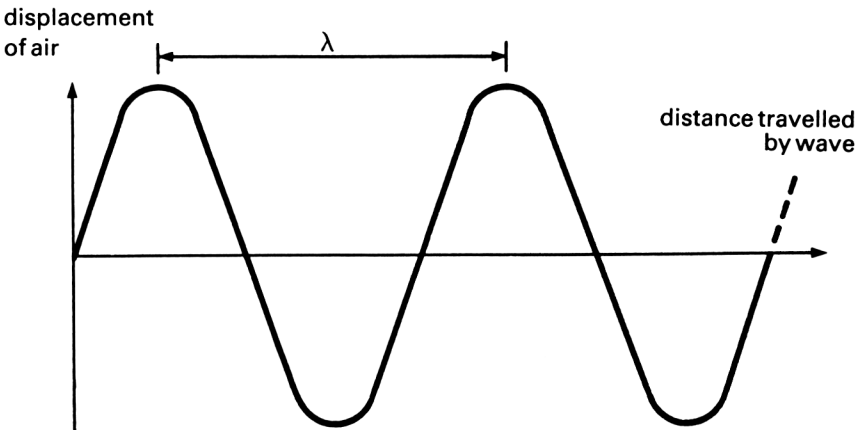


Fig. 10.1 : Sound Wave

The pitch of a note is dependent on the frequency of the sound wave produced; a high frequency produces a high pitched note and a low frequency produces a low pitched note. The musical interval between two notes is proportional to the ratio of the frequencies rather than the difference. Thus, the difference in pitch to the ear is the same if a note increases from 250 Hz to 500 Hz as from 1000 Hz to 2000 Hz. If one note has twice the frequency of another then the two notes are said to be one octave apart.

A tune which is played on several different musical instruments can still be differentiated even if exactly the same notes are played. This is because the quality of the waveforms from each instrument differs. For example, while a tuning fork would produce a waveform like that in Figure 10.1, a musical instrument would probably look more like that in Figure 10.2

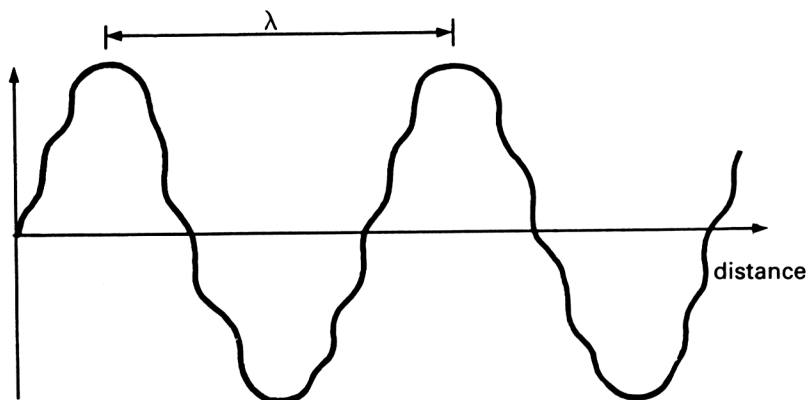


Fig. 10.2 Waveform with added "Noise"

SOUND and the Amstrad

Sounds can be obtained on the Amstrad by using the BASIC command **SOUND** which has a list of parameters; all are optional except for the first two. When a parameter is absent a default value is used.

SOUND *cs, tp, d, v, ve, te, np*

where *cs* - channel status
tp - tone period
d - duration
v - volume
ve - volume envelope
te - tone envelope
np - noise period

We shall now carry out an in depth investigation into each of the parameters:-

CHANNEL STATUS

The Amstrad has three independent sound oscillators (sometimes referred to as voices on other computers) and so in our **SOUND** statement we must specify the voice channels to where our sound output is to be sent. We can combine two or more voices together but must take care that the notes are synchronised correctly. The three voice channels are referred to as A, B and C; each of which can have up to five sounds queued up. The channel status parameter is a bit pattern which directs the sound output to the required channels as shown in Figure 10.3.

	7	6	5	4	3	2	1	0	bit
command	FLUSH	HOLD	rendez-vous C	rendez-vous B	rendez-vous A	send to C	send to B	A	
	128	64	32	16	8	4	2	1	value

Fig. 10.3 Channel Status

In the next section, remember that binary numbers are understood by the Amstrad if preceded by &X.

To output sound to channel A the channel status needs to have just the first bit set i.e. 0000001 (binary) or 1 (decimal). Similarly, to output to channels B or C the channel status needs to be set to either 0000010 (binary), 2 (decimal) or 0000100 (binary), 4 (decimal) respectively.

