

INTRODUCTION TO THE FIRST EDITION

This book was written to satisfy the need for a suitable volume to accompany a twenty-six week Evening Centre course in Basic Computer Programming and the information contained herein is based on the author's tutorial notes and deals in detail with the essentials of standard core BASIC

Short demonstration programs known as "lists" are used during tutorials to illustrate and explain the current topic under discussion. In a few instances, these demonstrations are "run" on a computer during the tutorial session. If this is not the case, the programs are discussed in detail during the tutorial and students are then advised to type in and run the programs at the end of each tutorial session to help with an understanding of the new topic.

Each topic is dealt with by a separate chapter within the book and associated with each chapter there are generally a number of "set" projects. Practical periods follow each tutorial and these projects are designed to encourage the use and provide a better understanding of the newly introduced subject material. The practical periods may occupy one, two or three weeks depending upon the topic and the number of projects to be dealt with. Students are encouraged to complete as many of these projects as possible, either at home on their own computers if available, or at the Evening Centre during the practical sessions. Many of the projects will be dealt with in detail by the tutor to ensure that important aspects are fully explained.

INTRODUCTION TO THE SECOND EDITION

Subtle changes in the course, and new advances in technology have highlighted the need for a new revised edition. Although the book still deals with a "core" BASIC, the dialect for the course has been standardised on BBCBasic as I find most students who have a home computer either have, or can easily acquire, a BBCBasic Interpreter (and it is my personal favourite!).

The demand for the course is such that it now takes place at more than one centre, and not only in the evenings. The course will not necessarily cover every topic in the book, as many people do not have access to a home computer and value all the practical time they can get at the centres. However, for those who wish to take programming further, a companion Workshop has been started.

BASIC

The BASIC language was introduced at Dartmouth College, New Hampshire, USA in 1964. It was originally designed as a simple "high level" computer language intended for the use of absolute beginners and contains many easily understood English words such as "PRINT" and "END". It proved so popular that where a programming language is supplied with a

computer it is almost always a "dialect" of BASIC, which stands for "Beginners All-purpose Symbolic Instruction Code". It has been much modified, and now incorporates many more instructions than the original. At one time of day every different make of home computer had its own version, or dialect, of Basic, and the methods of loading and saving programs also varied which meant that programs written on one computer could not be used on another.

In recent years, much improved versions of the language, addressing and overcoming many of the limitations of the original, have been issued. Programs written in these can now be compiled, giving a "stand-alone" program which can be used without ever revealing the language the program is written in.

There are now two dialects of BASIC that predominate: Microsoft Basic, mainly intended for the IBM PC family of computers and their compatibles, and BBCBasic as used by Acorn in all their machines, and widely used in schools. Both versions can be obtained in forms which can be used on computers other than for which they were written, but the version concentrated on in the course and this book is BBCBasic, since Acorn computers are used where it is taught. One area in which there are major differences in syntax is in the field of graphics, and thus no attempt is made to introduce the subject in this book.

Whatever version of BASIC be used, there are very formal rules of syntax (grammar), and a computer will not understand a particular command or statement unless it be entered in the correct manner. A simple spelling mistake will cause the computer to stop execution of a program and supply an error message indicating what is wrong and at which line. An example of this is the PRINTTAB statement. Some computers insist on PRINT TAB, whilst others will only recognise PRINTTAB!

All BASICs can handle words, symbols, and numbers, and generally have a built-in ability to carry out "Library Functions", which include standard mathematical and trigonometric functions and text handling.

No computing course can be non-mathematical, but as far as possible only a simple everyday maths ability is required in order to be able to benefit from, and handle effectively, the requirements presented by this book and course. Where a particular formula be required, it is given and students are only expected to know how apply and express it in BASIC. The normal rules of mathematics are in any case discussed and revised at an early stage.

The need for program planning cannot be over-emphasised. This is dealt with in detail in one of the early chapters, but suffice to say at the moment that students should have a supply of quarter-inch squared paper. This will be used to plan displays, and to assist in writing programs, and may be likened to the coding forms used by professional programmers.

A general word about computer languages. The language used by the computer itself (machine code) is a "low level" language and very fast. Computers can be directly programmed in machine code, or via an assembler, but it is difficult and prone to errors. BASIC is a "High level" interpreted language which permits the writing of programs in an almost English-like style. When a BASIC program is run or executed by the computer, each program line is first converted to machine code and then executed. This has to happen every time the program is run, and thus results in interpreted BASIC being noticeably slower than many other languages. At

the level of programming taught in this book this will not be very noticeable.

Where speed of operation is important, either a compileable BASIC or another language is usually used. Programs written in these are written to produce a source code, which will not be run "as is", but will be converted, or "compiled" to an object code which will run directly on the computer.

BASIC is still the single most widely-used computer language, and many programmers use nothing else. It provides a good grounding for those who may wish to go on to other more specialised languages. The modern compiled versions have ensured its progression and adaption to the new generation computers, and it will continue to be the first choice for many programmers, both serious and amateur, for the foreseeable future.

Copyright 1988, 1991, Bill Hayles

CONTENTS

HISTORY & MICROCOMPUTERS - I	I
COMPUTER ARITHMETIC - II	II
PROGRAM PLANNING - III	III
COMMANDS & STATEMENTS - IV	IV
NUMERIC VARIABLES & THE LET STATEMENT - V	V
THE PRINT & PRINT TAB STATEMENTS - VI	VI
STRING VARIABLES - VII	VII
THE INPUT & GOTO STATEMENTS - VIII	VIII
THE DATA, READ & IF-THEN STATEMENTS - IX	IX
THE REM STATEMENT, ASCII CODES AND THE CHR\$ AND ASC FUNCTIONS - X	X
THE RESTORE STATEMENT & MULTISTATEMENT LINES - XI	XI
ROUNDING & RANDOM NUMBERS - XII	XII
COUNTERS - XIII	XIII
LOOPS & THE FOR-NEXT STATEMENT - XIV	XIV
ARRAYS & ARRAY VARIABLES - XV	XV
LOGICAL OPERATORS & THE EXTENDED IF-THEN STATEMENT - XVI	XVI
THE ON GOTO & GOSUB STATEMENTS - XVII	XVII
LIBRARY FUNCTIONS - XVIII	XVIII

CHAPTER I

A BRIEF HISTORY OF COMPUTING

Computing really began a little over three hundred years ago with the introduction in 1642 of the first mechanical addition/subtraction machine. The designer and maker of this machine was Blaise Pascal, after whom a modern computing language has been named. In 1843, Charles Babbage introduced the 'Analytical Engine', a complex mechanical system which in it's design anticipated every section of the modern digital computer. The first electronic computer was introduced in 1941 by Konrad Zuse and in 1944, the IBM Mk I electromechanical computer appeared on the market. This computer was a mass of gears and electronic relays, was slow in operation, measured 51 feet long, 8 feet wide and weighed 5 tons!

In 1945, the first fully electronic computer arrived. This was known as ENIAC (Electronic Numerical Integrator And Computer) and it was contained in a room the size of a small house. It used literally hundreds of control switches, had more than 18,000 electronic valves and weighed 30 tons! In 1952 alone this machine suffered 19,000 valve failures. The power requirement was 200,000 watts (200 Kw) and the machine was finally retired in 1958.

The arrival of transistors changed the situation considerably as far as computers were concerned. These replaced valves, were much smaller in size, offered greater reliability with a smaller power requirement and were cheaper. The early 1960's saw the arrival of integrated circuits (I.C.'s). These were small and contained the equivalent of a dozen or so transistors etched on the surface of a small chip of silicon. (Today's equivalent to these early 'chips' contain hundreds of transistor equivalents and are multi-layered.)

The first 'Mini' computer was produced in 1965 by the Digital Equipment Corporation and was known as the PDP-8. It was the size of a small filing cabinet and cost about £10,000. Many more mini's followed this machine.

1971 saw the discovery and introduction of the first microprocessor. The microprocessor is the complete operating system for a microcomputer on a silicon chip much smaller than the key of a pocket calculator. The microprocessor was, in fact, invented for calculator use. The Intel Corporation were trying to design a single I.C. that would add, subtract, multiply and divide. They came up with something considerably more versatile, the microprocessor. Microprocessors can be programmed to function as calculators, to control traffic lights, time ovens and many other useful functions. If the microprocessor is used as the central operating system and if a memory capability is incorporated with the microprocessor then we have the basis of the microcomputer. Microprocessors originally cost several hundred pounds each, now they may be bought for less than ten pounds. They are small, versatile and were first used in microcomputers in 1975.

The story today is different from that of ten years ago. Computing is advancing at a startling rate and each year sees the introduction of new ideas and equipment. The range of

personal computers and the uses to which they are put is now very extensive and their price bears no relation to that of even a few years ago. For home computing there are many cheap and very efficient systems available and in business, the IBM PC system has set an industry standard which should sustain for some time yet, as well as becoming the choice for many as their home machine. The INTEL 80386 and 80486 chips which offer multitasking facilities will probably become the standard. Operating speed appears to be the factor which most manufacturers are concerned with and faster processors have changed the face of personal computing to the extent that many people now own home computer systems much more powerful than the early minicomputers.

The following are the mains components of a micro-computer.

The CPU or Central Processing Unit - The circuitry in this section accesses data from memory, works on it and transfers it back to memory again. It also controls all other system components including input-output and mass-storage. The heart of the CPU is a microprocessor 'chip', sometimes called a microprocessor unit or MPU. Some well known chips are the Intel 8080, 80286 and 80386, and the Motorola 6502 as used in BBC machines.

RAM or Random Access Memory - This is where the programs and data which we feed the computer are stored. All information is stored in a binary form, a series of 0's and 1's. Each 0 or 1 is known as a 'BIT'. Inside the microcomputer the binary bits of information are usually organised in groups of eight bits. Eight bits is known as 1 'BYTE'. RAM memory size varies from one computer to another and it is not uncommon to have RAM memories of 640K, 1024K or 4096K. 'K' represents 1024 bytes therefore a RAM memory of 640K will provide storage for 655,360 bytes (5,242,880 bits).

In order to picture what 655,630 bytes means, assume that each character stored, for example the letter A, requires one byte. This is not strictly true particularly for number storage range but it does provide some guide as to memory capacity.

Random Access Memory is termed as such because of the computer's ability to go directly to any byte or position in this memory to either 'read' its contents or to 'write' or store new information at that point. The computer can rapidly access bytes of information at random, there is no need to start at the beginning of the memory and to sequentially search all the way through for some item of information or location.

RAM is generally 'volatile', this means that when the computer is switched off the contents of memory are lost. It is also erasable: memory content can be wiped out to enable new programs or data to be stored. Together with this, the contents of memory can also be modified or overwritten as required. Some more recent computer systems provide battery 'back-up' of RAM, thus avoiding the loss of information stored when the computer is turned off.

ROM or Read Only Memory - Individual bytes of data in this section of computer memory may be 'read' but no 'writing' is permitted. ROM is used to store programs or data that will never be changed. It is 'non-volatile', meaning that the information stored therein is permanent and will not be lost on power-down. ROM is used in computers to store programs used by the CPU to control system components and in computers with a resident BASIC language, this is where the BASIC interpreter resides. You will recall from the introduction that this interpreter is used by the system to convert the

BASIC instruction into machine language, the low-level language used by the computer itself.

Some ROM's can be erased and these are known as EPROM's, Erasable Programmable Read Only Memory. EPROM's can be erased by ultraviolet light and then reprogrammed as necessary. An EPROM in a computer system will allow a special program or set of programs to be utilised by the system, these generally being intended for very specific operations such as equipment control or data logging. In everyday life EPROMs are used to control traffic lights, ovens and vehicle electrical systems amongst numerous other applications.

Some computers have no resident language and BASIC, for example, must be 'loaded' each time the system is used. In this instance, the BASIC interpreter usually occupies an area of RAM memory which is protected against erasure while the computer is on.

ROMPACKS are another way in which programs may be made available to the computer. In many cases, these are simply plugged in to a socket provided

and are immediately operative without the need to 'load' the programs concerned. The BBC system utilises this option, which is less used these days.

VDM or Video Display Module - This converts the electronic signals from the computer into 'dot patterns' which show up on the VDU or Video Display Unit. These dot patterns form letters, numbers, symbols or graphic shapes. Many computers have dedicated VDU's supplied with the computer system itself but home computers are often supplied with a VDM which permit the use of a television to display information. Generally, the resolution obtained from a TV display is not as good as that produced by a dedicated VDU. High resolution versions of dedicated VDU's provide an extremely readable display which is often required when graphic work is being undertaken. In the past, the computer system and VDM were only capable of providing a monochrome display but many modern-day computer systems offer colour and high resolutions.

INPUT-OUTPUT COMPONENTS

These are often called input-output devices. An input device is used to transmit both programs and data to be manipulated by such programs, into the computer.

The most common microcomputer device is the keyboard. This looks very much like a typewriter keyboard but generally has a larger number of keys, some with special functions so that symbols, graphics and maths formulas can be typed. The equivalent to the carriage return on a typewriter is provided by a key generally marked 'RETURN' or 'ENTER'. A TAB is also provided and this works in very much the same way as the conventional typewriter TAB. Some keyboards have a set of cursor control keys, these being used to position the cursor correctly on the screen and in some cases in conjunction with other control keys for the purposes of editing text on the screen.

When a key is depressed on the keyboard, a unique binary code is generated. For example, when you press the A key, the code 01000001 is sent to the computer. This code can be sent in either of two ways depending upon the system. If each bit is sent simultaneously on a separate wire then the connection or transfer is known as parallel and when the bits are sent sequentially one after the other on a single wire the connection or transfer is known as serial.

Computer peripherals such as printers and plotters are also designed for use with either serial or parallel connection and this has to be considered when purchasing accessories for your computer. There are two standards commonly in use, for serial connection the standard is known as RS232 and for parallel, Centronics.

In a computer, bits are represented by two different voltages, usually called 'high' for a 1 bit or 'low' for the

0 bit. The 1 and the 0 are also referred to as 'bit on' and 'bit off'. One bit gives only two possible codes, 1 and 0 (2^1). With eight bits, there are 2^8 or 256 different possible codes. A number of these codes are used to account for the alphabet (lower and UPPER case), numbers, special symbols and the codes required for control functions such as carriage return and cursor movement.

Most manufacturers use the same basic codes as far as alphanumeric and standard control functions are concerned. However, different manufacturers often utilise the remaining codes in different ways to provide for example, graphic characters, reverse field characters and various other functions.

The standard code system is known as the American Standard Code for Information Interchange (ASCII) and on the next page is a table of some of the codes for the upper case alphabet showing the binary code, the character represented or produced and the decimal equivalent to the binary code.

<u>VALUE</u>	<u>BINARY CODE</u>	<u>CHARACTER</u>	<u>DECIMAL</u>
	01000001	A	65
	01000010	B	66
	01000011	C	67
	01000100	D	68
	01000101	E	69
		
	01011010	Z	90
	00100000	SPACE	32
	00001101	RETURN (CR)	13

For the purposes of programming in BASIC, you will not need to know anything about binary codes as this is taken care of by the electronic circuitry and components which comprise the computer system. It is sometimes useful to know the decimal value for different characters as there are statements in BASIC which can utilise this information. The decimal value for any character can be found from the ASCII table usually supplied in the computer systems' manual.

Conversion of binary to decimal is quite simple and some examples are shown below for an 8 bit code:-

128	64	32	16	8	4	2	1	
0	0	0	0	1	0	0	0	= 8 decimal
1	0	0	0	0	0	1	0	= 130 decimal
0	1	0	0	0	0	0	1	= 65 decimal
1	0	1	1	0	0	0	0	= 176 decimal
1	1	1	1	1	1	1	1	= 255 decimal

Other input devices are the mouse, joystick, light pen and scanner. The mouse, a hand held device whose movements on a flat surface are echoed to the screen, is becoming increasingly used both as a simple to understand tool by newcomers to computers and as an aid to CAD, or Computer Aided Design. The joystick is much used by computer games players. The main use of light-pen technology is nowadays confined to use in connection with bar codes and computerised retailing and stock control, its use with personal computers having largely replaced by the mouse and the scanner. The scanner is run over a picture, or some text, which it will convert into a screen image and computer file, which can then be modified or imported into a document using a desktop publishing program. In the case of text, special programs

can perform Optical Character Recognition, and convert the image into an ASCII text file.

Printers and plotters are both output devices and the necessary code conversion circuitry needed to convert the electronic signals from the computer system is built in to both of these devices. As previously mentioned, these may be connected to the computer via either a parallel or serial interface. Printers, or 'line printers' as they are sometimes known provide 'hard-copy' or printed output from the computer, this output might be a copy of a resident program (listing) or data or information produced by running such a program. The quality of the output varies from one printer to another, apart from 'draft quality' some provide what is known as 'near letter quality' (NLQ) output and others 'letter quality' (LQ). This particularly applies to dot-matrix printers where the characters printed are produced by a multi-wire print head. This head prints characters and shapes made up of a series of dots, these characters being of higher quality, clearer and easier to read the more wires the print head uses. Dot-matrix printers are quite capable of producing good quality

graphic output and because of the vast number of characters that can be formed with a multi-wire print head, are very useful when using software such as a Desk Top Publisher (DTP) or graph drawing software.

Daisy wheel printers produce high quality text output but cannot be used for graphic output of any sort and are therefore becoming less frequently used. They are only good where the main use of the computer system is Word Processing (WP) and different varieties of print wheel can be obtained and used to provide different output styles. Their disadvantage is speed and the noise produced during printing, although the latter can be overcome by enclosing the printer in a sound-proof enclosure. Such enclosures can also be used for dot-matrix printers which can also produce a noise problem.

Ink Jet printers form characters in a different way to both of the above as do Laser printers and both of these are almost silent in operation. The quality of the output from a laser printer is extremely high and this type of printer is gradually replacing the others in many situations.

Plotters are used for very high quality graphic output, often employing multicoloured pens to provide attractive and clear presentations. High resolution work is possible with the better quality plotters and many are capable of directly producing transparencies for overhead projection.

A 'modem' or modulator-demodulator may be used to transfer to or receive data as a serial transfer from another computer by means of the public telephone system. The modem generally contains the necessary circuitry to convert the signals from the sending computer into audible tones which can be sent down the telephone line and conversely, to convert the incoming audible tones received from another computer systems into signals which will be understood by the receiving computer. Modern modems plug directly into the standard telephone socket and then to the computer, leaving the telephone still available for normal use.

Networks are another way of connecting computer systems. A local area network (LAN) can be used to connect a series of systems so that software, data and peripheral equipment can be shared. For example, the BBC microcomputer system may be networked and this is often found in schools so that hardware costs can be reduced and teaching material made available to all users. The BBC network system is known as ECONET and gives the tutor the ability to look at the work being done on any terminal and if necessary to send messages down to the terminal concerned.

MASS STORAGE COMPONENTS

Another name for this is OFF-LINE storage. Any programs or data in RAM memory will be lost when the computer is turned

off, unless the system has a built-in battery back-up. Mass storage avoids such a loss and the need to retype a program when next required or to recreate any data lost on power-down by providing a medium on which such programs or data may be saved or stored for later use. The two most popular forms of mass storage are hard disk and floppy disk systems. Both employ magnetic recording techniques such as that used for audio recordings of music and speech and utilise an interface which converts the bit-signals from the computer into audio tones which are recorded on the disk. Conversely, when reloading programs or data, the same interface converts the audio tones received from the disk and converts them back into bit-signals which can be understood by the computer. Floppy disk systems are commonly used for home computers since they offer a cheap and readily obtainable way of providing this storage facility. Standard disks are now cheaply available. Floppy disk storage has taken over from cassette storage. The transfer speed when loading or saving a file is much faster, for example a program file which takes one minute to load from tape may well load from disk in just a few seconds. Part of the reason for this is due to the actual data transfer speed but it can also be attributed to the way in which files are

saved to and loaded from a floppy disk. On each disk there is a 'directory track' which is used to catalogue the files stored on that disk together with their locations. Each file has a name as with cassette systems and when that name is used to request a load from disk, the system looks in the directory to find the location of the program, proceeds directly to that location and immediately loads the program. If a program name which does not exist is entered, the system informs the user that the file is not available.

Floppy disks as purchased have to be prepared before use. During this operation which is known as 'formatting', a series of invisible circular tracks where files will be stored are written on the disk and a directory is prepared at the same time. Depending upon the computer system, there may be 40 tracks or 80 tracks, each being divided into sectors, and either one or both sides of the disk may be used. Together with this, storage on the disk may be in either single or double density, this meaning that twice as much may be stored in the same space. An 80 track disk, double-sided and double-density will store 720 Kbytes and quad-density disk, which are available for some systems, will store 1.2 Mbytes (million bytes) of data or information. Program and data files may be stored on disk as one contiguous unit or may be split into several parts if the space situation requires it. This situation is totally invisible to the operator and is dealt with by the disk operating system (DOS).

Floppy disks are loaded into a disk drive when required and removed at the end of the operating session before the computer is switched off. They comprise a protective casing inside which the disk itself is free to rotate, this being driven by the disk drive motor at about 300 revolutions per minute. A drive spindle clamps onto the central hole in the disk and the 'read-write' head contacts the disk surface through a slot in the casing. This head actually rests on the surface of the disk and may be likened to the pick-up of a record player, the disk is rotating and the head can move from the inside track to the outside track of the disk so that any particular area of the disk can be covered. The directory normally occupies the central track and all other tracks are used for data storage. Because of the fact that floppy disks are removable from the disk drive, they can be damaged easily and must therefore be treated with care. Bending, leaving near a magnetic field or allowing dirt or liquids onto the disk surface can cause data loss and they should always be stored carefully in sturdy storage boxes. Together with this, the fact that the read-write head actually contacts the disk surface means that they will eventually wear out and copies (backups) should be frequently made so that duplicate files are always available when the inevitable happens. Floppy disks come in two common sizes, 5.75" and 3.5".

Hard disks are a permanent part of the computer system and are mounted in a dustproof enclosure. Their capacity is greater than that of a floppy disk, a common capacity being 20 Mbyte, although capacities of 100mb or more are becoming increasingly common. The read-write head floats on a cushion of air just above the disk surface and they are therefore much more reliable than floppy disks. Access and transfer speeds are much faster than for floppy disk systems and they are almost essential for business applications where the programs and data to be stored are extensive. Like all electronic systems, they can fail and data stored on these disks should also be periodically backed up. A device known as a tape streamer is often used for such an operation.

A hard disk system generally comprises two or more disks mounted one above the other on a common drive spindle and there are multiple read-write heads. The more disks used to make up the system, the greater the overall capacity of the system as a whole.

CHAPTER II

COMPUTER ARITHMETIC

Three types of numeric constants or numbers are allowed in BASIC:

<u>TYPE</u>	<u>EXAMPLES</u>
Integer	34, 1, -16, 0, 3752
Decimal	34.67, -16.016, 100.56
Exponential	21E+05, 2.54E-02

Integer constants are whole numbers, or numbers which do not contain a decimal point.

Decimal constants are numbers which contain decimal points.

Exponential constants are the BASIC way of expressing numbers in a 'scientific notation', a way to display or store numbers in a small space. Please note that the computer denotes powers by the carat (^) sign, i.e. 4^2 is shown as 4^2 and this is how we shall show it in the book.

E.g. 21E+05 is the same as 21×10^5 which equals 2,100,000.

Conversion with a positive exponent is achieved by moving the

decimal point, in this example, five places to the RIGHT.

and 2.54E-02 is the same as 2.54×10^{-2} which equals 0.0254.

Conversion with a negative exponent is achieved by moving the

decimal point, in this example, two places to the LEFT.

It is important to be able to recognise the values of exponential constants as the computer will occasionally display results in this form. The point at which different computers resort to exponential number displays depends on the make of the computer, but generally very small or very large numbers will be displayed in this form. Together with this, different computers sometimes display exponential constants in different ways. For example, on some computers 2,100,000 would be displayed as 21E+05, on others it would be displayed as 21E5.

We use commas to make large numbers easier to read: in the example above we have written 2,100,000 using commas to separate the millions and thousands. Since BASIC does not allow the use of commas for this purpose, this number would be entered as 2100000!

