CENTURY
COMMUNICATIONS

THE MICRO ENQUIRER

COMMODORE 64

Christopher Bidmead and Benjamin Woolley

COMPUTER Answers

The Micro Enquirer is based on updated and rewritten material which originally appeared in *Computer Answers* — a monthly micro magazine with a reputation for lucid articles on all areas of computing.

Managing Editor
Benjamin Woolley, first editor of *Computer Answers*, is a highly experienced computer journalist. He has been involved in home computing since the launch of the ZX 80, which began it all, and has covered the subsequent development of this fast moving field.

Editor
Christopher Bidmead is a technical writer contributing articles to a variety of computer publications, including *Computer Answers* and *Practical Computing*. He is also a freelance television scriptwriter and was a script editor for BBC TV's *Dr Who* when he first became interested in computers and used one to help with his work.

Technical Editor
Dr Peter Turcan has a PhD in computer science from the University of Reading and is author of the *Scrabble* software package for the Sinclair Spectrum and Apple II computers. He also writes on computing and was technical editor and is now editor of *Computer Answers*.

MICRO ENQUIRER: COMMODORE 64

Due to an error at the printers, a few program listings in this book are rather faint. Here are the programs which are difficult to read:

Page 45
Fig. 1. 'Ellipse' effect:

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,3 : NEXT
100 SIZE = 100 : RATIO = 0.5
110 FOR ANGLE=0 TO 2*π STEP π/60
120 X = 160 + SIZE * SIN(ANGLE)
130 Y = 100 + SIZE * RATIO * COS (ANGLE)
140 GOSUB10000
150 NEXT
9000 GETA$:IFA$=""THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2↑(7-(XAND7))
10020 RETURN
```

Page 45
Fig. 2. Spiral:

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,3 : NEXT
100 FOR ANGLE=0 TO 30  STEP 0.1
110 X = 160+3 * ANGLE * SIN (ANGLE)
120 Y = 100+ 2 * ANGLE * COS (ANGLE)
130 GOSUB10000
140 NEXT
9000 GETA$:IFA$=""THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2↑(7-(XAND7))
10020 RETURN
```

THE MICRO ENQUIRER

THE MICRO ENQUIRER

COMMODORE 64

Christopher Bidmead and
Benjamin Woolley

Guild Publishing
London

The Micro Enquirer is based on updated and rewritten material which originally appeared in *Computer Answers* – a monthly micro magazine with a reputation for lucid articles on all areas of computing.

Managing Editor
Benjamin Woolley, first editor of *Computer Answers*, is a highly experienced computer journalist. He has been involved in home computing since the launch of the ZX80, which began it all, and has covered the subsequent developments of this fast moving field.

Editor
Christopher Bidmead is a technical writer contributing articles to a variety of computer publications, including *Computer Answers* and *Practical Computing*. He is also a freelance television scriptwriter and was a script editor for BBC TV's *Dr Who* when he first became interested in computers and used one to help with his work.

Technical Editor
Dr Peter Turcan has a PhD in computer science from the University of Reading and is an author of the *Scrabble* software package for the Sinclair Spectrum and Apple II computers. He also writes on computing and is now editor of *Computer Answers*.

CONTRIBUTORS TO THE MICRO ENQUIRER

Steven Linderman
Robin Webster
Alan Thomas
Gordon Stevenson
Sheridan Rosser
Bob Robinson
Chris Preston
Robert Palmer
Simon Orebi Gann
Fiona Newman
Peter Van Der Linden
Kathy Lang
Alistair Kelman
Ray Hammond
Charles Christian
Jonathan Burchell
Peter Brameld
Phil Manchester
Graham Bland
Eric Bagshaw
Norman Wilson
Max Phillip
Peter Jackson
Shirley Fawcett
Chris Cunningham
Geoff Conrad
Janet Rothwell
Waldo Watkins
Phil Garratt
Steve Adams
Chris Preston
Ron Stewart
Margaret Prince
Ros Earlle
Jenny Lodder
Val Hudson
Jackie Searle
Tony Dennis
Stephen Applebaum
Kathryn Custance

Designed and produced by The D & A Partnership, Dorking, Surrey RH4 1PS

Based on material first published in *Computer Answers* magazine.

This edition published 1984 by
Book Club Associates
By arrangement with Century Communications Ltd.

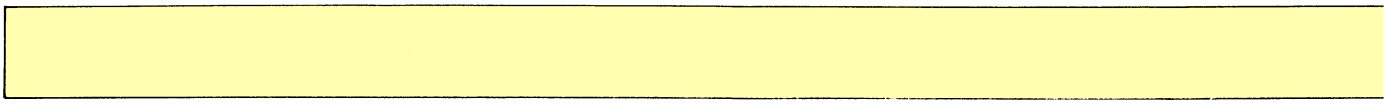
Flair TV graphics on page 33
reproduced by permission of Logica UK Ltd.

Illustrations by David Parr, Steve Cross and John Parr.

Printed in Great Britain by Butler & Tanner Ltd.

This book is dedicated
to the memory of
Waldo Watkins

Contents



Introduction: The Commodore 64

Elbowed on by heavy marketing, the popularity of the Commodore 64 has had a snowball effect, making it, in the UK at least, the best selling micro in its price range. But it isn't just big sales that has drawn the attention of so many of the world's best commercial software writers to the machine. Hardware features abound, exemplified by the VIC-II graphics chip and the SID four channel sound chip that can be driven as a very fully-featured synthesiser. You'll find a discussion about sound and about synthesisers in general under the entry on *Music* in this book.

Its extensive hardware features make the Commodore 64 ideal for games, and for the smaller business user the machine's popularity has produced a body of 'serious' software. But, it has to be said, the Commodore 64 can be a discouraging machine for people who are hoping to learn something about the fascinating subject of computing.

Firstly, if your experiments aren't just to evaporate when you turn off the machine, you will need some way of storing your programs. This is true of all micros, of course, but many of them allow you to use an ordinary domestic cassette recorder (see entry on *Cassettes*). Commodore 64 owners, unfortunately, will need a purpose-built cassette mechanism. Commodore aim to supply you with this, but you may think it worthwhile to look around for special recorders sold by other manufacturers. You'll find there are some that fit the Commodore but can also be used for other recording.

Another shortcoming of the Commodore from the (would-be) programmer's point of view is the absence of any instructions in the built-in Basic to drive the graphics and sound facilities. It's rather like supplying a sports car with overdrive in the engine but with no controls to switch it on from the driver's seat. In the entry on *Graphics* we show you some ways round this problem, and give you a broader picture of methods used by other machines to cope with sound and graphics, now and in the future.

Commodore 64 owners do have the big advantage of belonging to what manufacturers like to call a large 'user base'. The bigger the user base the more software is available, and the Commodore 64 gives you an enormous choice. There are already enough games, for example, for you to spend the rest of your life battling with aliens or questing for gold in dragon infested mountains (for more about what is available see the entry on *Games*). But while this sort of software is terrific for bringing your computer to life, and is a splendid way to start your relationship with the new technology, it is only the beginning of what you and your Commodore can do together. This book is for computer owners who want to go further than simply learning to press buttons and get rewards.

Understanding a computer and getting it to do what you want it to, rather than just running a few pre-packaged games, is a fascinating adventure. This book will help you on your travels into the interior of your machine, but its main purpose is rather more than that. Computers and computing are transforming the way we live, rather as the advent of trains transformed the landscape and lives of those who lived during the first half of the 19th century. It is no insult to your computer to suggest that it represents only the beginning of a long and rapidly developing relationship you are destined to have with the new technology, as we move towards the end of this astonishing 20th century.

'Computer literacy' is really quite the opposite of what people learn when they immerse themselves in the idiosyncrasies of a particular micro. The entries on the *Operating System* and *Memory Map* will help lay bare many of the intimate and mysterious PEEKs and POKEs that are the short cuts to your particular hardware features. But this is computing with a very small 'c'; the PEEKer and POKEr who has, for example, missed the point of the general principles of software design (see *Programming*) will, when the next generation of hardware arrives, be left with a useless handful of dry facts, rather than a growing body of understanding.

Like the best software, the literate computer user is machine-independent. Part of the object of this book is to help you understand the strengths and limitations of your own machine in the much broader context of computing as a whole. Our scope ranges across all the machines and techniques available today, reaches back into the history of computing on mainframes (and beyond – see *Numbers*), as well as forward into the future.

Your home computer is an exciting investment, a door onto a limitless future. There are many computer books that show you how to paint the door, oil the hinges and polish the handle. The object of this particular book is to help you open that door and step through to the world beyond.

Analogue: Squaring The Circle

To us everything is curved, chipped or in some way irregular. We think and live in an analogue world, the features of which tend to blend in uncountable transitions. Digital computers, on the other hand, work in the stark black and white underworld of binary numbers. To them, everything is either one or zero – there's no in between. So to interact with the world as we see it, the digital computer has to be given a translating device.

There are many ways of converting the analogue data of the real world into a digital form suitable for, say, a graphic display. Systems readily available to the micro user take two basic forms:

Analogue-to-digital conversion using paddles, *some* joysticks and mice.
Direct digital input using light pens, digitizing pads and video digitizers.

The mouse takes control

The mouse has had a lot of publicity of late, lead by its appearance on machines such as the Lisa and the Macintosh. This peripheral screen driving device serves as a kind of overdrive to the normal keyboard cursor controls. It's known as a mouse because it fits in the cupped hand and has a long tail curling away to the back of the micro, this connection lead plugs into an RS-232 connector, a parallel port or, in one ingenious version, into the same plug as the keyboard. The name 'mouse' has an added ring of truth because of what the device does to the cursor, sending it scurrying across the screen at a rate unapproachable with ordinary cursor keys.

Its advantages in interactive graphics are self-evident. With the proper software supporting the codes it sends, you can use the mouse like a pencil to draw lines of all shapes and sizes, building up pictures on the screen. Its role in text handling and spreadsheet packages is less obvious, particularly to keyboard users who have become adept at finding their way around the historically quirky world of QWERTY. But the advantages are real nevertheless.

Word processing software often makes far too little provision for easy cursor control when marking or moving blocks of text, and in this respect the chief offender is WORDSTAR, still the de facto standard for micro word processing. Rodent aid is able to give a much needed boost to these underdeveloped features.

Spreadsheets usually suffer from similar shortcomings, particularly noticeable in some of the new 16-bit versions which encourage the creation of very large layouts. It takes appreciable time to scroll around a sheet of four million cells when the only tools at your disposal are cursor keys. Jumping is allowed, of course, but this tends to fragment the user's image of the spreadsheet.

Manipulation of spreadsheets often involves inserting data into rows at the top and chasing down to the 'bottom line' for the result. This is where the mouse scores, allowing you to zip effortlessly between the two parts of the accounts. Even where this facility is offered by the spreadsheet software, it usually requires setting up by, for example, giving names to cell ranges. It's nice to be able to cut through this formality and roam through the spreadsheet as the fancy takes you.

Because the thumb and little finger grip the mouse at the sides, leaving the normally endowed earthling with three fingers surplus to requirements, mice typically offer one or more control buttons which can be used to send additional signals to the computer. Three buttons are the most that designers judge manageable without subverting the essential simplicity of the system, but there are ways of multiplexing control signals onto the buttons to increase the range of commands.

On a mouse with only three buttons it is possible to implement many more than three commands by activating the buttons in combination, or by using software which counts the number of clicks that you send. Usually the mouse control buttons put out purely arbitrary codes which are translated by driver software augmenting the normal operating system.

Unfortunately for most mice, a package like WORDSTAR uses many more control combinations than the buttons can handle, and the user has to revert to the keyboard. This feels unnatural and awkward: intuitively you want the whole range of WORDSTAR commands at your finger-tips – literally – without having to take them off the mouse. This uncomfortable situation doesn't arise where the mouse hardware and the application package have been designed together, as exemplified by Microsoft's WORD.

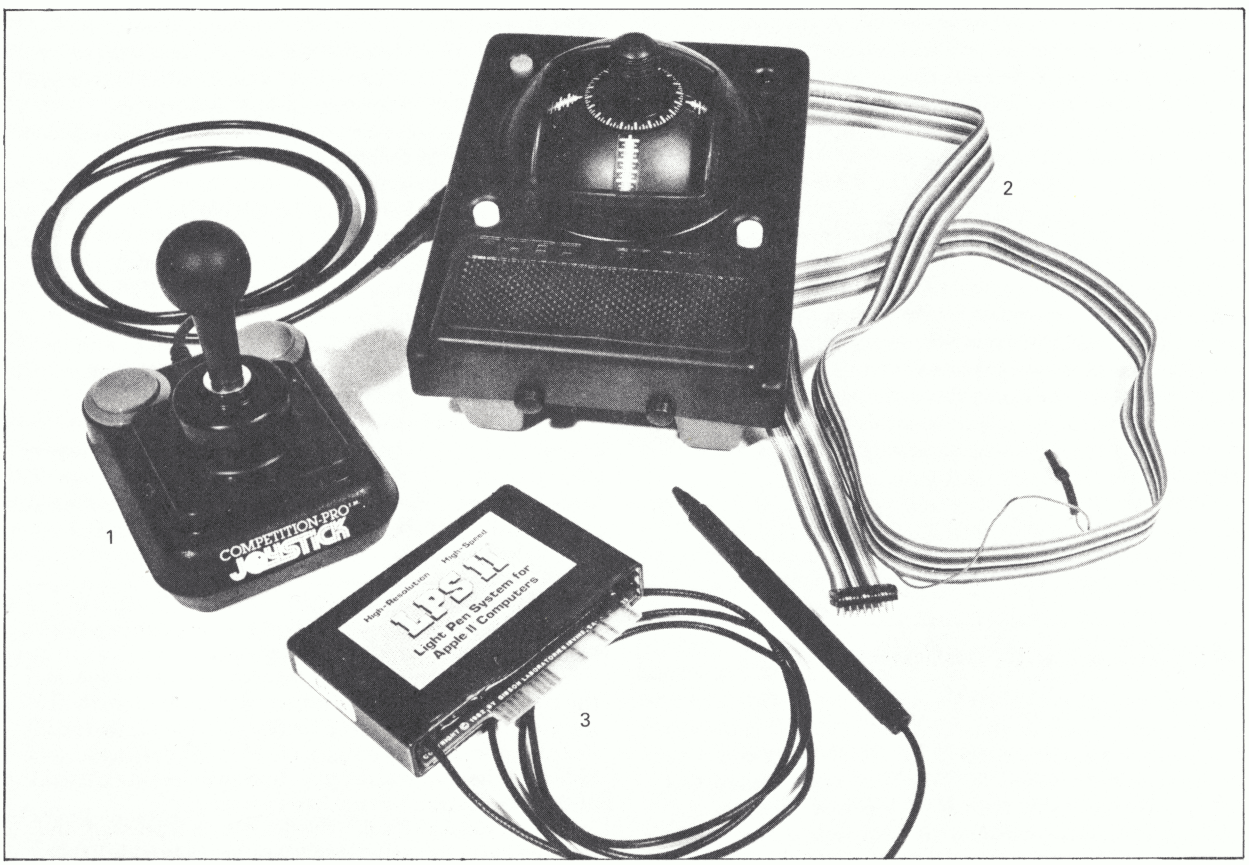
Other analogue devices

It remains to be seen whether the mouse craze proves to be like the bicycle, a long-lived boon and blessing to mankind, or like the hula-hoop, a fad that graunched a thousand hips before passing away without a trace. In the meantime, don't overlook other non-keyboard methods of speeding the cursor. **Touch-sensitive screens** and **Tracker balls** have been given less publicity, but have their own advantages over the mouse. The tracker ball – a sort of inverted mechanical mouse with an over-sized ball moved or spun with the finger tips – takes up hardly any extra desk space. None at all is required by touch-sensitivity, the talent by which an ordinary-looking monitor reads the precise position of the operator's finger when he or she touches the screen.

Digitizers

The digitizer is literally a device which collects analogue data and turns it into digits, and in this sense there is little to distinguish it from an analogue-to-digital converter. But in the parlance of the computing fraternity the word has taken on two distinct meanings:

The video digitizer accepts input from a video camera or video tape, and converts it to machine-readable form, often to appear subsequently on a high-resolution screen image.
Digitizing pads, or graphics tablet digitizers, accept input from a special pen or hand-held cursor which is moved over a grid, and send position coordinates back to the computer.



1. The Competition-Pro joystick works with the Spectrum and is imported from the USA by Kempton Micro Electronics.

2. The Bit Stik designed and produced by Robocom comes complete with systems software for the BBC, but requires the second processor too.

3. The LPS II light pen by Pete and Pam Computers

Most graphics tablet digitizers operate on similar principles to the ferrite bead core memory used in the first computers – a closely-packed mesh of very fine wires with a ferrite ring at every junction of two wires. Passing a small current down the wires in a certain direction alters the magnetic field in the ferrite ring. The magnetic change represents one bit of information, which could be read back later by passing the current in a different direction.

The graphics tablet is similar to a drawing board, with an 'active area' on it. This area is typically about 30cm (12") square, and is the portion that contains the ferrite bead matrix. When the ferrite nib of the pen attached to the board is placed on the active area, the tablet can read and convert the position into a location on the grid.

Obviously, with the tablet using a grid, the position of the pen is already digitized, and it needs only a little further encoding to output in computer readable form. This output might be a direct interface with your computer, or one of the standard protocols such as IEEE-488 or RS-232.

There is usually a small, very low-resolution area somewhere around the main active area of the tablet which allows menus to be set up easily. When the pen goes inside a menu option's area, that option is selected. Often the menus are drawn up on a Mylar sheet overlaying the pad so that the options are represented with informative colour coding.

To convert a picture to digital form for use in computer graphics, the picture is placed on the active area and the `Draw` option is selected from the menu. Tracing the picture with the pen will usually cause it to appear on the display from where it can normally be saved to disk for future use.

As well as being able to draw pictures on the screen, the digitizer's ability to convert shapes into numbers enables the software to compute areas and the lengths of irregular objects when they are placed on the pad and outlined with the stylus.

This kind of tablet can be dangerous to use around disks, since the magnetism inside the tablet will very effectively erase the contents of any disk placed on it. *Beware!*

The light pen

A light pen is an ingenious way of interacting with the picture on your monitor screen. A monitor display projects an electron beam across the face of a cathode ray where

they strike a very small phosphor dot on the screen, which glows as a result. To provide a readable display the electron beam traverses the screen at a fast and very accurately pre-determined speed, in a pre-determined sequence. Once the full screen has been scanned, the beam then starts at line one again.

The 'home position' is marked by a synchronisation signal (the 'sync pulse'), from which all other timing is taken. Because the beam sweeps the screen at a regular rate, its position on the screen at any given moment can be expressed as an elapsed time since the last sync pulse – it is from this timing that light pen gets their bearings. At each 'home position' it reads the sync pulse and starts timing. If the pen is on the screen when the dots under the pen light up, the pen's program stops timing. The exact location on the screen can then be found, either by calculation, or by a look-up table which correlates the elapsed time and screen position.

Light pen resolutions vary from an area of about 13mm square down to an individual dot on the screen. If you want to recognise dot points, it's no use buying a light pen that can only see a block the size of a character.

Joysticks and paddles

Many personal computers accept a joysick or games paddle input. There are basically two designs. In the first, four switches are mounted North, South, East and West of the stick itself. When the stick is moved in one of the four primary compass directions, just one switch is activated. But if for instance the stick is pushed to the North-west direction, both the North and West switches are closed. So the stick reacts to eight different directions altogether.