To send output to several channels the channel status needs to be set to the sum of the status of the corresponding single channels. So, to output sound to all three channels, the channel status has to have all three bits set, i.e. 0000111 (binary) or 7 (decimal).

As you can see from Figure 10.3 there are several other bits that may be set. Two channels may be synchronised (rendezvous) so that they are actioned simultaneously by setting the rendezvous bits of the opposite voice channel in each channel status.

Bit 6 in the channel status is called the hold bit and when set instructs the sound queue of the channels flagged to be frozen, i.e. no more sounds on those particular channels are emitted. The queues may be thawed by the **RELEASE** command which enables the sound processing of specified channels to continue. The value passed is a value between 1 and 7 (3 bits) and specifies the following channels:

- Bit 0: Channel A
- Bit 1: Channel B
- Bit 2: Channel C

In a similar fashion, bit 7 is called the flush bit and when set deletes all sounds queued up in the flagged channels and activates the current **SOUND** statement immediately. The flush bit can also be used to thaw a sound channel but remember that the queued sounds are lost. The sound queues are also flushed when the control character **CHR\$(7)** is printed.

TONE PERIOD

The period of a note gives the pitch characteristics and is inversely proportional to the frequency which was introduced at the start of this chapter. The exact relation between the two is found using the following formula:

frequency = 125000 / period.

Values that we can use for the musical scale are given in Figure 10.4. Remember that notes one octave apart have a frequency ratio of 2; as a result, the period of any note can be easily calculated.

Setting a period of 0 achieves 'white noise' where the emitted waveform is randomly generated; however a noise period parameter (1 to 15) must be passed as the seventh parameter to SOUND.

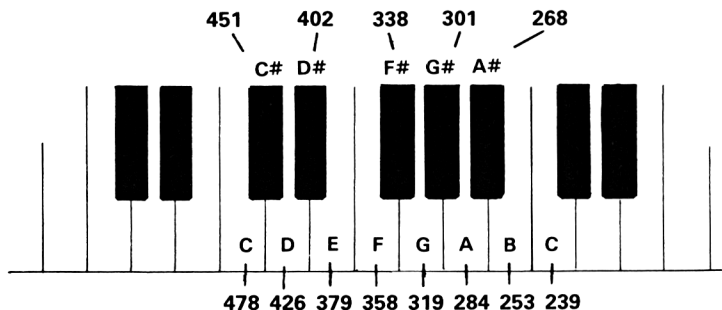


Fig. 104: The Musical Scale - Period values

Although our sound control at this stage is very elementary, program 38 demonstrates a crude key instrument operated by the keys 0 to 9.

PROGRAM 38: SIMPLE KEYBOARD INSTRUMENT

```

10 DIM n(10)
20 ON BREAK GOSUB 80
30 SPEED KEY 5,1
40 FOR j = 1 TO 10 : READ n(j) : NEXT j
50 k$ = INKEY$ : IF k$ = "" THEN 50 ELSE k =
ASC(k$)-47
60 IF k>0 AND k<11 THEN SOUND 1,n(k) ELSE 50
70 GOTO 50
80 SPEED KEY 10,3
90 END
100 DATA 284,478,451,426,402,379,358,338,319,301

```

DURATION

At present, the length of each sound note defaults to 1/5th of a second but this may be altered by specifying a third parameter which gives the length in 1/100th second units. The range of values is -32767 to 32767; negative and zero values have their uses when we study sound synthesis.

VOLUME

The loudness of the sound is given by the fourth parameter and should range from 0 (off) to 7 (full); values 8 to 15 are used with sound synthesis.

SOUND SYNTHESIS

Before we explore the synthesiser capabilities of the Amstrad let us look at what makes the synthesiser so different from other musical instruments.

In Figure 10.2 we saw how the waveform of a sound emitted from a musical instrument differed from a pure note such as the sine wave of Figure 10.1. It is the way these waveforms are modified that gives a sound its specific character, and this modification is called 'envelope control'. The volume envelope is split into three sections: Attack, Decay and Sustain (ASD) as shown in Figure 10.5.