ARITHMETIC OPERATORS & EXPRESSIONS

<u>OPERATOR</u>	<u>MATHS SYMBOL</u>	<u>BASIC SYMBOL</u>
<u>Examples</u>		

Addition	+	+	5 +
7			
Subtraction	-	-	27.5
- 6.4			
Multiplication	x	*	15 *
8.35			
Division	_	/	2/3

Note: The _ sign on the BBC computer cannot be used as the division sign!

Note: Precision varies somewhat with different makes of computer. Some permit double precision, that is sixteen significant digits.

Computers can be forced to round results by use of a specific formula or by a statement provided in some varieties of the BASIC language. One use of this is to enable cash transactions to be presented to two decimal places (for example £23.67). Forced rounding may also be used where the precision of the figures presented is too great and where such precision is not necessary. Figures may be force-rounded to the number of decimal places required.

ORDER OF OPERATIONS

The accepted rules are:-

1. If an expression contains parenthesis (brackets), then the operation within the brackets is performed first.

For example: $(2 + 6) * 2 = 8 * 2 = 16$

2. Then, any EXPONENTIATION is carried out.
3. This is followed by DIVISION AND MULTIPLICATION
4. Finally, ADDITION & SUBTRACTION are carried out.

This order can be remembered by the word BEDMAS:

Brackets, Exponentials, Division, Multiplication, Addition, Subtraction

For example: $8/2 + 6 * 2^3 - 1$
 $= 8/2 + 6 * 8 - 1$
 $= 4 + 48 - 1$
 $= 52 - 1$
 $= 51$

If two adjacent operations have the same 'priority', then the operations are performed from left to right:

For example: $3 * 4/2 = 6$

Find the value of the following expressions:

- | | | |
|----------------|------------------------|------------------|
| a) $3 + 4 * 2$ | b) $3 + 4 * 2^2$ | c) $(3 + 4) * 2$ |
| d) $8/4/2$ | e) $(2^3 - 5) * 3 - 9$ | |

Note:

Numbers are stored in the computer memory in an exponential format. This can cause problems at times particularly when the numbers have been produced by an exponential calculation. For example 9^2 will, in some computers, produce a result like 81.000001 rather than the correct value of 81. If the result had been derived by $9 * 9$, it would be 81! This anomaly will be dealt with at a relevant point later in this book.

CHAPTER III

PROGRAM PLANNING

As has been mentioned in the introduction to this book, the importance of careful program planning cannot be over-emphasised. Just as the creator of any object, be it artist, architect or carpenter, will make a sketch of the product before working on the details of the construction, so should the computer programmer take this approach. Refinements can be added at later stages but the initial planning is important because this is the basis on which the remainder rests.

The primary aim of a computer program is to produce meaningful information in an easily read and understandable form, the way in which this information is produced being just one of the factors to be considered. The key point to be remembered is that good programs are rarely produced at the computer keyboard; they should be planned and written away from the computer, only being typed in and tested when you are satisfied that all the requirements have been met.

Professional computing often involves several stages, each stage perhaps being carried out by different individuals or groups. For example, displays and graphics output is generally the concern of a specific group. The final stage is the coming together of each part in the form of the program which is then typed into the computer system and tested. Systems Analysis is one of the early stages in this planning procedure, the programming forming the final stage and both of these stages are often dealt with by different individuals or groups. Most budding programmers find that it is not the conversion of the plan into BASIC instructions which gives them most problems, it is the plan itself which requires most thought. It is for this reason that you should take an approach similar to that mentioned above and also to carefully retain any written work produced during the planning stage. This work is known as 'documentation' and can be extremely useful at a later date if a program needs to be modified, particularly if the program concerned was written some time ago and you have forgotten what was done to produce it.

Let us examine the planning of a program which will result in a display something like DISPLAY 1 showing the conversion of a temperature in degrees Centigrade into the equivalent Fahrenheit temperature. Quite often, deciding upon the layout for the final display will give you a very good idea of what you want the program to do and how to achieve that aim. Together with this, if you are able to sit down with a pencil and paper and calculate or work out what you want then you already know what the program should do!

DISPLAY 1

TEMPERATURE CONVERSION

CENTIGRADE TO FAHRENHEIT

TYPE THE CENTIGRADE TEMPERATURE? 35

How should be plan such a display to determine the ideal locations on the screen for each of the items comprising this display?

In professional computing, specially designed forms are available for both display design and program instruction writing. These are known generally as coding forms and make the job easier in many respects. For display design in particular, it is important to use some such guide and for this purpose we shall use squared paper of the variety that is found in school mathematics books.

to find the screen locations for those items comprising the display. To locate the position for an item so that it is centred within the width of the screen, we simply count the number of characters comprising that item, subtract this figure from 40, divide the result by 2 and use this value to locate the item across the screen. For the phrase 'TEMPERATURE CONVERSION', this comprises 22 characters including the space between the two words. Subtract from 40 leaves 18, divide by 2 gives a value of 9. You will note that this is the TAB position used in the program on the next page to place this phrase centrally on the display. The TAB statement will be dealt with fully in a later chapter and is only one way of positioning items to begin in a specific column or location across the screen.

Considering the tabular display above it is clear that the following stages are required in the final program:

1. We firstly need the program to 'clear' the VDU of any irrelevant information remaining from previous operations. A clean, clear display is a must!
2. We next want to print a title to indicate what the program does.
Always work on the assumption that any program you write will be used by others as well as yourself. Displays should therefore clearly indicate the purpose of the program.
3. Thirdly, we want the computer to ask the operator for the centigrade

PROGRAM PLANNING

Chapter III

temperature which needs to be converted.

4. We now need to print the 'headers', these being the column titles.

5. The computer should now calculate the Fahrenheit equivalent to the centigrade temperature supplied at stage 3. It is fairly simple to go to a book and to find the formula necessary for this conversion. (In this course, any formulae required for project work are supplied.)

6. We want the computer to then display both the centigrade temperature and the associated Fahrenheit equivalent in the columns on the display.

7. Lastly, we want the computer to stop operation (execution).

Having planned the display, we now actually know what we want the program to do and how this is to be done. All of this by simply looking at and thinking about the final display! Always consider the final product before going forward with the planning and construction of a computer program.

What we have produced here is an ALGORITHM, a defined process or set of rules for solving a given problem. Conversion of this algorithm to BASIC program instructions would now be easy and these program instructions or LINES are shown below;

```
10  CLS
20  PRINTTAB(9);"TEMPERATURE CONVERSION":PRINT
25  PRINTTAB(8);"CENTIGRADE TO FAHRENHEIT":PRINT
30  PRINTTAB(4);:INPUT "TYPE THE CENTIGRADE TEMPERATURE ";C
40  PRINTTAB(7);"CENTIGRADE";TAB(23);"FAHRENHEIT"
45  PRINTTAB(7);"-----";TAB(23);"-----":PRINT
50  LET F=(C*9/5)+32
60  PRINTTAB(10);C;TAB(27);F
70  END
```

At this stage we have not covered any BASIC statements but it is relatively easy to relate the instructions above to the written stages produced during planning. Each of the lines begins with a number. Line 10 containing the BASIC statement CLS, this being the instruction understood by the computer as meaning 'clear the screen'. Lines 20 to 45 and line 60 all have PRINT and TAB statements and these relate to the requirement to display certain information at a particular

position on the screen. Line 30 also contains the INPUT statement and you might guess correctly that this statement is the one used to request information (input) from the operator, via the keyboard. Line 70, END needs no explanation.

In conclusion, always plan your program carefully, remember that displays often give many clues as to what the program should do. Use squared paper to plan these displays and do not approach the computer until you have a fully written set of documentation and a display design or designs. Desk check your written program by playing the part of the computer in an endeavour to see what will result when your program is executed. If you adopt this method of program writing it cannot be guaranteed that your programs will be error free, but they should contain far fewer errors than if you compose them directly at the keyboard. Certainly your displays should be far cleaner, clearer and attractive having been given more forethought. It is always permissible to make modifications afterwards to give a final polish to the programs and this may be done at the keyboard, but only after typing in and testing the original plan.

SCREEN OUTPUT

It has been already noted that many home computers have a standard screen width of 40 characters. This screen is also divided into four (invisible) 'zones' or 'fields', each being 10 characters wide and in a few computers this zone width may be changed to suit a particular circumstance. Zones are useful for multi-column tabular displays and there is a specific BASIC instruction used in conjunction with the PRINT statement to force the printing of text and numbers in zones. We have already seen that the TAB statement may also be used to locate an item at a particular position across the screen and both of these methods for controlling screen output will be dealt with at greater length in a following chapter.

In most computers the use of the zoning instruction with both text and numbers will produce the following effect. For example, suppose we wish to print a table of four column, each column having a header and associated with each header, a number:

TEST 1	TEST 2	TEST 3	AVERAGE
65	50	47	54

This would be how most computers would display such a requirement using the zoning instruction in BASIC. Each item would be printed starting at the beginning of each zone, that is columns 1,11,21 and 31.

The BBC microcomputer and others when using BBCBASIC however, use a different method to zone text from that which is used for numbers. The same display on the BBC would be as follows:

TEST 1	TEST 2	TEST 3	AVERAGE
65	50	47	54

The text zoning instruction will position each text item starting at the beginning of the zone, numbers being placed at the end of each zone and working backwards!

This problem may be easily overcome by ignoring the zoning instruction and using the TAB statement to position all items exactly where you want them.

LINE PRINTER OUTPUT

Most printers have a paper width of 80 characters; the same width as is standard on many computer systems or can be selected on the BBC microcomputer. The zone widths are still 10 characters. Therefore on a printer there are eight

available zones under normal circumstances. Again, the TAB statement may be used to position items at the required location across the paper and this is the method that is recommended.

We will deal with using printers, "hard copy" at a later stage.

CHAPTER IV

COMMANDS & STATEMENTS

There are two ways of giving instructions to the computer. The first is to give it **COMMANDS** which it will act on immediately, the second is to give it a series of numbered instructions or **STATEMENTS** which it will store in memory and carry out in sequence when told to do so. Such a stored series of instructions is called a **PROGRAM** and the program may be **EXECUTED** when required. Words used by BASIC to describe a desired operation are known as **KEYWORDS** or **RESERVED WORDS**. Many of the keywords in BASIC can be used as both commands and as statements in a program. When a computer is being used to enter and immediately carry out commands, this is known as the **COMMAND** or **DIRECT MODE**. Conversely, when a series of statements are entered, to be executed at a later stage, the mode in use is known as the **PROGRAM MODE**.

COMMAND or DIRECT MODE

The keyword 'PRINT' is used to instruct the computer to print something on the screen. For example, if you type the following lines and press RETURN at the end of each line, the result obtained will be as shown on the right:

<u>TYPE THIS & 'RETURN'</u>	<u>RESULT</u>
PRINT 35 * 2	70
PRINT 6 * 2 - 7	5
PRINT 3 + (9 * 8)	75
PRINT "HELLO"	HELLO
PRINT "PLEASED TO MEET YOU"	PLEASED TO MEET YOU

The computer in the first three examples is acting very much like a calculator and can in fact be used as such to do simple or complex calculations. The last two examples show the computers ability to handle words or phrases (text). The keyword PRINT can also be used as a program mode statement as will be shown later. Some other important commands and their use are listed below:

RUN - instruct the computer to execute or run the resident program.

LIST - displays on the screen the 'list' of program statements which go to to make up the resident program.

NEW - clears memory and erases the resident program so that a new program can be entered. This command should always be used before starting on a new program.

The RETURN key must always be pressed at the end of each line. Until this is done, the command will not be executed.

PROGRAM MODE

If the keyword and associated information is preceded by a number the computer assumes that you are entering a program line and therefore does not act upon it immediately. It simply stores that line in memory for later use and will act upon it when told to do so by means of the RUN command. If several lines which form a program are stored in memory then these will operate in line number order when the program is executed. For example, if you type the following, the program so formed will be stored until you decide to execute or 'run' it:

COMMANDS & STATEMENTS

Chapter IV

```
10   CLS
20   PRINT "HELLO"
30   PRINT "PLEASED TO MEET YOU"
40   PRINT
50   PRINT "MY NAME IS FRED!"
60   END
```

Remember to press RETURN at the end of each line but before you do this, check that the line is correct and that you have made no typing errors. If you have, use the DELETE key to rub out the line and retype it. If you discover a mistake in your line after you have pressed RETURN then retype the line completely. The new line that you type will automatically replace the old one. To verify this, type LIST [RETURN] to list the latest version of the program. (The editing facilities on most computers make it relatively easy to edit or correct lines which contain a mistake without the need to retype the line in full.)

When ready type RUN [RETURN] and the program will execute.

LINE NUMBERS

These serve two purposes. Firstly, they serve as labels, as a particular statement may be referred to by it's line number. Secondly, and more important, the line number tells the computer in which order to execute the statements. It is good practice to number lines in increments of 10 or more, for example 10, 20, 30, 40 and so on. This allows extra lines to be inserted into the program later if so required. Consider the part-program below:

TYPED IN THIS ORDER

```
10   LET A = 5
20   LET B = 6           If we then type 25 LET D = A + B
[RETURN]
30   LET C = 7           and LIST the result:
40   PRINT A + B + C
```

```
10   LET A = 5
20   LET B = 6           This will be displayed showing that
we can insert
25   LET D = A + B       extra lines by using a line number in
between
30   LET C = 7           existing lines.
40   PRINT A + B + C
```

If you wish to remove a line completely, this can be done by typing the relevant line number only followed by RETURN. For example, 25 [RETURN] would remove line 25 from the program and you will be back to the first version typed.

The standard format for a program line is therefore: LINE
NUMBER statement

CHAPTER V

NUMERIC VARIABLES & THE LET STATEMENT

The term "numeric variable" is used to simply describe an area in computer memory where a number can be stored. It is called "variable" since the content of that area in memory can be changed or varied, and may be pictured as a "box" in which the number resides. This "box" is labelled with a name so that data may be stored and retrieved from it by referring to that name. Most BASICs permit lengthy variable names such as "TOTAL" and "ADDRESS" to be used, known as "descriptive variable names", but since some words are invalid in BBCBASIC when used in upper case, and to reduce the amount of typing we have to do, we shall normally limit ourselves to small, simple labels. These will usually be one or two letters of the alphabet, or a letter of the alphabet followed by one digit. The letters I and O are best avoided as numeric variables because of possible confusion with the figures 1 and 0.

The following are all examples of sensible simple variable names: A, B, C, F, Q, Z, AG, B2.

A few two-letter combinations cannot be used because they clash with keywords (e.g. IF, ON).

Numbers or values can be stored in numeric variables by means of the LET statement. Consider the two program lines below:

```
100 LET P = 25      When executed, 25 will be stored in numeric
variable P
110 LET X = 5       When executed, 5 will be stored in numeric
variable X.
```

Now consider this line; 120 LET X = P + X. If this were associated with the two lines above, then when executed, the values stored in P and X would be added and the result stored in numeric Z. The values in both P and X would still be there; they would not have been altered by the operation in line 120. With all expressions such as that on line 120, the operation on the right of the equals sign is first done and then the result is stored in the variable on the left of the equals sign.

We shall in the future be meeting what at first sight seems to be a rather odd situation and this is shown below:

```
200 LET W = W + 1.
```

Applying the rule given above concerning expressions, when this line is executed we would first add 1 to the current value stored in W and then store the result of this back in W! For example, if W contained the value 7 before line 200 was executed then we would add 1 to 7 and store the result 8 back in W. Therefore after line 200 had been executed, W would be equal to 8. This particular use of a numeric variable is quite often found when a requirement to count items is concerned, for example, we might use it to permit the computer to count the number of times a particular action had taken place, such as the wrong answer being given to a question supplied by the

computer to the operator. Such a situation might arise in a quiz program.

Another way to picture this situation is:

```
NEW VALUE OF W = OLD VALUE OF W + 1
```

Remember that although we have used W for this example, we might equally have used any other valid numeric variable and the line,

```
250 LET Z = Z - 1
```

 would work in exactly the same way. Lines such as

```
300 LET Z = Z - 1
```

 will also be encountered. Variables must have an initial value given to them, or "initialised". Line 200 above would be nonsense if the computer did not already hold a value for Z.

NUMERIC VARIABLES & THE LET STATEMENT

You will probably have noted that each item on a program line has been typed with an intervening space, for example 200 LET W = W + 1. Many computers take no notice of such spaces and would understand 200 LETW=W+1 as meaning the same thing. It is much easier however for you to read program lines if you use spaces and you are encouraged to do so. In any case, on some computer systems certain program statements require intervening spaces otherwise the computer will not understand the statement and will halt execution of any program containing the incorrect statement.

It is wise to name numeric variables in a way which indicates the purpose for which that variable is being used. For example, if a numeric variable is to be used to hold a temperature value then perhaps T should be the name of that variable; similarly, if a figure representing VAT must be stored, V could be used.

THE LET STATEMENT

The LET statement is known as an assignment statement. It assigns values to numeric variables and we shall meet it again when we deal with STRING VARIABLES, these being used to store textual items such as names and addresses.

In most BASICs, the LET statement is optional. The computer will understand 100 L = 3.987 equally as well as 100 LET L = 3.987.

The full format for the LET statement is:

```
LINE NUMBER      LET      NUMERIC VARIABLE = VALUE OR
EXPRESSION
```

PROGRAM TRACING

This is a technique which is used to 'desk-check' a program before implementing it on a computer. The programmer must play the part of the computer and 'run' the program, checking the result of each operation by hand and recording it. In this way, errors and mistakes can often be found before the program is typed on the computer. A simple example follows to indicate how this desk-check is carried out:

	A	B	C
	—	—	—
10 LET A = 6	6		
20 LET B = 9		9	
30 LET C = A * B			54
40 PRINT;C		Value 54 printed	
50 END		End execution	

The numeric variables used in the program are listed and when their content changes, the new value is written under the

variable name. Comments such as those shown for lines 40 and 50 are self-explanatory.

Variables must have an initial value given to them, or be "initialised". 200 LET W = W + 1
would be nonsense if the computer did not already hold a value for W

CHAPTER VI

THE PRINT & PRINTTAB STATEMENTS

The aim of most computer programs is to present the result of a particular operation as a display (screen output) or hard copy (printer output) in a form that can be easily read and understood by any operator executing those programs. Programs should always be written on the assumption that other people will use them. Consequently careful thought must be given to the style and format of any outputs produced by the program so that there will be no confusion over what is being presented.

THE PRINT STATEMENT

This statement is used to present such results and to display on the screen textual and numeric information, both of these also being known as data. In the vast majority of cases a PRINT statement will produce the same, if not a very similar output in different versions of BASIC. BBCBasic, however, produces a non-standard output when printing numeric information (numbers). This has already been referred to in connection with screen zones in an earlier chapter. With some BASICs, when a number is printed on the screen it will be printed with both a leading space and a trailing space. The leading space is used for the 'sign' of the number, if that number is positive a leading space will be printed, if negative, a minus sign would replace the leading space before the number. BBCBasic does not use a leading or trailing space at all with numbers but will, of course, print a minus sign in front of a number if it is negative. Together with this, whereas most computers print numbers from the left of a display zone, BBCBasic places them at the end of the zone. This is best illustrated by the simple program below and the resulting displays:

```
10 CLS                Clear the screen
20 LET A = 3          Store the value 3 in numeric variable A
30 PRINT A            Print the value stored in numeric variable A
40 END                Terminate program execution
```

```
-----
| 3      .      .      .      |      DISPLAY 1
| >_     .      .      .      |      <- The display produced with
|        .      .      .      |      most BASICs
-----
|          3      .      .      |      DISPLAY 2
| >_     .      .      .      |      <- The display produced
using                                         BBCBasic
|        .      .      .      |
-----
```

The dots down each of the above displays indicate the four invisible screen zones. Note the leading space before the number 3 on DISPLAY 1 and the way in which the BBC places the number at the end of zone 1 as shown in DISPLAY 2.

Demonstration programs used throughout this book are written mainly for use with BBCBasic.

For the program above, line 30 for the BBCBasic should be: 30 PRINT;A. Substituting this line would produce a display similar to that shown in DISPLAY 2 but with no leading space before the number. The '>' symbol on each of the displays is called the 'prompt', it indicates that the computer is ready for use and awaiting the next action. The '_' is the cursor, or position where the next character will be printed. From now on, we will assume that we are using BBCBasic when formatting screen output.

THE PRINT & TAB STATEMENTS

Chapter VI

We have seen that the PRINT or PRINTTAB statements may be used to print the value stored in a numeric variable. We can also use either of these statements to print words or a sequence of keyboard characters. These are called 'character strings' and an example line follows:

```
230 PRINT "AREA"           This line prints the word AREA
```

The computer prints all of the characters between the quote marks which serve only to indicate the beginning and the end of the string. The string on line 230 is AREA. Here are some further examples of PRINT statements with strings:

```
60 PRINT "*****"           70 PRINT
"-----<>-----"
80 PRINT "ABC DEF GHI"       90 PRINT
"A1B2C3D4E5F6G7H8I9J0"
```

Note that on line 80, there are spaces between the groups of letters. Spaces are 'characters' and may be used within strings. Consequently, we may print a persons name by using a PRINT statement;

```
100 PRINT "WILLIAM BROWN"    Prints the string WILLIAM
BROWN
```

Also, on line 90, there is a mixture of letters and numbers. When numbers are enclosed by quote marks, they are 'characters' and are no longer true values. Maths cannot be done directly on string numbers.

Consider the following program:

	<u>TRACE</u>	<u>L</u>	<u>W</u>	<u>A</u>
90 CLS	Clear the screen			
100 LET L = 20		20		
110 LET W = 8			8	
<u>List 1</u> 120 LET A = L * W				160
130 PRINT "AREA"	Print the string AREA			
140 PRINT;A	PRINT the value of numeric			
variable A				
999 END	Terminate program execution			

When this program is executed, the following display would ensue:

```
-----
|AREA          |      <- The string AREA printed
hard-left     |
|             |
|160          |      <- BBCBasic output. No
leading space. |
```

Now consider this program:

	<u>TRACE</u>	<u>S</u>	<u>B</u>	<u>P</u>
	90 CLS	Clear the screen		
	100 LET S = 300	300		
	110 LET B = 450		450	
<u>List 2</u>	120 LET P = S - B			-150
	130 PRINT "PROFIT"	Print the string PROFIT		
variable P	140 PRINT;P	Print the value of numeric		
	150 END	Terminate program execution		

THE PRINT & TAB STATEMENTS

Chapter VI

Execution of program List 2 would produce the following display:

```
-----
|PROFIT          |           The minus sign indicating
a negative      |
|-150           |           <- value is printed before
the number.     |
|>-            |           Both items are printed
hard-left.      |
```

THE PRINTTAB STATEMENT

The keyword PRINT used with the keyword TAB form the PRINTTAB statement. This works in very much the same way as the tabs on a typewriter. The following line would print *** starting in column 21 across the screen:

```
120 PRINTTAB(20);"***"
```

This line instructs the computer to skip (or to move the cursor) 20 columns from the left and to begin printing in the next column. The general format for a simple PRINTTAB statement such as this is:

```
LINE NUMBER PRINTTAB(X);"***" This tells the computer to
skip X columns from the left and to begin printing in column X
+ 1.
```

Consider the following program:

```
90 CLS
100 PRINT "COLUMN NUMBERS"
110 PRINT;"12345678901234567890"
List 3 120 PRINT
200 LET A = 3.14159
210 PRINTTAB(5);"ANSWER"
220 PRINTTAB(5);A
999 END
```

When this program is executed, the following would be displayed:

```
-----
|COLUMN NUMBERS  |           <- String printed hard-left
|12345678901234567890 |           <- String printed hard-left
|                |           <- 'Blank' line printed
|      ANSWER    |           <- String printed starting
column 6         |
|      3.14159   |           <- Value printed starting
column 6         |
```

|>_

|

The number or value in brackets following the PRINTTAB statement is the tab position from which printing will commence. This number or value may also be a numeric variable or expression. The following program illustrates this:

```
          90 CLS
          100 PRINT;"12345678901234"
List 4    110 LET X = 2
          120 PRINTTAB(X);"ABCD";TAB(X+6);"EFG"
          999 END
```

Note that we have a somewhat more complex multiple PRINTTAB statement in List 4. When executed, it would produce the display on the next page:

THE PRINT & TAB STATEMENTS

Chapter VI

```
-----  
|12345678901234      |      <- String printed hard-left  
|  ABCD  EFG          |      <- X = 2, therefore string  
ABCD is  
|>_                  |      printed starting in  
column 3 and  
|                    |      string EFG starting in  
column 9.
```

It may make things easier to think of the left most column as column 0. Then PRINTTAB prints at the appropriate column number.