A number of commonly available joysticks use this strategy, and also use the same pin connections to a nine-pin miniature D connector. You can easily tell joysticks of this type by the way the microswitches click as the stick is moved.

The alternative design uses variable resistors instead of switches to achieve a more accurate and continuous response to the movement of the stick. Between the joystick and the computer is an analogue-to-digital converter which measures the voltage coming from the interface for a given period, calculates its size relative to the known maximum and minimum, and places a number related to the current position on its output port.

The drawback in the use of joysticks is the limited range of movement available and the accuracy of the interface. The movement range can make the cursor jump about all over the screen, sending it everywhere but where you want it.

Software

The graphics software used in conjunction with these analogue-to-digital add-ons ranges from children's games, to the very best in vector display techniques. The vast majority of the software is of the bit-mapped type, where each point of the picture is represented as part of a byte in computer memory. It is a very easy technique to program and generally has adequate display characteristics.

The catch is that if you want to expand the image size much above standard, the display takes on the appearance of a disordered array of bricks. If you reduce the size, the detail is liable to vanish, as the image is packed into fewer bytes.

A few of the newer, more expensive, software packages store the image as a list of vectors, rather than a bit image. These vectors tell the program the distances and angles to draw a line relative to the rest. Using this technique, when you expand or contract the image, the line lengths are changed to suit the size set. Everything else is kept in proportion since the vector data is constant, regardless of the dimensions.

Despite these many fashionable input devices, keyboard entry will continue as a fast and reliable way of talking to your computer for a long time to come. For serious computing, the mouse nibbling at your coffee cup or the joystick by your elbow doesn't buy you out of the need to learn to type.

The 64 handles its analogue I/O by way of the pair of nine-pin C-connector games ports. These can be used for joysticks, games paddles, or even a light pen.

The digital joysticks are of the simple switch-type, and the state of the switches is reflected in the bottom five bits (bits 0 to 4) of the byte at 56320 (for joysticks 1) and 56321 (for joysticks 2). A bit pattern of 11111 (31 decimal) indicates that none of the direction switches are closed, and the fire button has not been pressed.

Bit four turns to 0 when the fire button is pressed, and this can easily be tested by ANDing the byte with 10000 (decimal 16). If the answer is 0 the button has been fired. The lower four bits each represent the directions NSWE in that order, and again a zero in the bit pattern indicates that the corresponding switch has been closed. So a combination of switches 1 and 3, closed when the joystick is moved South-East, would produce the pattern 0101 (decimal 5).

Analogue joystick and paddles affect the values at locations 54297 and 54298, but require machine code routines to read them because they change too quickly for BASIC. But handling the light pen input is relatively simple: the x and y coordinates can be read with the formula:

```
100 X = 2*PEEK(53267): Y = PEEK(53268)
```

Arrays: Exploring The Dimensions

DIM, short for dimension, is an instruction that has always been necessary to BASIC, although its exact meaning varies between dialects. Its job is to warn the computer how much data you are going to use so that it can set aside an appropriate space in its memory.

Think of it in terms of a trip to the theatre. If a theatre box-office has just sold seat number 10 and a party of 20 makes a booking, the box office would reserve a block of 20 seats and continue selling with the next free seat, number 30. Similarly, although BASIC is quite happy for you to create new single variables when you need them simply by writing LET X=4 or LET A=2100 and so on, it will have problems if a party of 20 related numbers turns up unannounced on its doorstep.

So DIM is used to create **arrays**, – block bookings, if you like, of either numbers or words. DIM A(10) would create a list of 10 numbers called A(1), A(2), A(3) and so on up to A(10) – at least in most BASICs.

Similarly for strings, 200 names could be stored in an array of strings, once the statement DIM N$(200) has been executed.

Arrays do more than just create masses of variables. Any particular value in the array (an **element**) can be referenced by any numerical expression in the brackets after the array name (the **subscript**). So to print out the 10 numbers in array A we could write:

```
50   FOR P = 1 TO 10
60   PRINT A(P)
70   NEXT P
```

or to print out the 200 names in two columns:

```
50   FOR P = 1 TO 200 STEP 2
60   PRINT  N$(P), N$(P+1)
70   NEXT P
```

Anywhere a group of related items has to be processed, for example sorting or searching, arrays will be used.

Multiple dimensions

So far the examples have referred only to lists – arrays having one subscript. But arrays with more than this one dimension are frequently used, both in computing and in the real world. For instance, a railway timetable is a two-dimensional array, often called a table or matrix. The two dimensions might be station names (rows) and departure times (columns). If we wanted to create a table of names and sexes on the computer, we could use DIM T$(10,2). The table could be visualised like this:

```
T$(R,C)    name          sex
============================
entry 1    Mark          boy
entry 2    Susan         girl
entry 3    John          boy
entry 4    Mary          girl
etc.
```

Each row is the entry for a different person and the two columns are for name and sex. To refer to a particular element of the array, we need only write T$(R,C), where R is the row and C is the column desired. It doesn't matter whether the column or row comes first as long as you stick to the same order throughout a program.

One example of what can be done with T$ is to print out the names of all the girls. We need to look down column 2 and whenever we find girl, we should print the corresponding name, found in column 1 of that row:

```
100   PRINT "Here are the girls..."
110   FOR R = 1 TO 10 120 : IF T$(R,2) = "girl"
      THEN PRINT T$(R,1)
130   NEXT R
```

In many BASICs it is possible to use more than two dimensions. For example, DIM A(4,4,4,4) creates a four-dimensional array. The computer finds these multi-dimensional arrays completely reasonable, although they are extremely difficult for us to visualise.

Traps to avoid

Using arrays requires care and planning. You must ensure that subscripts are never negative or greater than the value given by DIM. Many BASICs allow you to use a subscript of zero. Thus you might get one more element than you asked for: DIM A(10) in some dialects creates an 11-element array from A(0) to A(10). In variants of Microsoft BASIC the command OPTION BASE lets you set the minimum subscript to either 0 or 1.

One hint is not to use numbers literally in controlling arrays. Instead of saying DIM A(20) use LET N=20, followed by DIM (N). All your loops and tests should look at the value of N and not at 20. This means that if you run out of memory or decide to alter the size of the array, you need only change the one LET statement rather than a lot of instructions. Be aware though that if you ever use a compiler you may find this 'dynamic' dimensioning rejected and have to rewrite the DIM statements.

Once you've dimensioned an array, you can't normally change its size. Imagine the problems if the party of 20 theatre-goers turned out to have 30 members. Some BASICs have an ERASE instruction which allows the computer to forget an existing array and reset its dimensions. If yours hasn't, the best you can do is dimension arrays to the largest size needed and if necessary only use part of the array for some of the time.

Odd-shaped arrays

Arrays and DIM instructions are not especially difficult to use. With some experimenting it should become obvious why they are such a fundamental part of computing. You'll find you will be able to improve your programming by using the idea to cope with awkwardly structured data, but as you become more experienced you'll run into problems that require storage that is not easily accommodated into a simple array. What if you want a triangular array or one shaped like a pyramid or a ball or a cylinder?

The answer is to make use of the concept of a **mapping function** to persuade a simple array that it's

something more exotic than a simple rectilinear shape. To show how mapping functions work, let's take the very simple case of creating a two-dimensional array in a dialect of BASIC that only allows arrays to have one dimension.

Say, for example, that the logic of your program implies a two-dimensional array of 10 x 8 elements. There are at least two ways of solving the problem posed by our supposed BASIC limitation. Obviously you could declare eight separate arrays, each of length 10. But you could also write a mapping function to convert between the two-dimensional array that is logically required, and the single-dimension array that is actually possible.

The advantage of this method is that after writing the map the rest of the program can refer to the two-dimensional structure as if it really existed, enabling you to simplify the logic and improve the flow of your program. It is often said that to achieve a structured modular program, the data structures should match the nature of the problem. This is the real bonus of using a mapping function.

```
100 WRITE = 1 : RD=0
110 DIM ARRAY(80)
1000 REM MAPPING FUNCTION
1010 PSN = (ROW-1)*8 + (COL-1)
1020 IF OP=RD THEN V=ARRAY(PSN
)
1030 IF OP=WRITE THEN ARRAY(PS
N)=V
1040 RETURN
2000 REM SET ARRAY(4,6) TO 127
2010 V=127
2020 OP=WRITE : ROW=4 : COL=6
: GOSUB 1000
3000 REM SET VARIABLE X TO ARR
AY(9,3)
3010 OP=RD : ROW=9 : COL=3 : G
OSUB 1000
3020 X=V
```

Fig. 1. Mapping function for a 2D array

Figure 1 shows a skeleton section of code that creates and uses the map for our two-dimensional array. The mapping function requires four variables:

- `OP` telling it whether to read or write.
- `ROW` and `COL` giving it the logical position in the two-dimensional array.
- `VALUE`, which is used to hold the value of the element.

The function takes the value in `ROW`, subtracts one and multiplies by the length of the rows, then adds the column position and subtracts one. What it is in fact doing is adding up the number of complete rows before the row that the element is contained in, then adding the column number to find the exact location. Obviously a one-dimensional array of sufficient length must be declared at the start of the program.

```
100 WRITE = 1 : RD=0
110 DIM ARRAY(5^8)
1000 REM MAPPING FUNCTION
1010 PSN=(D1-1)*(5^7)+(D2-1)*(
5^6)+(D3-1)*(5^5)+(D4-1)*(5^4)
1020 PSN=PSN+(D5-1)*(5^3)+(D6-
1)*(5*5)+(D7-1)*5+(D8-1)
1030 IF OP=RD THEN V=ARRAY(PSN
)
1040 IF OP=WRITE THEN ARRAY(PS
N)=V
1050 RETURN
2000 REM SET ARRAY(1,2,3,4,5,4
,3,2) TO 127
2010 V=127
2020 D1=1:D2=2:D3=3:D4=4:D5=5:
D6=4:D7=3:D8=2
2030 OP=WRITE : GOSUB 1000
3000 REM SET VARIABLE X TO ARR
AY(5,4,3,2,1,2,3,4)
3010 OP=RD : LEVEL=2 : ROW=8 :
 COL=7 : GOSUB 1000
3020 D1=5:D2=4:D3=3:D4=2:D5=1:
D6=2:D7=3:D8=4
3030 OP=RD : GOSUB 1000
3040 X=V
```

Fig. 2. Mapping function for an 8D array

Most BASICs these days will of course cope with two-dimensional arrays, but the example above is a useful demonstration of everything you need for setting up mapping functions. The principle can be extended very simply: figure 2 shows how to set up an eight-dimensional array, all 'sides' of length five. The only problem here is the amount of memory required for an array of this size – 5 to the power of 8 is about 400,000, which is more memory addresses (let alone memory

chips!) than most micros can cope with. But there's no harm in looking to the future!

More interesting than massive array structures are ones with irregular shapes. Figure 3 gives a diagrammatic representation of a triangular array. One way of creating this would be to declare a two-dimensional array and

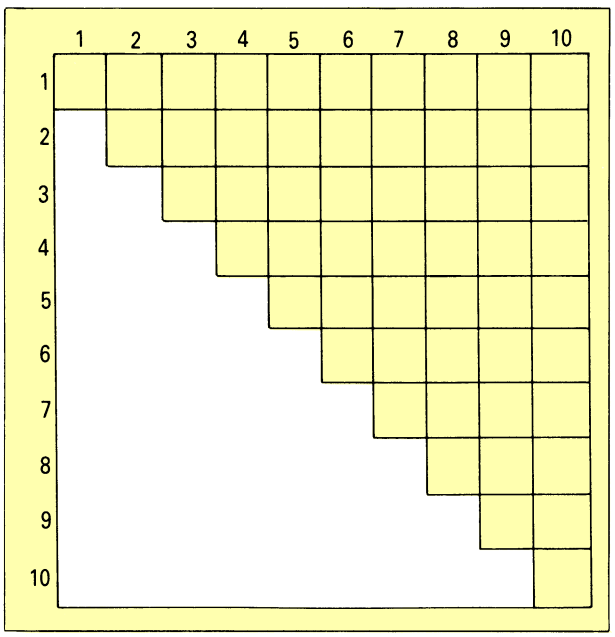


Fig. 3. A triangular array

```
100 WRITE = 1 : RD=0
110 DIM ARRAY(55)
1000 REM MAPPING FUNCTION
1010 PSN=(ROW-1)*10 +(COL-1) -
 ROW*(ROW-1)/2
1020 IF OP=RD THEN V=ARRAY(PSN
)
1030 IF OP=WRITE THEN ARRAY(PS
N)=V
1040 RETURN
2000 REM SET ARRAY(3,8) TO 127
2010 V=127
2020 OP=WRITE : ROW=3 : COL=8
: GOSUB 1000
3000 REM SET VARIABLE TO ARRAY
(9,10)
3010 OP=RD : ROW=9 : COL =10 :
 GOSUB 1000
3020 X=V
```

Fig. 4. Mapping function for a triangular array

simply not use half of the elements. But as well as being inelegant this is also a waste of memory space. Instead we will extend the idea explored in the two previous examples and declare a single-dimension array, using a mapping function so that the program can think of it as a triangle.

Figure 4 shows the example code. The triangle is stored in the one-dimensional array: first row 1, then row 2, then row 3 and so on. The only tricky bit in the mapping function is working out how much to lop off as the row numbers get higher. The formula

```
n(n-1)/2
```

gives the following sequence of values (starting when n=1):

```
0 1 3 6 10 15 21
```

and so on. This is exactly what is needed, as the values produced equal the number of 'missing' elements from the straight two-dimensional array. Notice in figure 4 that the size of the initial array contains the right number of elements, so no space is wasted.

Now we're ready to tackle something a bit more ambitious: a pyramid. Just suppose the problem on hand is crying out for a pyramidal array, and nothing else will do. For the sake of argument the pyramid has a base of 10 x 10 elements, which will make the next layer 9 x 9, the next 8 x 8, and so on up to the top layer of one element. There's a picture of our pyramid in figure 5.

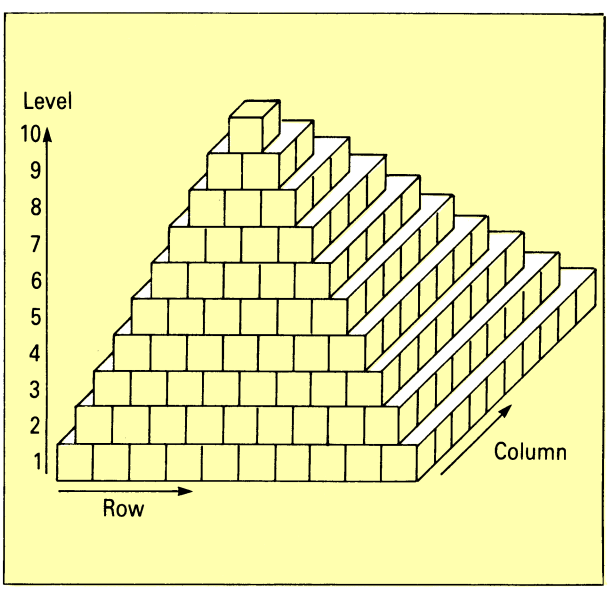


Fig. 5. A pyramidical array

In the previous example of the triangle the mapping function calculated the position by lopping off a value from a basic two-dimensional array. For the pyramid we'll take a different approach. Rather than take something off, the position of an element is found by totalling up the number of elements beneath it. Figure 6 shows the sample code.

Consider an element on the fourth level up, third row along, fifth position. Our mapping routine begins by counting the levels. If the element is on the fourth level (the 7 x 7 one) then the preceding levels are 10 x 10 and 9 x 9 and 8 x 8. The first operation of the mapping function is to total up the number of elements in these lower levels, which is achieved within the loop between lines 1040 and 1070. Once that has been done the problem resolves itself into the simple two- to one-dimensional mapping of our first example.

```
100 WRITE = 1 : RD=0
110 DIM ARRAY(385)
1000 REM MAPPING FUNCTION
1010 PSN=0
1020 BASE=10
1030 IF BASE=11-LEVEL THEN 110
0
1040 PSN=PSN + (BASE*BASE)
1050 BASE=BASE-1
1060 GOTO 1030
1100 PSN=PSN+(ROW-1)*BASE+(COL
-1)
1110 IF OP=RD THEN V=ARRAY(PSN
)
1120 IF OP=WRITE THEN ARRAY(PS
N)=V
1130 RETURN
2000 REM SET ARRAY(4,3,1) TO 1
27
2010 V=127
2020 OP=WRITE : LEVEL=4 : ROW=
3 : COL=1 : GOSUB 1000
3000 REM SET VARIABLE X TO ARR
AY(2,8,7)
3010 OP=RD : LEVEL=2 : ROW=8 :
 COL=7 : GOSUB 1000
3020 X=V
```

Fig. 6. Mapping function for a pyramid

Artificial Intelligence: Expert In The Machine

The field of artificial intelligence has had a rough time historically. It has traditionally been looked on by the world of 'legitimate' computer science as an attempt to make computers do the impossible, that is to say, think for themselves. But every step of ground gained towards that 'impossible' goal – for example the development of languages like LISP or PROLOG which deal in rules and concepts – has been seized by the legitimate computer world as its own property . . . as soon as the impossible is implemented and working it no longer lies in the domain of artificial intelligence.

Intelligence is a vague concept. The quality exhibited by many of these systems is actually more like the intelligence' that people attribute to their pet dogs, and a more appropriate description would be friendliness and predictability.

Applied to computers, this suggests a very 'user-friendly' system, which doesn't do anything nasty unexpectedly. This seems to be quite a good objective. Apart from being house-trained, we want the machine to understand us. Unfortunately, we are trying to communicate at a level a bit higher than 'sit' or 'beg', so there is a considerable amount of work still to be done.

Much excitement has been generated by the advent of the so-called **Expert System**, not least because of the massive investments in this area reportedly planned by the Japanese computer industry. An expert is someone who is knowledgeable in a particular field, able to dispense useful advice after having established the facts relevant to a particular case. An expert computer system fulfils the same role, though of course the dialogue will take place through a conventional terminal – at least for the present.

Even in a rapidly changing subject like computing, any new developments must draw upon earlier work in the subject. In the earliest stages of computing, most programs were devoted to processing numeric data. The handling of non-numeric information proved less tractable than had at first been expected. Nonetheless progress was made, even if at a less spectacular rate than at first hoped for, giving rise to the development of such languages as LISP and PROLOG.

LISP and PROLOG

Examples of LISP applications include Expert Systems and **problem solvers**. These may be distinguished by the way they interact with the user. Any Expert System, or intelligent database, asks questions to find out what is happening in the real world. The information is held not as a set of discrete data items, but as a complex web of related information.

Problem solvers know what is going on already, and will define a problem in the form of a desirable situation. They then work out the sequence of actions required to achieve this solution. In practice, the boundaries are blurred, and most systems lie between the two extremes. Examples of applications are medical diagnosis, chemical analysis, geological surveys, circuit design, applied maths – all implemented in LISP.

Learning and reasoning have been modelled using LISP. These are both just ways of manipulating knowledge: learning is a means of combining new information with that already known; reasoning is used to extrapolate that knowledge. Both of these are closely tied together, and have implications for other areas of artificial intelligence work, because they are fundamental to the idea of knowledge (whether it is being fed in as speech, text or vision or being extracted as in an Expert System).