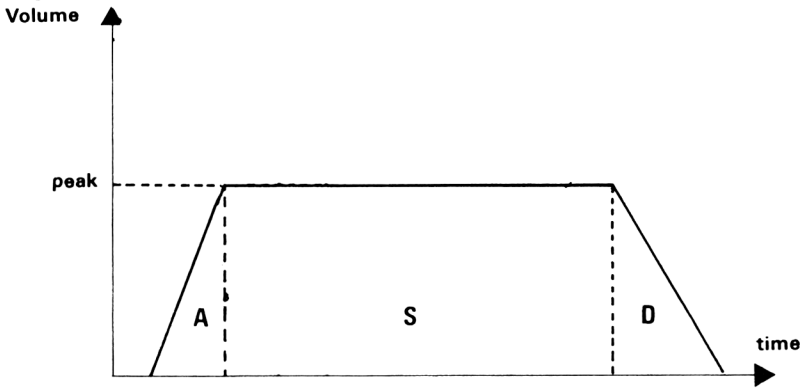


Fig. 10.5: The ASD (volume) Envelope

ATTACK:

This is the length of time it takes to reach the peak of the wave. This is best seen by examining two musical instruments, the piano and the violin. The piano uses a hammer mechanism to create the sound; the hammer strikes the string, the string vibrates and the sound is created. Because the sound reaches its peak virtually as soon as the hammer strikes, the attack is short or steep. The violin, however, requires a bow to be stroked across the strings and the vibration of the string is built up to a peak as the bow moves across. This, therefore, has a long attack.

DECAY:

This is the length of time it takes to travel from the peak of a wave to its end. When the piano hammer has struck the string it continues to vibrate, until the energy created by the hammer is dissipated; therefore the decay is long. A flute's sound, however, ends almost immediately the player ceases to blow; therefore, it has a short decay.

SUSTAIN:

This is the amount of time a wave remains at its peak.

Rapid changes about a frequency, which causes an effect called vibrato, can be obtained by controlling the tone envelope, as for example in Figure 10.6

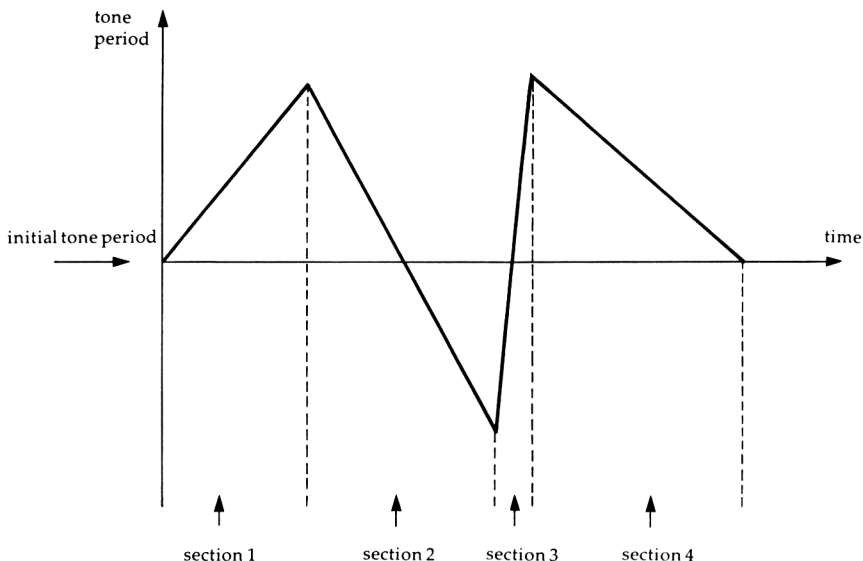


Figure 10.6: The Tone envelope

So to produce characteristic sounds, as opposed to simple beeps, we use envelope shapers to control the shapes of its synthesised waves. How do we program the facilities of envelope control on the Amstrad?

VOLUME ENVELOPE CONTROL

To control the volume envelope, it must first be defined using the statement **ENV**, and then referenced in the **SOUND** statement. In fact, up to 15 envelopes (labelled 1 to 15) are allowed, each of which may have up to 5 sections (attack, delay or sustain periods).

To define the envelope we use:

ENV number, $P_1, Q_1, R_1, P_2, Q_2, R_2, \dots, P_5, Q_5, R_5$

where P_i, Q_i, R_i define the characteristics of each section of the envelope

P_i = Step counts (0 → 127)

Q_i = Step size (-128 → 127)

R_i = Pause time (0 → 251)

Considering an envelope with only three sections:

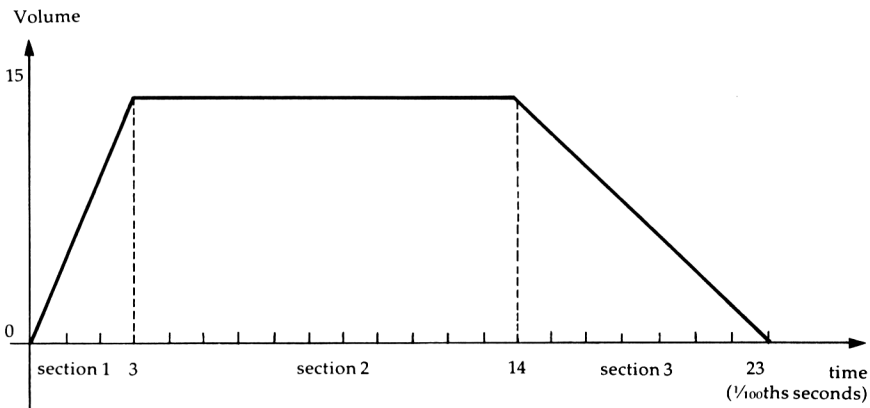


Fig. 10.7

For each section, subdivide it into steps, each a multiple of $\frac{1}{100}$ th seconds. For example we could divide Figure 10.7 as follows:

$$\text{section 1} = 3 * \frac{1}{100}$$

$$\text{section 2} = 1 * \frac{11}{100}$$

$$\text{section 3} = 3 * \frac{3}{100}$$

↑ step counts ↙ pause times

and so the following parameters are required:-

$$P_1 = 3, R_1 = 1; P_2 = 1, R_2 = 11; P_3 = 3, R_3 = 3$$

The step sizes are evaluated by calculating the number of volume units increased within one step count.

$$\text{i.e. } Q_1 = \frac{15}{3} = 5; Q_2 = 0; Q_3 = \frac{-15}{3} = -5$$

Finally, we require a label in the range 1 to 15 so that we can refer to this envelope definition in our **SOUND** statements; e.g. 1.

Thus our **ENV** statement has become

ENV 1, 3, 5, 1, 1, 0, 11, 3, -5, 3

The **ENV** label is passed to the **SOUND** statement as the fifth parameter. The initial volume is controlled by the fourth parameter as before but when an envelope is featured the range is 8 (off) to 15 (full). The number of times the volume envelope should be repeated is passed as a negated value to the third (duration) parameter.

You have probably realised by now, that unless you are an expert on acoustics, the only way of achieving the exact sound you require is to experiment.

TONE ENVELOPE CONTROL

The tone envelope is controlled in a similar fashion to the volume envelope; again five sections can be defined using the ENT command:

ENT number, P₁, Q₁, R₁, P₂, Q₂, R₂,, P₅, Q₅, R₅

where P_i, Q_i, R_i are as before but have the following ranges

P_i = Step counts (0 → 239)

Q_i = Step size (-128 → 127)

R_i = Pause time (0 → 255)

The tone envelope will be repeated for the length of the sound envelope if a negative envelope number is used (though its positive argument is specified in the SOUND statement).

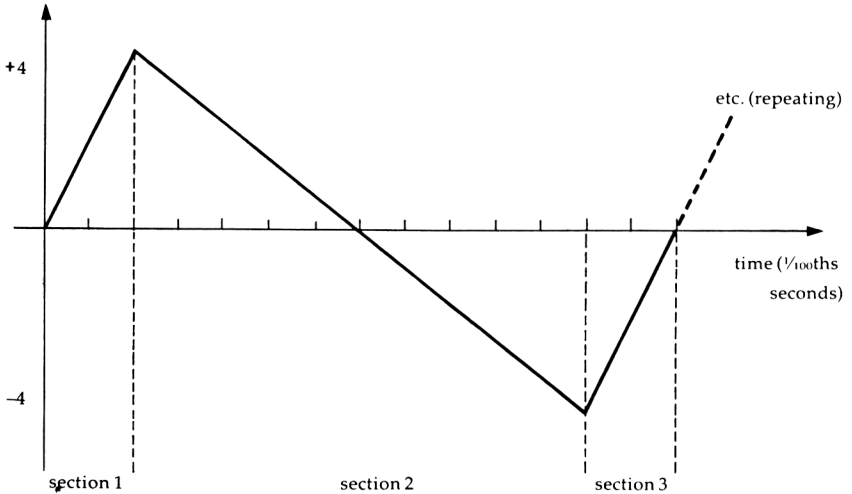


Fig. 10.8

In figure 10.8 we define the sections as:

section 1 = $1 \times \frac{2}{100}$

section 2 = $2 \times \frac{5}{100}$

section 3 = $1 \times \frac{2}{100}$

↑ step counts ↘ pause times

so that:

P₁ = 1 Q₁ = 4 R₁ = 2

P₂ = 2 Q₂ = -4 R₂ = 5

P₃ = 1 Q₃ = 4 R₃ = 2

So, for envelope number 1 we would use

ENT -1, 1, 4, 2, 2, -4, 5, 1, 4, 2

The tone envelope number is the sixth parameter to SOUND.