Let us look more carefully at line 120 of the above program:

```
120 PRINTTAB(X);"ABCD";TAB(X+6);"EFG"
```

Note:

- 1 The PRINT statement occurs only once on the program line.
- 2 The brackets containing the TAB position.
- 3 The semi-colon (;) used between each part of the statement on the line.
- 4 No space between TAB and (X).

Many computers do not always require the use of a semi-colon on a program line such as that above. There are some instances however, where an error may occur if semi-colons have not been used. In order to avoid any confusion, it is recommended that semi-colons are ALWAYS used in conjunction with PRINTTAB statements.

Further examples of multiple PRINTTAB statements:

```
400 PRINT "NAME";TAB(15);A;TAB(25);B;TAB(35);C
```

```
350 PRINT TAB(0);T1;TAB(10);T2;TAB(20);T3;TAB(30);T4
```

```
600 PRINT TAB(0);"NAME";TAB(10);"TEST 1";TAB(20);"TEST 2";  
TAB(30);"AVERAGE"
```

Note that the first part of line 400 PRINT "NAME" and line 600 PRINT TAB(0);"NAME" will in fact produce exactly the same output, this being the word NAME printed hard-left on the screen.

In Chapter III, Program Planning, we discussed the use of squared paper in the planning of screen displays. It will now be apparent that this is a sensible way to approach such a problem and that careful planning will make it much easier for you to determine the number or value that should be used in any particular PRINTTAB statement. If you examine the display plan on page ??????? and the subsequent program on the

following page, you will appreciate the significance of this method!

The main uses of the PRINTTAB statement are:

- 1 to create tabular displays
- 2 to 'draw' pictures
- 3 to plot rather crude graphs.

The format is: LINE NUMBER PRINTTAB(X);Data

For multiple statements:

 LINE NUMBER PRINTTAB(X);Data;TAB(Y);Data;TAB(Z);

Data

 or LINE NUMBER PRINT Data;TAB(A);Data;TAB(B);Data;

TAB(C);Data

'Data' represents STRINGS, VARIABLES or EXPRESSIONS.

THE PRINT & TAB STATEMENTS

Chapter VI

THE PRINT STATEMENT AND SEMI-COLONS

The effect of semi-colons in a PRINTTAB statement is to ensure that after moving to the TAB position indicated in brackets, the cursor will then remain in that position and print the next item from that point. In other words, the cursor is prevented from dropping to the next screen line. The effect which is produced by the semi-colon may also be used in standard PRINT statements as shown below in List 5 which is a modified version of List 1:

```
          90 CLS
          100 LET L = 20
          110 LET W = 8
List 5      120 LET A = L * W
          130 PRINT "AREA = ";A
          999 END
```

The following display would ensue on execution of this program:

```
-----
|AREA = 160          |
|>-                 |
|                   |
```

Some further examples of print statements using semi-colons and their outputs. Assume in each case that the variable W has been assigned a value of 4:

```
100 PRINT "THE AREA IS";W;"SQUARE INCHES"      -> THE AREA IS 4
SQUARE INCHES

230 PRINT "YOU GAVE";W;"WRONG ANSWERS"         -> YOU GAVE 4
WRONG ANSWERS

350 PRINT "YOUR BEST MARK WAS";W              -> YOUR BEST MARK
WAS 4
```

Remember that BBCBASIC does not print a leading space in front of numbers or a trailing space after them. Consequently, line 100 above, for example, executed in BBCBasic would give:

```
THE AREA IS4SQUARE INCHES
```

To obtain the same output as above in BBCBasic, these spaces would have to be provided within the strings before and after the numeric variable:

```
100 PRINT "THE AREA IS ";W;" SQUARE INCHES"
```

THE PRINT STATEMENT AND COMMAS

Earlier in this chapter and in the chapter on Program Planning, we briefly discussed screen zones and mentioned an instruction which could be used to force the printing of items in these zones. With the standard screen width of 40 characters there are four zones, each of ten characters width. On systems employing an 80 character width and on line printers, there are eight zones. Some computer systems permit the standard zone width to be modified but as this facility does not apply in many cases we shall not consider it in this book. These zones provide a simple method for obtaining tabular displays, providing that they do not have more than four columns of information. We have seen above that semi-colons may be used within PRINT statements to keep all output on the same screen line. Commas may be used in a similar way but these have the effect of forcing the next item output into the next screen zone.

THE PRINT & TAB STATEMENTS

Chapter VI

Consider the following program:

```
90 CLS
100 PRINT "LENGTH", "WIDTH", "AREA"
110 LET L = 10
List 6      120 LET W = 12
130 LET A = L * W
140 PRINT;L,W,A
999 END
```

Execution of this program would provide the following output:

```
-----
|LENGTH      WIDTH      AREA  |    <- Strings printed at start
of zones
| 10          12         120   |    <- Values printed at start
of zones
|>_          |                and with leading spaces,
line 140.
```

For simple tabular displays where there are no more than four columns of information comma zoning may be used or string output, but the TAB statement is often a better choice and because of the BBC's anomaly in printing numbers at the end of each zone, it is probably the best choice if you are writing programs for use in BBCBasic. In any case, the TAB statement gives you full control over output and you can easily place items at any position across the screen.

Both commas and semi-colons may be used in the same PRINT statement and an example of this follows:

```
90 CLS
100 LET L = 20
110 LET W = 8
List 7      120 LET A = L * W
130 PRINT "LENGTH = ";L;"WIDTH = ";W
140 PRINT "THEREFORE, AREA = ";A
```

On execution, the following display would ensue:

```
-----
|LENGTH = 20      WIDTH = 8      |    <- LENGTH = 20 extends
into zone 2
|THEREFORE, AREA = 160          |    and therefore WIDTH =
8 is in
|>-                          |    zone 3!
|                              |
```

THE PRINT STATEMENT AND APOSTROPHES

An apostrophe (') in a PRINT statement forces the cursor to the beginning of the next line in the same manner as a fresh PRINT statement can do, and can therefore reduce the number of PRINT statements necessary.

```
100 PRINT "JOHN SMITH" ' "32 HIGH STREET" ' "WELLING"
```

will produce the same display as

```
100 PRINT "JOHN SMITH"  
110 PRINT "32 HIGH STREET"  
120 PRINT "WELLING"  
shown on the next page
```

THE PRINT & TAB STATEMENTS

Chapter VI

```
-----  
|JOHN SMITH      |  
|32 HIGH STREET |  
|WELLING        |  
|>-            |
```

Semi-colons, commas and apostrophes

```
          90 CLS  
          100 LET L = 20  
          110 LET W = 8  
LIST 6A 120 LET A = L * W  
          130 PRINT "LENGTH", "WIDTH" ' "AREA" ' ;L;TAB(10);W;  
TAB(20);A  
          140 END
```

Note the apostrophe and semi-colon after "AREA", i.e. new line and first column.

THE EXTENDED PRINTTAB STATEMENT

The PRINTTAB statement can be used to print to ANY position on the screen. It takes the general form.

PRINTTAB(A,D)

Where A indicates the position from the left (from column 0) and D indicates the number of rows down (the top row is row 0).

```
100 PRINTTAB(20,12);"*"  
will print a star in the centre of the screen.
```

Any character already on the screen in the designated position will be deleted and overwritten.

```
          100 CLS  
          110 PRINTTAB(16,13);"FLASHING"  
List 6B 120 PRINTTAB(16,13);"          "          <- 8  
spaces  
          130 PRINTTAB(16,13);"FLASHING"  
          140 PRINTTAB(16,13);"          "  
          150 PRINTTAB(16,13);"FLASHING"  
          etc ...
```

will produce a flashing display as the word "flashing" is alternately written and then overwritten with spaces. List 6B can be changed to give a continuous flashing display by adding the line
155 GOTO 110
which will be explained later in the course.

PRINTER OUTPUT

All of the program outputs above have been shown as displayed on the VDU or screen. Output may of course be directed to the printer and this has been mentioned in Chapter IV, Commands & Statements. There will be instances where you will require the actual program output (the result produced on execution) to be in the form of a hard copy. The statements required for this purpose do however, vary somewhat from Basic to Basic. In a large number of cases modification of the PRINT keyword to LPRINT will achieve the required result, But, BBCBasic needs to be given

THE PRINT & TAB STATEMENTS

Chapter VI

instructions to both turn on the printer (VDU2) and to turn it off again after use (VDU3).

Given below is one version of the previous demonstration program, List 3, which illustrates how VDU2 and VDU3 may be used for BBCBasic programs to direct output to the printer.

90 CLS	The VDU2
statement tells	
95 VDU2	the computer to
provide	
100 PRINT "COLUMN NUMBERS"	<u>both</u> a screen
display <u>and</u>	
110 PRINT "12345678901234567890"	a hard copy of
what is	
<u>List 3a</u> 120 PRINT	produced when the
program	
200 LET A = 3.14159	is executed.
Every PRINT	
210 PRINT TAB(5);"ANSWER"	statement
between the VDU2	
220 PRINT TAB(5);A	statement at
line 95 and the	
230 VDU3	VDU3 statement
at line 230	
999 END	would be so
affected. PRINT	
a	statements not 'inside'
	VDU2 - VDU3 would
display to	the screen only!

To obtain a printout 'hard copy' of your program listing using BBCBASIC, enter the following commands:

```
VDU2
LIST
VDU3
```

You will be told of any additional commands necessary in conjunction with the networked printers used in the course workshops.

CHAPTER VII

STRING VARIABLES

In the last chapter we discussed 'character strings'. Another name for these items is 'string literals' or just simply a 'string'. In line 230 PRINT "AREA", the word AREA is a character string or string literal. Every time this line is executed by the computer this string would always be printed literally as shown. We have introduced numeric variables in a previous chapter, these being used to store numbers or values that can change (vary) according to the program instructions. In a similar way, there are areas in memory where we can store strings which can vary and these are referred to as STRING VARIABLES. Here, we have the ability to store names, addresses, item descriptions and any other textual or symbol string. BASIC requires that string variables be named differently from numeric variables.

As with numeric variables, most BASICs permit lengthy variable names such as "TOTAL" and "ADDRESS" to be used, known as 'descriptive variable names', but again, since some words are invalid in BBCBASIC when used in upper case, and to reduce the amount of typing we have to do, we shall limit ourselves to small, simple labels. A string variable name must end in a dollar (\$) sign. It will often be one or two letters of the alphabet plus a \$, or a letter of the alphabet followed by one digit and a \$.

The following are all examples of sensible simple string variable names:

A\$, B\$, C\$, F\$, Q\$, Z\$, AG\$, B2\$.

Again, a few two-letter combinations cannot be used because they clash with keywords (e.g. IF\$, ON\$).

From now on we will refer to a \$ as the string sign, and A\$ is verbally called "A-string".

The number of characters that can be stored in a string variable differs from computer to computer. With some, it is as little as 18, quite often it is 256 and in a few cases it can be as much as 4095 or more.

The assignment of strings to string variables may be done in the same way as with numeric variables. The LET statement (optional) can be used:

For example; 100 LET S\$ = "COMPUTER" OR 100 S\$ =
"COMPUTER"

Note the quotation marks surrounding the string. These simply indicate the beginning and the end of the string and are not a part of it.

A string variable may be used to store complex information such as an address and the following lines illustrate this:

120 LET A\$ = "312 GIPSY ROAD, WELLING, KENT."

340 LET T\$ = "0223 34452"

450 LET M\$ = "THE ANSWER THAT YOU GAVE WAS WRONG. PLEASE TRY AGAIN!"

Consider this program:

	<u>TRACE</u>	<u>B\$</u>	<u>C\$</u>
	10 CLS		
	20 LET B\$ = "HAPPY"		HAPPY
<u>List 8</u>	30 LET C\$ = "BIRTHDAY"		
BIRTHDAY			
variable B\$	40 PRINT B\$		Print the string in string
variable C\$	50 PRINT C\$		Print the string in string
execution.	60 END		Terminate program

STRING VARIABLES

Chapter VII

When executed, the following would display:

```
-----  
|HAPPY          |      <- String printed  
|BIRTHDAY      |      <- String printed  
|>_           |
```

If we now modify line 40 to: 40 PRINT B\$;C\$ and remove line 50 the following display would ensue:

```
-----  
|HAPPYBIRTHDAY |      <- Strings printed one after  
the            |  
|>_           |      other.  
|             |
```

By now changing the semi-colon in line 40 to a comma: 40 PRINT B\$,C\$ we would obtain:

```
-----  
|HAPPY    BIRTHDAY |      <- Strings printed in zones  
by the     |  
|>_       |      comma instruction.  
|         |
```

Further modification by the addition of line 35: 35 LET S\$ = " " and retyping line 40 again to read: 40 PRINT B\$;S\$;C\$ would display:

```
-----  
|HAPPY BIRTHDAY |      <- Strings B$, S$ and C$  
printed         |  
|>_           |      one after the other.  
|             |
```

A change to a string assignment in List 8 could be made by retyping the relevant line or by using the editing facilities of the computer. On Acorn computers, there are four editing keys marked with arrows. These keys are used to position the cursor at the beginning of the line to be edited. The COPY key is then used to copy that line to the bottom of the display and this line can be edited as required. Suppose for example, that we wished to change line 30 to read: 30 LET C\$ = "CHRISTMAS". We would position the cursor under the 3 at the beginning of the line and press COPY until the following had appeared at the bottom of the display:

```
30 LET C$ = "BIRTHDAY
```

Then the DELETE key would be used to 'rub out' BIRTHDAY, the word CHRISTMAS would be typed followed by a quote mark ("), and the RETURN key pressed. The new line 30 would now read:

30 LET C\$ = "CHRISTMAS", and since this is the last version of that line typed, it would be the current version. The old line 30 would no longer exist even though it would still show on the screen. 'List' would show this!

The rules which apply to numeric variables also apply to string variables, with one major difference. You may carry out maths operations on numeric variables but not on string variables.

Consider these programs:

<u>List 9</u>	10 CLS	10 CLS
LET C\$ = A\$ + B\$	20 LET A = 2	20 LET A\$ = "2"
	30 LET B = 5	30 LET B\$ = "5"
	40 LET C = A + B	<u>List 10</u> 40
	50 PRINT;C	50 PRINT C\$
	60 END	60 END

STRING VARIABLES

Chapter VII

Execution of these programs would display the following:

```
-----  
|7          | <- List 9 would display the  
value      |  
|>_       | stored in variable C, this  
being      |  
|          | the result of adding A and  
B.         |  
-----
```

```
-----  
|25        | <- List 10 would display the  
result of  |  
|>_       | 'adding' A$ and B$. When  
you use a  |  
|          | plus sign between string  
variables  |  
           | they are 'joined' together.  
-----
```

This process of joining string variables together is known as CONCATENATION, and it is used in Word Processing software during editing and insertion of text. It is useful in BASIC in that, for example, full names may be constructed from first and last names. Let us assume that variable F\$ has been assigned the string ALFRED and that string variable L\$ has been assigned the string TENNISON.

The program statement: 500 LET N\$ = F\$ + " " + L\$ would store the string ALFRED plus one space plus TENNISON (ALFRED TENNISON) in string variable N\$.

Another example: Assume that the following assignments have been made:

```
N$ = "312"      S$ = "GIPSY ROAD"      T$ = "WELLING"      C$ =  
"KENT"  
P$ = "DA16 1JJ"
```

```
The statements: 100 LET A1$ = N$ + ", " + S$ + ", " + T$ +  
"."  
                110 LET A2$ = C$ + ", " + P$ + "."
```

would store the full address in two string variables. If these were then printed by the statements:

```
200 PRINT A1$  
210 PRINT A2$ the following would display:
```

```
-----  
|312 GIPSY ROAD, WELLING |  
|KENT, DA16 1JJ         |  
|>_                     |  
-----
```

Consider the following programs and determine what they would output to the screen:

	10 CLS		10 CLS
	20 LET T\$ = "JOHN"		20 LET G\$ =
= "READY..."			
	30 LET S\$ = " "		30 LET H\$ =
"STEADY..."			
<u>List 11</u>	40 LET U\$ = "WILLIAMS"	<u>List 12</u>	40 LET I\$ =
= "GO!"			
G\$,H\$,I\$	50 PRINT T\$;S\$;U\$		50 PRINT
	60 END		60 END
	10 CLS		10 CLS
	20 LET B\$ = "UNHAPPY"		20 LET A\$ =
"DOG"			
<u>List 13</u>	30 LET B\$ = "HAPPY"	<u>List 14</u>	30 LET A\$ =
" "			
	40 PRINT " I FEEL ";B\$		40 PRINT A\$
	50 END		50 END

STRING VARIABLES

Chapter VII


```
10 CLS
20 LET H$ = "HELLO"
List 15 30 PRINT H$ + H$ + H$
40 END
```

MORE ABOUT EDITING ON ACORN COMPUTERS

Assume that the following program has been typed and that the errors in it were not noticed and corrected during typing. These errors must be corrected before the program is executed.

```
10 CLS
20 LET A = 56
List 16 30 LRT B = 8          <- This line should read 30
LET B = 8
40 LET C = A/B
50 PRONT C          <- This line should read 50
PRINT C
60 END
```

The actions to be taken are:

1 Use the 'arrow' edit keys to position the cursor under the 3 of 30.

```
30 LRT B = 8
-
```

2 Press COPY a number of times until this appears at the bottom of the display:

```
30 LR
```

3 Press DELETE once. This will be the situation:

```
30 L
```

4 Press E once. This will now display:

```
30 LE
```

5 Press COPY until the remainder of the line appears:

```
30 LET B = 8
```

6 Press RETURN to complete the editing:

A similar correction procedure may be used for line 50.

Similar procedures usually apply when using a BBCBasic interpreter on a non-Acorn computer.

CHAPTER VIII

THE INPUT & GOTO STATEMENTS

The LET statement which we have used to assign numeric values to numeric variables and string values to string variables is perhaps the simplest BASIC statement. It has a disadvantage however, in that each time a program containing LET statements is executed, the same result will be obtained unless the lines containing the LET statements are modified by retyping or editing. Obviously, this is not very satisfactory since there is no guarantee that the person using the program will know how to do this. We need another statement which will allow values to be entered and assigned to variables as the program executes, thus permitting that program to be used for a number of different applications. The statement which allows this is known as the INPUT statement and it might be used for example, where several different values have to be entered in order to calculate a number of different areas.

Consider this simple program which uses an INPUT statement:

```
List 17      5 CLS
             10 PRINT "TELL ME YOUR NAME PLEASE"
             20 INPUT N$
             30 PRINT "HELLO ";N$
             40 END
```

When executed, the following display will ensue:

```
|TELL ME YOUR NAME PLEASE |      <- Line 10 string
printed.                  |
|?_                       |      <- INPUT 'prompt',
line 20.                  |
|                           |
```

At line 20, the computer waits for a name to be typed (entered) from the keyboard. Until this is done, no further action will be taken. The question mark is known as the INPUT prompt and is asking for something to be entered from the keyboard. If we now type FRED [RETURN], the string FRED will be assigned to variable N\$ and the following will then display:

```
|TELL ME YOUR NAME PLEASE |      <- FRED entered in
response to prompt        |
|?FRED                   |      <- OUTPUT from line 30.
|HELLO FRED              |
|>_                      |
```

Obviously, every time we run this program, we can if we so wish enter a different name. The name entered is assigned by the computer to the variable associated with the INPUT statement, in the case above, N\$. This method of computing where the operator supplies information during program

execution and where that information is used to supply an immediate result is known as INTERACTIVE COMPUTING. Another example, where the 'question' is stored in a string variable:

```
90 CLS
100 LET A$ = "NAME PLEASE"
110 PRINT A$
120 INPUT N$
130 PRINT " PLEASED TO MEET YOU ";N$
140 END
```

List 18

THE INPUT & GOTO STATEMENTS

Chapter VIII

Execution of List 18 would first display the following:

NAME PLEASE		<- Line 110 string
variable printed		
?_		<- INPUT prompt, line
120.		

Entering a name (say, JOHN) followed by [RETURN] assigns that name to N\$ and would result in:

NAME PLEASE		
?JOHN		<- JOHN entered in
response to prompt		
PLEASED TO MEET YOU JOHN		<- Output from line 130.
>_		

There is no limit to the number of INPUT statements which you can use in a computer program. Here is an example which uses two such INPUTs to get the information required so that the area of a carpet can be calculated:

```
10 CLS
20 PRINT "WHAT IS THE LENGTH OF THE CARPET
(YARDS)"
30 INPUT L
40 PRINT "WHAT IS THE WIDTH OF THE CARPET
(YARDS)"
50 INPUT W
60 LET A = L * W
70 PRINT "THAT WILL BE ";A;" SQUARE YARDS"
80 END
```

After both pieces of information (say, 5 at line 30 and 4 at line 50) have been entered and the computer has calculated the area at line 60, the following would be displayed:

WHAT IS THE LENGTH OF THE CARPET (YARDS)		<- Line 20 string
printed		
?5		<- 5, in response
to prompt		
WHAT IS THE WIDTH OF THE CARPET (YARDS)		<- Line 40 string
printed		
?4		<- 4, in response
to prompt		
THAT WILL BE 20 SQUARE YARDS		<- Output from
line 70		
>_		

The advantage of the INPUT statement are obvious here. This short program could be used to calculate the areas of many different pieces of carpet or by a simple adaptation of the question asked, to calculate the carpet requirements for any room or area.

This program lacks some detail however. For example, it does not have a title, and the screen display is not altogether attractive and clear. We have mentioned before that all programs should be written so that other operators will be able to use them easily and with no fear of a misunderstanding occurring as to what the program is trying to achieve.

Here is a program which will add two numbers and which provides a much clearer display, less likely to be misunderstood by the user:

THE INPUT & GOTO STATEMENTS

Chapter VIII

List 20

```
70 CLS
80 PRINTTAB(13);"ADDING MACHINE"
90 PRINTTAB(13);"===== "
100 PRINT
110 PRINT "FIRST NUMBER PLEASE"
120 INPUT A
130 PRINT "SECOND NUMBER PLEASE"
140 INPUT B
145 PRINT
150 LET T = A + B
160 PRINT "THE SUM IS ";T
170 END
```

When executed, the computer will wait for an input at line 120 and then at line 140. The result will then be calculated at line 150 and displayed at line 160. The display at the end of execution will look like this (assuming 47 input at line 120 and 53 at line 140).

	ADDING MACHINE		<- Line 80 string printed
	=====		<- Line 90 string printed
			<- 'Blank' line printed,
line 100			
	FIRST NUMBER PLEASE		<- Line 110 string
printed			
	?47		<- 47 entered in response
to prompt			
	SECOND NUMBER PLEASE		<- Line 130 string
printed			
	?53		<- 53 entered in response
to prompt			
			<- 'Blank' line printed,
line 145			
	THE SUM IS 100		<- Output from line 160.
	>_		

The following is a modified version of List 20. You will note that a new line, 165, has been added and that line 170 now contains a new statement, GOTO 80, rather than the END statement above.

List 21

```
70 CLS
80 PRINT "ADDING MACHINE"
90 PRINT "===== "
100 PRINT
110 PRINT "FIRST NUMBER PLEASE"
120 INPUT A
130 PRINT "SECOND NUMBER PLEASE"
140 INPUT B
145 PRINT
150 LET T = A + B
```

```
160 PRINT "THE SUM IS ";T
165 PRINT "-----"
170 GOTO 80
```

The GOTO statement is known as an 'unconditional transfer'. Every time the computer executes line 170 in List 21, program execution will be transferred back to line 80 and execution will continue from this point. This is known as a continuous or 'infinite loop' and provides a repetitive operation facility so that the program will run over and over again until it is brought to a halt by pressing the ESCAPE key. On some computers, the key which will stop such a loop is labelled BREAK or STOP. When using BBCBasic, you should press ESCAPE and not BREAK!

THE INPUT & GOTO STATEMENTS

Chapter VIII

Repetitive operation is a very useful facility in many situations. Suppose for example that you are an employer and you want a program which will calculate your employees wages at the end of each week. Such a program might be repetitive so that you could progress through the employee list entering the necessary requirements ofr each employee using INPUT statements, and the program could then produce their wage slips without the need to re-execute the program for each employee. There are many other similar situations where repetitive operation is useful, this being just one use of the GOTO statement.