PROLOG, first developed around 1970, enables the programmer to deal relatively easily with non-numeric data, and breaks away from the more traditional 'Von Neumann' approach of setting out a step-by-step sequence of instructions. Early computing was largely concerned with the certainties of physical laws. To write a program, one needed to understand the physics and mathematics of the physical entity involved, and to produce the programming algorithm which embodied these. Statistical analysis could also be performed, as could the simulation of systems with statistical properties (for example, traffic flows), but even here the presence of well determined underlying physical laws was generally assumed.

But many of the problems of real life are concerned with incomplete and unreliable information. Driven by this need to deal with inter-relationships where estimates are involved, the mathematics of probabilities and 'fuzzy' logic have now evolved to quite a useable state. These developments in non-numeric computing and in the mathematics of uncertainties have taken place at the same time as the advent of cheaper and more powerful computer hardware and the consequent spread of interactive computing. These ingredients have been put together by those working in the area of artificial intelligence to develop genuinely practical Expert Systems.

Who needs it?

Medicine provides a useful example of what can be done with Expert Systems. In dealing with the diagnosis and treatment of diseases we may have:

A complex combination of symptoms, identifiable with varying levels of certainty ('well, it does hurt a bit here').
A wide range of potential causes from the commonplace to the extremely rare (and quite possibly a combination of causes).
A range of potential treatments, some well proven, some experimental, many with possible restrictions according to circumstances.

Perhaps one of the best known of the pioneer Expert Systems was **MYCIN**, developed at Stanford in the mid-1970s to assist doctors in the diagnosis and treatment of diseases.

Another area of notable pioneer work is in geological exploration. In the search for mineral deposits, evidence can be acquired from a whole range of disciplines, including various scientific analyses of bore-hole samples,

seismology, gravity and magnetism. Much of this evidence will be very expensive to obtain, making it be desirable to obtain accurate predictions from the minimum of data.

The role of a manager is to take executive decisions in the light of incomplete information. Very many factors, both internal and external to the organisation, may need to be balanced. Expert systems applied in this area are sometimes used to help in making decisions, and so are known as decision support systems.

If the problems of diagnosis are considerable, those of creative design can be even more intractable. A design specification may include many parameters which have to be satisfied, and there are likely to be even more design parameters (many of them interconnected) which may be varied to meet these specifications. As you might expect, the application of Expert Systems in creative design is fairly restricted at the moment. One example has been in the configuration of complete computer systems from standard modules to meet requirements of performance, of cabinet and rack mounting restrictions, of power-supply demands, and of cable length restrictions.

A human expert may need help from a system to remember all the facts and factors to be taken into account, particularly where many scientific or social disciplines are involved, or where the state of knowledge is constantly changing. Help may also be needed where very many inter-related factors have to be balanced or reconciled. The human expert will expect to be able to intervene and to direct the lines of search followed by the Expert System, in order to take short-cuts in the light of experience.

Often the human expert will want to be able to extend and refine the knowledge held by the Expert System as more information comes in. It might be preferable, however, if the Expert System, having followed the suggestions of the human expert, is prepared to argue back where the evidence gives stronger support to some different interpretation.

The novice in training may use an Expert System to test real or simulated cases. In this situation the user will need to be able to ask the system just why it is seeking particular information, what hypotheses it is currently pursuing, and why, when it has finished, it has accepted some hypotheses and rejected others.

Systems like this may be used:

Because the appropriate human experts are scarce (for example, because of the breadth of the subject, the time needed to gain practical experience after training, the amount of new information being thrown up by research).

Because the experts are in the wrong place (it may not be possible to keep an expert standing by on a remote drilling rig, nor to send out experts on disk-drive faults to every reported computer fault simply on the off-chance that they may be needed).

Because the experts themselves need help (for example, to keep in mind all the hundreds of rare diseases as well as the common ones).

Because the time of experts is expensive (they can not reasonably be provided for long periods of training on a one-to-one ratio with trainees, for example).

Let's try it out

Consider the development of a system to give advice to a computer user on what to do when a program stops with an error message. We will assume that there are just two hypotheses or 'goals' to investigate:

There is a hardware failure, which we will assume is unlikely unless confirmed by subsequent evidence (the jargon for this is that its initial or prior probability is low).
There is a programming error, which we will assume is very likely until proven otherwise (its prior probability is high).

To confirm or reject our hypotheses we will need the following items of evidence:

Whether the program that has failed has run successfully many times before (in other words it is a production program).
Whether other production programs have also recently failed.
Whether the hardware test programs fail when we run them.
Whether we have isolated the exact statement which caused the failure in the program.
Whether the isolated statement is definitely logically correct.

Figure 1 shows how these items of evidence contribute to decide the two hypotheses.

Next we must express these relationships or rules in the language supported by our Expert System. As an illustration Figure 2 uses the MICRO EXPERT package from Isis Systems. The five items starting `QUESTION` relate to our five items of evidence on which the system will question the user. In each case the heading `QUESTION` is followed by the name of the question (for example '`hware-tprogs-fail`' on line 1) and then by the actual text of the statement as it will appear on the screen of the terminal ('`Hardware test programs fail?`' on line 2).

In our very simple model, all the questions will query the user's degree of certainty that the statement is true, hence the use of `CERT` (on line 3, for example) to specify this type of question. The system expects the answer to be a number between `+5` (signifying 'extremely likely') through `0` (signifying 'uncertain' or 'don't know') to `-5` (signifying 'extremely unlikely'). Other forms of question expect an answer in the form Yes or No, or as a simple numeric quantity.

On line 13 of figure 2 you will see one of our rules of inference, relating observations (elicited from the user as the answers to the three previous questions) to one of our hypotheses – that there is a hardware failure. Line 13 starts with the label `GOAL` because this hypothesis is one of the major points that we definitely want the system to decide for us.

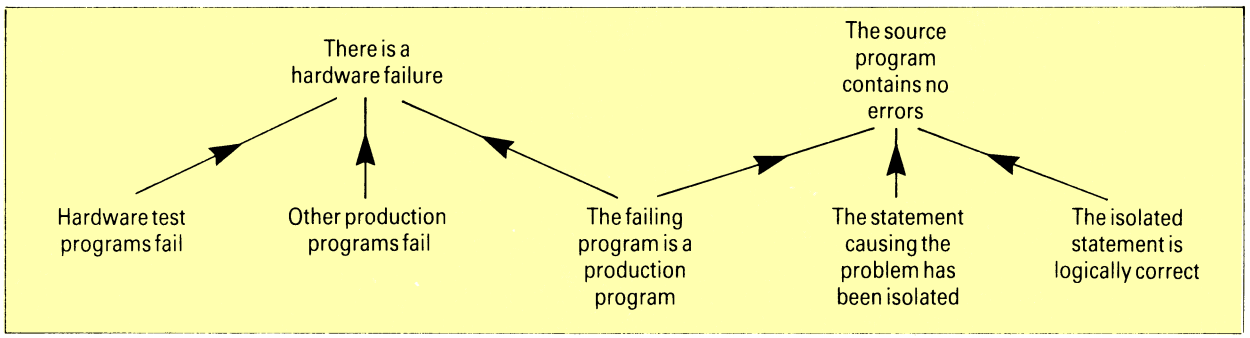


Fig. 1. How evidence is used to decide two hypotheses

```
 1  QUESTION hware-tprogs-fail
 2           'Hardware test programs fail'
 3           CERT
 4
 5  QUESTION other-pprogs-fail
 6           'Other production programs also fail'
 7           CERT
 8
 9  QUESTION production-prog
10           'This program has run many times successfully'
11           CERT
12
13      GOAL hardware-fail
14           'There is a hardware failure'
15           BLOCKED production-prog – 1.9.5.0
16           BAYESIAN hware-tprogs-fail LS 125 LN 0.315
17                    other-pprogs-fail LS 63 LN 0.315
18                    production-prog LS 12 LN 1.0
19                    PRIOR 0.01
20
21  QUESTION statement-isolated
22           'The exact statement causing the problem has
been isolated'
23           CERT
24
25  QUESTION statement-ok
26           'The isolated statement is definitely ok (no
program error at all)
27           CERT
28
29      RULE program-ok
30           'The source program contains no errors'
31           BLOCKED production-prog – 5.0 1.9
32           BAYESIAN production-prog LS 2 LN 0.1
33                    statement-isolated LS 10 LN 0.1
34                    statement-ok LS900 LN 0.0001
35           PRIOR 0.01
36
37      GOAL program-error
38           'The source program is in error'
39           NOT program-ok
40
```

Fig. 2. Example expert program, using MicroExpert package

On line 16, this goal is identified as of the type **BAYESIAN**. This indicates the type of mathematics by which the calculations to decide the certainty of the goal are to be computed. This is followed by the names of the three observations or **QUESTIONS** which will be used to decide on the certainty of the **GOAL**.

The first of these is '**hware-tprogs-fail**'. **LS** stands for *Logical Sufficiency* and indicates the factor by which the certainty of the **GOAL** is to be increased if the question has received the answer 'Yes' or 'very likely'. In this case if the system is told that the hardware test programs *do* fail, then we can be much more certain that there is a hardware failure, and so we will apply a relatively high Logical Sufficiency factor of 125 to reflect this.

Conversely, **LN** stands for *Logical Necessity* and indicates the factor by which the certainty of the **GOAL** is to be reduced if the question has received the answer 'No' or 'very unlikely'. In this case, if the system is told that the hardware test programs *do not* fail, then a hardware failure may be taken to be less likely, and a reduction factor of 0.315 will be applied.

For this **GOAL** the three Logical Sufficiency factors have been chosen so that if all three questions receive the answer 'Yes' or 'very likely' then the certainty of a hardware failure may also be taken as 'very likely', while the three Logical Necessity factors have been set so that if all the answers are 'No' then the chances of a hardware failure may be taken as 'very unlikely'.

The figure for **PRIOR** in line 19 indicates that for this **GOAL** the initial probability for a hardware failure in the absence of any evidence should be taken as 0.01.

The second half of the definition of the system (from line 21 onwards) deals similarly with the problem of deciding if there is a program failure. In this case it was more convenient to decide the **GOAL** '**program-error**' as the logical opposite of the intermediate **RULE** 'program – ok'.

When faced in real life with the problem of a failure during a program run, it might save us time in making investigations if we assumed a hardware error for a well-proved production program, whilst assuming a programming error in the case of a newly written program.

In a simple attempt to reflect this approach, the **BLOCKED** statement on line 15 indicates that the 'hardware – fail' **GOAL** is not to be investigated unless the certainty that it is a production program which has failed

lies between −1.9 ('possibly not') and 5.0 ('it definitely is'). The 'program–ok' **RULE** is blocked in a similar way (using different values) in line 31.

So much for imparting our knowledge to the system: how does it work out when we run the system? The output from one run is given in Figure 3. The system is as informative as possible when running, and generates quite a lot of console output which has been edited out for the purposes of this illustration.

Lines 100 and 101 announce the goal the system wishes to explore first, whilst the next line indicates that so far the system has no information to decide on (a 'certainty' of 0.0). The system next asks whether it was a proven production program which failed (lines 104, 105). It receives the answer 'Yes' (on line 105 the user enters 4, just a little down from the highest certainty of 5). This reply increases somewhat the certainty of the goal (line 107).

When the system next discovers that hardware test programs fail (lines 109, 110), the certainty of the goal is dramatically increased (line 112). The information that other production programs are possibly still OK (lines 114, 115) does not much diminish the certainty of the final

```
100  The current goal is whether or not:-
101  There is a hardware failure
102  Certainty Factor is 0.00 Certainty range is −4.50 to + 4.99
103
104  How certain are you that This program has run many times
105  successfully [−5..0..+5]?4
106
107  Certainty Factor is 0.40 Certainty range is −0.14 to +4.99
108
109  How certain are you that Hardware test programs fail
110  [−5..0..+5]?5
111
112  Certainty Factor is 4.64 Certainty range is 3.97 to 4.99
113
114  How certain are you that Other production programs also
fail
115  [5..0..+5]?−1
116
117  This goal was whether or not
118  There is a hardware failure
119  Certainty Factor is 4.57
120
121  There are no more questions for this goal
122  There are no more goals to answer
123
124  Type Q to end this session. Type R to see the Report.
     (user input is underlined)
```

Fig. 3. Example run of Fig 2

conclusion (lines 117 to 121).

Because of the blocking mechanism we described above, the system does not bother to explore the question of a program error (line 122). At this point a detailed printout of the results of the session can finally be requested (line 124).

You may well object to the extreme naivety of our Expert System. For example:

> We have not allowed for multiple causes of failure.
> We have subsumed data errors within program errors.
> We have not allowed for the possibility of a system software failure (for example of a compiler).
> We have not allowed for the fact that in the end even the most trusted of production programs may (will?) still contain a programming bug.

These objections are, of course, perfectly valid. But we have at least managed to draw a number of significant points:

> The actual formal or logical design of even the most straightforward deductive process can be quite subtle (if it were not we would not need to build the Expert System).
> But given the logical design, the actual encoding of our system is pretty straightforward.
> We shouldn't necessarily believe the answers just because they come out of a so-called 'Expert System'. The answer will be reliable only if the knowledge obtained from the expert was correct, if that knowledge was then correctly encoded, and finally if the user consulting the system answered the questions accurately.

MICRO EXPERT was run on an ordinary disk-based micro, and is capable of supporting quite complex knowledge systems. In this particular system only the knowledge structure (the relationship between the questions, rules and goals) is kept in RAM, whilst the voluminous text messages which may be presented at the terminal are all stored on disk.

This allows maximum space for the structure without unduly affecting response times. In principle it would be possible to partition distinct aspects of the knowledge structure, for example the diesel and the electric parts of diesel-electric locomotives, and to load into RAM and work on each part as appropriate.

In conclusion, it is now possible to model an 'Expert System' in a quite modest micro using subsets of the knowledge held by experts from many different fields. It is the formalisation of the knowledge into questions, rules and goals which presents the real challenge. Existing generalised Expert System packages can then help to make it easier to encode that formalised structure, and can provide all the built-in mechanisms necessary for asking the questions, making appropriate inferences, and displaying the conclusions.

Of course, this should not be taken to imply that all the problems have been solved and that it is all plain sailing from here, but at least things are off to a genuinely practical start.

ASCII: The Standard For Storing Characters

In 1880 J.E.M. Baudot invented what has become known as the Baudot Code, which by the 1950s had become the main standard for international telegraph communications. This is a code made up of five binary digits which, with a system similar to that used by the 'shift' key on a typewriter, allows the 32 unique combinations to represent 58 distinct characters – six are duplicated so they can be used in either mode (*see* Figure 1). Unfortunately a single mistake in transmitting a shift character was likely to produce a stream of special symbols instead of letters, which could be very confusing.

In the 1950s, with the computer appearing on the horizon, came the need to represent all of the 26 lower-case characters as well as the upper-case characters, the numerals 0-9, punctuation, and special characters (such as * / # _ & ' + = ?) – many more symbols than the 58 characters which had proved so adequate for telegraph communication.

Virtually every different computer company came up with a different way of encoding these characters into binary numbers but, unsurprisingly, it was IBM's system which came to be predominant. This was eventually refined into Extended Binary Coded Decimal Interchange Code, affectionately known as EBCDIC (*see Figure 2*).

EBCDIC uses eight bits to define 256 unique characters, which is where the idea of calling eight bits a 'byte' comes from. This scheme not only includes all the upper- and lower-case letters, as well as the numerals and special symbols, but it also leaves plenty of byte patterns over to represent special machine-dependent codes to do things like ringing bells on terminals and ordering a printer to move to a new line.

Figure 2 shows why it is a terrible mistake to type a capital O instead of zero, or a capital I instead of a one.

Letters						Figures						
A	1	0	0	0	0	1	Q	1	0	1	1	1	/
B	0	0	1	1	0	8	R	0	0	1	1	1	–
C	1	0	1	1	0	9	S	0	0	1	0	1	.
D	1	1	1	1	0	Ø	T	1	0	1	0	1	NA
E	0	1	0	0	0	2	U	1	0	1	0	0	4
F	0	1	1	1	0	NA	V	1	1	1	0	1	'
G	0	1	0	1	0	7	W	0	1	1	0	1	?
H	1	1	0	1	0	+	X	0	1	0	0	1	'
I	0	1	1	0	0	NA	Y	0	0	1	0	0	3
J	1	0	0	1	0	6	Z	1	1	0	0	1	:
K	1	0	0	1	1	(	LS	0	0	0	0	1	LS
L	1	1	0	1	1	=	FS	0	0	0	1	0	FS
M	0	1	0	1	1	)	CR	1	1	0	0	0	CR
N	0	1	1	1	1	NA	LF	1	0	0	0	1	LF
O	1	1	1	0	0	5	ER	0	0	0	1	1	ER
P	1	1	1	1	1	%	NA	0	0	0	0	0	NA

Fig. 1. Baudot code

They may look similar on the screen or when reproduced on the printer, but a computer treats them as differently as A and Z. In EBCDIC all numbers begin with the binary code 1111. The code for capital I is 11001001; the code for one is 11110001.

The emergence of ASCII

In the sixties an attempt was made by the American National Standards Institute (ANSI) to define a national standard for data transmissions between computers. The protocol chosen was not EBCDIC but a new code called ASCII – the American Standard Code for Information Interchange.

The ASCII set uses only seven bits, to give 128 characters – 32 special codes and 96 alphabetic, numeric

	0000 0001 0010 0011	0100 0101 0110 0111	1000 1001 1010 1011	1100 1101 1110 1111
0000	**NULL**	**HT**		
0001	**TM**	**LF BS**	**DC1 DC2 DC3**	**STOP**
0010	**DS SST FDS**		**VT**	**FF CR SO SI**
0011			**EM SUB ESC**	**FS GS RS US**
0100	**SP**		£, .	< (+ 1
0101	&		! $	*) ; –
0110	– /		< '	% -o- > ?
0111	.		: #	((' = "
1000	a b c	d e f g	h i	
1001	j k l	m n o p	q r	
1010	s t	u v w x	y z **SOH DLE**	**CAN NAK SYN ETB**
1011			**STX ETX**	**EOT ENQ ACK BELL**
1100	A B C	D E F G	H I	**CAK**
1101	J K L	M N O P	Q R	**DAK**
1110	S T	U V W X	Y Z	**DOS**
1111	0 1 2 3	4 5 6 7	8 9	◇
Example: Code for A = 1100 + 0001				

Fig. 2. EBCDIC code

COLUMN	0	1	2	3	4	5	6	7
ROW	BITS 4321 765	000	001	010	011	100	101	110	111
0	0000	**NUL**	**DLE** control P	SP	0	@	P	/	p
1	0001	**SOH** control A	**DC1** control Q	1	1	A	Q	a	q
2	0010	**STX** control B	**DC2** control R	"	2	B	R	b	r
3	0011	**ETX** control C	**DC3** control S	#	3	C	S	c	s
4	0100	**EOT** control D	**DC4** control T	$	4	D	T	d	t
5	0101	**ENQ** control E	**NAK** control U	%	5	E	U	e	u
6	0110	**ACK** control F	**SYN** control V	&	6	F	V	f	v
7	0111	**BEL** control G	**ETB** control W	'	7	G	W	g	w
8	1000	**BS** control H	**CAN** control X	(	8	H	X	h	x
9	1001	**HT** control I	**EM** control Y	)	9	I	Y	i	y
10	1010	**LF** control J	**SUB** control Z	*	:	J	Z	j	z
11	1011	**VT** control K	**ESC**	+	;	K	[	{	
12	1100	**FF** control L	**FS**	,	<	L	/	|	|
13	1101	**CR** control M	**GS**	-	=	M	]	m	}
14	1110	**SO** control N	**RS**	.	>	N		n	~
15	1111	**SI** control O	**US**	/	?	O	-	o	**DEL**

Example: Code for A =

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
			1	0	0	0 1

LEGEND for Control Codes in Columns 0 and 1:

NUL	Null	**VT**	Vertical Tabulation	**SYN**	Synchronous Idle
SOH	Start of Heading	**FF**	Form Feed	**ETB**	End of Transmission Block
STX	Start of Text	**CR**	Carriage Return	**CAN**	Cancel
ETX	End of Text	**SO**	Shift Out	**EM**	End of Medium
EOT	End of Transmission	**SI**	Shift In	**SUB**	Substitute
ENQ	Enquiry	**DLE**	Data Link Escape	**ESC**	Escape
ACK	Acknowledge	**DC1**	Device Control 1	**FS**	File Separator
BEL	Bell (audible signal)	**DC2**	Device Control 2	**GS**	Group Seaparator
BS	Backspace	**DC3**	Device Control 3	**RS**	Record Separator
HT	Horizontal Tabulation (punched card skip)	**DC4**	Device Control 4	**DEL**	Delete
LF	Line Feed	**NAK**	Negative Acknowledge		

Fig. 3. ASCII code

and other printable symbols, (*Figure 3*). The other bit needed to make up an eight bit byte can be used for 128 additional (often graphic) characters, as a parity bit, or anything else you can dream up. WordStar, for example, uses it to store formatting information while Digital Research reserves some of the extra bits in CP/M file names to mark files for special treatment.