The next program is written with apologies to all music lovers. Its idea is to play the tune 'Ogonek' but has been written in a skeleton form in order that you can experiment with envelope control. Tone periods for 40 notes are set up in the array n(39). The subscripts of required notes are stored in DATA statements, and, when read, they are played on channels A and B which are synchronised by setting the rendezvous bits. Channel B is organised to play two octaves lower than A by dividing its tone period by four. Having played around with the envelope controllers you will have seen the problems of trying to get a required sound in all its glory!!

PROGRAM 39: OGONEK

```

10 GOSUB 200 : REM initialisation
20 ENV 1,3,20,1,1,0,20,1,-1,1
30 ENT -2,1,1,20,2,-1,50,1,1,20
40 READ v : IF v = -1 THEN END
50 SOUND 16+1,n(v),-1,15,1,2
60 SOUND 8+2,n(v)/4,-1,15,1,2
70 GOTO 40

200 DIM n(39)
210 FOR k = 0 TO 3
220 RESTORE
230 FOR j = 0 TO 9
240 READ v
250 n(k*10+j) = v/(k+1)
260 NEXT j
270 NEXT k
280 RETURN
290 DATA 478,451,426,402,379,358,338,319,301,284

1000 REM example tune
1005 DATA 0,4,9,12,16,4,14,2,6,4,14,4,8,1,14,4,
11,12,14,4,11,12,9
1010 DATA 0,4,9,12,9,4,0,4,16,17,19,8,17,16,19,
8,11,17,17,9,14,9,17,9,17,19,21
1015 DATA 9,19,21,16,4,8,11,16,14,11,8,0,12,24,
24,24,12,23,21,24,12,16,23,17
1020 DATA 9,14,9,17,9,17,9,1,7,19,21,9,19,21,16,
8,12,16,8,12,16,0,8,11,16
1025 DATA 0,4,9,12,16,4,14,2,6,4,14,4,8,1,14,4,
11,12,14,4,11,12,9
1030 DATA 12,16,12,9,21
1035 DATA -1

```

SQ

With the SQ function we can enquire into the state of one of the sound channels. The passed argument is either 1 (channel A), 2 (channel B) or 4 (channel C) and returns a value comprised of the following bits specifying the state of the channel:

7	6	5	4	3	2	1	0
Set if Active	Set if Held	Rendezvous state (as in Fig. 10.3)			number of free entries in queue (0-4)		
128	64	32	16	8	4	2	1

Fig. 10.9: Channel status function

SOUND INTERRUPTS

There will be many occasions when sound is wanted as a background facility to the actual program. Trying to place sound commands within a program such that the sound queues were always primed to ensure a continuous tune/sound would be extremely difficult. We can set up sound interrupts to occur whereby a sound subroutine is executed whenever a gap occurs within the sound queue. As with other subroutines, on completion, control returns to the position at which the interrupt occurred. By controlling all the sound output within this subroutine we have the sound facility running as a background task to the main program. Separate interrupts, of equal priority, are enabled by:

`ON SQ (x) GOSUB line no` where x takes the values 1,2, or 4, as usual depending of the channel.

However, when sound interrupts occur, or the SOUND or SQ statements are executed, then sound interrupts are disabled; thus if sound interrupts are to continue then they should be re-enabled in the sound subroutines.

Our final program demonstrates sound interrupts. The program displays a simple arcade-type screen, on which a number of fearless aliens scroll continually across the screen and approach the earth's surface. You have control over a laser, by using the left and right cursor control keys, and a fire [COPY] button. The objective is to destroy all the aliens before they land, but take care - if you shoot and miss, an extra alien will join those already present.

A sound interrupt is set up to produce a sound of the aliens groaning - a good incentive for you to quickly destroy them all! Note that the interrupt is reset whenever the sound subroutine is called. Another sound is emitted whenever an alien is destroyed. Since we want this sound to be produced immediately, rather than just put on the end of the sound queue, we flush the sound channel by setting the seventh bit of the channel status. As before, the sound interrupt is reset.