The display produced by List 21 after one complete loop would be as follows with the INPUT prompt for the second time through displayed and the computer awaiting an input or answer from the operator via the keyboard:

	ADDING MACHINE		<- Line 80 string printed
	=====		<- Line 90 string printed
			<- 'Blank' line printed,
line 100			
	FIRST NUMBER PLEASE		<- Line 110 string
printed			
	?47		<- 47 entered in response
to prompt			
	SECOND NUMBER PLEASE		<- Line 130 string
printed			
	?53		<- 53 entered in response
to prompt			
			<- 'Blank' line printed,
line 145			
	THE SUM IS 100		<- Output from line 160.
			<- <u>Now back to line 80!</u>
	ADDING MACHINE		etc
	=====		.
	FIRST NUMBER PLEASE		.
	?		.

Pressing the ESCAPE key at this point would stop program execution. If it was later found necessary to resume execution from this point, this could be done by means of the CONT command (short for CONTINUE). Simply type CONT [RETURN] and execution will re-commence from the point at which it was stopped.

THE ENHANCED INPUT STATEMENT

The above examples have all used a PRINT statement to indicate what information is required and an INPUT statement to get that information from the keyboard operator. There is however,

a statement which combines both of these requirements and which is available in most BASICs.

Consider this program:

```
5 CLS
7 PRINT "CIRCULAR AREA CALCULATOR"
8 PRINT "-----"
9 PRINT
10 LET P = 3.14159
20 PRINT "WHAT IS THE RADIUS"
30 INPUT R
40 LET A = P * R^2
50 PRINT "THE AREA IS ";A;" SQUARE UNITS"
60 END
```

List 22

THE INPUT & GOTO STATEMENTS

Chapter VIII

In List 22, the question indicating what is required is being provided by the PRINT statement at line 20. On initial execution the display would be:

<pre>CIRCULAR AREA CALCULATOR ----- WHAT IS THE RADIUS ?_</pre>	<pre><- Line 20 string printed <- Input prompt, line 30.</pre>
---	--

Now compare this modified version which uses the ENHANCED INPUT STATEMENT:

```

          5 CLS
          7 PRINT "CIRCULAR AREA CALCULATOR"
          8 PRINT "-----"
          9 PRINT
List 23   10 LET P = 3.14159
          20 INPUT "WHAT IS THE RADIUS";R
          40 LET A = P * R^2
          50 PRINT "THE AREA IS ";A;" SQUARE UNITS"
          60 END
```

You will note that line 30 has been removed and that the question and means of getting the required information from the operator has now been combined in one statement at line 20, the enhanced input statement. The display follows:

<pre>CIRCULAR AREA CALCULATOR ----- WHAT IS THE RADIUS?_</pre>	<pre><- Line 20 input statement produces this on the display.</pre>
--	--

The text within the quote marks on line 20 is printed and the input prompt follows that text immediately on the same screen line. This is obviously a neater way to use the INPUT statement.

INPUT STATEMENT FORMATS

SIMPLE INPUT: LINE NUMBER INPUT Variable (String or Numeric)

Examples: 100 INPUT R (numeric input requested)
 200 INPUT R\$ (string input requested)

NOTE: If a statement such as that in line 100 is being used to request a

numeric input and a string is entered, the computer will ask for the entry to be made again. A message such as 'REDO' will be displayed. Conversely, if the operator supplies a numeric input to a string request such as that in line 200, it will be accepted. Remember that the computer 'sees' all input to a string request as 'characters' and these may be either numbers or text or symbols such as ';'.

THE INPUT & GOTO STATEMENTS

Chapter VIII

If the response to the INPUT request at line 200 above contains a comma, the computer will ignore the comma and all text after it.

If the following was the response:- 345, Berwick Road, only the '345' would be assigned to R\$, the remainder would be ignored.

This problem can generally be overcome by enclosing the response

in quote marks:- "345, Berwick Road", in which case the whole response would be assigned to R\$.

ENHANCED INPUT: LINE NUMBER "INPUT REQUEST STRING" ; VARIABLE
(String or
Numeric)

```
Example: 350 INPUT "WHAT IS THE LENGTH";L
          400 INPUT "TELL ME YOUR NAME PLEASE";N$
          460 INPUT "HOURS WORKED THIS WEEK";HW
          570 INPUT "MARKS FOR TEST 1";M1
```

The alternative version to the enhanced input statement is:

LINE NUMBER PRINT " STRING " ; : INPUT Variable (String
or Numeric)

```
Examples: 780 PRINT "WHAT IS THE LENGTH";:INPUT L
           840 PRINT "TELL ME YOUR NAME PLEASE";:
INPUT N$
```

THE GOTO STATEMENT FORMAT

The format is: LINE NUMBER GOTO line number

```
Examples: 500 GOTO 650
           870 GOTO 100
```

GOTO is entered as one word.

CHAPTER IX

THE DATA, READ & IF-THEN STATEMENTS

We have seen how both the INPUT and LET statements can be used to provide data (information) for processing by a computer program. The interactive INPUT statement allows the entry of such data while the program is running and permits the same program to be re-executed so that different sets of data may be entered and processed. The major use of the LET statement is to provide a calculation or processing facility within the program using lines such as 100 LET A = (B + C) * D. We have also used LET to assign values to variables with lines such as 10 LET A = 6, but it is obvious that if the amount of data to be assigned is extensive then the program will be very lengthy due to the large number of LET statements that would be required. In everyday computing, extensive amounts of data are normally stored on disk as external 'datafiles' and the program accesses these as required. Where there is only a moderate amount of data however, this can be contained within the program and 'read' when required. Two new statements are used for this type of internal data handling, these being DATA and READ. The INPUT, LET, DATA and READ statements will be often seen in use together in the same program since they tend to be complementary to one another.

The DATA statement:- Provides a list or set of data items, each item separated from the next by a comma.

The READ statement:- Tells the computer to read an item (or items) from the data list and to assign that data item to a variable (numeric or string).

Consider the following program:

```
1990 CLS
2000 DATA 17,45,61,85,92      <- Data list
2010 READ F                    <- Read and
assign value to F
List 24 2020 PRINT;F          <- Print the
value of F
2030 GOTO 2010                <- Transfer back
to 2010 to
2040 END                      read and
assign the next value to F
```

The following is the BBCBASIC display on executing List 24:

```
|17      | <- First data item
printed
|45      | <- Second data item
printed
```

```

|61          |          <- Third data item
printed
|85          |          <- Fourth data item
printed
|92          |          <- Fifth data item
printed
|Out of DATA at line 2010 |          <- ERROR MESSAGE!
|>_         |

```

When this program is executed, the screen is cleared, the computer notes that a DATA line exists at line 2000, and then the READ statement is encountered for the first time. The first value, 17, is read and assigned to variable F and at line 2020 this value is printed. Line 2030 transfers control back to 2010 where the second value, 45, is read and assigned to F and at line 2020, this value is printed. This process continues until all of the other values in the DATA list have been read and printed, after which line 2030 will again transfer control back to 2010 where the computer will attempt to read but not be able to find any further values on the DATA line. This is recognised by the computer as an error in the program and the BBC, for example, would print the error message shown on the display above and program execution would be halted. You will note that the END statement on line 2040 is never executed in this program!

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

When a program containing a DATA statement or statements is executed, the computer uses a 'pointer' to keep track of the item that is the next to be READ on the list of DATA items. Initially, that pointer will be set to the first item on the first DATA line and this is illustrated below:

```

      |
10 DATA 17,45,61,85,92

```

The 'pointer' is pointing at 17 and this will be the first value to be assigned to F by the READ statement on line 2010. The 'pointer' will then move to the next value on the DATA line, the value 17 will be printed at 2020 and line 2030 will transfer control back to 2010 again. The situation now will be:

```

      |
10 DATA 17,45,61,85,92

```

The pointer is now pointing at 45 and this will be the next value to be READ and printed. The program continues 'looping', the pointer moving as shown below:

```

      |
10 DATA 17,45,61,85,92      Next value to be read is
61

```

```

      |
10 DATA 17,45,61,85,92      Next value to be read is
85

```


85		<- Fourth data item
printed		
92		<- Fifth data item
printed		
>_		<- NO ERROR MESSAGE!

SYMBOLS WHICH MAY BE USED WITH THE IF-THEN STATEMENT

<u>Computer Symbol</u>	<u>Meaning</u>
=	is equal to or the same as
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
<>	is not equal to or the same as

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

Examples of some simple IF-THEN statements:

<p>100 IF A = 98 THEN 140 variable A</p> <p>line 140.</p>	<p>If the value stored in numeric is 98 then transfer execution to line 140.</p> <p>If not, execution will continue at the line following line 100.</p>
<p>230 IF K <> 23 THEN 500 numeric variable K</p> <p>to line execution following line</p>	<p>If the value stored in is not 23 then transfer execution 500. If it is equal to 23, then will continue at the line 230.</p>
<p>3509 IF K > 100 THEN 240 numeric variable F</p> <p>value is execution will line 350.</p>	<p>If the value stored in is greater than 100 then transfer execution to line 240. If that equal to or less than 100, continue at the line following</p>
<p>500 IF X > 1000 THEN PRINT "TOO HIGH!" in numeric</p> <p>than</p> <p>is equal</p>	<p>If the value stored variable X is greater 1000, TOO HIGH! will be printed. If that value</p>

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

THE DATA STATEMENT AND 'VALUES'

The 'values' or items in DATA statements must be separated by commas:

```
120 DATA 8,27,-3.5,0,2.3E+6
350 DATA FRED,JOHN,BILL,ANNE,TERRY,RUTH
```

DATA statements may contain integers, decimals, exponential numbers, strings, but not expressions. The following DATA statements are not valid:

```
260 DATA 75,86;0,-17.6,83
      |
      Should be a comma
```

```
560 DATA 21,-1,A+7,26
      |
Expressions not permitted except as strings
```

```
590 DATA 1,-7,96,85,
      |
      No comma after last item
```

DATA statements may occur anywhere within a BASIC program but as a matter of style and readability, it is suggested that they are sited either right at the beginning or immediately before the END statement. Putting DATA statements at the beginning makes the program run marginally faster.

```

                                     TRACE   L   W   A
90 CLS
100 PRINT "LENGTH";TAB(10);"WIDTH";TAB(20);"AREA"
110 READ L
8   160
120 IF L = -999 THEN GOTO 810
9   153
130 READ W
6.5 292.5
List 26 140 LET A=L*W
150 PRINT;L;TAB(10);W;TAB(20);A
160 GOTO 110
800 DATA 20,8,17,9,45,6.5,-999
810 END
```

When executed, this program would display the following:

LENGTH	WIDTH	AREA
20	8	160
17	9	153

45	6.5	292.5	
>_			

This program illustrates:

- 1 Two READ statements (separate).
- 2 The IF-THEN statement
- 3 The 'dummy value', -999
- 4 The DATA statement at the end of the program
(before END).

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

LINE 100 - prints the headers
LINE 110 - READS the first value for L from the DATA list at
line 800
LINE 120 - checks to see if the value just READ is the dummy
value
LINE 130 - READS the next value for W from the DATA list at
line 800
LINE 140 - calculates the area A, of the rectangle using L and
W
LINE 150 - prints the values of L, W and A under the headers
LINE 160 - transfers execution back to line 110 where the next
value for
 L is READ.

The DATA values are being READ in pairs using TWO separate READ statements. When the pointer finally reaches -999 and this value is READ for L, line 120 will transfer execution to line 810 where program execution will terminate, thus avoiding an OUT OF DATA error.

Programs of this nature are easily modified to find other rectangular areas by simply changing the values on line 800 or by adding further values or DATA lines to the program.

More than one value at a time may be assigned by a READ statement. For example, the two READ statements on lines 110 and 130 could be combined as follows, remembering to delete line 130:

110 READ L,W (Note the comma between L and W)

If this modification were to be made to list 26, then we would have to add an additional 'dummy value' to the DATA list at line 800:

800 DATA 20,8,27,9,45,6.5,-999,-999

As two values are now being READ at one time, we must have sufficient values on the DATA line to avoid an OUT OF DATA error. When these dummy values are finally READ by the combined READ statement at line 110, -999 will be assigned to L and -999 to W. The second dummy value (last item in the DATA list) may in fact be any value whatsoever since it is only the first dummy value which is being used to check for the end of the list.

Consider another program:

	<u>TRACE</u>	<u>S</u>	<u>N</u>	<u>X</u>
2090 CLS				
2100 DATA 18,48,73,63,15,34,-999			0	0
18				
2110 LET S = 0			18	1
48				
2120 LET N = 0			66	2
73				

63	2130 READ X	139	3
<u>List 27</u>	2140 IF X = -999 THEN GOTO 2180	202	4
15	2150 LET S = S + X	217	5
35	2160 LET N = N + 1	252	6
34	2170 GOTO 2130	286	7
-999	2180 PRINT;S/N;TAB(10);S;TAB(20);N		
	2190 END		

This program produces the following display:

40.85	286	7		Displayed are the
average of the				values, the sum of the
>_				the number of values!
values and				

At LINE 2180 - $S/N = 286/7 = 40.85$ (Average of the values)
 $S = 286$ (Sum of all DATA values, excluding the dummy value)
 $N = 7$ (Number of DATA values READ and processed).

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

NOTE: In this program, a calculation is being carried out on a PRINT line, (2180). There is no LET statement involved. In a situation such as this, the computer first calculates the value of S/N and then prints the result.

For example, 1200 PRINT A/B + C^3. The expression is first evaluated and then the result is printed.

List 27 could be used to determine the average and the sum of any number of values by simply retyping line 2100 to suit, adding more DATA lines if required and remembering to include the dummy value, -999, as the last item on the last DATA line:

For example: 2100 DATA 18,48,73,63,15,35,34
 2101 DATA 67,32,90,78,14,27,26
 2102 DATA 55,43,-999

ANALYSIS OF LINES 2110, 2120, 2150, 2160

LINE 2110 - sets numeric variable S to zero. This is known as 'initialising' a variable and is used in this case to ensure that S is at 0 before the program starts summing the values. THE INITIALISATION OF VARIABLES IS A COMMON OPERATION. IT DOES NOT NECESSARILY MEAN SETTING A VARIABLE TO ZERO; IT MAY BE SET TO SOME OTHER STARTING VALUE. Many BASICs have a program statement, 'CLEAR'. This has the effect of setting all variables, both numeric and string, to zero. Setting a string variable to 'zero' effectively clears out any string content and leaves the variable empty. This is generally called a 'null string'. The statement LET A\$ = "" has the same effect for string variables as does the CLEAR statement.

LINE 2120 - initialises numeric variable N to 0.

LINE 2150 - LET S = S + X. This type of statement should be viewed as shown below:

LET	S	=	S	+	X
	NEW VALUE		OLD VALUE		LAST DATA VALUE
	of S		of S		READ

(increment)

It must be remembered that when the computer encounters a statement such as this, it firstly calculates the value of the expression on the right of the equals sign and then assigns the result of this back to the variable on the left of the equals sign.

now 18	First time through:	S = 0 + 18	Therefore S is
now 66	Second time through:	S = 18 + 48	Therefore S is
now 139	Third time through:	S = 66 + 73	Therefore S is

LINE 2160 - LET N = N + 1. This works as above except that each time this statement is executed, the value of N will increase by only 1.

now 1	First time through:	$N = 0 + 1$	Therefore N is
now 2	Second time through:	$N = 1 + 1$	Therefore N is
now 3	Third time through:	$N = 2 + 1$	Therefore N is

In this case, the variable N is being used as a 'counter' and we shall be dealing with these in more detail in a later chapter.

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

DATA statement may also be used to hold strings and READ statements will read these strings provided that the correct format is used. Consider the following program:

```
          90 CLS
          100 DATA FRANCE,PARIS,ENGLAND,LONDON,IRAN,
TEHRAN,END,END
          110 PRINT "COUNTRY";TAB(10);"CAPITAL"
          120 PRINT "-----";TAB(10);"-----"
List 28   130 PRINT
          140 READ CO$,CA$
          150 IF CO$="END" THEN GOTO 180
          160 PRINT CO$;TAB(10);CA$
          170 GOTO 140
          180 END
```

Execution of list 28 would produce:

COUNTRY	CAPITAL		<- Print headers (110)
-----	-----		<- Underline headers (120)
			<- Print 'blank' line
(130)			
FRANCE	PARIS		<- Read & print country &
capital city			
ENGLAND	LONDON		<- ditto (lines 140,
160)			
IRAN	TEHRAN		<- ditto
>_			<- Terminate execution

Now consider this program and the display produced on execution:

```
          100 CLS
          110 DATA JOHN BETTS,01-303-6605,FRED SMITH,
01-387-7667
          120 DATA ANNE BROWNE,0244-67654,PAULA WILLIAMS,
01-854-8773
          130 DATA END
          200 PRINTTAB(11);"TELEPHONE DIRECTORY"
          210 PRINT
List 29   220 PRINTTAB(5);"NAME";TAB(25);"TEL.NO."
          225 PRINT
          230 READ N$
          240 IF N$ = "END" THEN 900
          250 READ T$
          260 PRINTTAB(5);N$;TAB(25);T$
          270 GOTO 230
          900 END
```

	TELEPHONE DIRECTORY		<- Print title (200)
--	---------------------	--	----------------------

```

|                                     |   <- Print 'blank' line
(210)
|   NAME                             |   TEL.NO.                 |   <- Print headers
(220)
|                                     |   <- Print 'blank' line
(225)
|   JOHN BETTS                       |   01-303-6605            |   <- Read & print name
& tel.no.
|   FRED SMITH                       |   01-387-7667            |   <- ditto      (lines
230, 250
|   ANNE BROWNE                     |   0244-67654             |   <- ditto      & 260)
|   PAULA WILLIAMS                   |   01-854-8773            |   <- Terminate
execution
| >_                                 |

```

Although in List 29 we are READING two items, there is only the need for one dummy value since the READ statements are separate, lines 230 and 250. This is an alternative way to READ more than one item but still use only one dummy value. Compare this with the combined READ statement in List 28.

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

As with numeric values, the strings in a DATA statement must be separated by commas. This means therefore, that we have to be careful IF THE STRING ITSELF CONTAINS COMMAS (EMBEDDED COMMAS). If we look at the following line which is supposed to represent ONE DATA ITEM, an address, we might normally hand-write such an address using commas as shown on the line below. The computer however, sees this DATA line as containing FOUR ITEMS, each separated by a comma!:

```
200 DATA 312, GIPSY ROAD, WELLING, KENT.
```

If we want the information contained on line 200 to be treated as ONE DATA ITEM, we must surround that information by quote marks:

```
200 DATA "312, GIPSY ROAD, WELLING, KENT."
```

ALWAYS USE QUOTE MARKS WHEN STRING ITEMS HAVE EMBEDDED PUNCTUATION! The following lines represent THREE DATA ITEMS:

```
200 DATA "312, GIPSY ROAD, WELLING, KENT."
210 DATA "25, BERWICK ROAD, WELLING, KENT."
220 DATA "THE MANOR, BEXLEY, KENT."
```

Each of these DATA items could be READ by ONE STRING VARIABLE, for example:

```
500 READ A$      1st READ:  A$ = 312, GIPSY ROAD,
WELLING, KENT.
                  2nd READ:  A$ = 25, BERWICK ROAD,
WELLING, KENT.
```

etc

DATA statements may contain mixed string and numeric items:

For example: 10 DATA FRED,43,1.15,ALAN,27,3.86
 20 DATA ANN,21,2.20,JUNE,35,3.5
 30 DATA etc

```
40 DATA END  
  
                  .  
                  100 READ N$  
                  110 IF N$ = "END" THEN 9000  
                  120 READ A,H
```

This set of DATA items could well be an employees name, age and hourly rate of pay. We would need to READ 3 items for each employee using READ statement such as those shown on lines 100 and 120. By splitting these statements into a single and double READ, we need to use only one dummy value and in this example, END has been used.

Another example: 1000 DATA BOTHAM,78,COWDREY,196,RICHARDS,
35
 1010 DATA EDRICH,78 etc

```
1030 DATA END,END  
  
                  .  
                  2000 READ P$,S
```

This example uses a combined READ statement and therefore needs TWO dummy values. The DATA items could be a players' name and his score.

THE DATA, READ AND IF-THEN STATEMENTS

Chapter IX

LARGEST AND SMALLEST VALUES

We can use the DATA and READ statements with a set of numbers to illustrate a common computer application, the determination of the largest and the smallest value in such a set. Weather forecasters for example, use this type of operation in determining the hottest and coldest days; statisticians to check the least and most popular choices in a survey. Consider this simple example. We wish to find the largest number in the following list of numbers, 3, 8, 6, 9 and 2. We need to use a temporary storage variable (T) for this process and must first initialise T to a very small value, say -9999. Then we compare each of the above values (N) with the current value stored in T and if N is larger we transfer it to T. Therefore, T always contains the largest value READ so far and at the end of the program will hold the largest overall value.

The process look like this:

	<u>N</u>	<u>T</u>	
		-9999	<- Initialise to -9999.
Value 1	3		<- Compare N with T. If larger,
transfer N to T		3	
		3	<- T is now 3
Value 2	8		<- Compare N with T. If larger,
transfer N to T		8	
		8	<- T is now 8
Value 3	6		<- Compare N with T. If larger,
transfer N to T		8	
		8	<- T is still 8
Value 4	9		<- Compare N with T. If larger,
transfer N to T		9	
		9	<- T is now 9
Value 5	2		<- Compare N with T. If larger,
transfer N to T		9	
		9	<- T is still 9.

The largest value is now stored in T.

The following short program will achieve this aim:

List 30

```
100 DATA 3,8,6,9,2,-1      <- Data list, -1 is dummy
value
110 LET T = -9999          <- Initialise T to -9999
(small value)
120 READ N                 <- Read a value for N
130 IF N = 1 THEN 160      <- Terminate at dummy
variable
```

```
140 IF N > T THEN LET T = N      <- Check if N greater. If so,
transfer
150 GOTO 120                      <- Go for next number
160 PRINT "LARGEST NUMBER = ";T  <- Print largest number
170 END                          <- Terminate execution
```

To find the smallest value, only 3 changes need to be made:

```
110 LET T = 9999                  <- Initialise T to 9999
(large value)
140 IF N < T THEN LET T = N      <- Check if N smaller. If so,
transfer
160 PRINT "SMALLEST NUMBER = ";T <- Print smallest number
```


'Action' may be a BASIC statement, for example:

```
200 IF G = 9 THEN PRINT "CORRECT!"  
300 IF A$ <> "YES" THEN PRINT "GOODBYE!"
```

If the condition is FALSE, the computer proceeds to the next line

number following the IF statement. If the condition is TRUE, the

computer will take 'action' indicated after the THEN statement.

CHAPTER X

THE REM STATEMENT, ASCII CODES AND THE CHR\$ AND ASC FUNCTIONS

The need for program planning has been emphasised in an earlier chapter and the documentation produced by such a procedure, in the form of the original written plan and any display designs, is essential if at a later date the program requires modification or updating. Such documentation avoids the need to go through the program line by line to rediscover how it works, what variables are used and any other relevant facts. BASIC provides a statement which permits the inclusion of comments or remarks within the program itself and these remarks are also a useful aid when trying to determine the facts about that program. The new BASIC statement is 'REM', short for REMARK, and when the computer 'sees' a program line beginning with REM, it ignores it and passes on to the next line. Any information whatsoever may be typed after REM and this can be used to provide built-in information about the program action. The computer will never execute anything on a line beginning with the REM statement.

The following program is a fully REM'd version of List 27:

```
100 REM PROGRAM TO DETERMINE THE SUM,
110 REM AVERAGE AND NUMBER OF VALUES
120 REM PROCESSED, BY ALAN WILLIAMS.
130 REM WRITTEN 10TH JUNE, 1987.
140 REM
150 REM VARIABLES USED:-
160 REM S - SUM OF ALL VALUES
170 REM N - NUMBER OF VALUES
180 REM X - 'READ' VARIABLE
190 REM DUMMY VALUE IS -999
200 REM
2090 CLS
2100 DATA 18,48,73,63,15,35,34,-999
2105 REM THE NEXT TWO LINES INITIALISE
2106 REM S AND N (TO ZERO)
2110 LET S = 0
2120 LET N = 0
2130 READ X
2140 IF X = -999 THEN GOTO 2180
2142 REM LINE 2150 SUMS THE DATA VALUES AND
2144 REM LINE 2160 COUNTS THE NUMBER OF VALUES
2150 LET S = S + X
2160 LET N = N + 1
2170 GOTO 2130
2180 PRINT;S/N;TAB(10);S;TAB(20);N
2190 END
```

List 31

This version would obviously be much easier to understand than the original version, particularly for people other than the actual programmer. In lengthy programs the use of REM statements is almost mandatory. These statements can be inserted to describe what each component part of the program does. The 'run-time' version of such a program often has all of the REM statements removed as each statement does use some

of the computer memory, although this is not a problem on a computer with a lot of memory.