IBM's EBCDIC is still used internally in almost every mainframe and minicomputer, although ASCII rules on the micros. The standard was designed to let computers talk to each other, with a number of codes (such as the ETB code shown in Figure 3) set aside to help with the management of the flow of communication.

Things didn't turn out quite as expected by the American standard setters – a phenomenon that the setters of standards must be used to by now – and most developers of micros chose to use the ASCII code, if not always in the same way. Nowadays most micros will store text files in ASCII code, and many will also use the ASCII codes to perform functions like ringing the bell and backspacing (*see* the chapter on printers). But other ASCII control character codes have different uses on different machines. For example, the CP/M operating system uses CONTROL-C (in ASCII this represents the end of a piece of text) to restart the system. Perversely, CP/M's end of text code is CONTROL-Z which has a completely different meaning in ASCII . . .

Most versions of BASIC have a command to convert decimal values into ASCII characters. For example to to get the micro's bell to ring while a program is running, you might insert the statement `PRINT CHR$(7)`. **7** is the decimal value of the ASCII binary code BEL. The function `ASC()` also found in most standard BASICs, works the other way round, turning ASCII characters into their decimal values. To find out the decimal ASCII value of a particular character, you would use the function `ASC()`. Thus `ASC('A')` would return the value 65.

Despite this level of standardisation, ASCII alone doesn't offer much hope for micro users wanting in-depth chats with other micros. For a start, ASCII contains no standard for the speed at which the micros would communicate, nor for starting off the conversation (the *handshake*).

It is only useful for exchanging text files, and for sending instructions to machines which expect their instructions in ASCII form. What ASCII does not provide, in short, is a set of protocols needed in common between two machines wanting to communicate.

Assembler: Ground Level Programming

Getting down to the nitty gritty of assembly language programming is an excellent way to learn about microcomputers. Machine coding – entering the values that form a program directly into memory as a series of obscure hex bytes – is certainly an interesting exercise on a long winter evening, but is an impossibly ponderous way to develop programs of any length. On the other, hand-high level languages like BASIC which shield the programmer from the machine's tedious details, also mask its true performance by creating slow and bulky code.

Where an assembler fits in

All useful program development depends on being able to write 'source code' in a language which can achieve a compromise between the English a human understands and the simple 'ons' and 'offs' that the processor can read. Yet however the program is written initially, it will still have to end up as machine code, or 'object' code as it is sometimes called. Since the computer's processor is only able to understand a fixed number of very simple instructions, its power comes not from the number of elementary actions it can undertake, but from the very high speed at which its electronics can carry them out. This speed makes it possible for a high level instruction from a language like BASIC to correspond to many machine code instructions, which in turn enables the programmer to write in relatively meaningful 'real world' terms like `PRINT"Hello,Printer"`.

But in making this translation from the high level of the language to the low level of the processor, the compiler or interpreter can't help introducing redundancies and inefficiencies. Because translation programs have to cope with all the possible expressions that might be used in a language, they never produce object code that is as fast or as compact as the program would have been if it had been written directly in machine code.

This is where the assembly languages come in. Sandwiched between easy-to-use high-level languages and almost impossible-to-work-with machine code, assembly languages provide a convenient way of producing machine code programs. Unlike a high level language, one instruction in assembly language corresponds directly to one particular machine code instruction. The difference is that rather than have a binary number for each instruction, programmers can use a 'mnemonic', a short name that serves as a human memory-jogger to the meaning of the instruction. And instead of being restricted to numbers which have no meaning for humans, programmers can use whatever names they want for the various data and addresses they use in the program. Instead of the raw lists of numbers that comprise a machine code program, assembly language allows the programmer to add comments to the program, showing what it does and how it does it.

Using an assembler

A powerful editor/assembler is the best tool yet devised for investigating and successfully programming a micro, particularly where memory is limited. An assembly package usually includes an editor, assembler, linker, debugger, monitor and disassembler. Sometimes the monitor and disassembler will come with the computer (stored in ROM) and sometimes as separate packages.

The **editor** is tailored to writing assembly programs, and can either be line or screen based (*see* Text editing). A line editor may include syntax checking just after the line is entered, but will usually not offer very powerful editing features. A screen based editor makes the entry and correction of a program much easier, with full cursor control for modifying and inserting characters. Screen editors are not really suited to syntax checking, but their powerful editing features make them preferable to the simpler line editors. A good development system will include features such as the ability to insert files from disk, global find and replace and copying and moving sections of the source code.

An **assembler** will take the source code produced using the editor and translate the mnemonics into machine readable form. Assemblers can make either one or two 'passes' of the source file. On a two pass assembler the first operation checks the syntax and may produce a tokenised temporary file (often residing in RAM); then the second pass takes this intermediate code and produces the binary code. A tokenised file is one where single symbols (tokens) are used to replace a string of characters to save space and supply values for variables. A single pass assembler tries to do both operations at once; a full two pass system is usually preferable.

A **linker** allows you to develop large programs in chunks that are easily manageable by the programmer and also by the editor. These 'modules' are assembled independently, and later combined into a single file of code that comprises the program. One advantage of this is that when a bug is detected in a single module, you only need edit the source code of that module, run it through the assembler again, and use the linker to patch it back in the program. Otherwise you would have to reassemble the entire program. As a linker has to have access to special tables of global variables and addresses, which the assembler must know how to set up, it must be an integral part of an assembler package. A linker does not come as a stand-alone program.

A **debugger** is an aid to checking a program while it is running. It allows operations such as the setting of break points, where the program is temporarily stopped allowing the programmer to examine various memory locations or registers. Sophisticated debuggers enable the program to be stepped through, one instruction at a time, effectively having a break point after every single instruction.

A **monitor** – not to be confused with the same word used to describe a screen display device – is a collection of software routines, usually kept in ROM, which can be used instead of a debugger. Generally a debugger will present information in a more helpful manner, and will

include more powerful commands. A monitor allows memory contents to be examined, changed and compared with other locations, but will almost certainly present the information solely in lines of hex responding to obscure single letter commands.

A **disassembler** is sometimes included in a debugger or a monitor. Its role is to take the object code of a program and try to translate it back into assembler mnemonics. The disassembler has a difficult job, as there is no way it can know by looking at a single byte whether the value represented is an instruction or data. Good disassemblers, however, are able to make intelligent guesses, applying simple rules like 'if a previous instruction executed a data fetch from this address the value here is probably meant to be data', 'if the program counter is set to this address by a call or jump instruction then it is almost certainly intended to be code', and 'if a particular patch of bytes are never visited by the program counter in the course of execution a reasonable assumption is that they represent data.'

When debuggers go astray they do so disastrously, generating reams of impossible assembler mnemonics, sometimes sprinkled with question marks. Using a debugger to examine other people's code – inspecting the inner workings of an operating system perhaps – requires a good knowledge of the ins and outs of assembler, and a great deal of patience.

Cassettes and disks

An editor/assembler package can come on cassette, disk or cartridge. Cassette-based systems are perfectly useable in the early stages of learning assembler, but for serious development work a disk-based system is far more convenient. Cassette-based systems do not usually include linkers, may not include debuggers, and usually have less powerful editors. Overcome these disadvantages how you might, programs – which almost certainly will have taken a long time to develop – still have to be stored on those notoriously unreliable cassette tapes (*see* Backing Store). A cartridge system usually allows programs to be saved either to cassette or disk, but this flexibility means that disks cannot be assumed, so the features it offers will usuallly be similar to its cassette equivalent.

What to look for

Probably the two most important features to investigate in an assembler are whether and to what extent it offers **macros**, and what **pseudo-ops** are included. A macro is the assembly programmer's equivalent of the rubber stamp, allowing an identifying name to be associated with a small chunk of code the programmer expects to use several times in writing a main routine. Instead of having to write out the code each time, the programmer simply includes the name of the macro, and the assembler takes care of the necessary expansion.

Beginners understandably confuse macros with subroutines. A subroutine will appear only once in the final assembled program, although it may be called by the running code from many different places, exactly like the target of a `GOSUB` statement in BASIC. Macros on the other hand are expanded wherever they occur in the source code, and are executed 'in line', as the jargon has it. Unlike subroutines they therefore take up space in the final program, and the more times you use a particular macro the more space it takes. But macros do make life easy for the programmer, and as an added refinement can usually take parameters, allowing each expansion to be slightly different. The instruction sets of some processors, the Z-80 for instance, suit the use of macros better than others such as the 6502.

Pseudo-ops are special instructions directed to the assembler, which obeys them directly, instead of trying to translate them into object code. The most common pseudo-ops are those used to initialize constants; bytes, words, or strings. Most assemblers will allow a number of constants to be defined by a single directive. Pseudo-ops may also direct the assembler to switch on or off its listing to the printer, begin new pages and add title lines. The listing, which may be a file or a printed copy, provides a valuable cross-check for the programmer, as it contains both the source code and the assembled object code expressed in hex.

```
LAB1    DEFB 1,2,3,4
LAB2    DEFW AOOOH, BOOOH
LAB3    DEFS "this is a string"
```

Fig. 1. Initializing pseudo-ops

Some packages will allow constants to be defined in several different number bases, allowing you to use decimal for simplicity, hex to save space, binary to express constants whose bit patterns are more important than their numerical values, or even – for the Americans and old timers – octal. String constants can normally be entered in ASCII, to be translated automatically to binary by the assembler. Figure 1 shows a few examples. Labels (for example, `LABI`) are used to address the data. The first example sets up four consecutive bytes, holding 1, 2, 3 and 4 respectively. The second directive sets up two 16-bit, or 'word' addresses, and the third sets up 16 consecutive bytes holding the binary values of each character.

```
LST     (turn list file on)
UNL     (turn list file off)
PAGE    (skip to a new page)
TITLE   (title at top of each page)
```

Fig. 2. List output directives

Figure 2 shows a number of useful list output directives. A third set of pseudo-ops, in Figure 3 alters the addresses assigned to the start of the program. These are available because it is quite likely that the area in memory where the program will be finally run is different from the area used during its assembly.

The more sophisticated the system, the more pseudo-

ops are required. When using a linker, for example, special operations are needed to allow variables to be accessed by other independently assembled program modules. Figure 4 shows the sort of pseudo-ops which might be included to do this. Some are only to be found in the best of the packages.

```
AORG 0800H    (set the absolute origin
               of the program to hex 0800)

RORG + 20     (set the origin 20 bytes further
               up than the current location)
```

Fig. 3. Changing program origins

```
COPY < file >          (include another file
                        into the assembly)
END                    (end of program marker)
REF < name >           (refer to a label not
                        in this source file)
DEF < name >           (define a label to be used
                        by other source files)
DFS < n >              (leave space for < n >
                        bytes of data)
EVEN                   (start on an even
                        byte boundary)
IF . . .THEN
ELSE                   (conditional assembly)
```

Fig. 4. Pseudo-ops found in more sophisticated assemblers

Assembler and BASIC

Compare the two sections of program in Figures 5 and 6. Both search through a list of 100 numbers and find the largest one present. One is in BASIC and the other is in assembly language for the 6502 processor. Notice that the 6502 version needs more instructions and is less comprehensible, even with its comments. The 6502 listing was produced by an assembler as it performed the translation. The source program as written and entered by the programmer appears on the right. On the left, the assembler has output the object program by giving the locations in memory and the values stored there opposite each corresponding instruction in the assembly language version.

```
BASIC PROGRAM

100   REM FIND LARGEST NUMBER
110   LET B = 0
120   FOR C = 1 to 100
130   IF N (C)> B THEN B = N (C)
140   NEXT C
150   REM B NOW = TO LARGEST VALUE
```

Fig. 5. BASIC routine to search through a list of 100 numbers

ASSEMBLY LANGUAGE PROGRAM

```
asm
                    1  * Find largest number
                    2  *
                    3  * Searches a list of 100 numbers
                    4  * Returns the largest in 'BIGST'
                    5  *
                    6  * First, tell the assembler
                    7  * What 'BIGST' and 'BASE' refer to
                    8  *
                    9  BIGST Equ 100
                   10  BASE  Equ 512
                   11  *
                   12  * The actual program starts here
                   13  *
7000:A9 00         14  Begin     LDA $00          ; set biggest to zero
7002:85 64         15            STA BIGST
7004:A0 64         16          LDY 100; set count to 100
7006:B9 01 00      17  Loop      LDA BASE,Y       ; get next number
7009:C5 64         18            CMP BIGST        ; is less than BIGST?
700B:90 02         19            BCC Less than    ; if yes, try next
700D:85 64         20            STA BIGST        ; else set new value of BIGST
700F:88            21  Less than DEY              ; take one off the counter
7010:DO F4         22            BNE LOOP         ; and repeat if any more to do
                   23  *
                   24  * BIGST now contains the
                   25  * Largest value
                   26  *
Machine code       Assembly language
object program     source program
```

Fig. 6. Assembly equivalent of Fig 5

The assembler has given the machine code values in hex rather than binary simply to keep the listing in a manageable form. Although the BASIC version is apparently shorter, remember that once assembled, only the actual machine code will be used in running the assembled program. In contrast, BASIC will need to keep the source program and the entire BASIC interpreter in memory during execution. In fact the 6502 version will run some hundreds of times faster than its BASIC equivalent.

Overcoming the limitations

Although assembly language is not especially difficult to learn or use, it is a very different skill to programming in a high level language. The programmer does have to learn how the computer works and detail every little operation that takes place.

Apart from the simplistic nature of the instructions, the lack of built-in aids for testing and correcting programs in assembly language make the programmer's job that bit more difficult, so that a disciplined approach to programming is absolutely essential. The programmer must concentrate on design and advanced planning to make programs as reliable as possible from the start. The sensible approach to a task of starting with a general statement and dividing it up into ever more detailed sub-tasks is just as important here as it is in the so-called 'structured languages' like PASCAL.

The experienced assembly programmer has several tricks up his sleeve. One is to invent, learn or steal short sequences of instructions to do the commonly needed tasks not included in the processor's ownset of

instructions. These can be kept in a library, perhaps in the form of ready-to-use macros, and automatically written out when needed in a program. Standard sub-routines can be introduced in the same way. Adding a previously tested subroutine to a program provides a way of combining instructions into useful segments which can be confidently used by the rest of the program. Subroutines and macros are thus often used like building blocks.

In assembly language, the programmer must be concerned about the way the program fits into the computer as a whole – not just how it shares memory with the operating system, the memory-mapped screen and ROM software, but also how it addresses ports and peripherals. Sometimes existing routines to handle the program's input and output will not exploit a machine's potential to the full, and by using assembly language it is sometimes possible to improve performance by adding, for example, faster disk access and better screen handling.

The joys of assembler

Unlike programming at a high level, assembly language has no built-in facilities for structuring and processing data and programmers must design and implement whatever they need. This does mean they need to know the many ways in which the humble binary word can be used to store information, and that those primitive instructions can be combined to do more useful calculations. In return they provide themselves with whatever structures they want, and with the maximum economy of space and speed.

The benefits that assemblers have to offer the programmer show up in a variety of different applications, from system software such as compilers and operating systems, to word processors and database managers, and – particularly – arcade games.

The inherent speed of assembler is its main attraction but there is a trade-off between taking a long time to develop a fast program and using a high level language to dash off a less efficient program quickly. Often the use of an assembler removes problems of speed, but dramatic improvements can also come from changing the way a program works rather than the language it uses. As an alternative, some compiled languages (for example C) and some interpreted languages (for example FORTH) produce surprisingly fast object programs. Before these were available, assembly language was the only way of producing programs quickly enough to communicate with peripherals and professional typists, to provide sorts and searches faster than people, or to give a space invader a sporting chance of success. Even if a high level language is used for the bulk of the program, sneaking off into a short machine code routine to sort or update a display is a common trick for combining the best of both worlds. Frequently, machine code routines are used as extensions to high level languages: routines to draw shapes, play music and mess about with textual data are often added to BASICs which were originally provided without them.

The compactness of machine code is an obvious benefit in any system with a limited memory space. Writing ever shorter routines to do the same job is the strange obsession of many programmers. Even if you don't want to write a mind-bending word processor or a full spec. space invader game, machine code can come in handy for the odd short routine – a patch to allow your database management program to drive an unusual printer or to add a word count facility to a word processor package.

Finally, assembly language teaches a discipline which will prove invaluable even when working with other programming languages and trying to understand and solve the inevitable problems which occur with software and manuals. Apart from the enjoyable experience of learning it, assembly language gives you the final say in exactly what your computer is and does.

Backing Store: Keeping A Hold On Software

Microcomputers, unlike elephants, tend to forget things. Switch a micro off and any information in its volatile RAM memory will be lost – anything more than casual computing demands a securer way of retaining data and the general description of this long-term storage is 'backing store'. Especially designed reel-to-reel tape recorders, and large, rapidly rotating magnetically coated steel drums have played an important part in the development of backing store technology and today three types predominate: audio cassette, floppy disks and hard disks. However technological developments are rapidly bringing new and exciting solutions to the problems of backing store.

Audio cassettes

Although never originally designed to store computer data, audio cassette (*qv*) technology has been pressed into service for the purpose as a result of its relatively low cost and its wide availability. Using a cassette machine to store data from a computer requires a special interface – normally built into the computer – to turn the digital data into a series of audible tones (for recording) and back into data when replaying. Such a system also needs a correctly made up cable to transfer the signal from the computer's cassette output to the microphone input of the recorder, and again from the recorder's earphone output to the computer cassette input. There is often a third connection which taps directly into the computer's power supply to turn the cassette recorder on and off so that the computer has some, albeit limited, control over the tape transport mechanism. With an arrangement like this you will usually still have to rewind and fast wind the tape manually.