PROGRAM 40: ALIEN ATTACK

```
10 ENV 1,1,15,5,1,0,20,5,-3,2
20 ENT 1,10,1,1,40,-1,1,60,1,5
30 SPEED KEY 1,1
```

```

40 INK 0,0 : INK 1,24 : INK 2,15 : INK 3,13,22
50 PAPER 0 : BORDER 0
60 CLS
70 z$ = CHR$(95) + CHR$(95) + CHR$(244) +
  CHR$(95) + CHR$(95)
80 p = 1 : z = 20 : pp = 800
90 t$ = ""
100 FOR k = 1 TO 20 : t$ = t$ + SPACE$(1) +
  CHR$(225) : NEXT k
110 PLOT 0,16 : DRAWR 639,0,2
120 ON BREAK GOSUB 800
130 EVERY 500 GOSUB 400
140 ON SQ(1) GOSUB 1000
150 FOR j = 1 TO 40
160 PEN 3
170 LOCATE 1,p : PRINT MID$(t$,j,40);MID$(t$,1,j-
  1)
180 PEN 2
190 LOCATE z,24 : PRINT z$
200 k$ = INKEY$ : IF k$ = "" THEN 240 ELSE k =
  ASC(k$)
210 IF INKEY(8)<>-1 AND z>3 THEN z = z-2
220 IF INKEY(1)<>-1 AND z<36 THEN z = z+2
230 IF INKEY(9)<>-1 THEN GOSUB 600
240 NEXT j
250 GOSUB 150

400 p = p+1
410 pp = pp*0.9
420 IF p<32 THEN 460
430 LOCATE 12,25
440 PRINT "Invasion complete";
450 GOTO 800
460 LOCATE 1,p-1 : PRINT SPACE$(40)
470 RETURN

600 v = j+z+1 : IF v>40 THEN v = v-40
610 q = ASC(MID$(t$,v,v))
620 MOVE 16*z+22,25 : DRAWR 0,380,1
630 FOR k = 1 TO 50 : NEXT k
640 DRAWR 0,-380,0
650 IF q<>225 THEN 740
660 LOCATE z+2,p : PRINT CHR$(238)
670 SOUND 128+1,1000,40,15
680 ON SQ(1) GOSUB 1000
690 MID$(t$,v,v) = SPACE$(1)
700 IF t$<>SPACE$(40) THEN 750
710 LOCATE 12,25
720 PRINT "Invasion repelled";
730 GOTO 800
740 MID$(t$,v,v) = CHR$(225)
750 RETURN

```

```
800 PRINT CHR$(7)
810 INK 1,24 : PEN 1
820 SPEED KEY 10,3
830 LOCATE 1,1
840 END

1000 SOUND 1,pp,-5,15,1,1
1010 ON SQ(1) GOSUB 1000
1020 RETURN
```































































































































That, then, concludes both this chapter on sound and the whole book. Hopefully, as you have worked your way through all ten chapters, you have increased your repertoire of programming skills and improved your technique. If this is the case, then the aim of this book has been achieved.

APPENDIX A

The ASCII Character Set

Remember that 0 to 31 are control characters and so are only printable if preceded by CHR\$(1).

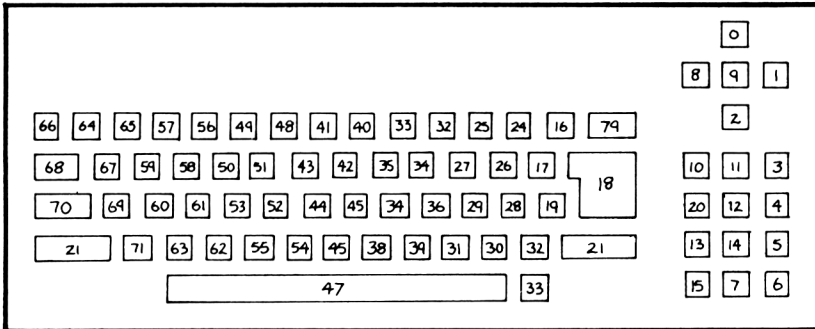
	0		1		2		3		4
	5		6		7		8		9
	10		11		12		13		14
	15		16		17		18		19
	20		21		22		23		24
	25		26		27		28		29
	30		31		32		33		34
#	35	\$	36	%	37	&	38	'	39
(40)	41	*	42	+	43	,	44
-	45	.	46	/	47	0	48	_	49
2	50	3	51	4	52	5	53	6	54
7	55	8	56	9	57	:	58	;	59
<	60	=	61	>	62	?	63	@	64
A	65	B	66	C	67	D	68	E	69
F	70	G	71	H	72	I	73	J	74
K	75	L	76	M	77	N	78	O	79
P	80	Q	81	R	82	S	83	T	84
U	85	V	86	W	87	X	88	Y	89
Z	90	[91	\	92]	93	↑	94
	95	`	96	a	97	b	98	c	99
d	100	e	101	f	102	g	103	h	104
i	105	j	106	k	107	l	108	m	109
n	110	o	111	p	112	q	113	r	114
s	115	t	116	u	117	v	118	w	119
x	120	y	121	z	122	{	123		124
}	125	"	126		127		128		129