ASCII CODES

When the computer sends text to the printer, to another computer, or even to the screen, it does not actually send the pattern of dots which would make up the text characters, but rather a number between 0 and 255. The other device will then convert this number to a character or an operation such as a carriage feed. Handling text in this way is much faster, and requires much less memory than sending a dot pattern or 'bit-map'. It is therefore rather important for the devices to have the same code for the same character. The set of codes almost universally used by all micros is known as the ASCII set. ASCII stands for American Standard Code for Information Interchange, and was agreed in 1966 for use in teleprinter and other datacommunications applications. It was derived from an earlier five-bit code, named Baudot after its inventor, which had been used since the last century for tickertape machines, telexes and teleprinters. ASCII was originally defined as a seven-bit code, allowing 128 combinations (0-127). Codes 32 to 126 are defined characters, whilst 0-31 and 127 define operations, and are called Control Codes. For micro-computers, whilst the characters assigned to 32-127 are more or less universal, the same cannot be said for some of the control codes 0-31. The BBC in particular goes its own way with these codes. The standard ASCII codes 0 to 31, with their BBC meanings, are as follows:

CODE	NAME	STANDARD MEANING	BBC MEANING
0	NUL	Null or blank (i.e. nothing)	Does nothing
1	SOH	Start of header	Send 1 character to printer
2	STX	Start of text	Printer output on
3	ETX	End of text	Printer output off
4	EOT	End of transmission and graphic cursors	Separate text
5	ENQ	Enquiry	Join text and graphics cursor
6	ACK	Acknowledge (positive)	Enable VDU drivers
7	BEL	Bell, buzzer or sounder	Sound 'beep'
8	BS	Backspace	Backspace

9	HT	Horizontal tab	Horizontal tab
10	LF	Line feed	Line feed
11	VT	Vertical tab	Vertical tab
12	FF	Form feed	Form feed
(printer) /CLS	(screen)		
13	CR	Carriage return (enter)	Carriage return
14	SO	Shift out	Page mode on
15	SI	Shift in	Page mode off
16	DLE	Data link escape	CLG (Clear
graphics screen)			
17	DC1	Device control 1	Colour (p)
18	DC2	Device control 2	GCOL (graphics
colour) 1,(p)			
19	DC3	Device control 3	New actual
colour for col. no.(p)			
20	DC4	Device control 4	Restore default
actual colours			
21	NAK	Negative acknowledge	Disable VDU
drivers			
22	SYN	Synchronisation	Mode (p)
23	ETB	End text block	Create
user-defined character (p)			
24	CAN	Cancel	Define graphics
window (p)			
25	EM	End of medium	PLOT (p)
26	SUB	Substitute or end of file	Restore default
windows			
27	ESC	Escape	Does nothing
28	FS	File separator	Define text
window (p)			
29	GS	Group separator	Define graphics
origin (p)			
30	RS	Record separator	Move text cursor
to TAB(0,0)			
31	US	Unit separator	MOVE (p)

Code 127 means backspace and delete both as standard and on the BBC.

(p) means that the code has to be used with parameters.

THE REM STATEMENT, ASCII CODES AND THE CHR\$ AND ASC FUNCTIONS
Chapter X

The codes 32 to 126 refer to actual characters and are common to most micro-computers.

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

Computers commonly use an eight-bit (1 byte) character set, which doubles the number of characters to 255. There is no common agreement as to the use of these codes. On the IBM P.C. and its derivatives, 128-168 represent 'foreign' characters (156 is the £ sign); 169 to 223 are 'graphic' characters usable in text mode; 224 to 239 are Greek alphabet characters, and 240 to 253 are additional mathematical symbols. Most printers have the Epsom Italic Set as their default setting. This is incompatible with the IBM set beyond number 127, and this gives rise to the problems when trying to print out, for example, the boxes obtainable using the IBM graphic characters. Many printers can be configured to the IBM set to avoid these problems. On the BBC, codes 128-255 vary in meaning depending on the mode.

You may wonder what relevance this has to our study of BASIC. BASIC supplies us with two functions, CHR\$ and ASC which enable us to make use of ASCII codes and convert between the character and the code and vice-versa.

CHR\$(n) will give a one-character string corresponding to the ASCII code n i.e. 100 PRINT CHR\$(65)
will print a capital A on the screen
and 100 PRINT CHR\$(49)
will print the figure 1.

In addition, the use of CHR\$ will enable us to print characters not obtainable on the keyboard:

 100 PRINT CHR\$(252)
in mode 7 on Acorn computers will produce two parallel lines.

THE REM STATEMENT, ASCII CODES AND THE CHR\$ AND ASC FUNCTIONS

Chapter X

CHR\$ can also be used with certain control codes to produce effects:

```
100 PRINT CHR$(7)
```

will cause almost any computer to 'beep' and

```
100 PRINT CHR$(12)
```

will have the same effect as

```
100 CLS
```

One word of warning: when CHR\$() is used with certain control codes, you will be asking the computer to do the impossible and it will 'hang'. However, this will cause no damage to the machine, which will have to be reset. The first time this happens to you, call the tutor who will show you the recovery technique.

ASC is the reverse of CHR\$, and will give a number corresponding to a character

i.e.

```
100 PRINT ASC("a")
```

will print the figure 97.

The following program will give you a screen display of the main character set:

```
100 REM LISTING TO GIVE SCREEN DISPLAY OF ASCII SET
110 LET N=31
120 CLS
130 LET N=N+1
140 PRINT;N;" ";CHR$(N),;
150 IF N=127 THEN GOTO 170
160 GOTO 130
170 END
```

List 31a

If you would like the screen display to be tidier add the following line:

```
105 @%=8
```

amend line 170:

```
170 @%=10
```

and add line 180:

```
180 END
```

For the time being don't worry how lines 105 and 170 work.

Two final words: when using BBCBasic on a machine other than a genuine BBC, the character set is that for the machine (although as already stated in most cases they are identical). Secondly, a peculiarity of BBCBasic is that its VDU statement, which is not found in other basics, is for many purposes exactly the same as PRINT CHR\$, i.e. VDU65 is the same as PRINT CHR\$(65).

CHAPTER XI

THE RESTORE STATEMENT & MULTISTATEMENT LINES

Many applications in computing require that the same DATA be READ over and over again. We have seen however, that if having READ all of the items in a DATA list, we try to READ again, then an OUT OF DATA error will occur! This is illustrated below:

```
100 DATA 4,63,-9,0,-999
```

After having READ all items and the dummy variable (-999) the 'pointer' will be pointing at the position after the -999. Any further attempts to READ again will result in an error. To overcome this problem and to enable us to RE-READ the DATA we need some method of returning the pointer to the first value on the first DATA line. The RESTORE statement makes this possible

Below, is a simple program, the function of which is to display the location of an item in a store. The program allows a search to be made through the DATA items and if the item required is found the shelf and bin numbers are displayed. The DATA list entries comprise the item name, its shelf number and the bin in which it is stored:

```
10 DATA SCREWS,2,3,NAILS,5,27      <- Item names, shelf
numbers &
20 DATA NUTS,1,14,BOLTS,2,14      bin numbers
30 DATA PINS,3,7,WASHERS,7,23
40 DATA END,0,0                    <- Dummy values
50 CLS
60 INPUT "WHAT ITEM ";Q$           <- Ask operator for
item name
70 READ I$,S,B                      <- READ item and its
location
List 32 80 IF I$ = "END" THEN 140   <- Check if list
end reached
90 IF I$ = Q$ THEN 110              <- Check if item
required
100 GOTO 70                          <- Go back for next
READ
110 PRINT "SHELF ";S,"BIN ";B       <- Print location if
found
120 RESTORE                          <- Set pointer back
to start
130 GOTO 60                          <- Go back for next
search
140 PRINT "NOT STOCKED!"           <- Indicate item is
not listed
150 RESTORE                          <- Set pointer back
to start
160 GOTO 60                          <- Go back for next
search.
```

Execution of this program might display:

WHAT ITEM? NUTS NUTS?		<- Location of
SHELF 1 BIN 14 displayed		<- Location
WHAT ITEM? NAILS NAILS?		<- Location of
SHELF 5 BIN 27 displayed		<- Location
WHAT ITEM? HAMMERS HAMMERS?		<- Location of
NOT STOCKED! message.		<- Not found
WHAT ITEM?		

This program is an infinite loop. The ESCAPE key would need to be depressed to terminate execution.

NOTE: 1. Line 40 - 3 dummy values since at line 70, 3 items are being READ.

2. Line 60 - The answer INPUT to this request is known as the 'target string'. For this to be found, it must exist in the list exactly as typed in at line 60. For example, a space typed after the word SCREWS_ would result in a 'NOT STOCKED!' message!. An exact match is required.

3. Line 80 - After each READ, we must check to see if the end of the DATA list has been reached. This must be done before the comparison at line 90.

THE RESTORE STATEMENT & MULTISTATEMENT LINES

Chapter XI

4. Line 90 - Check to see if the target string (Q\$) is the same as the item (I\$) just read at line 70. If so, execution transfers to line 110 where the shelf and bin numbers are displayed.
5. Line 100 - Goes back for next READ if no match.
6. Lines 120/150 - Reset the DATA pointer back to the first item (SCREWS) in the DATA list.
7. Lines 130/160 - Goes back for the next item search.

List 32 contains the simplest example of a search routine, the likes of which have any applications in both the business and personal computing fields and there are a number of different search routines available which are very much faster than this. For the sake of space, the DATA list in this program is very short but it could be much longer if required and might contain for example, names and addresses, stamp collection information, telephone numbers, record/compact disc information and so on. The DATA, READ and RESTORE statements used together are a simple and straightforward method for providing such a Database facility on a personal computer system. The disadvantage of this program is that any modifications required to DATA items must be done by editing the DATA lines accordingly. Consequently, if, for example, the location of SCREWS was changed in the store, the DATA entry would have to be edited to accommodate this change. Thus the reason for employing external datafiles on disk where such modifications can be made and recorded while the program is running. The methods for handling external datafiles are not covered in this book, but are covered in the companion second year course.

MULTISTATEMENT LINES

Lines 120/130 and 150/160 in List 32 duplicate the same operation, namely:

```
RESTORE  
GOTO 60
```

We could modify the program to avoid this duplication and the subsequent waste of program space by making the following changes:

```
80 IF I$ = "END" THEN PRINT "NOT STOCKED":GOTO 120
```

Lines 140, 150 and 160 may be deleted as they are no longer required. Line 80 is an example of a multistatement line. There are now two statements on this line, the IF-THEN statement and the GOTO statement. Note that they are separated by a COLON. When the computer executes line 80 the comparison

is made to check if the end of the DATA list has been reached. If so, the message is printed and execution transfers to line 120. If not, execution 'drops through' to line 90 and continues from there.

Lines 120 and 130 may also be combined to read, 120, RESTORE: GOTO 60.

The second half of the program would now look like this:

```
80 IF I$ = "END" THEN PRINT "NOT STOCKED!":GOTO 120
90 IF I$ = Q$ THEN 110
100 GOTO 70
110 PRINT "SHELF ";S,"BIN";B
120 RESTORE:GOTO 60
```

THE RESTORE STATEMENT & MULTISTATEMENT LINES

Chapter XI

The computer 'sees' lines 80 and 120 as each having two separate statements, the colon being used to separate these statements. Multistatement lines can provide a notable saving in memory in lengthy programs and sometimes make a program more efficient. They can however make a program listing difficult to read. It is generally obvious where some advantage is to be gained in a program by using multistatement lines.

Examples of multistatement lines:

1. 10 A=4:B=-7:C=0:D=345
2. 20 IF X = 8 THEN PRINT "WRONG!":W = W + 1:GOTO 20
3. 50 PRINT "HELLO":PRINT "NICE TO SEE YOU!"
4. 90 PRINT:PRINT:PRINT

1. Values are assigned to four variables using one program line.
2. If the value of X = 8 then the message WRONG! is printed, the value of W is incremented by 1 and execution is transferred to line 20. If not, program execution would continue at the line following line 20.
3. This line, when executed would print: HELLO
4. Three 'blank' lines would be printed on the screen.

Some care must be taken with multistatement lines. For example, when there are two IF-THEN statements on one multistatement program line, providing the first IF-THEN statement is true the second IF-THEN statement will be executed. If however, the first IF-THEN statement is false, the second will not be 'seen' by the computer!

STATEMENT FORMATS

The RESTORE statement:

LINE NUMBER RESTORE

Action: The computer resets the DATA pointer to the beginning of the DATA

list. (Note that some BASIC's have a RESTORE statement that will

reset the pointer to any designated item in the DATA list, but this

is not available on most systems.)

MULTISTATEMENT LINES:

LINE NUMBER STATEMENT : STATEMENT : STATEMENT :
STATEMENT

Action: Permits the use of more than one statement on a
program line.

Statements are separated by COLONS.

CHAPTER XII

ROUNDING & RANDOM NUMBERS

BASIC has a built-in set of 'Library Functions'. We have already encountered CHR\$ and ASC, and we shall meet more in a later chapter. Two of these functions are particularly useful in many situations and we shall therefore deal with these two now. It was mentioned in the chapter on Computer Arithmetic that microcomputers generally work to a precision of six or seven significant digits but if, for example, we are writing a program involving cash transactions we simply require figures presented to two decimal places (pounds and pence). We therefore need the ability to correctly 'round' results to this format and some computers have their own particular statement for this purpose. In this book, we shall use a rounding expression which will work in any BASIC and which uses one of the BASIC library functions, INT. This is short for INTEGER and it has effect of removing any figures after a decimal point, leaving a whole number result, so if we 'INT' 5.678, the result is 5, the decimal component .678 having been removed. The effect of this statement is to produce a whole number result not greater than the original number! Below are some examples of this statement in use:

```
100 LET X = INT(98.54532)      <- 98 stored in X after
execution
230 LET Y = INT(12.1)         <- 12 stored in Y after
execution
345 LET Z = INT(-5.6)         <- -6 stored in Z after
execution!
                                (Note: -5 is greater than
-5.6!)
```

The statement to round any value stored in say X, to two decimal places is

LET X = (INT(X * 100 + 0.5))/100. Following, is an explanation of this statement. Let us assume that the value stored in X is 6.5367 and we want this value rounded to two decimal places:

Step 1 Move the decimal point two places to the right by multiplying by 100. X * 100
(653.67)

Step 2 Add .5 to the number.
X * 100 + 0.5 (654.17)

Step 3 INTEGER this value.
INT(X * 100 + 0.5) (654)

Step 4 Move the decimal point back two places by dividing by 100.
(INT (X * 100 + 0.5))/100 (6.54) This is the rounded value of X!

To round to one decimal place use LET X = (INT(X * 10 + 0.5))/10.

RANDOM NUMBERS

BASIC has the ability to produce random numbers which may be used for the purposes of simulations, games of chance or random tests. These numbers are not entirely random but for most uses may be considered so. A few BASICS require the use of an additional statement before the random number generator, usually RANDOMIZE, but this is not a general requirement. In BBCBasic the statement LET X=RND will store a random whole number between -2147483648 and 2147483647 in the variable X. In other common BASICS to use the same statement would produce a decimal value between 0 and 1. On its own, this is not particularly useful, but it can be used to obtain a random number between any given limits:

ROUNDING & RANDOM NUMBERS

Chapter XII

Considering BBCBasic only, a statement in the form `LET X=RND(Y)`, where Y is a whole number, will give X a random value between 1 and Y. Therefore, to simulate the throwing of a dice we would use the statement;

```
LET X=RND(6)
```

Which would assign a value between 1 and 6 to X.

One special case is `RND(1)`, which will give a decimal random number between 0 and 1.

In other BASICs, the general form would be `LET X=INT(RND(1)*Y)+1`, which, although a longer form will work equally well in BBCBasic.

Games of skill and chance are made considerably more interesting when an unpredictable outcome, made possible by the use of `RND`, is built in to the program.

CHAPTER XIII

COUNTERS

Counting is a common operation in computer programming and we can use counters to both control the number of times an operation is carried out, or to count the number of times an operation is carried out. When using DATA and READ statements, we have had to employ a dummy value to determine when the end of a DATA list has been reached. We could use a counter instead and the following program illustrates this:

```
      5 CLS
      10 DATA 3,7,-6,4.5,23          <- DATA list. No dummy
value!
      20 LET C = 0                    <- Initialise counter
to zero
      30 LET C = C + 1                <- Increment counter
by 1
List 33 40 READ N                    <- READ value for N
      50 PRINT;N                      <- Print the value
      60 IF C = 5 THEN 80             <- Check if more
values to READ
      70 GOTO 30                      <- Go for next value
      90 END
```

Execution of this program would result in the following display:

```
-----|
|3      | <- Each value in the DATA
list is |
|7      | printed. There is no
dummy  | <-
value, | but the counter ensures
|-6     | that only
|4.5    | five values are READ and
|23     | that we
OF DATA| do not end with an OUT
|>_    | error.
```

Line 20 initialises the value of C to zero. Line 30 increments this value by one each time this line is executed. Lines 40 and 50 READ a value from the DATA list and print it. Line 60 checks to see if the value of C is 5. If so, program execution is terminated; if not, control is transferred back to line 30 where C is again incremented. In this program, C is being used to ensure that only five READ operations take place and to avoid the use of a dummy value and OUT OF DATA error. The value 5 on line 60 represents the number of items in the DATA list. If there were 20 items in the DATA list, line 60 would be: 60 IF C = 20 THEN 80.

Here is a program which uses a counter and which prints the squares of a set of numbers held in a DATA list:

<u>X</u>	90 CLS	<u>TRACE</u>	<u>C</u>
	100 DATA 3,9,12,14,16		
	110 LET C = 0		0
	120 LET C = C + 1		1
3			
<u>List 34</u>	130 READ X		2
9			
	140 PRINT;C;TAB(10);X;TAB(20);X^2		3
12			
	150 IF C = 5 THEN 170		4
14			
	160 GOTO 120		5
16			
	170 END		

The following display would ensue on execution of List 34:

1	3	9		<- The counter value, the
DATA value				
2	9	81		and the square of the
DATA value				
3	12	144		is printed for each item
on the				
4	14	196		DATA list. Execution
terminates				
5	16	256		after five 'READS' and
'PRINTS'.				
>_				

COUNTERS

Chapter XIII

Another example of counter operation, in which the counter is being used as a 'value'.

```

    90 CLS
    100 LET C = 0          <- Initialise counter (C) to
zero
    110 LET C = C + 1     <- Increment counter by 1
List 34 120 PRINT;C,C^2,C^3  <- Print counter, counter
squared, cubed
    130 IF C = 10 THEN 150 <- END after 10 loops
    140 GOTO 110          <- Go for next value
    150 END
```

This program would provide the following display:

1	1	1		<- Prints counter, squared,
cubed for				the numbers 1 to 10.
Program				execution terminates at
3	9	27		130 was IF C = 100 THEN
10. If line				150, then
4	16	64		the program would produce
etc				for the numbers from 1 to
a table				100.
10	100	1000		>_

Consider these lines: 120 PRINT;C,(C*9/5)+32
 130 IF C = 100 THEN 150

Substitution of these two lines in the above program would provide a display showing the Fahrenheit equivalents for the Centigrade temperatures from 1 to 100°C!

A further example of the use of a counter as a 'value' in the program:

```

    5 CLS
    10 LET C = 0          <- Initialise counter to
zero
    20 INPUT "YOUR NUMBER ";N      <- Request number from
keyboard
List 36 30 LET C = C + 1          <- Increment counter
by 1
    40 PRINT;C;" MULTIPLIED BY ";N;" = ";C*N  <- Print result
    50 IF C = 12 THEN 70          <- END when counter = 12
    60 GOTO 30                  <- Go for next value of C
    70 END
```

This program is a 'Multiplication Table' program and when executed would provide the display below. The range of the table could be modified by changing the value (12) in line 50 to any desired value.

YOUR NUMBER? 2		<- Assume that 2 is input here
1 MULTIPLIED BY 2 = 2		<- Line 40 output 1st time through
2 MULTIPLIED BY 2 = 4		<- Line 40 output 2nd time through
3 MULTIPLIED BY 2 = 6		<- Line 40 output 3rd time through
etc		
12 MULTIPLIED BY 2 = 24		<- Line 40 output 12th time through
>_		

The preceding examples have used a counter to control the number of times an operation is carried out (and as a 'value' within the program). There is in fact a more efficient way of controlling program operation known as the FOR-NEXT statement and we shall be dealing with this in the next chapter.

The next examples deal with the use of a counter to count the number of times that something happens; for example, the number of times a wrong answer is given to a question or perhaps the number of times the answer

COUNTERS

Chapter XIII

to a question is 'Yes'. There are many uses of the counter in this form. A college registrar might use it to count the number of students admitted, an inventory clerk to count the number of items in stock or an employer to find how many employees have worked overtime.

Let us assume that a survey has been taken in the locality to determine the number of people from a representative group who smoke and drink. Some other characteristics of the respondees were also considered, namely age and sex. We want the program when executed to provide a display similar to the following:

QUESTIONNAIRE RESPONSE

SEX	AGE	SMOKE	DRINK
F	32	N	Y
M	24	Y	Y
F	55	Y	N
F	30	N	N
M	65	N	Y

NUMBER OF RESPONDEES: 5
NUMBER OF SMOKERS : 2

On this display, F and M represent Female and Male, Y and N represent the answers (Yes and No) to the questions 'Do you smoke?' and 'Do you drink?'. The number of respondees is the total number of people questioned during the survey.

We shall use the variable N in the program to count the number of respondees and the variable S to count the number of smokers.

We have to remember that all counters must be set to zero (initialised) before the counting process begins and each time we find a smoker we must increment the variable S by 1, that is LET S = S + 1.

To avoid complications with too many counters, we shall use dummy values to indicate the end of the DATA list and each DATA item will consist of four parts, the sex of the person, their age, whether they smoke and whether they drink.

An example of this is:

List 37

```
10 DATA F,32,N,Y,M,24,Y,Y
20 DATA F,55,Y,N,F,30,N,N,
30 DATA M,65,N,Y,END,0,0,0
90 CLS
100 PRINTTAB(9);"QUESTIONNAIRE RESPONSE":PRINT      <- Print
title
200 LET N = 0:LET S = 0                               <-
Initialise
210 PRINT "SEX","AGE","SMOKE","DRINK":PRINT          <- Print
headers
```

```

300 READ S$,A,Q1$,Q2$                                <- READ
data
310 IF S$ = "END" THEN PRINT:GOTO 1000              <- Check
for END
320 LET N = N + 1                                    <-
Increment N
330 IF Q1$ = "Y" THEN LET S = S + 1                 <-
Increment S
400 PRINTTAB(1);S$;TAB(10);A;TAB(22);Q1$;TAB(32);Q2$ <- Print
result
410 GOTO 300                                         <- Go for
next
1000 PRINT "NUMBER OF RESPONDEES: ";N                <- Print
total
1010 PRINT "NUMBER OF SMOKERS    : ";S               <- Print
smokers
1020 END

```

COUNTERS

Chapter XIII

The actual display produced by this program is:

	QUESTIONNAIRE RESPONSE		<- Title centred on
screen (line 100)			
			<- 'Blank' line
printed (end line 100)			
	SEX AGE SMOKE DRINK		<- Headers printed in
zones (line 210)			
			<- 'Blank' line
printed (end line 210)			
	F 32 N Y		<- DATA items printed
number headers			
	M 24 Y Y		(line 400 operates
5 times)			
	F 55 Y N		
.....			
	F 30 N N		
.....			
	M 65 N Y		
.....			
			<- 'Blank' line
printed (mid line 310)			
	NUMBER OF RESPONDEES: 5		<- Line 1000 output
	NUMBER OF SMOKERS : 2		<- Line 1010 output
	>_		

One further example program illustrating the use of a counter to check the number of wrong answers given in a simple maths test:

```
5 CLS
10 DATA 15,7,8,14,13,19,21,29,99,89,-999,0
20 LET W = 0
30 READ A,B
40 IF A = -999 THEN 500
50 PRINT "WHAT IS ";A;" + ";B;" ";INPUT Z
List 38 60 IF Z <> A + B THEN PRINT "WRONG!":LET W = W +
1:GOTO 50
70 PRINT "CORRECT!":PRINT
80 GOTO 30
500 PRINT "YOU GAVE ";W;" WRONG ANSWER";
510 IF W <> 1 THEN PRINT "S"
520 PRINT
530 END
```

The display from List 38:

	WHAT IS 15 + 7? 22		<- Question, and the
answer given			
	CORRECT!		<- Answer is right!