Some micros use a custom-built cassette recorder connected to the micro with a more multi-way connector than most. This kind of integrated cassette recorder may be completely under the control of the computer which can wind and rewind the tape, all the time keeping track of the tape's position – sometimes even using this as a value in program calculations.
(*See* the chapter on cassettes).

Floppy disks

Because of the high-precision engineering required, floppy disks are a considerably more expensive way (in capital cost terms, although not usually in cost-per-bit terms) of storing data than cassette machines, but for those who can afford them there is no doubt of their superiority in terms of speed and storage capacity. A program which takes minutes to load from a cassette can be up and running within seconds when loaded from a disk. With a standard capacity of 800k for a double-sided double-density disk, floppies are also considerably more convenient for storing a large number of files.

The standard 8" disk pioneered by IBM is seldom used in micros today, the norm being 5¼" 'mini-floppies'. A new range of sub-4" micro-floppies (*qv*) pioneered by manufacturers such as Sony are now bringing floppy technology to small portables and desk top machines.
(*See* the chapter on floppy disks).

Hard disks

Hard, or Winchester (*qv*) disks are as much an advance on floppies as floppies are on cassettes in respect of speed and capacity – and the capital cost is correspondingly greater. But they have revolutionised the mid to top-range of the business micro market, and as sales volumes increase and prices fall, the home micro market is bound to reap some benefit.
(*See* the chapter on Winchester disks).

RAM disks

Many systems now allow disk drives to be emulated in RAM, so that the system can read and write to something which appears to behaves exactly like a disk, but which is, in fact, a reserved area of the core memory. This approach is very useful in systems doing a great deal of disk thrashing – software development houses for example, with their seemingly endless compiling – but it

Media	Approx. capacity	Approx. speed relative to standard floppy	Comments
8 inch floppy	1.2Mb	1	Most 8 inch systems use a standard IBM format
5.25 inch floppy	200K/1.2Mb	1	Vary in capacity 70K-1.2Mb. No standard formats. Some fast, others slow.
Sony 3.25 inch microfloppy	380K	2	The most popular microfloppy so far
Bats NC13 inch microfloppy	150K	1	The first microfloppy system
Hitachi-Maxwell 3 inch micro- floppy	250K	1	Double sided system
Hard disk	3-20Mb	20	Usually fixed
RAM disks	256K	100	Using spare RAM to simulate a disk/drive. Virtually instant access.

Table of different types of disk

suffers from the disadvantage that all the 'disk' contents are lost when the system is turned off so that initialising and closing down can be time-consuming and tedious.

A RAM disk hardly qualifies as backing store, but there is a closely related memory system which is beginning to become very popular in small portables. Low power battery-backed CMOS RAM can be used as a RAM disk to store files and to generally emulate a disk without any of the problems of moving parts. When the machine is turned off, a tiny trickle of current from a rechargeable battery is enough to retain all the data permanently – well, at least for the several months it takes the batteries to run down.

Bubble memory

Bubble technology is a technique for truly non-volatile storage of data in a compact 'solid state' form with no physical moving parts. The word 'bubble' isn't really helpful in trying to explain how the system works, unless you can visualise a chain of tiny magnetic 'domains' or bubbles being driven around a loop in a film of magnetic material by powerful electro-magnetic coils. In fact, under a microscope, these domains do look remarkably like bubbles although no physical material actually travels in the process.

Bubbles drives are capable of operating roughly four times faster than floppy disks. The high reliability of the system, due largely to its total absence of moving parts and lack of need for a standby power supply, has made it a favorite for military applications, where price is a trivial consideration. In the commercial market the relatively high cost has held back this otherwise very promising line of development. One of the main companies pursuing bubble research, Texas Instruments, dropped out of the race, although with the surge of interest in portable computers a number of bubble-based machines are beginning to appear on the market.

Optical disks

The very high density of data-packing has made the technology used to develop the compact optical disk for home audio use of considerable interest to commercial computer manufacturers. Optical disks use a laser beam rather than a fluctuating magnetic head to write to the disk, creating a tiny and predictable deformity which can be read back when amplitude or phase shift changes are detected in a lower powered read beam scanning the surface.

A standard disk can store 55 megabytes – about 350 times the capacity of a single-sided single-density floppy disk. Dedicated computer peripherals based on the same technology measure their capacity in gigabytes. This enormous size goes a long way towards compensating for the optical disk's biggest snag – data, once written, cannot be changed. To make an optical disk system look like a read/write device means abandoning outdated sectors and copying new data to unused tracks. This process is repeated until the disk becomes full, whereupon the current data is copied to a new disk and the old disk is discarded.

Backup: The Safety Factor

A great deal goes into the creation of software, both in the commercial programs that you buy and the ones you write yourself. Electronically stored data too has often been gathered and sifted at the cost of a lot of time and effort. Software security is about making sure that all that effort isn't wasted, either through piracy, unauthorised access, or the accidental corruption of data.

Anti-piracy

Security in the first sense must be sophisticated to deter the determined pirate. Recognising this, some software houses don't bother with security at all, partly because it can prevent legitimate users from making the back-up copies that are such an important part of security in the second sense of the word. But recent estimates suggest that between #30m and #50m are being lost annually through software piracy in the UK alone, and the software house of Visicorp, originators of Visicalc, reckons that for every copy sold, an additional pirated copy is made.

Software on cassette is very difficult to secure against piracy, and generally companies make their money by producing the software cheaply and aiming for a large initial turnover. Products like this have a fairly short life time, so that by the time the pirates have got in on the act the company has moved on to something else.

However, cassette software producers use a number of safeguards to discourage prying. The most common method is to disable methods of breaking out of the program. This means the program has total control, preventing you from using other features in the machine's ROM which would enable you to find out the machine's memory contents. This is usually done by disabling the BREAK or ESCAPE keys, or by getting them to corrupt the program when they're pressed.

This technique is often used in conjunction with a loader program. When you type in `LOAD "name"` you call up a short program called a 'loader', the function of which is to pull the main program into memory and run it automatically. The loader will usually define the start memory address for the main program and this makes it indispensable, because without the loader the main program won't know where to locate itself. But because the loader automatically runs the main program it isn't easy to break into it to find out how it works.

Some software houses, especially those that are connected with the manufacturers, use calls to undocumented code in the ROM in order to obscure the logic of the program. The disadvantage is that undocumented code is very likely to be altered in subsequent revisions of the ROM (which manufacturers typically undertake something like four times in a machine's life span), leaving the software high and dry.

None of these measures makes a cassette program totally secure, but they do make life very difficult for the casual pirate.

At the business end of the micro software market, where programs cost hundreds rather than tens of pounds, protection from piracy becomes more important. Software of this kind is almost certainly disk-based, and this gives the opportunity for several ways of securing the information. Unsurprisingly, software companies are reluctant to disclose how they protect their software, but we can at least examine some principles.

One is to record the information on to the disk in an unusual way, bypassing the operating system. Operating systems usually come with utilities to copy disks. The copying program expects to find the information in a certain format: one thing it may look for is address headers which mark the beginning of a sector, and so tell the disk operating system its position on a track.

Your copying program will need to find the address header to make any sense of the disk, and because things like address headers are standard codes on disks, your program will know what to look for. Unless, of course, the software house changes the values. This method of protection is a popular and simple one, and is often used in combination with other methods.

Some disks use synchronised bytes. This is where an extra one or two bits are added to a series of bytes, to mark the start of a track or a piece of data. Changing these bytes confuses copy programs. A variation of this is to add bits to other parts of the program, which will also throw a copy program out of sync. The permutations of this method are almost endless, and vary for different makes of disks.

But 'clever formatting' only protects the software until the user buys (or borrows!) one of the sophisticated track/sector recognition programs like Locksmith, which automatically work out how the disk has been tweaked and apply the necessary correction. Companies selling these programs claim that they are not aimed at pirates, but at sensible users who want to make back-up copies for their own use.

Some systems can squeeze an extra track on to the disk after the last track. If the copy program doesn't know about this, it will stop after the last track, and miss the additional information. The first thing the protected software does when you run it is to see that this extra track has its information intact. There are also other ways of re-aligning the tracks to confuse the copier.

An alternative anti-piracy technique is to use the operating system in an undocumented way, but there are always dangers that the hardware will react unpredictably. Another way is to use a hardware device called a dongle. The dongle is a small gadget you plug into your computer, either as an additional PROM or at a spare printer interface. Software designed to be dongle-dependent will check for the presence of the device and read a security code from that location before proceeding.

Dongles are generally machine specific. The more sophisticated dongles will contain a ROM or a shift register, and actually enter into a dialogue with the protected software before allowing it to run – a sort of 'who goes there, friend or foe?' When you start the software rolling, the program sends a code to the dongle, which then sends back a message containing the serial number and any copyright details. If the message is the right one, the program will continue on its way - if not, a warning flashes up on the screen.

One professional approach, requiring cooperation from the hardware manufacturers, is to incorporate a unique

serial number into the ROM. When a particular software package is run on the machine for the first time it reads the serial number and displays a message asking the user to ring the software vendor, who then supplies the user with another number that ties the package into the processor.

But as fast as software manufacturers come up with new forms of protection, other people are devising ways of breaking and entering secure software. And the market for these ideas isn't composed of pirates, but the army of computer users, who resent being prevented from copying their own software for back-up.

The more sophisticated experienced programmers can often break codes themselves by examining the guts of the program. Dongle protection can be defeated by altering the code to jump over the checking routine, or by adding routines that simulate the presence of the dongle.

Those prying eyes

Protecting your own information in a system, so that other people cannot access it is becoming increasingly vital as multi-user systems spread. A multi-user system is where several users hook their terminals onto the same central processing unit. As with networking, this means that each user has access to the data held on the computer, sharing the files and programs.

The multi-user operating system, Unix, demonstrates what can be done to offer several levels of security. Each user has a unique password, any combination of about a dozen letters or figures, which is used to log in to the system. The password is held in an encrypted form in the machine and does not appear on the screen when typed in. After this first level of security comes a file specific level where each file in the system has an owner, who assigns a nine digit security code to the file. The code is divided into three groups of three digits. The first group defines what the owner of the file can read write or execute in the file. The second group defines which members of the owner's group can read, write or execute, and the third group of digits defines what anybody who taps away at the keyboard can do to the file.

For instance, the accounts manager of a company may own a file which has details of all employee's salaries, and will need to be able to read, write and execute all part of the file. Other people in the accounts department could read everything in the file, apart from details of their colleagues salaries, but would not be able to write to or execute any information. The rest of the company's employees could only access the file directory. They could not read the rest of the file, nor could they write to or execute any part of the file.

There is a facility in Unix to hide a file, so that it doesn't even appear on the directory. There is also an encryption algorithm, so that you can encrypt a whole file for extra security.

On a personal business computer or a home computer the simplest method of hiding information in a program is just to lock away your disks or tapes, in a secure box. A well-stocked computer shop will have secure boxes, which are not only protection against other people accessing your private data, but will also protect your disks and tapes form fire and flood.

Archives

Archiving and backing up describe the process of making off-line copies of files for security purposes. The term 'off-line' means that the file is not immediately available for machine use, but some simple process has to be gone through before bringing it back 'on-line'. In the traditional mainframe world this often meant going down to the basement to hunt for a large reel of tape, but with business systems it is usually a matter of digging out the appropriate floppy disk from some safe place and slotting it into the drive.

This makes backing up and archiving sound simple, but in reality it is a difficult art. Firstly there is a distinction to be made between the two terms. Archiving is the process of building up a reference library of past work done on the machine. You might go to such a library, for example, for a machine readable copy of the sales data for the first quarter of 1984. Backing up on the other hand is more concerned with preserving the full state of the machine, should the worst happen and a crash occur.

Archiving, then, is a continuous process of maintaining a library. Backing up is a guarantee against disaster. If they are to do their job both processes have to be approached methodically with two main objectives:

Organising the routines for backing up or archiving so that they are simple. If the process is complicated it won't get done.

Avoiding the creation of an overwhelming mass of duplicated material. Ideally you need two off-line copies of every vital file, but often one off-line copy will be enough.

These days mainframe machines usually have backup and archiving working mysteriously behind the scenes without interference from human operators. In a micro installation the ideal is to emulate this situation as closely as possible, which means devising end-of-the-day security routines that are simple to carry out, but do the job thoroughly. Sometimes this will involve using special software, but with care the ordinary utilities supplied with the operating system should be enough.

Archiving deals with products of the system, such as programs created on a software development machine, the articles written by a journalist, the financial data accumulated and processed by the accounts department, or whatever. The object is to transfer files that are no longer needed on-line into a library or libraries where they can be readily found if they have to be referred to again. Usually this will mean saving work of different kinds onto separate library disks, where each disk is suitably classified and labelled.

Depending on the kind of work, you may want to keep:

only the latest edited version of a file

several different versions of the same file

This second case is more complicated than the first.

Some operating systems – Unix for example – include exotic utilities for keeping track of multiple versions, but in most ordinary circumstances it is enough, to give each different version a distinct but related file name, and then treat each file as in case one.

To isolate the problem from the specifics of the hardware – Winchester disks, floppies, cassettes or whatever – it is only necessary to think of two separate areas where files will reside: the working area and the archive area. On a dual floppy machine the work area may be a series of floppies that are in constant use; on a hard disk system it will usually be part of the Winchester disk, but the principle will be the same.

It is rarely practical to archive each file as soon as it is finished with and, as work is done on the system, files will tend to accumulate in the working area. It is very easy to turn the work area into an unholy tangle of vital current files, unnecessary files that have been already archived, and 'junk' files which have served their purpose and ought to be deleted. The problem with cleaning up a work area like this is that an enormous amount of investigation is usually needed to discover which files are which. Similar remarks apply to the archive area, which readily becomes filled with files that have to be retained 'just in case.'

The archive routine should make sure the following tasks are carried out:

Remove junk files. This is made easier if junk files are always identified by name. Files only intended for temporary use should always be created with a name that reflects their status (eg `TEMPFILE.$$$` under CP/M). Software packages like WordStar create backup files as they go along and give them a distinctive name. At the end of the work session these will often be wiped, particularly where disk space is at a premium.

Copy all work files to the archive area. When running under an operating system that doesn't include automatic dating it's a good idea to keep a record of the transaction. The hard copy of the directory listing, or output from the copying program if – like CP/M's PIP – it produces a list of files transferred, could be slipped into the diskette jacket or tape pack with the date added by hand. A satisfactory alternative is to use the editor to create a `READ.ME` file in the archive area that records the date and details of the archiving.

Delete all work files no longer needed. This usually means all the completed work files. Files that still reflect work in progress will now exist in both the work area and the archive area under the same name – a potential cause of confusion. One way round this is to give work in progress files a distinct name (under CP/M the type extension `.WIP` would do the job), and rename them when they're finished. But this isn't essential, as most copy programs used for archiving will automatically delete the work in progress copy in the archive area when the finalised file is copied to it.

Backing Up

Backing up is concerned with preserving a working copy of all the files that constitute the system, including the operating system itself, its utilities, and all the application packages currently in use. Under this heading it is also usual to include any files used as databases, because although they contain data they are not end products of the system. Obviously the original distribution disks will have been kept in a safe place, but copies of these will have usually been configured specially for the system, a process it would be tedious to repeat.

Essentially these system files remain unchanged, but occasional modifications may be made, particularly to the databases, and the backup process should reflect this. Consequently it's usual to make an initial *global* backup of all the system files, and update this with periodic *incremental* backups of only those files that have changed or been added since the last backup.

Deciding which these files are can be the hardest part of backing up. The date-stamping feature of operating systems like MSDOS or CP/M+ is obviously very helpful, but it is also possible to exploit an under-documented feature of CP/M 2.2 called the 'archive bit'. This flag is actually part of the file name, the highest bit on the last character of the extent. Most CP/M implementations will always set this bit to 0 on creation or update of the file. The trick then is to use copying software that sets it to 1 whenever a file is backed up, and can also detect this bit and skip files where it is not 0.

Without either of these automatic aids, incremental backup can be rather tedious, though usually less so than having to make continual global backups. The simplest way is to follow a similar procedure to archiving, without of course deleting the work files once the backup has been made.

Archiving on cassette

Archiving and backing-up on a cassette system is not as scientific, nor as easy as on a disk system. Firstly, piracy protection techniques will normally prevent you from taking back-up copies of programs you've bought on cassette. However an intelligent precaution where two cassette recorders are available would be to make an audio copy of valuable programs. Be aware though that by opening the packaging on a proprietory cassette you are often entering into a licencing agreement to make no copies of any kind; the cassette inlay should explain the position exactly. In cases like this insist on a replacement from the shop if something goes wrong.

When developing your own programs there are a number of measures which must be observed. Before you start writing a program, check out your tape, save a short program and verify (or reload it if you haven't a verify function). When you're sure the tape's properly set up, you can begin writing your program. Have a rewound cassette in the recorder, set the counter to zero, and save after writing the first 50 or so lines. Continue to save at regular intervals, noting down each time the tape counter

setting, until you've finished the program.

Give each saved version of the program a different name. A good idea is to have a name followed by the number of the last line written (for example, **TEST50**, **TEST110**, **TEST160**) until you have reached the end of the first draft. At this point you should give it a name indicating that it's the first finished version, say, **TESTFINISHED1**. *Never run an unsaved program.* If it causes the system to crash, and you need to reset or pull the plug to recover, you will lose all your code. So after a programming session, save, verify and only then run the program.

Now you have a tape which will document the development of your program from start to finish. With one finished version on tape, flip over the cassette, rewind it, set the counter and save and verify it. As you develop the completed program, save every time you make radical revisions – again, giving each version a different and meaningful name. When you have finished, save and verify the program at least twice, preferably on separate tapes. Don't put finished programs on the same tape as the development programs, and don't put different programs at different stages of development on the same tape. Always clearly label your cassettes, and keep a book which shows what's where on which tape. It's also good discipline to datestamp all versions of your programs, both with a **REM** at the beginning, and in your cassette log.

Baud Rate: Speeding Bits Down The Wire

The term baud (pronounce it *bode*, though you'll sometimes hear it as *bored*) immortalises the name of Monsieur J.E.M. Baudot, inventor in 1880 of the Baudot code, which by the 1950s had become the main standard for international telegraph communications (*see* ASCII).

In the field of microcomputing, baud rate can be thought of roughly as the transmission speed of serial data, in bits per second (with complications . . .).

A stream of bits can be sent down a wire as a voltage switching between high and low to represent the *ON*s and *OFF*s. But digital transmission of this kind is suited only to short distances; in wires longer than about 500 feet, capacitance and resistance effects rapidly begin to corrupt data.

However, the telephone network, which carries analogue signals (more or less continuous signals of varying magnitude), can be pressed into use as a data carrier over large distances by converting the data at each end of the transmission line. Although received and sent in digital form, data transmitted in this scheme rides on the line as an analogue signal. The converter at each end of the line that makes this possible is called a modem (*qv*) – it stands for **MO**dulator/**DEM**odulator. Figure 1 shows a modem at work.