	130		131		132		133		134
	135		136		137		138		139
	140		141		142		143		144
	145		146		147		148		149
	150		151		152		153		154
	155		156		157		158		159
	160		161		162		163		164
	165		166		167		168		169
	170		171		172		173		174
	175		176		177		178		179
	180		181		182		183		184
	185		186		187		188		189
	190		191		192		193		194
	195		196		197		198		199
	200		201		202		203		204
	205		206		207		208		209
	210		211		212		213		214
	215		216		217		218		219
	220		221		222		223		224
	225		226		227		228		229
	230		231		232		233		234
	235		236		237		238		239
	240		241		242		243		244
	245		246		247		248		249
	250		251		252		253		254
	255								

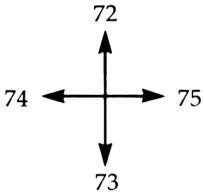
APPENDIX B

Key Handler Codes

(as used with INKEY)

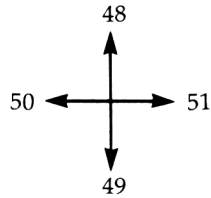


Joystick 0



fire 1 : 77
fire 2 : 76

Joystick 1



fire 1 : 53
fire 2 : 52

APPENDIX C

COLOUR CODES

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		

APPENDIX D

ERROR CODES

ERROR CODE	ERROR
1	Unexpected NEXT
2	Syntax error
3	Unexpected RETURN
4	DATA exhausted
5	Improper argument
6	Overflow
7	Memory full
8	Line does not exist
9	Subscript out of range
10	Array already dimensioned
11	Division by zero
12	Invalid direct command
13	Type mismatch
14	String space full
15	String too long
16	String expression too complex
17	Cannot CONTinue
18	Unknown user function
19	RESUME missing
20	Unexpected RESUME
21	Direct command found
22	Operand missing
23	Line too long
24	EOF met
25	File type error
26	NEXT missing
27	File already open
28	Unknown command
29	WEND missing
30	Unexpected WEND

INDEX

Note: definitions of BASIC keywords are all contained in Chapter 1. Entries in the index are selected references which illustrate specific uses of the keywords.

ABS	53	DRAW	111
AFTER	70	DRAWR	111
Alphabetical sort	100	Edges	85
Animation	33	ENT	126
Arrays	75	ENV	124
ASC	17	Envelope (sound)	123
ASCII	17	ERL	72
Assignment statements	7	Error taps	72
ATN	58	EVERY	70
Attack	123	EXP	52
Banked ROMs	67	Exponential	46
BIN\$	64	Fibonacci	61
Binary	64	FIX	48
Bit	64	FRE	67
BORDER	41	Frequency	120
BREAK	72	Functions statements	10
Bubble sort	96	GOSUB	60
Byte	64	Graph	85
Cassette file	102	Graphic statements	10
Channel	121	Heuristic	91
CHR\$	17	HEX\$	65
CINT	48	High level	4
Circular List	82	High resolution	110
CLOSE IN	103	HIMEM	67
CLOSEIN	103	I/O functions	14
CLOSEOUT	103	INK	36
Colour	36	INKEY	28
Compiler	5	INKEY\$	27
Control characters	32	INPUT	26
CONTROL key	26	Input/Output statements	9
Control statements	7	Insertion sort	97
COPY	39	INT	48
COS	57	Integer	46
CP/M	2	Interpreter	5
CREAL	48	Interrupt	72
Cursor keys	32, 38	KEY DEF	29
DATA	21	LEFT\$	18
Decay	123	LEN	18
DEFINT	47	LINE INPUT	27
DEFREAL	47		
DEFSTR	47		
DIM	75		