WHAT IS 8 + 14? 22		
.....		
CORRECT!		
.....		
WHAT IS 13 + 19? 33		
.....		
WRONG!		<- Incorrect answer!
WHAT IS 13 + 19?		<- Question repeated
etc		
YOU GAVE 1 WRONG ANSWER		<- Final message
>_		

The numbers for the test are held in the DATA list as five pairs; and dummy values are used to indicate the end of the list. The counter (W) which is to be used to record the number of wrong answers given is initialised to zero at line 20. The first two values to be added are READ at line 30 and the end of list check is made at line 40. The question using the two READ values is asked at line 50 and the answer checked at line 60. If incorrect, the message WRONG! is printed, and wrong answer counter W is incremented by 1 and control is transferred back at line 50 where the question is asked again! If correct, execution proceeds to line 70 where the message CORRECT! is printed and then control returns to line 30 for the next question.

COUNTERS

Chapter XIII

When the end of the DATA list is reached, line 40 transfers execution down to line 500 where a message is displayed showing the number of wrong answers given. Note that line 500 ends with a semi-colon and therefore the cursor will remain at the end of the message YOU GAVE X WRONG ANSWER_ until line 510 has been executed. If the number of wrong answers given is anything other than 1, then an 'S' will be added to the message. If only 1 wrong answer was given then an 'S' will not be added! Consequently, the output produced by lines 500/510 will be either of:

YOU GAVE 1 WRONG ANSWER or YOU GAVE X WRONG ANSWERS

The program could be easily modified to show the correct answer when a wrong answer is given and then proceed to the next question. Together with this, a random number generator could be used so that the questions asked are different each time!

STATEMENT FORMAT

LINE NUMBER LET VARIABLE = VARIABLE + VALUE

Action: Each time this line is executed the value stored in the variable
 will be incremented by VALUE.

For example: 100 LET P = P + 2

 The value of numeric variable P will be increased
by 2 each
 time this line is executed.

Other examples: 200 LET R = R * 2
 300 LET O = O + W
 400 LET Q = Q - 1

All counters must be initialised before use, either to zero or to a chosen starting value.

CHAPTER XIV

LOOPS AND THE FOR-NEXT STATEMENT

One area in which computers excel is that of repetitive operations. In many cases these operations are contained in a 'loop', the statements in such a loop being executed a certain number of times and in the chapter on DATA and READ we used a counter to control the number of times that a loop was executed. The two new statements, FOR and NEXT make loop operation much easier and these two statements, like DATA and READ, are always used together. The two programs which follow show both a counter and the FOR-NEXT statements used to achieve the same result:

	<u>List 39</u>		<u>List 40</u>
	5 CLS		5 CLS
	10 PRINT "DEAR JOE"		10 PRINT "DEAR
JOE"			
	20 PRINT		20 PRINT
	30 LET C = 0	=> =>	30 FOR C = 1 TO
5			
=> =>	40 LET C = C + 1	/\	40 PRINT "HAPPY
BIRTHDAY"			
/\	50 PRINT "HAPPY BIRTHDAY"	<=	50 NEXT C
	60 IF C = 5 THEN 80		60 PRINT
<= <=	70 GOTO 40		70 PRINT
"SINCERELY, FRED"			
	80 PRINT		999 END
	90 PRINT "SINCERELY, FRED"		
	999 END		

In List 39, the loop occupies lines 40 - 70 and when C attains a value of 5, execution transfers from line 60 to line 80. List 40 has a loop which occupies lines 30 - 50 and when C equals 6, execution drops through to line 60.

The FOR-NEXT statement shortens the program somewhat and this can be a useful attribute in lengthy programs where memory availability is at a premium. The FOR statement defines the start of the loop, the NEXT statement the end. Statements in between FOR and NEXT are said to occupy the 'body of the loop', in List 40 the body of the loop is only one line, line 40, but it could be many program lines. FOR and NEXT must be used together; for each FOR statement there must be an equivalent NEXT statement. Let us consider List 40 in more detail, particularly the action of lines 30 to 50:

- 1 Variable C on line 30 is first set to 1 by the FOR statement.
- 2 The body of the loop (line 40) is executed.
- 3 Line 50 tells the computer to determine the 'next' value for C and unless otherwise indicated, this will be one more than its previous value.
Consequently, the value of C is now 2.
- 4 If the value of C is 5 or less, the computer executes the body of the

loop again and 1 is again added to the value of C when the NEXT statement is encountered for the second time. This process continues until the value of C becomes greater than 5 and at that point, the computer exits from the loop to continue execution at line 60.

```
20 .....          TRACE          C  1  2  3  4  5  6
30 FOR C = 1 TO 5
40 PRINT "HAPPY BIRTHDAY"    TIMES printed  _  _  _  _  _
50 NEXT C
60 .....          The value of C is 6 at the end of
the loop!
```

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

Both Lists 39 and 40 would provide the following display:

DEAR JOE		<- 'DEAR JOE' message,
line 10		
		<- 'Blank line', line
20		
HAPPY BIRTHDAY		<- Message printed 5
times by		
HAPPY BIRTHDAY		loop
HAPPY BIRTHDAY		
HAPPY BIRTHDAY		
HAPPY BIRTHDAY		
		<- 'Blank line'
printed		
SINCERELY, FRED		<- Final message
printed		
>_		

Here are two further examples using FOR-NEXT loops where the value of the loop variable is being used in the output. What would be output by these programs?

5 CLS	90 CLS
10 FOR K = 1 TO 25	100 FOR T = 1 TO
5	
<u>List 41</u> 20 PRINT;K;	<u>List 42</u> 110 PRINT;T,
T^2,T^3	
30 NEXT K	120 NEXT T
40 END	130 END

FOR-NEXT loop normally count in steps of 1 unless otherwise instructed:

<u>LOOP</u>	<u>LOOP VARIABLE</u>	<u>SEQUENCE</u>
FOR N = 3 TO 8	N	3, 4, 5, 6,
7, 8		
FOR Q = 0 TO 4	Q	0, 1, 2, 3,
4		
FOR W = -1 TO 2	W	-1, 0, 1, 2

The sequence can be changed by specifying a STEP size or can be forced to count backwards:

<u>LOOP</u>	<u>LOOP VARIABLE</u>	<u>SEQUENCE</u>
FOR S = 1 TO 10 STEP 2	S	1, 3, 5, 7,
9 *		
FOR I = 0 TO 1000 STEP 20	I	0, 20, 40,
60, ... 1000		
FOR C = 10 TO 0 STEP -1	C	10, 9, 8, 7,
.....0		

```
FOR Z = 50 TO 20 STEP -5           Z           50, 45, 40,
.....20
```

* The loop variable will never exceed the limit; 9 is the last value displayed. If a STEP size is not shown, the step will default to 1.

A further two programs. What would be output?

```
          5 CLS                      5 CLS
          10 FOR Y = 0 TO 40 STEP 5    10 FOR X = 1 TO
2 STEP .25
List 43   20 PRINT;Y;                List 44   20 PRINT;X,
          30 NEXT Y                  30 NEXT X
          40 END                      40 END
```

Variables may be used to specify loop parameters

- (i) FOR F=N TO S
- (ii) FOR G=H TO U STEP V

In (i), if, say, N=7 and S=20, the sequence of values for F would be

7,8,9,10,20.

In (ii), let us assume H=10, U=20 and V=2. The sequence would then be

10, 12, 14, 16, 18, 20.

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

The FOR variable MUST be the same as the NEXT variable:

```
250 FOR P=1 TO 500
```

```
|  
|  
|
```

```
500 NEXT S
```

is not valid, and will produce an error message such as NEXT without FOR.

It is permissible, and often desirable to "nest" FOR - NEXT loops, i.e. for one loop to be inside another.

```
50 CLS
```

```
100 FOR F=5 TO 100
```

```
=====
```

```
110 FOR G=1 TO 100
```

```
===== ||
```

```
||
```

```
120 PRINT;F;TAB(10);G;TAB(20);F*G
```

```
|| ||  
Loop 2
```

List 45
Loop 1

```
130 NEXT G
```

```
===== || ||
```

```
||
```

```
140 NEXT F
```

```
||
```

```
=====
```

```
150 END
```

```
TRACE      F          G          F*G  
          5           1           5  
          5           2          10  
          |  
          5          100          500  
          6           1           6  
          |  
          100         100         10000
```

Many BASICs allow 24 nested loops!! They are very useful when used in conjunction with multi-dimensioned arrays, which will be dealt with in a later chapter.

Some BASICs, including BBCBasic, permit the omission of the variable on the NEXT line. For example, a loop may start with FOR F=1 TO 100 and end with just NEXT. The computer will not get confused as to which variable the NEXT statement refers to, but you might, so you are advised to continue to specify a variable after NEXT, particularly when nesting loops.

The FOR-NEXT statement may be used to READ items from a DATA line. This is a far better method than using a counter or

dummy value and simply requires that the upper limit of the loop is the same as the number of DATA items:

			<u>TRACE</u>	<u>D</u>
<u>A</u>	5 CLS			
	10 DATA 7,8,-5,6.23,28	<- 5 items to READ		1
7				
<u>List 46</u>	20 FOR D = 1 TO 5	<- Loop set to 5		
2	8			
	30 READ A:PRINT;A	<- READ & PRINT item		3
-5				
	40 NEXT D	<- End of loop		4
6.23				
	50 END			5
28				

Line 20 could equally be: 20 FOR D = 0 TO 4 (0,1,2,3,4 = 5 operations).

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

Consider the following program which may be used to perform a computation a fixed number of times. This program will determine the cost of say, a new car, in a number of years time assuming an inflation rate (fixed) of R:

```
190 CLS
200 INPUT "CURRENT RATE OF INFLATION (%)" ;R
210 INPUT "CURRENT COST OF NEW CAR (£)" ;K
220 INPUT "NUMBER OF YEARS HENCE" ;N
240 FOR Y = 1 TO N
```

Note the use List 47 250 LET K = K + (K *R/100)
of variable

```
260 NEXT Y
```

as the

```
270 PRINT:PRINT "COST OF CAR IN ";N;" YEARS = £";K
```

upper limit

```
280 END
```

the FOR -

NEXT loop!

Execution of this program would provide a display similar to this:

```
|CURRENT RATE OF INFLATION (%) ? 10 | <- Operator types
inflation rate
|CURRENT COST OF NEW CAR (£) ? 5000 | <- The current value
is typed in
|NUMBER OF YEARS HENCE ? 5          | <- And the number of
years hence
|                                     |
|COST OF CAR IN 5 YEARS TIME = £ ....| <- The computer
calculates and |>_                    |
prints the final message
```

The calculation at line 250 needs explanation. Variable K is being used to store the cost of the car; initially its current cost but also its future cost.

```
250 LET K = K + (K *R/100)
      |   |   |   ^
      New Old Increase
      Value Value due to
      of K  of K  inflation
```

Each time line 250 is executed, the new cost of the car after each year is calculated by adding the increase due to inflation to the old or previous value. This line, for the above sample display, would have been executed five times (five years). The R/100 part of the expression converts the value input for the inflation rate into a decimal value for calculation purposes. Consequently, as 10% was indicated by the operator, the new cost of the car after the first year

would be $5000 + (5000 * .1)$ or £5,500. This new value would now be used for the second year calculation, $5500 + (5500 * .1)$ and the process would continue a further three times to arrive at the final result. The variable N in the FOR-NEXT statement allows this program to be used for any value entered for the number of years.

In many programs, the value of the loop variable may be used in calculations carried out in the body of the loop. The following program illustrates this and provides a tabular display of temperature conversion from Centigrade to Fahrenheit:

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

```
190 CLS
200 PRINTTAB(9);"TEMPERATURE CONVERSION"
210 PRINTTAB(7);"CENTIGRADE TO FAHRENHEIT":PRINT
230 PRINTTAB(7);"CENTIGRADE";TAB(22);"FAHRENHEIT"
List 48 240 PRINTTAB(7);"-----";TAB(22);"-----"
250 FOR C = 1 TO 100 <-
Loop 1 to 100
260 LET F = (C * 9/5) + 32 <-
Convert C to F
270 PRINTTAB(10);C;TAB(26);F
280 NEXT C
290 END
```

The program will provide a tabular display of the Fahrenheit equivalents for Centigrade temperatures from 1 to 100 and would be easily modifiable to do this for any Centigrade temperature range chosen by the keyboard operator by incorporating the necessary input statements to obtain the lower and upper limits of the FOR-NEXT loop.

These limits would of course be signified on line 250 by two numeric variables replacing the correct values 1 and 100.

Consider the following program and determine what would be displayed on execution:

```
90 CLS
200 LET S = 0 <- Initialise
variable S to zero
210 PRINT "NUMBERS" <- Print header
220 FOR X = 1 TO 7 <- Loop from 1 to 7
List 49 230 LET S = S + X <- Increment S by the
value of X
240 PRINT;X <- Display X
250 NEXT X:PRINT <- End of loop
300 PRINT "THE SUM IS ";S <- Print result
310 END
```

A final example of FOR-NEXT action illustrates BREAK-EVEN ANALYSIS. The situation sometimes arises in real life where we have to make a decision whether to buy an item which will permit us to make something normally bought ready for use. Such a product might be yoghurt and we need to know that if we buy a Yoghurt Maker, the cost of that item will be recouped by making the yoghurt at home with relatively cheap and easily obtained starting materials. We shall assume that the Yoghurt Maker costs £12.95 and the ingredients for each batch of home-made product, 8 pence. Set against this, yoghurt purchased ready-made from the shops costs 40 pence for an equivalent batch. It is not easy to determine when the 'break-even' point will arise so we shall arrange for our program to run for three years on the assumption that it will occur within this period. What we need the program to do is to provide a tabular display showing the week number, equivalent

home and shop costs for that week, and the savings made if any. Initially, the savings made will be negative but at some point, savings will become positive, and when this first occurs this will be the break-even point. The first week's 'Home Cost' displayed by the program will represent the initial cost of the Yoghurt Maker plus the cost of one lot of ingredients. The second week's 'Home Cost' will be the cost of the Yoghurt Maker plus two lots of ingredients and so on. The 'Shop Cost' is simply the running total assuming the purchase of one pot per week. Savings represent the difference between shop and home costs.

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

```
90 CLS
100 PRINT "BREAK-EVEN ANALYSIS":PRINT
120 PRINT "WEEK", "HOME", "SHOP", "SAVED"
125 PRINT " NO.", "COST", "COST"
130 FOR W = 1 TO 156                                <- Go
for 156 weeks
List 50 140 LET H = 12.95 + .08 * W                  <-
Find home cost
150 LET S = .40 * W                                  <- Find
shop cost
160 LET V = S - H                                    <- Find
savings
162 LET H = INT(100*H+.5)/100                        <-
Round H
164 LET S = INT(100*S+.5)/100                        <-
Round S
166 LET V = INT(100*V+.5)/100                        <-
Round V
170 PRINTTAB(0);W;TAB(10);H;TAB(20);S;TAB(30);V    <-
Output result
180 NEXT W                                           <- Go
for next week
190 END
```

The display produced on execution.

BREAK-EVEN ANALYSIS				<- Program title
				<- 'Blank line'
WEEK	HOME	SHOP	SAVED	<- Headers
NO	COST	COST		
1	13.03	.40	-12.63	<- Week 1 output
2	13.11	.80	-12.31	<- Week 2 output
3	13.18	1.2	-11.98	<- Week 3 output
.....				
.....	etc			

This program will run for 156 weeks regardless of when the break-even point occurs. An extra program line could be added so that program execution terminates when the break-even point has been reached:

```
175 IF V>=0 THEN PRINT "BREAK-EVEN AT ";W;" WEEKS":END
```

Line 175 above, forces the program to 'jump out' of the FOR-NEXT loop at a certain point. This is usually quite permissible and you will find examples of it in many computer programs. However, please note that a value for the loop variable, in this case W, is still held in the computer 'stack'. If you jump out of too many loops, the stack will eventually become full and you will get an error message such as 'Too many FOR's'. If you want to be the paragon of perfection you would amend the line above to read:

```
175 IF V>=0 THEN PRINT "BREAK-EVEN AT ";W;" WEEKS":LET  
W=157
```

This would produce exactly the same result, but would end the loop by setting the loop variable to greater than the maximum specified by the FOR statement. That is, when line 180 was executed, W would have a value of 157 which would cause execution to drop to line 190. Although not strictly necessary at the level of this course, this is the correct way to drop out of a loop.

Jumping INTO a loop is NOT permitted:

```
100 F=0:G=0  
110 F=F+1  
120 IF F=4 THEN 150  
List 50a 130 GOTO 120  
140 FOR G=1 TO 7  
150 PRINT;F*G  
160 NEXT G
```

LOOPS AND THE FOR-NEXT STATEMENT

Chapter XIV

If this program were to be executed, the message
Can't match FOR at line 160
would produced.

OTHER LOOPS

Most other BASICs will allow the use of other loops, such as WHILE - WEND. In BBCBasic, there is a REPEAT - UNTIL statement which you may wish to investigate and use.

STATEMENT FORMATS

LINE NUMBER FOR Variable = Value/Variable TO Value/Variable
STEPValue/Variable

STEP is only required if a value other than 1 is to be used.

Examples:	120 FOR W = 1 TO 90	350 FOR X = 1
TO 10 STEP .1	230 FOR E = 1 TO K	670 FOR G = N TO B
STEP -2	340 FOR M = V TO Q	890 FOR Z = F TO
100 STEP 2		

LINE NUMBER NEXT Variable

Examples:	560 NEXT W	1200 NEXT X:NEXT Y
		(multistatement line!)

At this point in the course we come to a watershed. Until now, we have covered only main elements of BASIC, i.e. the most essential parts. The Adult Education courses will not proceed beyond this point until the majority of students are happy they understand what has gone before, and there is sufficient time.

CHAPTER XV

ARRAYS & ARRAY VARIABLES

Until now, we have been using 'simple' numeric and string variables to store numbers and text strings and there are a large number of this type of variables available. We shall now discuss subscripted variables which for certain operations provide facilities not available with simple variables. The following are examples of subscripted variables:

	A(7)	M\$(55)	book(250)	
book\$(250)				
	subscript	subscript	subscript	
subscript				

Subscripted variables consist of a variable name, either numeric such as book, or string such as book\$, followed by a number in brackets. Most BASICs, including BBCBasic allow the use of descriptive variable names such as above, but very often we still use single letters.

A set or list of subscripted variables is called an ARRAY, and subscripted variables are also known as ARRAY VARIABLES. An array used for storing numbers is a numeric array, for strings a string array. Values may be assigned to subscripted variables in exactly the same way as with variables, that is by using the LET, INPUT and READ statements.

ARRAYS

One way to picture an array is to think of it as a set of labelled pigeon holes:

A(0)	A\$(0)
A(1)	A\$(1)
A(2)	A\$(2)
HELLO	A\$(3)
A(3)	
A(4)	
A(5) 16	
A(6)	
numeric array named	
A(7)	
(0-11). The array	
A(8)	
A\$ and has four	
A(9)	
the numeric	
A(10)	
the value 16 and	

The array on the left is a A and has 12 compartments above is a string array named compartments. Compartment 5 of array is being used to store

| A(11) | _____ |
array, HELLO.

compartment 2 of the string

Array compartments generally start from zero, although this particular compartment is not often used in array processing routines. An array is named by the variable used and compartments are referenced by the number (subscript) in brackets following the array name. Both of the above are examples of single-dimension arrays and these are the type we shall mainly be using in this book. Multi-dimension arrays are also available and an example named X follows:

	1	2
X(0,		
X(1,	27	
X(2,		
X(3,		4
X(4,		
X(5,	105	

In this two-dimensional array named X, the value 27 is stored in X(1,1), the value 4 in X(3,2) and 105 in X(5,1). Multi-dimensioned arrays may be more difficult to visualise and handle, but suit situations where several related items need to be processed.

ARRAYS & ARRAY VARIABLES

Chapter XV

Consider the following program which has no real use but does illustrate the processing of array variables:

```

    90 DATA 36,-90,BOOK,SCOTCH      <- Mixed DATA list
    100 DIM Y(4),H$(3)               <- Array declaration
    110 LET Y(3) = 47                <- Assignment with LET
    120 LET H$(2) = "CHAIR"          <- Assignment with LET
    130 READ Y(1),Y(2),H$(1)         <- Assignment with
READ
    140 IF H$(1) = "BOOK" THEN 200   <- Decision with array
variable
List 51 150 INPUT Y(0)              <- Assignment with
INPUT
    200 LET P = 3                    <- Simple variable
assignment
    210 READ H$(P)                  <- READ with variable
subscript
    220 LET Y(P+1) = Y(1) + Y(P)     <- Adding array
variables
    230 FOR K = 1 TO 3              <- Loop from 1 to 3
    240 PRINT;Y(K);TAB(10);H$(K)     <- Output array
variables
    250 NEXT K                      <- End of loop
    260 END
```

LINE 90 DATA 36,-90,BOOK,SCOTCH - Contains the DATA items which will be processed by the program.

LINE 100 DIM Y(4),H\$(3) - Whenever an array is to be used in a program, the program must tell the computer (declare) the name of the array and how many compartments are required, by using the DIM statement. DIM is short for DIMENSION, thus the dimension or size of array Y in the above program is four (actually five because of the zero compartment which most BASICs set aside). Similarly, array H\$ has four compartments. As has been mentioned previously, many array processing routines do not utilise compartment 0. When line 100 is executed, the computer reserves space in memory sufficient for the array storage. Some BASICs automatically allow the use of arrays up to 10 compartments without the need for a DIM statement but you are advised to use it whenever an array is necessary. (The maximum array size which may be reserved by a DIM statement depends very much on the amount of computer memory available.) This particular line when executed would 'set up' the following in memory:

Y(0)	H\$(0)
Y(1)	H\$(1)
Y(2)	H\$(2)

Y(3)	
Y(4)	

H\$(3)

LINE 110 LET Y(3) = 47 - Tells the computer to assign the value 47 to compartment 3 in numeric array Y.
LINE 120 LET H\$(2) = "CHAIR" - Tells the computer to assign the string CHAIR to compartment 2 in string array H\$.
LINE 130 READ Y(1),Y(2),H\$(1) - Tells the computer to READ from the DATA list and assign the 'values' read to Y(1),Y(2) and H\$(1).

The situation would now be:

Y(0)	
Y(1)	36
BOOK	
Y(2)	-90
CHAIR	
Y(3)	47
Y(4)	

H\$(0)
H\$(1)
H\$(2)
H\$(3)

LINE 140 IF H\$(1) = "BOOK" THEN 200 - Since the string stored in compartment 1 of the string array H\$ is BOOK, execution would now transfer to line 200. Note that in this program, line 150 would not be ARRAYS &

executed!

In the program lines which we have covered so far, the subscripts (compartment numbers) have been constants (numbers) but one aspect of arrays which makes them extremely useful is that subscripts may also be variables or expressions. The next lines illustrate this.

LINE 200 P = 3 - Simple variable P assigned a value of 3.

LINE 210 READ H\$(P) - Substituting the current value for P, this really means READ H\$(3), thus the DATA item SCOTCH is read and assigned to compartment 3 of the string array H\$>

LINE 220 LET Y(P+1) = Y(1) + Y(P) - Similarly, this line may be read as

LET Y(4) = Y(1) + Y(3). Adding Y(1) and Y(3) gives 83 and this value is stored in compartment 4 of numeric array Y.

The situation would now be:

Y(0)		H\$(0)
Y(1)	36	H\$(1)
BOOK		
Y(2)	-90	H\$(2)
CHAIR		
Y(3)	47	H\$(3)
SCOTCH		
Y(4)	83	

LINE 230, 240 AND 250 - Use a FOR-NEXT loop to display the contents of the two arrays as below. Note that Y(4) is not displayed by these lines!