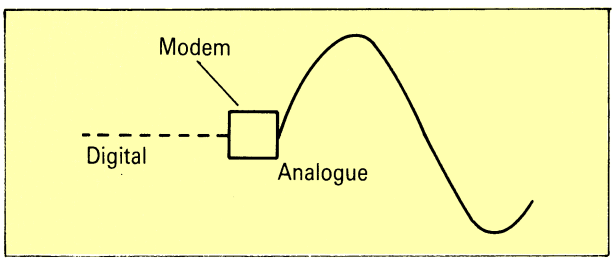


Fig. 1. A modem at work

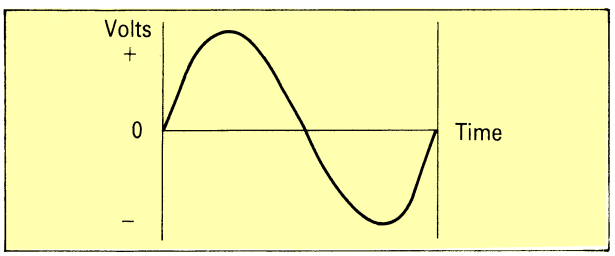


Fig. 2. A sine wave representing an analogue signal

Bits and waves

In transmission, a bit has to be represented by a change of some kind. Analogue transmission gives the opportunity to introduce changes into the amplitude, frequency or phase of the signal. Imagine an analogue signal as a wave on a graph, with the vertical axis representing amplitude of the signal, and the horizontal axis depicting time. You can then chart the change of a signal over a particular period. For example, in a milli-second the signal might start at zero voltage, peak to one volt, fall back to zero volts and then continue falling to minus one volt before returning to zero volts again. On our graph (figure 2) we reproduce this as a sine wave. In all probability the wave will continue beyond this one milli-second slice by repeating the same changes over and over again. Our slice represents one complete cycle.

A simple transmission system might distinguish between an *ON*-bit and and *OFF*-bit by two different levels of **amplitude**. In figure 3 we demonstrate this with a signal that peaks at two volts for *ON* and one volt for *OFF*, but the principle would be the same for any two distinguishable voltage peaks.

Alternatively, the bits might be distinguished by changes in **frequency**. For example, *ON* could be represented by one complete sine wave cycle every millisecond, and *OFF* by two cycles every millisecond, as in figure 4.

The third approach, varying the **phase**, is less easy to explain. Keeping the amplitude and frequency constant,

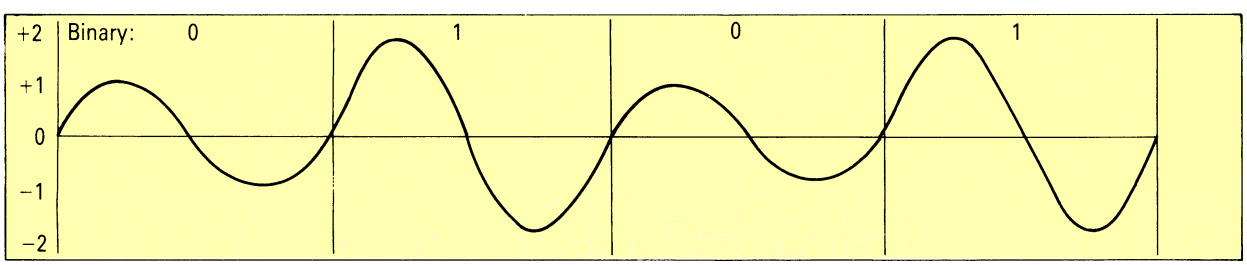


Fig. 3. Two level amplitude modulation

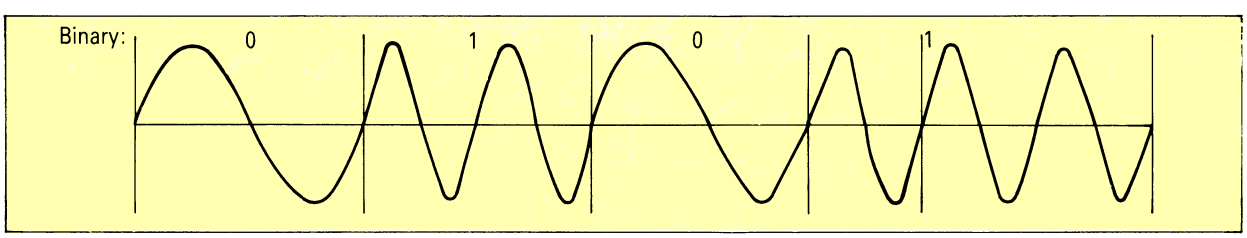


Fig. 4. Frequency modulation

the signal can be shifted backwards or forwards along the time axis, the degree of shift being used to represent the *ON*ness of *OFF*ness of each transmitted bit.

In discussing this kind of change, a complete cycle of a sine wave is thought of as passing through 360 degrees, as if it were a rotating wheel. (For the mathematically minded, the projection of one point on the circumference of a rotating circle onto a surface moving along the time axis actually defines a sine wave.) The zero voltage at the beginning of the cycle then becomes 0°. The peak is 90°, the second zero voltage is taken as 180°, the trough is 270° and the last zero as the cycle is completed makes the full 360°.

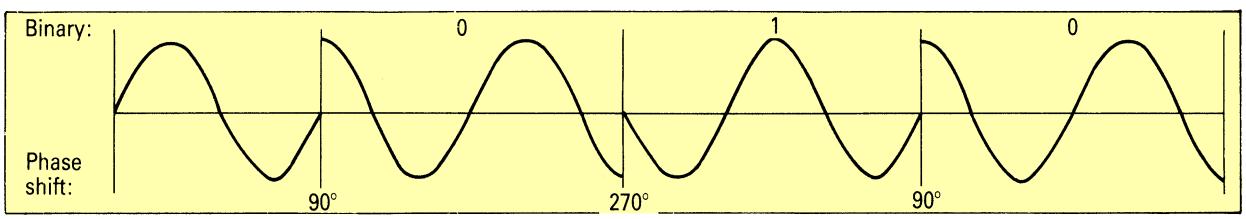


Fig. 5. Phase modulation

Phase, or position in the cycle measured in degrees, is varied to represent bits by shifting the signal a preordained number of degrees either forward or backward (270° forward for *ON* in figure 5 for example, and 90° forward for *OFF*).

We said earlier that baud rate is more or less the same as the bit rate, and this is certainly true for straightforward data transmission along a short wire. But strictly speaking the baud rate is a measure of the speed at which the transmission line is modulated, and can be applied to any of the above means of modulation. So a 300-baud signal that's changed by varying amplitude will involve 300 changes in amplitude per second. The same applies to phase and frequency changes.

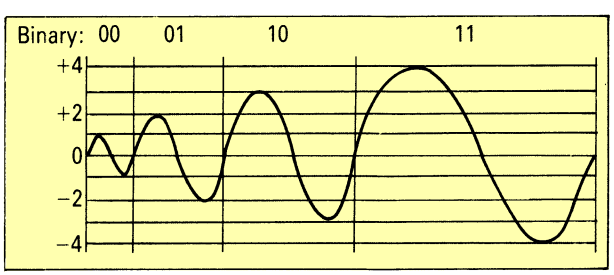


Fig. 6. Four level amplitude modulation

Multi-level modulation

So far we've assumed only two levels of modulation, one volt or two volts for the amplitude, one or two cycles per millisecond for the frequency, and 90° or 270° shifting for the phase. In each of these cases, the number of bits per second the line can carry approximates to the baud rate. One modulation represents one transmitted bit (and the system may add some extra bits as we shall see). But this position changes radically if the modulation is multi-level.

The simple amplitude modulation represented in figure 3 could be made to carry more information if it had four levels rather than two. The four levels could be five, four, three, two and one volts respectively (figure 6).

With a four-level modulation, four unique states can be represented. Instead of one bit being sent down the line at a time, two can be sent, as there are four possible two-bit numbers (00, 01, 11, 10). Four volts could represent **00**, three volts **01**, two volts **11** and one volt **10**. Under these circumstances the baud rate does not equal the bit rate: the bit rate will be twice the baud rate because one change in the modulation represents two bits.

The signal could carry even more information by increasing the number of levels. This can be done by mixing different methods of modulation, most commonly amplitude and phase. The more levels of modulation, the higher the bit rate will be in comparison with the baud rate.

Timing and checking the transmission

The extra bits added by the transmission system introduce another complication into the simple equivalence between baud rate and bit rate. In **asynchronous data transmission** extra bits are added at the beginning and end of each character, forming what is called the 'frame'. A further bit may also be used as a check against against data corruption.

Marking the beginning and end of each character with start bits and stop bits serves as an alternative to a clock signal to show precisely when data is to be taken to be valid. There is usually a single start bit, and one or more stop bits. Confusingly there can even be one and a half stop bits – a fact that serves to remind us that the 'bits' aren't really part of a bit pattern at all, just periods of known signal state (usually low to start and high to stop) which the receiver monitors to stay in synch with the transmitter.

The data-corruption check, called the **parity bit**, is sometimes omitted. If included, parity bits are either even or odd; in either case the principle is the same. With even parity, an extra bit is set (made 1) or reset (made 0) in order to make up the total number of set bits to an even number. For example, the letter **g** is represented in ASCII by the seven-bit pattern **1100111**. If transmitted with even parity an extra bit would be added to make the total number of set bits to 6, an even number. Odd parity applies the same rules to arrive at an odd number of set bits.

Bulletin Boards: The Micro Connection

Everyone with a microcomputer, or even a terminal to a remote computer, already possesses most of the equipment needed to communicate with other computers. With the development of dedicated modem chips, the cost of the remaining equipment has now started to drop to the point where even the hobbyist can give it serious consideration.

All you need is a computer, some simple software to make it emulate a terminal, and a matching modem – for most machines the standard RS-232 connection will form the interface. Micro users in the United States have already discovered this in a big way, tapping into literally hundreds of hobbyist and commercial networks.

The Source

Of those probably the largest, and certainly the best-known, is The Source. It will be a number of years before the image of the massive computer in the sky, overflowing with data on everything under the sun becomes a reality, but The Source is probably the closest thing to it. It is owned by Readers Digest and runs on a fleet of mainframes at a US-based bureau. It was based onset the concept that there is a massive amount of useful information sitting out there on computers; all it needs is to be linked together and made available and affordable to anyone with a microcomputer and a modem.

The Source has the entire United Press news database, together with investment support services, a whole mass of analytical data, business research, consumer services, home repair advice, health care information, and a shopping system with 30,000 brand name items at up to 40% discount prices. As well as teletext-style static information which the user can access but not change, The Source incorporates an electronic bulletin board system, through which users can communicate with each other and set up special-interest databases. Users can put up messages for each other, and scan across all the 80 boards by key words to track down messages for them personally or of particular interest to them. It also encourages the formation of new networks, and provides support for the development of new specialist databases. Two thousand new users join every month.

The cut-price night-time rates make The Source a better proposition for the British computer owner than might at first appear likely, since they coincide with peak business hours UK time. That advantage helps to offset the daunting cost of the telephone call; the disadvantage is that you will need a special modem that conforms to the US Bell communications standard, as well as to our own CCITT. Alternatively you could use a networking service like British Telecom's PSS service (*see* below).

In fact The Source claims over 1,000 overseas users. But the company has no plans to set up on this side of the Atlantic, so would-be British subscribers must choose between expensive telephone call charges, or an expensive subscription fee to join British Telecom's packet switching service to cut the cost of the long-distance calls.

Bulletin boards

The bulletin board movement is still new to the UK, but is extensive enough to offer a stock of 30 to 40 programs available for anyone who wishes to download onto their own computer. In many cases there is no charge apart from the cost of the telephone call. Hobbyists are free to leave or pick up messages from fellow-enthusiasts.

For the newcomer to networking, it comes as a pleasant surprise to discover how uncomplicated it is to log on to and use a bulletin board system. You dial up the bulletin board system number, and the system answers the phone automatically. Then you simply place the telephone handset in your acoustic coupler, hit the enter button, and a welcome message comes up on your screen.

After entering a password, or identifying yourself in some other way, a menu appears offering a number of options. You can enter or retrieve messages, and if a message is waiting for you there will be a message on the screen to let you know. Other options are to upload or download software (non-proprietory, of course), so that you can exchange games and utilities you have written yourself. One typical system, Forum-80, prompts the user through the system step by step, offering help when necessary.

BT and packets

British Telecom's Packet Switching Service (PSS) is a network specifically designed to carry computer data. It tends to be too expensive to be used for domestic calls; you may as well use the normal telephone network. For international calls, especially to the US, PSS is relatively cheap for those who use it regularly, even allowing for the annual subscription charge.

One of the main economic advantages is that you are charged on the number of 'packets' of data sent or received, rather than the time spent on line. However, bear in mind that you also have to pay the phone charges for connecting up to the nearest PSS 'node'. If it is more than a local phone call away, which is quite likely if you live outside one of the major cities, this could add substantially to the cost.

Using PSS is simple. You ring up a local node and, with the micro connected up through a normal modem, type in an identity number and the password supplied when you paid the subscription. This logs you onto the PSS network. You then supply a string of characters and numbers which make up a 'network address', and are routed through to any computer network that is connected up to the PSS system – including all the major American public networks – and in turn to any computer available on that network. Once you are connected, the improved reliability of the line is the only reminder that you are not using the normal telephone network.

Prestel

Prestel is a 24-hour service owned by British Telecom,

offering two way data communciation down the telephone line. It draws its data from hundreds of 'information providers', who book time on the system as a commercial venture. As a consequence, Prestel has many more pages than Ceefax or Oracle (see below), but there is a charge for accessing the information. Apart from rental charges for the installation and the local call charges to the nearest Prestel computer, a charge for each page viewed is displayed at the top of the screen. For 90% of the pages this charge is 0p, ie. free! But many of these pages are directories leading onward towards valuable information, which does incur a charge.

There are many thousands of pages of software on Prestel, amounting to hundreds of programs, mainly games written in BASIC. To run these on your micro you will need a special Prestel adaptor – there's one available for most machines with an RS232 interface. The necessary Prestel communications software comes with the adaptor. Your system will need to be able to receive data at 1200 baud and send at 75 baud, the Prestel standard.

With your adaptor you will also be able to access the other information on Prestel and use its electronic mail service, 'Mailbox'. Through this you can send a message to any other Prestel user, either as a pre-formatted greetings card or as your own message of up to 100 words. Next time the recipient connects into Prestel a notice will tell him that a message is waiting.

There is virtually no limit to the number of messages you can have waiting for you and once you have read them you can store up to three at a time. There is a directory of Mailbox users on Prestel, but be warned, if you decide to be listed, you are open to junk mail from advertisers.

Apart from access to the usual Prestel pages and Mailbox, you can also become a member of a closed user group (CUG), such as Micronet. A CUG is a group of people who have access to private pages on Prestel. Tens of thousands of private pages make up Micronet, many of them pages of programs for education, games and business. Some of these are free, others you will have to pay for. The remainder of the pages are taken up with news, a buyers guide, classified and display advertising, and also services for user groups. Apart from lists of clubs and events, there are also bulletin boards for user groups.

As a Micronet user you have access, through 'Gateway', to other databases on Prestel. Gateway, is an extension to Prestel that allows users to access private computers through the public Prestel service, and indulge in a little teleliving. You may have read about 'teleshopping' and 'telebanking'. The idea is that through Prestel you can order goods and services much as you would over a phone, or by post through a mail order catalogue.

In Germany, the public viewdata system has been used by a bank to give its customers their own home banking terminal where they can access details of their accounts, pay bills, ask for loans and order cash from their own homes. This is now beginning to happen in Britain. The first such service, called HOMELINK, was introduced in 1983 by the Nottingham Building Society in conjunction with the Bank of Scotland.

British Telecom's Prestel operates at different speeds from the ordinary modem connections used by the bulletin board community. Unless you buy a sophisticated – and slightly more expensive – dual system device, simple modems for one system will not be able to communicate with the other. Prestel not only uses different baud rates, but its own set of character and graphics codes, which means you need special software and display hardware, as well as the facility to communicate at 1200/75 baud.

Other services, for example British Telecom's Gold, can be accessed with an ordinary modem driven by general purpose comunications software. Gold uses the same sort of system as The Source in America, and allows access to electronic mail, a bulletin board, computer services and databases.

With a Source-style home services network system beginning to appear in the UK at last, the incentive to invest in a modem will be far greater than it has been up to now. Together with the recent arrival of dedicated modem chips which have dramatically reduced production costs, this new mass market is creating modems within reach of everyone with a micro.

Ceefax and Oracle

The communications techniques we've mentioned so far all use the telephone system. An alternative approach is offered by Teletext, an information service transmitted by 'borrowing' a few extra lines from the ordinary television broadcasts. The BBC call their teletext service Ceefax, while the service shared between ITV and Channel 4 is known as Oracle.

Because these services are transmitted by the ordinary television signal they are only available during TV broadcasting time (including TV test card transmissions). To those with sets able to decode the signals both services are free – or rather, are paid for by licence fees, in the case of the BBC, and by advertisers in the case of ITV. Many TV sets are built with a teletext capabilities, but for those that aren't it is possible to buy an adaptor.

Both services provide several hundred 'pages' of news, weather, sports, business information, jokes and games, all regularly updated.

The drawback of teletext is that it is a one way system – you can only receive signals, not transmit them back. Nevertheless, the system does hold interesting possibilities for micro users, now that you can buy a telesoftware adaptor to turn your micro into teletext set. And there's a bonus: you can download telesoftware directly.

Buying: Your Next Micro

Buying computers tends to revolve around issues like memory size, the number of colours available in the graphics, the quality of the sound chip and so on. At the home computer level, what the advertisers call 'brand loyalty' can often generate fanaticism for certain products and technologies, with users holding their own machines in quite irrational reverence and equally irrationally decrying other makes.

The choice of hardware is an important factor – but it is not the only one. A manufacturer might produce a machine at a giveaway price that offers hi-res colour graphics, a super-fast microprocessor, a sound synthesizer, a variety of ports, a proper keyboard, all the memory you could hope for, disk drives . . . You may find it hard to stop yourself from buying it – but what about the software? That often-overlooked invisible commodity should be every bit as crucial to your choice of micro as the hardware.

Making up your mind

To make the right decisions about buying a micro you have to know what you want your machine to do. Microcomputers are built to perform a single, rather vaguely defined function – to compute. The precise definition of what the machine does is supplied by the software. But micros are restricted in the types of program they can run – a limitation that brings us back to the hardware. For example, application software designed to turn your machine into a word processor usually requires a proper keyboard and a facility to connect up the sort of printer you'll need. A program that involves a lot of speedy complicated calculations will probably require a system with a fast processor and a lot of internal memory.

So the easiest way of telling what the hardware is capable of is by examining the software that runs on it. The old division between 'business' and 'home' software has now been eroded, and manufacturers of micros and software producers have become a lot more be more ambitious in the range of cheaper machines, offering 'business' software which appears to indicate that they are good for anything. Readers of this book will be canny enough to know that this can be misleading. Although these machines might claim to run electronic spreadsheets, they are not comparable with most business systems, basically because of their limited internal memory and absence of floppy disks. Many smaller systems are expandable, but the upgrade path can be expensive. If you know from the start that you are going to need floppy disks, then it may be cheaper to buy a system that incorporates them in its basic design.

The best way to begin your buying campaign is by setting a realistic budget of the most you are prepared to pay. It's helpful to set a minimum price too, but keep this very flexible as it isn't always safe to assume that a cheaper machine has less to offer. Next you have to do some soul-searching, pondering on precisely *why* you are buying a micro. Write out a list of reasons. If you haven't included 'just messing about' you probably aren't being honest with yourself. Tear up the list and start again, this time envisaging how much time you will be spending on each aspect. Word processing may be at the top of your list, but if most of your session time is going to be taken up with Pacman, then 'playing games' should actually be at the top of your list.