Linked List	79	RESUME	72
LNR	90	RIGHT\$	18
LOCATE	35	Ring	82
LOG	53	RND	49
LOG10	53	ROM	2
LOWER\$	20	ROUND	48
LRN	90		
		SGN	54
Master file	102	Shell sort	98
Mathematical functions	12, 52	SHIFT key	26
MAX	48	Simulation	61
MEMORY	67	SIN	57
Memory map	66	Sorting	95
Menus	37	SOUND	120
MID\$	19	Sound interrupt	128
MIN	48	SPACE\$	20
MODE	36	SPC	34
		SPEED INK	41
NLR	90	SPEED KEY	29
Nodes	85	SQ	127
Numeric functions	12, 47	SQR	53
		Stack	83
ON BREAK	72	STR\$	20
ON ERROR	72	Streams	41
OPENIN	102	String	17
OPENOUT	102	String functions	12
ORIGIN	114	STRING\$	20
		Structural programming	37
PAPER	36	Subscript	75
PEEK	67	Sustain	124
PEN 21	36	SYMBOL	108
Pitch	119	SYMBOL AFTER	108
Pixel	35	System commands	6
PLOT	111	System functions	14
Pointer (in lists)	79	System software	2
POKE	67		
Pop (stack)	83	TAB	34
PRINT	17	TAG	116
PRINT	30	TAGOFF	116
PRINT USING	31	TAN	58
PRINT zones	31	TEST	115
Push (stack)	83	TESTR	115
		Three dimensions	113
Queue	84	TIME	69
Quick sort	99	Transaction file	102
		Traversal (tree)	90
RAM	2	Tree	87
Random numbers	49	Trigonometric functions	56
RANDOMIZE	50		
Real	46		
Recursion	60	UPPER\$	20
RELEASE	121	User defined characters	107

VAL 20
Variance 49
Volume 119, 123

White Noise 122
Windows 42

XPOS 111

YPOS 111

Z80 processor 1
Z80A 1

Join the Winning Team!

Sigma Press have an enviable reputation for producing a top-class range of computer books. Soon there'll be over a hundred, spanning all aspects of computing from BASIC to Artificial Intelligence. We concentrate on the upper-end of the market, so that even our books on very popular micros tend to emphasise techniques and applications other than collections of games.

Perhaps you could write a book for us. If so, write to the publisher, Graham Beech at the address below for full details - including a helpful advice sheet on how to submit a synopsis of your book in the best possible way.

We are interested in **all** aspects of computing - after all, we ourselves are computer enthusiasts! Dealing with Sigma is dealing with someone who understands **your** subject - someone who can advise you on what to emphasise in your book to make it a success. That's so much better than dealing with a multinational corporation!

Sigma work **with** technology, not against it. So, we welcome authors who submit manuscripts on word-processor disks from which we typeset automatically. This is just one of the methods we use to get your book on the market in the shortest possible time.

Marketing and distribution is, of course, of the highest calibre - through John Wiley & Sons Ltd., who are a well-known international publisher and with whom we have worked for several years with great success.

So, when you want your book to be a success, just write to:

Graham Beech
Sigma Press
5 Alton Road,
Wilmslow,
Cheshire.
SK9 5DY.

or, telephone: 0625 - 531035

Easily the Best - for you and your 464!

A book has got to be really good to gain approval from AMSOFT, the computer products division of AMSTRAD. Just one glance through this spectacular new book by our top-selling author will tell you why its so good, and why every 464 owner needs it by his computer as a constant companion.

The book assumes that you have got your 464 working, and have already done some simple programming. But, even at a simple level, the organisation of the book is attractive as it opens with a description of how the 464 works, how it communicates with external devices, and a quick summary of BASIC. There is a comprehensive reference section for you to find an explanation of any BASIC command or keyword in the Amstrad's repertoire.

Other important sections of the book cover: *Strings and characters; Input/Output; Arithmetic; Memory Map; Time, Clocks and Interrupts; Data Structures; Data Processing; Graphics; Sound.*

The book contains FORTY complete programs ready-to-run on the 464, ranging from very short ones to demonstrate how your 464 works, through to large, challenging programs that are themselves worth the cover price.

Programs are included for:

Code breaking;
Simple databases;
Sorting information into any order;
Drawing 3-D Pictures;
Business applications.

And, most spectacular of all, you'll find arcade-style space-games and a complete SOUND SYNTHESISER program.

*For the best in personal computing
read a Sigma book!*

GB £ NET +006.95

ISBN 1-85058-014-6



Sigma Press
5 Alton Road
Wilmslow
Cheshire
SK9 5DY

THE AMSTRAD
ADORDER
CPC
MARKET
TAMRISON

MARKET
TAMRISON



Document numérisé avec amour par

AMSTRAD

CPC 

MÉMOIRE ÉCRITE



<https://acpc.me/>