36	BOOK	<- Y(1) &
H\$(1) printed		
-90	CHAIR	<- Y(2) &
H\$(2) printed		
47	SCOTCH	<- Y(3) &
H\$(3) printed		
>_		

STATEMENT FORMATS

LINE NUMBER DIM Variable(Subscript)

Examples:

100 DIM F\$(20)

10 DIM J(1000)

200 DIM A(10),B(20),X\$(100),Y\$(25) <- Multiple DIM

statement

DIM statements are invariably placed at the beginning of a program since they must have been 'seen' (executed) before any associated arrays are used! Note that the subscript in a DIM statement may also be a variable. If this is the case, the variable must have been defined before the DIM statement is executed:

```
Example:  10 INPUT "ARRAY SIZE ";S
          20 DIM A$(S)
```

Specific compartments or locations in an array are referenced or referred to by a subscript and this subscript may be a constant, variable or expression. If the simple variable Q has a value of 7 then $X(Q) = X(7)$, $X(Q*4) = X(28)$ and $X(Q+4) = X(11)$. The computer first evaluates the subscript in order to know precisely which compartment to refer to. If the subscript when evaluated is less than 0 or more than the maximum declared in the DIM statement, an error will ensue! If the value is not a whole number, say 3.4, the INTEGER of that number is the value generally used, in this case 3.

ARRAYS & ARRAY VARIABLES

Chapter XV

COMMON ARRAY PROCESSING ROUTINES

Many array processing techniques make use of the fact that the subscript can be a variable. If for example, the value of S is 5, then location A(S) actually refers to the fifth compartment of array A. A number of standard routines follow and each is written in two forms, the 'long way' and the 'short way'. The short way invariably uses a FOR-NEXT loop.

ROUTINE 1 - Initialising all array compartments to zero:

```
100 LET A(0) = 0
8
110 LET A(1) = 0
120 LET A(2) = 0
130 LET A(3) = 0
  ||
  \|
180 LET A (8) = 0
```

```
100 FOR K = 0 TO 8
110 LET A(K) = 0
120 NEXT K
```

ROUTINE 2 - Reading DATA into an array:

```
100 READ A(1),A(2),A(3)
TO 8
110 READ A(4) .....
  ||
  \|
120 NEXT K
```

```
100 FOR K = 1
110 READ A(K)
120 NEXT K
```

ROUTINE 3 - Inputting data into an array:

```
100 INPUT A(1)
110 INPUT A(2)
120 INPUT A(3)
```

```
100 FOR K = 1 TO 8
110 INPUT A(K)
120 NEXT K
```

ROUTINE 4 - Printing the array contents:

```
100 PRINT;A(1)
110 PRINT;A(2)
120 PRINT;A(3)
  ||
  \|
```

```
100 FOR K = 1 TO 8
110 PRINT;A(K)
120 NEXT K
```

ROUTINE 5 - Summing the contents of an array:

```
100 LET S = 0
110 LET S = S + A(1)
TO 8
120 LET S = S + A(2)
+ A(K)
130 LET S = S + A(3)
  ||
  \|
```

```
100 LET S = 0
110 FOR K = 1
120 LET S = S
130 NEXT K
```

ROUTINE 6 - Copying from one array to another:

```
      100 LET B(1) = A(1)                                100 FOR K = 1
TO 8   110 LET B(2) = A(2)                                110 LET B(K) =
A(K)   120 LET B(3) = A(3)                                120 NEXT K
      ||
      \/
```

The FOR-NEXT statement reduces the number of statements which have to be used. To adapt these routines to handle say, 50 compartments, would only require that the upper value of the FOR-NEXT loop be changed to 50. The 'long way' would be difficult!

ARRAYS & ARRAY VARIABLES

Chapter XV

ARRAY USAGE

The following three programs are examples of how arrays may be used.

PROGRAM 1 - TABLE LOOK-UP

This program will permit a Sales Manager to determine the commission payable to each of his sales representatives based upon their monthly sales. The commission payments will be made as follows:

<u>SALES</u>	<u>COMMISSION</u>
Up to £2000	£ 0
£2000-£4000	£100
£4000-£6000	£300
£6000-£8000	£500
£8000 upwards	£800

At the beginning of the program we need to set up two arrays, sales and commission, with the appropriate figures. We will not need compartment (0).

sales(1)	2000	commission(1)	0
sales(2)	4000	commission(2)	100
sales(3)	6000	commission(3)	300
sales(4)	8000	commission(4)	500

Sales over £8000 earn a set £800 commission. This is a suggested program:

List 52

```
10 REM LIST 52
20 DATA 2000,0
30 DATA 4000,100
40 DATA 6000,300
50 DATA 8000,500
60 DIM sales(4),commission(4)
80 CLS
100 PRINT "SALES AND COMMISSION":PRINT
120 FOR F=1 TO 4
130 READ sales(F),commission(F)
140 NEXT F
150 INPUT "NAME OF SALESMAN: ";name$
160 INPUT "TOTAL SALES FOR MONTH: ";total
170 FOR F=1 TO 4
180 IF total < sales(F) THEN commission=commission(F):
GOTO 210
190 NEXT F
200 commission=800
```

```
210 PRINT "NAME";TAB(16);"SALES";TAB(24);"COMMISSION":  
PRINT  
230 PRINT name$,total,commission  
250 END
```

Notes:

LINE 60 - Declares two arrays, sales and commission, each with five compartments.

LINES 120, 130 and 140 - Read sales and commission figures from the DATA list into the appropriate compartment in arrays sales and commission.

LINES 150 and 160 - Ask for the name of the salesman and their total sales for the month and store them as name\$ and total.

ARRAYS & ARRAY VARIABLES

Chapter XV

LINES 170, 180 and 190 - Search through the sales array. If total is less than the figure in sales, commission is set to the corresponding entry in the commission array. Note that the simple variable commission and the array commission() are completely separate and independent. For example, if our salesman's total sales are greater than £2000 but less than £4000, then commission would be set to the figure in commission(2), i.e. £100, and the program would jump out of the loop and GOTO line 210. If however, we have a top notch salesman with a total sales of £10000 then the FOR-NEXT loop will terminate and 'drop through' to line 200 where the default value of £800 will be awarded as commission!

PROGRAM 2 - EDITING INPUT DATA

In this program we shall use an array to store values entered from the keyboard. After all the values have been entered the program will print the current list of entries and ask if there are any modifications to be made. Following any necessary corrections, the program will provide the sum and average of all values entered.

List 53

```
90 CLS
100 DIM A(200)
105 CLS
(Y/N) ";
110 PRINT "CALCULATOR WITH MEMORY"
120 PRINT
130 PRINT "TYPE 0 TO END ENTRIES"
";
140 PRINT
300 FOR I = 1 TO 200
320 PRINT "ENTRY NO: ";I
330 INPUT A(I)
340 IF A(I) = 0 THEN 370
350 NEXT I
360 PRINT "NO MORE SPACE!"
A(I)
370 LET K = I - 1
400 PRINT
410 PRINT "CURRENT LIST"
AVERAGE IS ";S/K
420 FOR I = 1 TO K
430 PRINT "ENTRY";I;" ";A(I)
440 NEXT I
450 PRINT
500 PRINT "ANY CHANGES
520 INPUT Q$
530 IF Q$ = "N" THEN 600
540 PRINT "WHICH ENTRY
550 INPUT I
560 PRINT "NEW ENTRY ";
570 INPUT A(I)
580 GOTO 400
600 LET S = 0
620 FOR I = 1 TO K
630 LET S = S +
640 NEXT I
660 PRINT "THE SUM IS ";S
670 PRINT "THE
680 END
```

Notes:

LINE 100 - Declares the array named A with 200 (201) compartments.

LINE 130 - Tells the operator to terminate his entries with a zero!

LINES 300 to 360 - A loop which will allow up to 200 entries to be made. First time through, line 320 prompts the operator with the message 'ENTRY NO: 1 ?' and the number entered in response to line 330 is assigned to A(1). Line 340 checks to see if the terminator (zero) has been entered, if so, execution transfers to line 370, if not the next entry is requested. Should a full 200 numbers be entered, execution will drop through to line 360 where a message will be printed.

LINE 370 - the variable I contains the number of entries PLUS 1, so 1 is subtracted and the result stored in K.

LINES 410 to 440 - Print the current list of entries.

LINE 500 - Asks if any entries are to be changed. If the answer is no (N) then execution transfers to line 600. If yes (Y or any entry other than N) then line 540 asks for the entry number. The entry number (I) is input at line 550 and the new entry at line 570.

LINE 580 - Transfers back to line 400 to see if any other changes need to be made!

ARRAYS & ARRAY VARIABLES

Chapter XV

LINES 600 to 640 - Sum the array contents and lines 660 and 670 prints the results of the calculations.

This program simulates the operation of a Calculator with memory.

PROGRAM 3 - SORTING DATA

Sorting or 'ordering' means arranging information or data in alphabetic or numeric order. The main reason for this requirement as far as string information is concerned is that it enables lists of such information to be searched much more quickly than by using a 'sequential search' system where a comparison is made on each time in the list starting at the first and working through to the last. With a sequential search, if you are searching a list of say 1000 items and the item you want is the last on the list, then it will take 1000 comparisons to find that item! With an ordered list and using for example, a search system known as the 'Binary Search', an absolute maximum of only 10 comparisons are needed to find any item in a list of 1000 items!

Textual (string) information such as a name and address list does in fact look more sensible when presented in an ordered state and the sort routine could be used to achieve this somewhat more basic requirement. The ordering of numeric information is generally required for different purposes, it might be to determine highest and lowest values and to present the results of the sort as a printed list. An example of this might be to output a list of examination results in either ascending or descending order.

While there are a large variety of sort routines available, we shall consider the simplest of these, the 'Bubble Sort'. This sort routine is used for already established lists of data such as a random list stored in an external datafile or items on a DATA line within a program. An alternative routine, the 'Insertion Sort', maintains a permanently ordered list. It inserts new items in the correct place within the list and is generally used only for external datafiles stored on disc or cassette. The Bubble Sort is in fact a very slow routine but is useful for short lists of items. If large lists are envisaged then a faster alternative must be considered.

The following program lines illustrate the bubble sort routine used to sort five numeric items in an array named A. Optional LET statements have been omitted:

```
100 S=0                <- Set 'swap flag' to zero
110 FOR K=-1 TO 4      <- Loop from 1 to 1 less
    than items
120 IF A(K) <- A(K+1) THEN 140    <- Compare adjacent items
    in array
130 T=A(K):A(K)=A(K+1):A(K+1)=T:S=1 <- Swap and set 'swap
    flag' to 1
```

```
140 NEXT K           <- Go for next comparison
150 IF S=1 THEN 100  <- If swapped, repeat
process
160 .....           (if not, drop through
to 1.160).
```

Simple variable S is used as an indicator to check if a swap has occurred.

Simple variable T is used for temporary storage during a swap.

LINE 100 - The swap flag is initialised to zero.

LINE 110 - Sets up a FOR-NEXT loop of 1 to 4, that is 1 less than the number of items in the array to be sorted.

LINE 120 - Compares the value of each array variable with the next array variable:

When: K = 1 the comparison is between A(1) and A(2)

 K = 2 the comparison is between A(2) and A(3) etc

ARRAYS & ARRAY VARIABLES

Chapter XV

Let us consider the first pass through the loop with K equal to 1.

If A(1) is less than or equal to A(2) then execution simply transfers to

line 140 where K becomes 2 and the next comparison is made.

If however, A(1) is greater than A(2), line 130 executes.

Variable T is

used to store the value of A(1), A(1) is given the value of A(2), and

A(2) the value of T. The values have been 'swapped' between A(1) and

A(2). To record this, the swap flag S is set to 1.

ORIGINAL
ARRAY

AFTER EXECUTION OF
LINE 130 WITH K EQUAL TO 1

A(1)	9
A(2)	7
A(3)	2
A(4)	6
A(5)	4

stored in A(1)

been swapped.

A(1)	7
A(2)	9
A(3)	2
A(4)	6
A(5)	4

<- The values

<- and A(2) have

AFTER 1ST COMPLETE
PASS THROUGH F-N LOOP

A(1)	7
A(2)	2
A(3)	6
A(4)	4
A(5)	9

bottom of

the

beginning to

whole routine the swap flag will have

The highest value has now 'sunk' to the

the list after one complete pass through

FOR-NEXT loop. Lower values are

"bubble" to the top of the list. The

is now repeated because

been set to 1.

AFTER 2ND COMPLETE
PASS THROUGH LOOP

continues and after
pass, the
sorted. The third
the swap flag

AFTER 3RD COMPLETE
PASS THROUGH LOOP

The process
the third complete
array has been
pass will have set

A(4)	7		A(4)	7		to 1, so further
pass will take						
A(5)	9		A(5)	9		place. No swap will
occur and						
						therefore execution will
drop						through to line 160.

Following are two programs, the first of which is the program represented above and the second, containing an equivalent string sorting routine:

List 54

```

5 CLS:N=5
10 DATA 9,7,2,6,4
20 FOR K = 1 TO N:READ A:NEXT K
A$:NEXT K
100 S=0
110 FOR K = 1 TO N-1
120 IF A(K) <= A(K+1) THEN 140
THEN 140
130 T=A(K):A(K)=A(K+1):A(K+1)=T
A$(K)=A$(K+1):A$(K+1)=T$
140 NEXT K
150 IF S=1 THEN 100
160 FOR K=1 TO N:PRINT A(K):NEXT
PRINTA$(K):NEXT K
170 END

```

List 55

```

5 CLS:N=5
10 DATA PETER,SID,ANNE,JOHN,ANN
20 FOR K = 1 TO N:READ
A$:NEXT K
100 S=0
110 FOR K = 1 TO N-1
120 IF A$(K) <= A$(K+1)
THEN 140
130 T$=A$(K):
A$(K)=A$(K+1):A$(K+1)=T$
140 NEXT K
150 IF S=1 THEN 150
160 FOR K=1 TO N:
PRINTA$(K):NEXT K
170 END

```

ARRAYS & ARRAY VARIABLES

Chapter XV

String sorts are based on the ASCII values of the characters forming the strings. Letter A has an ASCII value of 65, B is 66, C is 67. Consequently, B is considered by the computer to be greater than A, and Z to be greater than all other letters in the alphabet. The string sort shown in List 55 will also differentiate between say, BROWN and BROWNE, or SMITH and SMITHSON and will order these correctly!

To sort a list into DESCENDING order, the comparators in line 120 are simply changed from `<=` (less or equal to) to `>=` (greater than or equal to).

CHAPTER XVI

LOGICAL OPERATORS & THE EXTENDED IF-THEN STATEMENT

BASIC provides two logical operators, AND and OR. The use of these operators extends the way in which the IF-THEN statement can be used and in many cases, makes comparisons and decisions easier to make. When AND or OR are used with IF-THEN, the statement so produced is said to be a compound condition, meaning that the outcome depends upon more than one simple factor being either true or false. Here are two imaginary examples:

IF S\$ = "F" AND A>65

A possible interpretation of this condition might be:

"if the sex of the person is female AND the age is over 65"

IF A>= 90 OR A\$ = "ATHLETE" This might be:

"if the average mark is greater than or equal to 90 OR the person is an athlete"!

Logical operators are frequently used in IF-THEN statements. Whether or not the statement following THEN is executed depends on the TRUTH of the condition between IF and THEN such as:

IF A=7 THEN LET B=2

If A has the value 7 then A=7 is TRUE, therefore LET B=2 is executed.

If A does not have the value 7 then A=7 is FALSE, therefore LET B=2 is NOT executed.

Looking at IF-THEN statements in this way appears to be a way of complicating the obvious, but its use becomes more apparent when dealing with COMPOUND CONDITIONS formed using logical operators, and at a later stage in conjunction with the TRUE and FALSE operators and Boolean logic.

Logical Operator rules

RULE 1 - A compound condition which is composed of two simple conditions joined by the AND operator is TRUE if and only if BOTH simple conditions are true. If only one or neither of the conditions is true, then the compound condition is false.

For example:	IF S\$ = "M"	<u>AND</u>	H > 72	
	^		^	
S\$	H	Simple	Simple	
TRUTH value of		<u>Condition</u>	<u>Condition</u>	
<u>Compound Condition</u>				
M	74	-	-	
-				
M	67	-	X	
X				
F	75	X	-	
X				

F
X

65

X

X

RULE 2 - A compound condition which is composed of two simple conditions joined by the OR operator is FALSE if and only if BOTH simple conditions are false. If only the one or both of the conditions are true, the compound condition is true.

LOGICAL OPERATORS & THE EXTENDED IF-THEN STATEMENT

Chapter XVI

For example: IF S\$ = "M" OR H > 72

S\$	H	Simple	Simple
TRUTH value of		<u>Condition</u>	<u>Condition</u>
<u>Compound Condition</u>			
M	74	-	-
-			
M	67	-	X
-			
F	75	X	-
-			
F	65	X	X
X			

The program which follows determines car insurance premiums. It uses only two factors to determine the premium and is not therefore very realistic, but it does illustrate the use of the AND operator to apply a compound condition based upon which a decision is made about the premium cost. The two simple conditions are that if the driver is under 25 the premium will be higher, and if the car is a 'sports' version rather than a 'standard' version the premium will be higher. The AND operator allows us to use various combinations of these conditions to determine the premium payable.

```
10 REM CAR INSURANCE CALCULATOR
90 CLS
91 DATA JOE,23,SPORTS,ANN,18,STANDARD
92 DATA JIM,45,SPORTS,CAROL,35,STANDARD
93 DATA FRED,101,SPORTS
100 PRINT "NAME","AGE","CAR","PREMIUM":PRINT
120 FOR K = 1 TO 5:READ N$,A,T$
200 IF A<25 AND T$ = "SPORTS" THEN P=400:GOTO 300
210 IF A<25 AND T$ = "STANDARD" THEN P=300:GOTO 300
220 IF A>25 AND T$ = "SPORTS" THEN P=250:GOTO 300
230 P=200:REM -SIEVE
300 PRINT N$;TAB(10);A;TAB(20);T$;TAB(31);P
310 NEXT K
320 END
```

List 56

NAME	AGE	CAR	PREMIUM	
1.100				<-Headers printed,
				<-'Blank' line
printed, 1.100				
JOE	23	SPORTS	400	<-First data set
print, 1.300				
ANN	18	STANDARD	300	<-Second data set
print, 1.300				
JIM	45	SPORTS	250	<-Third data set
print, 1.300				

```

|CAROL      35          STANDARD      200      | <-Fourth data set
print, 1.300
|FRED       101         SPORTS        250      | <-Fifth data set
print, 1.300
|>_
|

```

The first pass through the FOR-NEXT loop reads JOE, 23, SPORTS. Line 200 check to see if both of the simple conditions, $A < 25$ and $T\$ = \text{"SPORTS"}$ are true and as in this instance they are, a premium of £400 is assigned by this line, and execution transfers to line 300 where the screen output occurs. Line 310 returns of the next DATA read.

If, for example, we now consider the fourth DATA read, CAROL 35, STANDARD, it can be seen that the conditions on line 200 are not met, neither are the conditions on lines 210 and 220. In this instance execution will drop through each of these lines and the default premium of £200 will be awarded at line 200.

The AND and OR statements allow us to program the computer to make much more complex decisions than with the simple IF-THEN statement.

LOGICAL OPERATORS & THE EXTENDED IF-THEN STATEMENT

Chapter XVI

STATEMENT FORMATS

LINE NUMBER IF condition 1 AND condition 2 THEN
instruction(s)

Action: If BOTH conditions are TRUE the instruction(s) after the THEN statement will be carried out. If NOT TRUE, execution will drop through to the next program line.

LINE NUMBER IF condition 1 OR condition 2 THEN
instruction(s)

Action: If EITHER of the conditions are TRUE the instruction(s) after the THEN statement will be carried out. If NEITHER are TRUE, execution will drop through to the next program line.

```
EXAMPLES: 500 IF S$ = "M" AND A>=65 THEN PRINT "MALE SENIOR  
CITIZEN"  
          640 IF A = 3 AND C > 10 THEN 670  
          700 IF V < 1 OR V > 50 THEN PRINT "OUT OF RANGE!"  
          870 IF N$ = "FRED" OR N$ = "JIM" THEN 1000
```

IF-THEN-ELSE

In addition to the simple IF-THEN statement, most BASICs allow the use of ELSE to give instructions if an IF condition is FALSE.

```
10 IF A=7 THEN LET B=6 ELSE LET B=5
```

will have exactly the same effect, and is neater than

```
10 IF A=7 THEN LET B=6:GOTO 30  
20 LET B=5  
30 -----
```

Multiple ELSEs are allowed:

```
10 IF A=7 THEN LET B=5:GOTO 40  
20 IF A=6 THEN LET B=7:GOTO 40  
30 LET B=8  
40 -----
```

can be replaced by one line:

```
10 IF A=7 THEN LET B=5 ELSE IF A=6 THEN LET B=7 ELSE LET B=8  
40 -----
```

LOGICAL OPERATORS & THE EXTENDED IF-THEN STATEMENT

Chapter XVI

MULTIPLE CONDITIONS

When using multiple conditions, use brackets to determine the order of testing. Do not be afraid to use brackets even if you think they may be unnecessary as they can make it clear exactly what is to be tested.

The following two lines are very similar, but lead to different results:

```
100 IF (A=7 AND B=6) OR (A$="BOOK" AND B$="PAPER") THEN LET  
    C$="MAGAZINE" ELSE LET C$="LEAFLET"
```

```
110 IF A=7 AND (B=6 OR A$="BOOK") AND B$="PAPER" THEN LET  
C$="MAGAZINE"  
    ELSE LET C$="LEAFLET"
```

TRUTH TABLE

6	A	7	4	
-----	-----	-----	-----	-----
7	B	6	6	
-----	-----	-----	-----	-----
PAPER	A\$	CASE	BOOK	
-----	-----	-----	-----	-----
BOOK	B\$	PAPER	PAPER	
-----	-----	-----	-----	-----
Line 100:				
X	(1). A=7 AND B=6	-	X	
X	(2). A\$="BOOK" AND B\$="PAPER"	X	-	
X	(1) OR (2)	-	-	
-----	-----	-----	-----	-----
LEAFLET	C\$	MAGAZINE	MAGAZINE	
-----	-----	-----	-----	-----

Line 110:			
X (1). A=7	-	X	
X (2). B=6 OR A\$="BOOK"	-	-	
X (3). B\$="PAPER"	-	-	
X (1) AND (2) AND (3)	-	X	

CHAPTER XVII

THE ON GOTO & GOSUB STATEMENTS

Menu driven programs or programs which offer a multiple choice of actions are quite common in computing since they make it possible for relatively unskilled personnel to use what can be quite complex software. In the personal computing field, menus are often used and following is a display illustrating a simple example of such a use:

```
AREA CALCULATOR

1. RECTANGLE
2. TRIANGLE
3. CIRCLE
4. TERMINATE

SELECT OPTION REQUIRED (1-4)? _
```

This display offers the choice of an area calculation or to terminate program execution. The program segment which would produce this follows:

```
100 CLS
110 PRINT:PRINTTAB(12);"AREA CALCULATOR":PRINT
120 PRINTTAB(12);"1. RECTANGLE"
List 57 130 PRINTTAB(12);"2. TRIANGLE"
140 PRINTTAB(12);"3. CIRCLE"
150 PRINTTAB(12);"4. TERMINATE":PRINT
160 PRINTTAB(5);:INPUT"SELECT OPTION REQUIRED (1-4)";
N
```

At certain locations within the program, there would be the relevant routines to deal with the three area options on the menu. Perhaps the rectangular area calculation routine would be at line 1000, the triangular area routine at line 2000 and the circular area routine at line 3000. The program termination routine might be at line 4000 for example. We would need to incorporate program lines from 170 on, in order to divert program execution to the relevant routine depending on the value of N input at line 160. We could do this with IF-THEN statements as follows:

```
170 IF N = 1 THEN 1000      There could be problems here if the
operator
180 IF N = 2 THEN 2000      typed say 0 or 5. These are not
valid entries
190 IF N = 3 THEN 3000      since only four options are offered!
The
200 IF N = 4 THEN 4000      addition of a further line (165)
will deal
with this.
```

```
165 IF N < 1 OR N > 4 THEN 160      If the value of N is less
than 1                               or the value of N is greater
                                      4 go back.
```

If any entry other than 1, 2, 3 or 4 is input at line 160, line 165 will repeat the request SELECT OPTION REQUIRED (1-4). Line 165 is known as a 'guard against an invalid input' , or more colloquially a "mugtrap", and when using menus, such a line should always be used so that only legal entries can be made! If the above menu offered 40 options, we would obviously need 40 IF-THEN statements and this would make the program very unwieldy. There is however a new statement which suits the above situation admirably and this is the ON GOTO statement. Lines 170 to 200 inclusive could all be replaced in the program above by one new line (170) which would achieve precisely the same result as those lines:

```
170 ON N GOTO 1000, 2000, 3000, 4000
```

THE ON GOTO & GOSUB STATEMENTS

Chapter XVII

When this line is executed, if $N = 1$ program execution is transferred to the first line number in the ON GOTO statement, line 1000, if $N = 2$ to line 2000, if $N = 3$ to line 3000 and if $N = 4$ to line 4000. Consider the following sample line:

```
250 ON X GOTO 150,200,340,670,890,1200,5000
```

- a) If $X = 6$, the next line to be executed would be the sixth line number, 1200
- b) If $X = 2$, the next line to be executed would be the second line number, 200.