Find out which software will be most likely to satisfy the requirements you've listed. Then check off the machines in your price range offering the hardware and software to do the job – and be prepared to compromise.

You should now have a shortlist of machines and of the software you want to run on them. The next stage of your campaign is to investigate suppliers. In the best of all possible worlds, you'll be buying from a local friendly microcomputer dealer, but if that isn't possible, you may have to settle for a shop that happens to sell micros along with its hi-fi equipment, confectionery or whatever. As a last resort settle for mail order, but only if you've had a chance to test drive a machine of the same make. Wherever possible the rule is not to take out your chequebook until you've had some hands-on experience of what you're buying.

The right micro for the job

Here are some pointers to look for if you have specific applications in mind:

Programming can be learnt on just about any machine, but only up to a point. A BASIC (*qv*) which can handle structures like `REPEAT...UNTIL` will broaden your outlook, and if it can understand defined functions and procedures, so much the better. Investigate what other languages are available – is there a good assembler, PASCAL, LOGO and so on?

Education in a wider sense – learning foreign languages, mathematics or whatever – will be a matter of available software packages. All machines make some gesture in this direction, but the most interesting software tends to be written for (expensive) micros supporting hi-res graphics and interaction by way of analogue devices such as a mouse, a joystick or a touch-sensitive screen. The entry in this book on Education should give you some guidance.

Games need similar hardware support – good colour graphics with high resolution, sockets for paddles and joysticks, and so forth. But a great deal more ingenuity has been spent by resourceful software designers in overcoming the hardware limitations of the smaller and cheaper machines. The games market is fast moving, creating rapid obsolescence and if the games scene is your main consideration it might be worth buying into it as cheaply as possible. (*See* the entry on Games).

Control and monitoring of processes around the home – such as the central heating system – will depend on analogue-to-digital converters, so it is very useful to go for a machine that has them built in. The presence of ports for joysticks and paddles is

sometimes an indication of internal A/D converters.
Communications will almost certainly depend on your machine having at least one RS-232 interface, and the more the merrier. You'll also need a modem (*qv*) and the appropriate communications software package. Some machines come with integral modems, but make sure that whatever you get is suitable for use in this country – US and UK standards are not the same.
Word processing in all but its most elementary aspects almost certainly depends on floppy disks, or at least the facility for connecting them up when you can afford them. You will also need a decent keyboard with a proper QWERTY arrangement – the so-called touch sensitive or rubber-capped type will drive you up the wall in no time. Programmable function keys can be useful in taming an otherwise unwieldy word processing package. The provision of a parallel port will save you some frustration and money when it comes to connecting a printer, but an RS232 port just might allow you a wider choice, particularly if you are going for a heavy business-type printer.

Your cardboard guarantee

Basically, a guarantee is a bit of cardboard with writing on it that sets out what a customer can expect by way of redress from a manufacturer when something goes wrong with an item. Normally it will specify something suggesting that if you return the item to the supplier within a given period of time, it will be repaired or replaced for you free of charge for labour, parts or carriage.

There are three relevant pieces of legislation in the UK: the Supply of Goods (Implied Terms) Act 1973, the Unfair Contract Terms Act 1977, and the Sale of Goods Act 1979. Read together, their effect – as far as 'consumer transactions' are concerned – is negative. They lay down what manufacturers cannot do. In particular manufacturers can't use guarantees to deprive customers of their 'legal rights'. You have probably noticed that most guarantees now incorporate a phrase stating: 'This guarantee is in addition to your statutory rights' – this isn't generosity, it's the law.

Before the 1973 Act, many unscrupulous manufacturers used to try to deprive their customers of these rights with guarantees (which have the same effect in law as a contract), the terms of which more or less implied that if you were unfortunate enough to buy a device that turned out to be a defective heap of junk, that was just your plain hard luck.

Thanks to changes in the law you can now no longer be ripped off like this when you complete a guarantee card. But that really is about as far as the law goes. Guarantees do not give you any extra rights. For example, there are no Office of Fair Trading regulations governing the wording of guarantees or the length of the time for which they must run.

And even all these 'statutory rights', which can no longer be forfeited by signing a guarantee, are in fact not much more than rather vague principles. The goods must be fit for their usual use; with no dangerous loose electrical connections; of a proper quality (not scratched or faulty when you take them out of their box); and must be as described (your 48K of RAM really must be RAM and not 32K sitting on top of 16K of ROM).

Electrical domestic appliances like kettles, refrigerators and blenders are supplied with 12 month guarantees, not because of any positive intervention by the Government or the law, but merely due to market forces. To remain competitive manufacturers must offer terms as least as favourable as those offered by their trade rivals.

But with microcomputers, where there are so often shortages of suppliers and hold-ups on deliveries, it is still very much a 'seller's market', and that often means 'take-it-or-leave-it' when it comes to the guarantees. So although for many types of goods a 12-month guarantee period is now the norm, don't expect it the microcomputer market.

When you buy a micro then, the old legal principle of *caveat emptor – let the buyer beware* – still holds true. So, while you are studying technical specifications and all the other factors you take into account before deciding to make a purchase, take time out to check the guarantee terms you are being offered as well.

Some machines are offered on longer terms, but a 90-day guarantee is often the most you can expect.

The second user option

To avoid any uncomfortable comparisons with second-hand cars, mainframe computers returning to the marketplace to seek new owners tend to be called 'second user systems', particularly in the higher price range where depreciation is rapid and old-hat hardware needs all the help it can get. This is a market where the sellers tend to call themselves 'vendors'.

Micro owners are more relaxed about these things. You may feel that 'second hand' is a perfectly adequate, not to say apt description of that much-prodded keyboard with the lettering wearing away – a machine which, with much regret, you now need to sell to make way for the new wonder of technology on which you have set your heart.

But pause to think, if you can just take your eyes off the glossy brochure for a moment. Upgrading to a more powerful machine doesn't necessarily mean buying something newer. You are happy to sell your machine at a knock-down price – so perhaps there is somebody else out there with a second-hand bargain for *you*.

Buying a micro secondhand is not something the complete beginner should take lightly, but if it's your second micro you will probably know enough to proceed wisely. In particular you know:

What your interests are, and how they are likely to develop.
How those interests need to be supported by hardware and software.
The range and prices on offer in the current computer market.

First time buyers are more vulnerable, but armed with the wisdom in this book and any advice they can muster from fellow-enthusiasts and magazines there's no real reason why they shouldn't also buy secondhand successfully. With technological developments striding forwards faster than most people can keep up, and perfectly good used micros going at half list price, there are some terrific bargains to be had. There's a saying in the industry that 'if it works it's out of date'. If you don't mind being a touch out of date, and you do want something powerful that works and will go on working for a long time to come – the second hand market is not to be shunned.

However, despite the mainframe vendors, buying a used micro should be approached like buying a used car. Take along a checklist, and if possible a knowledgeable friend, and above all don't be afraid to ask for a thorough test.

Unless you are a complete novice with a set budget looking for the best deal at a fixed price, you'll want to narrow the field down before you start the search. All the remarks relating to new micros also apply here. Decide whether you want disk drives or cassettes, portability or a large screen, and if you'd prefer a typewriter style keyboard. You'll also need to assemble your facts about which models have a good range of readily available software and add-ons. Reviews in magazines like 'Computer Answers' will help you sort the sheep from the goats.

Draw up a shortlist. The better known names tend to be well supported and will command the highest prices. Obscure machines, home-built micros and machines from defunct companies are worth little, so be ruthless with your offer if you choose to take the risk. Always try to buy a micro with its manual – no proper documentation means hours and hours and hours of your time trying to sort out the simplest problems, so don't forget to cost that in. Make sure too that there are the necessary mains leads, transformer and connecting cables, and if there aren't then lower your price by the cost of replacing them. These things can be bought, but it will cost you time and money to track them down.

Once you've located a likely candidate, ask if the machine in question is still under guarantee, since most manufacturers will overlook a change of ownership and honour the warranty. Buying from a dealer has the advantage that you can take back a faulty machine and get it repaired or even replaced. But don't omit the hidden cost of travelling to and from the seller's address, since it will cost you the same again to return a faulty machine.

Don't forget to give the micro a careful look-over before buying. The general condition can tell you a lot – is it battered or covered in dust? This is a good point to open an informal conversation about why the machine is being sold, being ready to probe deeper if you are not convinced. Ensure that the machine is being sold with its faults openly declared, and make your regrets and leave if these faults sound serious – there are plenty of good micros on sale.

The first thing that experienced second-hand buyers look at are the screws holding the casing together. Burred screw heads could well indicate careless tampering by the owner, so be on your guard. Give the micro a discreet shake to see if there are any loose components inside, or better still, try to get the owner's permission to open up the micro to see if anything is missing. Watch out for damaged or bent pins on the interfaces and check that the mains cable and any connecting leads are in good condition.

Always see the micro working, and try to run at least three programs, since this will give enough time for problems such as overheating to show up. While the micro is running, give the casing a gentle tap; loose connections and 'dry joints' where the solder hasn't fused properly will probably show up as flickers on the screen.

It is useful to check that it performs simple mathematics correctly, that it runs a short BASIC program successfully, and that all the keys (especially rarely used ones) function properly – generally it isn't easy to fix a faulty key. Manufacturers often supply a diagnostic program to test this sort of thing.

Often the seller will produce extra peripherals and software, and negotiate to sell them as well. Generally these things hold their price and are worth having, because the new price of peripherals such as disk drives is higher than most micros, and because they are usually suitable for more than just a single model. But beware of software that only runs on a single model – it can be literally valueless.

As long as the keyboard and interfaces are still functioning, a microcomputer will in theory last forever. But extra care has to be exercised when buying secondhand peripherals. The screen tubes of secondhand or reconditioned monitors can be beginning to show something of their limited lives, and disk drives are prone to electromechanical faults, and have more parts to can wear out than does a micro. With printers and typewriters the number of possible faults is even higher.

If you enjoy the art of skilful buying (some quite worthwhile people don't!) you might try either:

> Settling on a price for the micro and buying the extras as an afterthought – bearing in mind that they're not much use to the seller without the micro. Driving a hard bargain for the micro as a condition for taking the extras off the seller's hands.

The price. Private sellers tend to be very optimistic in the prices they ask for used machines, so either expect success in knocking them down, or go away and come back a week later when the seller has tried the price on a few more buyers. Don't forget you should be working on a percentage of the current list price, not on what the owner paid for it new. It is particularly worth checking if the micro you are chasing (or one very like it) is being heavily discounted by a dealer. If so, take along the advertisement or catalogue, and ask the seller's advice about whether you would be better off buying new from the dealer. The 'invisible hand' that the monetarist economists have such faith in may well intervene at this point to tip the selling price a little more in your favour.

CAD: Computer Aided Design

Computer-aided design is usually associated with the design of electronic circuitry and machine tools or, on a larger scale, cars, aeroplanes and buildings. To the mechanical engineer, CAD is often a draughting system that produces detailed sectional or assembly drawings. In the construction industry CAD is used for architectural design, heating and ventilation systems, structural engineering, and finite element stress analysis.

Until a few years ago CAD's requirement for high resolution, large memory and great computational power restricted it to mainframe or minicomputers, supported by expensive plotters, monitors and disks. However, today many professional systems are based around micros, and as prices fall and low-cost hardware becomes more sophisticated, it won't be long before home computers get a look in on this fascinating combination of graphics and calculation.

Professional CAD

Most draughting systems allow you to create your own symbols, shapes and parts and to store them in a disk library, to be extracted again as necessary when putting together a design. They will also provide automatic shading as well as rotation and reflection of shapes. An assembly drawing consisting solely of standard parts, and on which the parts list must be shown, would be an ideal application; on the other hand a small company producing one-off sheet metal work could find a CAD system more time-consuming than a drawing board.

The essential items are the micro, disk, keyboards, and plotter. For a few applications, an A3 size plotter would suffice, but most real engineering drawings need at least an A1 plotter, able to handle paper eight times the standard A4 size. Interestingly the one ingredient you might think essential – sophisticated graphics – is not regarded as very important, and some draughting systems do not need a graphics screen at all. It is the quality of the drawing coming out of the plotter which is important.

Some of the standard engineering CAD software is too large to go on a micro – programs with 30,000 to 100,000 lines of FORTRAN are commonplace – but you can often access these programs through a computer bureau using a terminal.

CAD among the artists

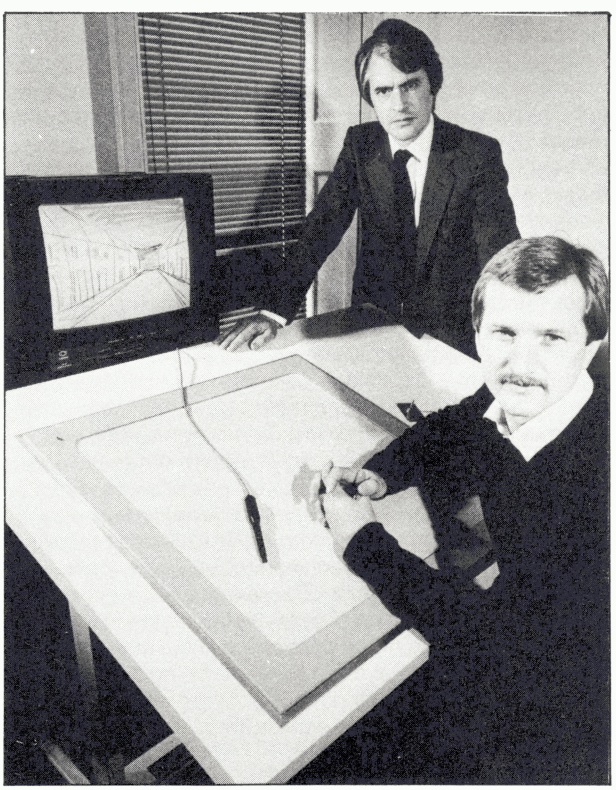
Painters, graphic designers and artists of all kinds have discovered CAD too. With the arrival of micros, the drop in price has coaxed several design studios to experiment with computers, triggering a corresponding growth in artistic CAD packages, some are even cheap enough to tempt the well-heeled home computer user.

Computers can save the graphic artist much of the more tedious work, such as drawing precise lines and filling in large areas of a picture. CAD gives the artist the luxury of changing colours on a whim, and of being able to move graphics around the screen. With good equipment artists can manipulate real photographs on the screen and record their pictures directly onto video or film. This offers enormous opportunities to animators, and some of the most extensive uses of computer-aided graphic design have been in the production of company videos and in the television industry.

The increased use of graphics on television has acted as a catalyst for the exploitation of computers in design. One of the best known computer art systems was developed by the BBC research department in conjunction with its graphic designers. It's called FLAIR and is now marketed under licence from the BBC by Logica. FLAIR costs about the same amount as a medium sized house, so that there are – understandably – only about a dozen systems in the country.

You might think that behind such a sophisticated and expensive system there must lie at least a minicomputer. In fact FLAIR uses a humble 8-bit micro – an Intel 8085A CPU with 48K RAM and 16K ROM, together with half a megabyte of display memory. Logica tends to think of FLAIR as a graphics system that just happens to have a micro in it. The micro is the cheap part of the system.

In 1982 FLAIR began to experience competition from the manufacturers of low-cost graphics boards which could be added on to ordinary micros. PLUTO, developed by a company called I/O Research, formed by two ex-Logica employees, was the first high quality graphics add-on to reach the enthusiast micro price market. The basic PLUTO board is a hundred times cheaper than the



Quantel's Paintbox system uses a graphics tablet to produce freehand images.

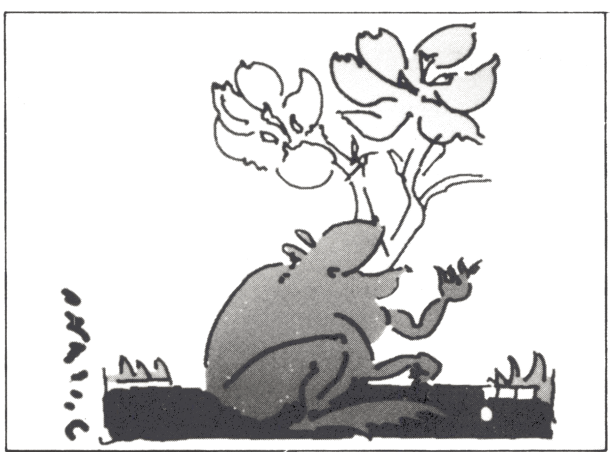
FLAIR system, and can be adapted to almost any computer you care to think of. For your money you get a pseudo 16-bit 8088 processor commanding enough memory space to hold more than two screens, so that you can draw on one screen while the other is being displayed. The board gives a resolution of 640 pixels by 288, in a choice of eight colours.

Of course just adding a board like this to your micro won't give you anything like the full capacity of FLAIR, either in terms of hardware or software. Because the PLUTO board has been sold mainly to manufacturers, hobbyists and professional TV design programmers, it doesn't come with much built in graphics software – users tend to write what they need for their own particular application. But a FLAIR look-alike package called GRAFITAS *is* available commercially for a few hundred pounds.

A combination like the GRAFITAS package and PLUTO will need a good colour monitor, a high quality graphics tablet, and a disk system to store the pictures. Typically you will be offered a grid of options around the border of the graphics tablet or screen, and you choose from them by moving the graphics stylus, or a mouse or – on touch sensitive screens – your finger. Options available include colours, standard shapes for circles, arcs, rectangles and so on, and a paint brush which lets you create you own shape and size of drawing implement. So, for instance, the line that appears on the screen can look as if it was drawn using an ink pen, a paint brush, or a piece of wood.

For more realistic image processing you might turn to a system called PAINTBOX, produced by Quantel. PAINTBOX is a good deal more expensive than FLAIR and is really a fine artist's or TV producer's tool, rather than a graphic designers sketchpad. Using the input from a video camera, you get an exact colour picture on the screen, you can then alter, this using effects looking like airbrush work, charcoal, or water colour. You can, for instance, draw a face in what looks like pen and ink, and then cover it with a water colour wash.

FLAIR and PAINTBOX, and to a lesser extent the combination of PLUTO and GRAFITAS, are expensive professional systems. But they are not entirely out of reach of everybody. At present these systems affect all of us in daily life: they are used in the production of the things we watch on television, the design of the towns and houses we live in, even the shoes and clothes we wear and the packaging they're sold in. They also have a more direct relevance for home micro users. The pioneers in computer-aided design set standards for micros, and as hardware prices continue to fall, home computer users are waiting to reap the benefits which time will certainly bring.



Flair graphics

Cassettes: Saving Serially

Loading a program from cassette can be a frustrating experience – but one which can be avoided if you know how to. Experienced cassette users know what to look for in a recorder, and are armed with a few tricks for dealing with difficult tapes.

Choosing a machine

When buying a cassette recorder it's worth making sure the dealer is prepared to exchange if it doesn't match your micro. If it is sold as an audio item, and works as such, you could be on tricky ground if it turns out to be unsuitable only for computer use. If you are buying from a computer dealer, ask for a demonstration of the cassette recorder linked to your particular micro.