Note the commas between each line number in the ON GOTO statement, with no comma after the last line number. Expressions may be used in the ON GOTO statement as well as variables:

```
1000 ON B+1 GOTO 1200,3490,5670,8970
```

STATEMENT FORMAT

LINE NUMBER ON Variable or Expression GOTO line 1, line 2, line 3 ...

Action: The computer uses either the value of the variable or expression to determine which line number to go to next. If an expression is used and the value is a decimal, say 4.56, most computers will take the integer of this and in this case, proceed to the fourth line number.

Although included in this book for completeness, ON-GOTO is very seldom used in modern BASICs because of the following limitations:

1. The ON variable must increase by 1 for each line number in GOTO.
2. GOTO must be followed by a line number. This becomes difficult in BBCBasic when using PROCedures (discussed in the advanced class) and limiting in such as QuickBasic which do not even need line numbers!

NOTE: It is important when using menus to ensure that invalid inputs are not allowed and a line such as the example line above (165) should always be incorporated in the program immediately after the relevant input and before any ON GOTO or other statements.

SUBROUTINES

It is often the case that the same set of operations (routine) will occur a number of times in a program. If you consider the menu driven program above, each of the three options to calculate an area will result in the need to output the result of the area calculation concerned. The program lines to perform that output could be incorporated as part of each of the three area calculation routines and might take the form of a message such as THE AREA IS X. Obviously, we would then have a situation where the same program line was occurring three times in the program, a waste of memory space and bad program planning! In a case like this, we can use a common output for the message on a single program line, this line to be used by all three of the area calculation routines. A common routine like this is known as a SUBROUTINE and this can be CALLED whenever required from any part of the program. As mentioned, BBCBasic has a facility called a PROCedure which provides a similar facility in a more advanced and flexible way, and most modern BASICs provide something similar. However, all still allow the use of subroutines. The 'programs' overpage represents the form that the above program might take without a subroutine and the alternative version with a subroutine.

THE ON GOTO & GOSUB STATEMENTS

Chapter XVII

100 Menu goes here	<u>List 58</u>
170 ON N GOTO 1000,2000,3000,4000	
1000 Rectangle area routine here	
1100 PRINT "AREA IS ";X	In this
'standard' version	
1110 INPUT "ANOTHER RECTANGULAR AREA (Y/N)";Q\$	of the
program, lines 1120 IF Q\$ = "Y" THEN 1000	
1100,2100 and 3100 do the	
1130 IF Q\$ = "N" THEN 100	same thing
(providing that	
1140 GOTO 1100	the relevant
area calculation	
2000 Triangular area routine here	store the
result in variable	
	X!).
2100 PRINT "AREA IS ";X	
2110 INPUT "ANOTHER TRIANGULAR AREA (Y/N)";Q\$	This example
is somewhat	
2120 IF Q\$ = "Y" THEN 2000	trivial since
only one line	
2130 IF Q\$ = "N" THEN 100	in each area
routine is	
2140 GOTO 2110	common.
However, you will	
3000 Circular area goes here	often find
situations which	
	are similar occurring,
where	
3100 PRINT "AREA IS ";X	the number of
program lines	
3110 INPUT "ANOTHER CIRCULAR AREA (Y/N)";Q\$	common to each
routine is	
3120 IF Q\$ = "Y" THEN 3000	considerably
more than 1!	
3130 IF Q\$ = "N" THEN 100	
3140 GOTO 3110	
4000 PRINT "PROGRAM TERMINATED":END	

100 Menu goes here	<u>List 59</u>
170 ON N GOTO 1000,2000,3000,4000	In this version of
the	
1000 Rectangle area routine here	program, there is
now only 1	
	'PRINT' line at 3500.
Lines	
1100 GOSUB 3500	3500 and 3510
are the sub-	
1110 INPUT "ANOTHER RECTANGULAR AREA (Y/N)";Q\$	routine. The
new GOSUB	

1120 IF Q\$ = "Y" THEN 1000	statements on
lines 1100,	
1130 IF Q\$ = "N" THEN 100	2100 and 3100
'CALL' the	
1140 GOTO 1100	subroutine.
When any of these	
2000 Triangular area routine here	are executed,
the GOSUB 3500	
	statement transfers
control	
2100 GOSUB 3500	to line 3500,
the subroutine	
2110 INPUT "ANOTHER TRIANGULAR AREA (Y/N)";Q\$	has been
'CALLED'. The 2120 IF Q\$ = "Y" THEN 2000	PRINT
statement on line 3500	
2130 IF Q\$ = "N" THEN 100	executes, and
then the RETURN	
2140 GOTO 2110	statement on
line 3510 is	
3000 Circular area goes here	encountered.
RETURN tells the	
	computer to transfer
control	
3100 GOSUB 3500	back to the
line <u>FOLLOWING</u>	
3110 INPUT "ANOTHER CIRCULAR AREA (Y/N)";Q\$	the relevant
GOSUB statement.	
3120 IF Q\$ = "Y" THEN 3000	If the GOSUB
on line 1100 was	
3130 IF Q\$ = "N" THEN 100	the called
statement, then	
3140 GOTO 3110	control would
be transferred	
3500 PRINT "THE AREA IS ";X	back from the
subroutine to	
3510 RETURN	line 1110.
4000 PRINT "PROGRAM TERMINATED":END	

THE ON GOTO & GOSUB STATEMENTS

Chapter XVII

List 60

```
100 PRINT "READY...."          <- Print message
110 GOSUB 1000                  <- Go to the subroutine at
line 1000
120 PRINT "STEADY..."        <- Print message
130 GOSUB 1000                  <- Go to the subroutine at
line 1000
140 PRINT "GO!!!"             <- Print message
150 GOTO 2000                  <- Jump round the
subroutine
1000 FOR T = 1 TO 2000:NEXT T   <- Subroutine
1010 RETURN                    <- Return to program line
following
2000 END                        the relevant GOSUB call.
```

The subroutine at line 1000, an 'empty' FOR-NEXT loop, is being used as a time delay. The loop will count from 1 to 2000 and in doing so there will be a pause in program execution. The length of the pause may be controlled by the size of the loop. A loop of 1 to 4000 would take twice as long to process as the loop above. The time taken depends very much on the make of computer since some computers operate faster than others (have a faster 'clock' speed). You should try different loops on your computer to determine the necessary upper limits to provide certain time delays as this facility is useful when the need to display messages for a certain length of time occurs.

NOTE: Line 150 prevents program execution 'falling' into the subroutine. If this is allowed to happen, an error will occur when the RETURN statement is met as without a GOSUB call, there is no relevant line to return to! Subroutines are often placed at the end of programs and 'falling' into them must be prevented.

Consider the following program and determine what would happen on execution:

```
5 X=0
10 PRINT "SUBROUTINE CALL DEMONSTRATION":PRINT
20 GOSUB 1000:GOSUB 2000:GOSUB 3000
30 GOSUB 1000:GOSUB 2000:GOSUB 3000
List 61 40 GOSUB 1000:GOSUB 2000:GOSUB 3000
50 GOTO 9999
1000 X = X + 1:RETURN
2000 PRINT "SUBROUTINE 2000 - CALL NUMBER ";X;
RETURN
3000 FOR T = 1 TO 5000:NEXT:PRINT:RETURN
9999 END
```

THE ON GOTO & GOSUB STATEMENTS

Chapter XVII

SPAGHETTI BASIC

With the complexities of program jumps using GOTO and GOSUB mastered, the average student now wishes to show off their knowledge. They complicate programs unnecessarily with many jumps, and finish with a program that becomes difficult, it not impossible, to unravel. GOTO and GOSUB should only be used when necessary, and certainly NEVER as a 'bodge' to insert a section of code into a program at a point where you would otherwise 'run out' of line numbers.

```
100 REM BAD PROGRAMMING
110 CLS
|
|
|
150 PRINT "-----"
152 REM WHOOPS! I need to insert a 30 line section between
150 and 160
154 GOSUB 1000
160 PRINT "======"
|
|
|
999 END
1000 REM 30 line routine
|
|
|
1300 RETURN
```

All BASICs have a line renumbering facility to help in situations such as the above. In BBCBasic it takes the form

RENUMBER S,I

where S is the line number you wish your program to start at, and I is the incremental factor
i.e. RENUMBER 100,10 would renumber our program with the first line set to 100 and the rest following in gaps of 10 (110,120,130, etc).

In the example above we could first RENUMBER (say) to 100,100. This would give us 100 'spare lines' between each program line. We then have enough room to insert our new routine. We could then use RENUMBER again to tidy our line numbers, and hide our mistake!

THE ON GOTO & GOSUB STATEMENTS

Chapter XVII

List 61a - Try to follow it!!!

```
10 GOTO 100
20 TO SORT OUT
25 GOTO 80
30 OUT
40 END
50 AND THE ORDER
55 GOTO 90
60 OF SPAGHETTI BASIC
70 GOSUB 110
75 GOTO 20
80 ANY PROBLEMS
85 GOTO 50
90 THINGS ARE CARRIED
95 GOTO 30
100 THIS IS AN EXAMPLE
105 GOTO 60
110 WHICH MAKES IT VERY DIFFICULT
120 RETURN
130 IN A PROGRAM
```

D O N ' T D O I T !!

CHAPTER XVIII

LIBRARY FUNCTIONS

Library functions are functions that are pre-programmed or built in to BASIC. Most BASICs have the functions discussed in this chapter, but there may be some minor differences with different versions of BASIC. These functions are divided into two main sections, numeric functions and string functions. Numeric functions allow semi-complex mathematics operations to be done and these will be discussed first but not in great detail. Students having a need for these mathematical functions will find them relatively easy to use. We have in fact, already used two of them, INT and RND, the statements used to integer numbers and to provide random numbers. String functions allow strings to be manipulated, changed and processed and will be dealt with somewhat more extensively.

NUMERIC FUNCTIONS

These are specified by a three letter name followed by the 'argument', that is the number of expression to be processed. The more commonly available functions are:

<u>Function</u>	<u>Definition</u>	<u>Examples</u>
SQR(X)	Square root of X	SQR(900) = 3 SQR(16) = 4 SQR(12.4) = 3.52136
ABS(X)	Absolute value of X	ABS(45.6) = 45.6 If X >= 0 then ABS(X) = X If X < = the ABS(X) = -X
INT(X)	Integer that is not greater than the argument	ABS(.01) = .01 ABS(-7.2) = 7.2 INT(7.85) = 7 INT(12.99) = 12 INT(-5.6) = -6
SGN(X)	The sign of X followed by a 1 (0 for zero)	SGN(88.7) = 1 SGN(-1) = -1 SGN(0) = 0
EXP(X)	The exponential of X i.e. e ^x where e = 2.71828	EXP(3) = e ³ = 20.0855
LOG(X)	The 'natural' log of the argument. To find log to base 10 of a number, LOG(X)/LOG(10)/
SIN(X)	Trigonometric sine of X where X is in radians. A radian is 3.14159/180 degrees. To obtain SIN(X) when X is in degrees use SIN(X * .017453)	SIN(.5286) = .5
COS(X)	Trigonometric cosine of X where X is in a radians. To obtain COS(X) when X is in degrees use COS(X * .017453)	COS(1.5708) = 0
TAN(X)	Trigonometric tangent of X where X is in radians. To obtain TAN(X) when X is in	TAN(.78539) = 1

degrees use $\text{TAN}(X * 0.17453)$

$\text{ATN}(X)$ The angle in radians whose tangent is X. To obtain $\text{ATN}(X)$ in degrees multiply $\text{ATN}(X)$ by 57.29578.

LIBRARY FUNCTIONS

Chapter XVIII

STRING FUNCTIONS

In spite of their name, computers are much more than machines for performing mathematical calculations. Computers can be and are very extensively used to manipulate, compare, organise and examine non-numeric (string) information. In fact, the ability of computers to store and retrieve string information (using a Database) is one of the major uses of computers today. Word processing is another example of non-numeric operations as is Desk Top Publishing and of course, newspaper production. In earlier chapters, we have seen how names and addresses can be stored in DATA statements and arrays, searched for, sorted and printed. In this chapter we shall discuss some of the more sophisticated string manipulations that are possible by means built-in library functions. We have already seen what is possibly the simplest of string manipulations, that of CONCATENATION or the joining of strings. The following program illustrates:

```
10 CLS
20 A$ = "HOT":B$ = "DOG"           <- Assign strings to
A$ and B$
List 62 30 C$ = A$ + B$           <- Join them and
store in C$
40 PRINT A$,B$,C$                 <- Print the three
strings
50 END
```

The following display would ensue on execution:

```
-----|
|HOT      DOG      HOTDOG          | <- Strings
printed in zones                    |
|>_                                     |
```

Any sequence of characters which appears in a string is known as a SUBSTRING. For example, ABC is a substring of AYZABCDEW. If we have a name such as FRED JONES stored in variable N\$, then FRED is a substring of N\$. The four following functions permit a wide variety of string manipulations.

<u>STRING FUNCTION</u>	<u>DESCRIPTION</u>
LEN(Z\$)	returns the number of characters in string Z\$.
LEFT\$(Z\$,N) of the	returns the substring in Z\$ that consists leftmost N characters.
RIGHT\$(Z\$,N) of the	returns the substring in Z\$ that consists rightmost N characters.

MID\$(Z\$,P,N) returns the substring in Z\$ that consists of N characters beginning at character number P.

Examples: 1. Suppose X\$ = "AFGHANISTAN"

LEFT\$(X\$,3) = AFG RIGHT\$(X\$,4) = STAN MID\$(X\$,4,3) = HAN

2. Suppose N\$ = "JAMES"

LEFT\$(N\$,5) = JAMES RIGHT\$(N\$,5) = JAMES MID\$(N\$,1,5) = JAMES
JAMES <- NOTE these

3. Suppose W\$ = "AFTERNOON"

LEFT\$(W\$,5) = AFTER RIGHT\$(W\$,4) = NOON MID\$(W\$,7,2) = OO

LIBRARY FUNCTIONS

Chapter XVIII

This program illustrates how these statements work:

```

    90 CLS
    100 A$ = "ABCDEFGHI"           <- Assign string A$
    110 L = LEN(A$)                <- Store length of the string L
    120 PRINT "STRING: ";A$        <- Print the string
    130 PRINT "LENGTH: ";L        <- Print the length
List 63 140 L$ = LEFT$(A$,5)      <- Put leftmost 5
characters in L$
    150 PRINT "LEFT PART: "L$     <- Print message and
L$
    160 R$ = RIGHT$(A$,3)         <- Put rightmost 3
characters in R$
    170 PRINT "RIGHT PART: ";R$   <- Print message and R$
    180 M$ = MID$(A$,4,3)         <- Put 3 characters
starting at
    190 PRINT "MID PART: ";M$     character number 4
in M$. Line 190
    999 END                       prints it.
```

Execution would provide the following display:

```

|STRING:  ABCDEFGHI           | <- Original string
printed
|LENGTH:  9                   | <- The number of
characters
|LEFT PART:  ABCDE           | <- Leftmost 5
characters
|RIGHT PART:  GHI            | <- Rightmost 3
characters
|MID PART:  DEF              | <- 3 characters
starting at
|>_                           | character number
4.
```

Another example using LEFT\$ only:

```

    5 CLS
    10 INPUT "FIRST NAME: ";F$    <- Get first name
from operator
    20 INPUT "LAST NAME: ";L$    <- Get last name from
operator
List 64 30 I$ = LEFT$(F$,1)      <- Find first name
initial
    40 PRINT I$;" ". ;L$        <- Print initial,
stop, last name
```

If, in answer to the question at line 10, the name JAMES was input and in answer to the question at line 20, SMITH was input, the program would output J. SMITH

Consider this program which demonstrates a text editing facility:

```

410 CLS
420 INPUT "TEXT: ";T$:T1 = LEN(T$):PRINT
430 INPUT "REPLACE: ";R$:R1 = LEN(R$):PRINT
450 FOR P = 1 TO T1 - R1 + 1
460 IF R$ = MID$(T$,P,R1) THEN 500
470 NEXT P
480 PRINT "NOT FOUND!":PRINT:GOTO 430
List 65 500 INPUT "REPLACE WITH: ";W$:W1 = LEN(W$):PRINT
520 T$ = LEFT$(T$,P-1) + W$ + RIGHT$(T$,T1-P+1-R1):T1
= LEN(T$)
560 PRINT "CORRECT TEXT: ";T$
570 INPUT "ANT MORE CORRECTIONS (Y/N)";Q$
580 IF Q$ = "Y" THEN 430
590 IF Q$ = "N" THEN 700
600 GOTO 570
700 END

```

The variables in List 65 are used as follows:

- T\$ - Original text (and later the corrected text)
- T1 - Length of original text (and later the corrected text)
- R\$ - The substring to be replaced
- R1 - The length of the substring to be replaced
- W\$ - The replacement string
- W1 - The length of the replacement string

LIBRARY FUNCTIONS

Chapter XVIII

The easiest way to understand this program is to assume a text entry and the trace the program. Let us assume therefore that the original text entry in response to line 420 was 'THIS IS THH PROGRAM'. We shall want to correct the misspelling 'THH' to 'THE'.

T\$ = THIS IS THH PROGRAM

T1 = 19

R\$ = THH

R1 = 3

W\$ = THE

W1 = 3

P LIMIT = 17

P WHEN FOUND = 9

LINE 420 - The string T\$ is assigned and its length (T1) evaluated.

LINE 430 - The substring to be replaced (THH) is entered (R\$) and its length evaluated.

LINE 450 - The limit of the FOR-NEXT loop is now known (T1-R1+1), 17.

LINE 460 - 1ST TIME THRU' - MID\$(T\$,P,R1) = 'THI' - No match so NEXT P

ditto 2ND TIME THRU' - ditto = 'HIS' -

ditto 3RD TIME THRU' - ditto = 'IS' -

ditto 4TH TIME THRU' - ditto = 'S I' -

ditto 5TH TIME THRU' - ditto = ' IS ' -

ditto 6TH TIME THRU' - ditto = 'IS ' -

ditto 7TH TIME THRU' - ditto = 'S T' -

ditto 8TH TIME THRU' - ditto = ' TH' -

ditto 9TH TIME THRU' - ditto = 'THH' - 'THH'

MATCHED!

Now execution transfers to line 500

LINE 500 - We type in the replacement substring, 'THE' and this is

assigned to W\$. Its length (W1) is evaluated.

LINE 520 - The new version of T\$ is now reconstructed:

LEFT\$(T\$,P-1) = 'THIS IS '

W\$ = 'THE'

RIGHT\$(T\$,T1-P+1-R1) = ' PROGRAM'

LINE 560 - The new T\$, 'THIS IS THE PROGRAM' is printed.

Further corrections may be made if required. Note that if the substring to be replaced, (entered at line 430) is not found during the search within the FOR-NEXT loop, program execution

BCP Page

will drop through to line 480 where the relevant message is displayed.

The most important part of this program is the SEARCH ROUTINE provided by lines 450 to 470. This is called an 'INSTRING' search, it checks for the existence of a substring within the main string. Many BASICS offer a statement which provides this search and in BBCBasic for example, this statement is INSTR. Instring searches permit items to be found by a 'PARTIAL MATCH', this meaning that it is not necessary to enter the whole of any item to be found. For example, if you were searching through a DATA list for an item such as 'WELLING' and you used the search string 'WELL', all entries for WELLING would be found. SO WOULD ANY ENTRIES LINE 'WELLINGTON', STOCKWELL' and any other strings containing the substring 'WELL'!

LIBRARY FUNCTIONS

Chapter XVIII

Even bearing this small problem in mind, the search with partial match is very useful and is generally found as a standard option on many professional database systems. It avoids the problems of the exact match search where, say, FRED_ (_ represents a space) has been typed and therefore an exact match with FRED would not occur. The instring search and a search string of FRE would find entries for FRED. Below is an alternative view of the comparisons made during the execution of List 65, assuming the entries were made as above:

```
      THIS IS THH PROGRAM
      ^^^
PASS 1      THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 2  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 3      THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 4  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 5  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 6  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 7  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 8  THH                      NO MATCH
      THIS IS THH PROGRAM
      ^^^
PASS 9      THH                      MATCHED!  Jump out of loop to
line 500
```

The general form of the INSTRING SEARCH is as follows and this should work in all BASICs. X\$ is the string to be searched and Y\$ is the substring to be found.

```
1000 FOR I = 1 TO LEN(X$) - LEN(Y$) + 1
1010 IF Y$ = MID$(X$,I,LEN(Y$)) THEN .....*
1020 NEXT I
```

* might go to a PRINT subroutine to display the found item.

This routine would permit a list of DATA items to be searched for a partial match:

```
800 INPUT "SEARCH FOR: ";Y$
900 FOR A=1 TO number of DATA items
```

```
910 READ X$
1000 FOR F=1 TO LEN(X$)-LEN(Y$)+1
1010 IF Y$=MID$(X$,F,LEN(Y$)) THEN drop to match found
routine
1020 NEXT F
1030 NEXT A
```

LIBRARY FUNCTIONS

Chapter XVIII

Using the BBCBasic INSTR statement, the general form for an Instring search would be as follows:

```
10 INPUT "LONG STRING ";A$
20 INPUT "SUBSTRING ";B$
30 A=INSTR(A$,B$)
40 IF A=0 THEN PRINT "NO MATCH":END
50 PRINT "MATCH FOUND AT POSITION ";A
60 END
```

If the match is found, A will have the value of the position in the long string at which the substring was found. If not match is found, A has the value 0.

THE VAL & STR\$ FUNCTIONS

These functions convert numeric variables into string variables (STR\$) and string variables into numeric variables (VAL).

Consider the following program:

```
10 Q$ = "1234":P$ = "4321" <- Assign strings
20 R$ = Q$ + P$:PRINT R$          <- 'Add' strings and
print (12344321)
List 66 30 Q = VAL(Q$):P = VAL(P$) <- 'Value' strings
40 R = Q + P:PRINT R             <- Add values and print
(5555)
50 END
```

The VAL statements on line 30 store the numeric values of Q\$ in Q, and P\$ in P. If you try to VAL a string such as "FRED" the result will probably be zero!

Now consider this program:

```
10 M=1111:N=2222                <- Assign values
20 R = M + N:PRINT R            <- Add values and print
(3333)
List 67 30 M$ = STR$(M):N$ = STR$(N) <- Convert to strings
40 R$ = M$ + N$:PRINT R$       <- 'Add' strings and
print
50 END                          (11112222)
```

These two statements were originally used in early versions of BASIC to 'compress' data before saving it in an external datafile on disk or tape. The reason for this is that numeric data requires more storage space than string data and such compression saves space. The data would then have to be VAL'd after being read back from the datafile before any numeric operations could be done on it. Many versions of BASIC have special statements which convert data to the required format

and some software today 'compresses' numeric information during program operation thus saving computer memory space, but the much larger memories currently available make this less necessary.

Finally, do not confuse STR\$ with the similar sounding STRING\$

```
110 A$=STRING$(3,"-***-")
120 PRINT A$
```

When executed prints:

```
_***_***_***_
```