The good news is that the best recorder for your micro may be the cheapest. Expensive cassette machines have a number of features that are either irrelevant in a data recorder, or even downright undesirable. In an expensive recorder you pay for the following:

Minimal wow and flutter – As their names imply, wow is the slow fluctuation of speed that affects pitch, and flutter is the rapid jerking of the tape transport system that produces an unpleasant and unwelcome tremolo effect. Both are at their most noticeable when replaying piano music. Though having a crucial contribution to whether music sounds realistic or just plain 'cassette-like', wow and flutter within the limits of the cheapest machine make very little difference to the integrity of your data.

Stereo – This adds a life-like 'third dimension' to music by recording slightly different sounds on two different channels, mimicking the way our ears work. The effect is not helpful when recording data, which only needs a single channel, and may actually be deleterious, as a pair of stereo heads scan less of the tape than a single monophonic head.

Noise elimination systems – The Dolby system is the best known of these techniques for eliminating hiss from tape by boosting the top frequencies when recording, and giving them an equivalent reduction on playback. The cassette recording techniques developed for computers are extremely tolerant of hiss, and straightforward, unDolbyed recording is to be preferred.

Tone controls – Though useful in music for compensating for the effects of the loudspeaker or of poor recording by altering the balance of frequencies, tone controls represent an unnecessary variable in the cassette/computer link. If your cassette machine has a tone control, set it at maximum and forget it.

Good low-frequency response – This is something not easy to achieve within the limitation of a cassette recorder, and so it isn't cheap. It gives distinct 'presence' to music, but your computer won't miss it in a cheap recorder, as micros don't use signals below 100Hz.

Automatic Level Control – The automatic recording level control (ALC) found on cheaper recorders is something of a mixed blessing. More expensive machines have one or more VU meters, and a control to adjust input volume, introducing one more variable in the link. A recording level meter and manual recording control are very useful features for giving a quick visual check as to whether the data signal is strong enough to work properly – a better arrangement than a simple neon indicator which lights when recording.

A good ALC system can be a real boon, adjusting the machine to the correct input setting immediately, and even making automatic compensation for poor leads or micros with different output volumes. The problem is that ALC circuits are designed for voice and music, and aim to get a 'reasonable' output level from a wide range of inputs. Some of them continually fiddle with the level when computer signals are used (technically the ALC control loop hunts). If you are unlucky enough to obtain one with a troublesome ALC circuit, the only sensible cure is to change it. Of course the recording side of the cassette system is only relevant when you are *saving* your programs.

Mains or batteries?

Mains/battery operation is now almost universal on cheap recorders, although the power pack may take the form of a mains adaptor that feeds a smoothed, rectified low voltage into the machine. Be wary of battery-only machines – depleted batteries can affect recording speed, and make saving and loading extremely difficult. On a very few machines can you remedy this by altering the playback speed. Some recorders with a poorly smoothed power supply can corrupt data when used on mains operation, but this can be detected by playing the tape back and listening for an audible hum. If the machine hums, change it.

Despite the caveat above about battery-only machines, the micro-cassette players developed for dictating can also be used, as long as you choose the kind which use a *capstan drive*. Rim drive machines have no capstan and simply pull the tape past the head by turning the take-up spool. While perfectly adequate for speech this produces quite large variations in tape speed, and should not be entrusted with data recording. At least one manufacturer produces a micro-cassette recorder specifically for use with home computers. Its small size makes it very easy to slip into a pocket, but of course you will not be able to play back standard cassette software from a machine of this kind.

Input and output sockets

Most cassette recorders intended for audio use come with sockets suitable for computers, but it is a point worth checking. Normal requirements are two 3.5mm

jack plug sockets, usually labelled **Mic** (Microphone) and **Ear**. A thinner 2.5mm jack plug socket for remote control is also handy. Some micros are supplied with a DIN socket. If the cassette does not have one, the problem can be overcome with a DIN-to-jack plug lead, but don't forget to count this towards the cost of the machine.

Keeping track

If you are going to record more than one program on a cassette a tape counter is extremely useful. It helps locate a particular part of a program when playing back and listening to the data signals by ear. All good recorders allow the user to listen to the data signals on fast forward (cue) and rewind (review). This helps the operator detect gaps in data transmission, speeding up program location. In fact, some machines specially tailored for computers allow you to listen to the data signal while playing and recording.

The best from your recorder

The ultimate in computer cassette recorders will have the facility for adjusting playback speed in case data has been recorded on a slower/faster machine, it will have AC bias and erase for clearer data signals, and a long life recording head – wear shows up earlier with computer use than with audio use.

Most micro users can live without such refinements, but your cassette machine has to be handled properly if you expect it to load correctly first time – and, more importantly, if you are going to trust it with your data. (*see* Backup) Here are some points to watch out for:

Keep the tape path clean. Data recordings are far more sensitive to oxide deposits left behind by tapes, so the recorder should be cleaned regularly. Proprietory cleaning cassettes are available, they either draw slightly abrasive tape across the head, or use the motion of the tape drive to work levers to scrub the head, pinchwheel and capstan with pads soaked in solvent. But if you are careful and your machine allows easy access to the tape path – as a good recorder should, there is no better way than to use cotton wool buds and cleaning fluid. Methylated spirits won't do any harm, but solvents such as benzene and similar thinners may ruin your pinchwheel and the plastic casing.

Tapes. Don't use the cheapest tapes, poor frequency response isn't the reason for avoiding cheap tapes – data recording doesn't need the highest of hi-fidelity. But cheap tapes often use troublesome coating material that flakes off, piling up oxide deposit along the tape path and prematurely wearing down your recording head.

Watch your head alignment. The recording head has an almost invisibly small vertical gap which enables it to magnetise the tape while recording, and to pick up the signal on playback. An incorrectly aligned gap, known technically as *azimuth misalignment*, can cause data to fail to load entirely.

Some tailor-made computer cassette recorders come with an accessible recording head adjusting screw, so that you can set up the alignment without having to dismantle the machine. It is easy to carry out the adjustment while listening to a tape – the signal sounds louder and sharper when the head is correctly aligned.

Paradoxically this adjustment may be useful even if your machine is perfectly aligned. A program which has been saved and loaded successfully on a machine with the azimuth out of alignment may not work on a recorder with correct alignment. Matching this degree of misalignment will enable you to load the program – but don't forget to readjust the machine afterwards.

Keep your recorder away from the computer. This may not be a problem, but the reason why the cassette leads supplied with computers are usually at least a couple of feet long is that there is a possibility of interaction between components like transformers which may interfere with loading and saving.

Verify your saves. Nearly all computers have a method of ensuring that a program has been saved correctly, usually in the form of a command which you can use after each save – but it only works if you use it!

Write-protect your data. Important software that you are not going to need to modify should be kept on a cassette with the record-prevention lugs at the rear broken out. All standard cassette machines are able to detect if these lugs have been removed, and will lock the record mechanism to prevent accidental erasure.

The 64 records data on cassette in 192-byte blocks. Each block of information read from tape is first held in a buffer and checked for validity. An invalid block is re-read – easily achieved without complicated rewinding because the 64 records each block twice. If the first read fails then the copy is read, and this should be correct. When you do an INPUT#, you are in fact reading data from the buffer, and when you come to the end of the buffer the cassette motor is started up again to refill it. The same principle applies to programs, but here the whole program is recorded on one long block, which is repeated for error correction. The validity check is carried out by exclusive-ORing each byte of data to calculate a 'checksum' byte which is recorded after the data. When the data is read back the 64 calculates a new checksum which is compared with the one on the tape. If the two agree then the 64 assumes that the data was read correctly, otherwise it retries with the copy block.

Communications: RS-232, IEEE And Centronics

The three common data communications standards are RS232, IEEE-488, and Centronics, a system named after the printer manufacturers who developed it. Centronics is used only with printers, but RS-232 and IEEE-488 can communicate with a variety of devices. Of the three, RS-232 is by far the least 'structured' – that is to say, the RS-232 does not define the total communications system, but only the pins and signals they carry.

Communication Principles

There are two basic ways of sending data:

Parallel transmission is the process of sending a whole character at once, so that the bit patterns of each byte appear simultaneously on a parallel set of 8 wires, one for each bit. Extra wires are also set aside for managing the communication.
Serial transmission breaks up the bytes into a bit stream, sending one bit at a time down the line. This will require at least seven transmissions for each single character.

Both IEEE-488 and Centronics are parallel transmission interfaces, while RS232 is serial. The theoretical top speed at which data can be transferred is higher for parallel than serial transmission, but the serial transmission is more reliable over longer distances, being less susceptible to corruption. Parallel transmission implies a heavy outlay in cables, and over distances greater than a few feet risks problems like 'skewing' – signals getting out of step due to varying delays on different lines.

The means by which elements of a system communicate are governed by sets of rules, known as protocols. These protocols have seven functional layers, but only three of the seven concern the stand-alone micro. The remainder are concerned with networking systems.

The lowest layer is the 'physical layer'. This defines the mechanical characteristics (pin layout, plug dimensions, and cabling), the electrical characteristics (voltage levels, data rates, and some circuit details), and the signal functions (names and functions).

At the next level is the 'data-link layer', which defines the way in which the data is transmitted across the previous layer. The problem with RS-232 is that as a standard is that its definition covers *only* the physical layer.

The **protocol layers** consist of:

The **physical layer** –, the plugs, pins and cables.
The **Data-link layer** – serial, parity and protocols.
The **Network layer** – switching and routing.
The **Transport layer** – communication between layers above and below.
The **Session layer** – communicates application to system.
The **Presentation layer** – conversion and formatting.
The **Application layer** – the application program.

IEEE-488

The IEEE-488 standard was approved in 1975 – the product of around four years work on an interface structure which could link not only peripherals, but whole systems. One of the major forces behind this development was Hewlett-Packard, and indeed the bus is still sometimes referred to as the HPIB (Hewlett-Packard Interface Bus).

Its main function was to link such esoteric gadgets as data-loggers, spectrum analysers, and digital voltmeters. For the user in the street, though, it was brought to prominence by its inclusion in the Commodore range of computers, which used the interface – somewhat eccentrically – to communicate with printers, disks, modems and other more conventional peripherals.

A device has one or more of the following statuses on the bus. It may be a **controller** – a master of other equipment and usually of the bus. In addition, it may be a receiver of information – a **listener**. A third role is as a provider of information – a **talker**.

How IEEE controls the lines. The first three lines after the data bus (eight lines in this since it is a parallel structure) control the data hand shaking. The remainder manage the bus and its facilities. The handshake is accomplished first by a talker setting DAV (Data AVailable) high, which, because it is an inverted signal, indicates 'data invalid'. The data is then placed on the bus. When all listeners who have been selected to receive the data have signified their readiness by freeing the NRFD (Not Ready For Data) line, DAV is reset as a signal that the data is now valid, and holds the data stable until all listeners have acknowledged their receipt of the data by dropping the NDAC (Not Data Accepted) line – *see* Figure 1.

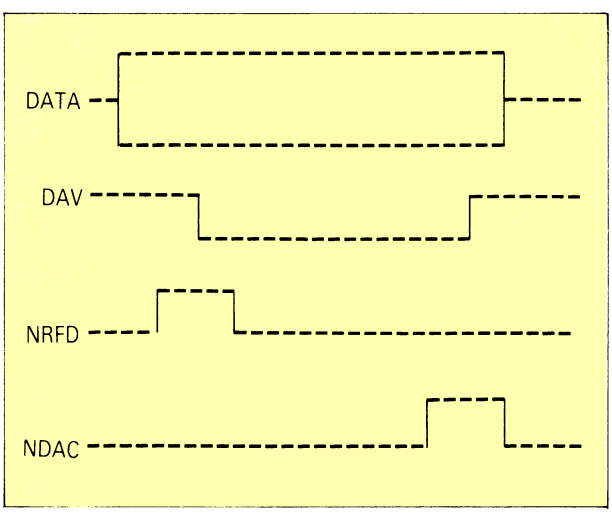


Fig. 1. IEEE timing sequence

One of the great advantages of the IEEE-488 convention is the ease with which communications can be established between many different devices using a set of simple commands.

PIN NO	CODE	MEANING	FUNCTION
1	D101	Data I/O	All data and address info
2	D102	Data I/O	All data and address info
3	D103	Data I/O	All data and address info
4	D104	Data I/O	All data and address info
5	E01	End or Identify	End of transfer
6	DAV	Data Valid	Stable data
7	NRFD	Not Ready For Data	Ready indicator
8	NDAC	Not Data Accepted	False when data accepted
9	IFC	Interface Clear	Interface "reset"
10	SRQ	Service Request	Device to signal need
11	ATN	Attention	Controller signal
12	SHIELD	Cable earth Shield	Instrument earth
13	DIOS	Data I/O	All data and address info
14	D106	Data I/O	All data and address info
15	D107	Data I/O	All data and address info
16	D108	Data I/O	All data and address info
17	REN	Remote Enable	Remote/local toggle
18	GND(6)	Earth for DAV	Earth for control line
19	GND(7)	Earth for NRFD	Earth for control line
20	GND(8)	Earth for NDAC	Earth for control line
21	GND(9)	Earth for IFC	Earth for control line
22	GND(10)	Earth for SRQ	Earth for control line
23	GND(11)	Earth for ATN	Earth for control line
24	GND.LOGIC	Earth for REST	Earth for control line

IEEE pin-outs

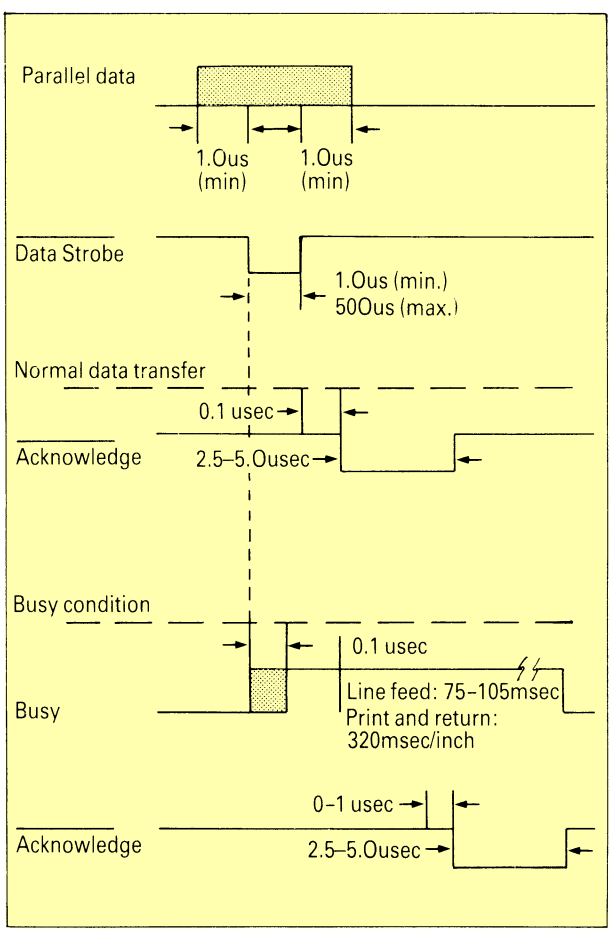


Fig. 2. Centronics timing sequence

Centronics

The obvious advantage of a parallel interface like the Centronics standard lies in its speed. It's also cheaper to implement in the hardware, which is why low cost dot matrix printers like those made by Epson and Centronics themselves usually have a parallel interface. Some, like Epson, offer RS-232 communication as an extra plug-in board.

Figure 2 shows the timing and signal relationships of this interface. The computer places the data to be transmitted on the pins of the printer interface connector. It then puts an electrical pulse on a line called 'data strobe', indicating that the printer interface pins have a new, valid data word on them waiting to be loaded into the printer. To indicate that it has successfully received the data strobe signal and has loaded the data into itself, the printer drives a line called 'acknowledge low'.

Data strobe and acknowledge low therefore constitute a sort of elementary conversation about the passage of the data: 'Here we are, the next byte's ready to go' and 'All right, I got that one, now you can send me to the next byte'. The printer also has a line called 'busy', used when it has to perform a relatively long, time-consuming task like making a carriage return or formfeed. As can be seen from the timing diagram, a successful interface need only monitor one of these lines.

The signal levels on the interface are 0 to 5 volts, known as 'standard TTL logic levels'. The Centronics interface uses a standard plug as well, the 36-pin Amphenol type 57-40360. Its pin connections are shown in Figure 3.

Personal computers for home use don't always use this particular connector for the Centronics output, but the documentation should still use the standard Centronics names for the pins or connections. Two other signals on

the interface, 'paper out' and 'printer not selected' are rarely used in personal computer interfaces and can be safely ignored.

If your micro has a port labelled Centronics, the chances are that you will be able to connect up a parallel printer simply by plugging it in.

PIN NO	CODE	FUNCTION
1	STROBE	Read Data Pulse
2	DATA 1	Data Lines
3	DATA 2	Data Lines
4	DATA 3	Data Lines
5	DATA 4	Data Lines
6	DATA 5	Data Lines
7	DATA 6	Data Lines
8	DATA 7	Data Lines
9	DATA 8	Data Lines
10	ACKNLG	Data received and ready for more
11	BUSY	Not ready for data *
12	PE	Set to high when out of paper
13	+5V	
14	AUTO FEED	Switch set extra line feed
15	NC	No connection
16	GND LOGIC	Logic earth
17	GND CASE	Case earth
19-30	GND	Signal earths (twisted pair)
31	INT	Roset and buffer clear
32	ERROR	See Busy
33	GND	Signal earth
34	NC	No Connection
35	+5V	
36	SLCT IN	Optional DC1/DC3 code

* Busy is set if:
1. Data is being received.
2. Printer is printing.
3. The printer is "off line" and
4. if an error condition is present.

Fig. 3. Centronics pin-outs

The RS-232 interface

Using RS-232 is unfortunately not as simple. Unless you understand the rules that govern the interface, connecting up computer equipment through an RS-232 so-called 'standard' port can be frustrating. But this is not really fair on RS-232; paradoxically its niggling difficulties stem from its success in being widely adopted – and equally widely adapted.

RS-232 really constitutes a specification for two standards: one for the modem, known as the Data Communications Equipment, or DCE, and another for the mainframe or terminal known as the Data Terminal Equipment, or DTE. Serial communication between computers or computers and printers is usually a two-way business. Data is sent out along one wire and read in along another in the form of rising and falling voltages which correspond to the bit patterns they represent. These voltages are measured with respect to a third wire, known as the SG (Signal Ground) line.

Other lines are set aside for managing the movement of the data, and these are called 'flow control lines'. All but two of the 25 pins of the full standard RS-232 'D' type connector have specified functions ranging from 'clear to send' to 'secondary data carrier detect'. Figuring out what these are for is just one the delights of RS-232, because computer equipment manufacturers tend to use combinations of them selected largeley on personal whims!

The function of each wire depends on which end you're looking at, that is to say, whether you are DCE or DTE. The standard lines are listed in Figure 4 along with the numbers of the pins in a standard 'D' type connector. The three already mentioned are designated for transmitting (TxD, on pin 2), receiving (RxD, pin 3) and signal ground (pin 7).

An immediate confusion arises: should the TxD or transmit line be wired up to the RxD or receive line at the other end? You would think so, but the answer should be no. This is where the DTE/DCE distinction first arises. The pins are described from the terminal's point of view, which means that the DTE uses the TxD line to transmit (to the DCE) and the RxD to receive (from the DCE). The DCE transmits through RxD, and receives through TxD.

PIN NO	TYPE	CODE	FUNCTION
1	E	–	Protective case earth
2	D	TxD	Transmit data
3	D	RxD	Receive data
4	C	RTS	Request to send
5	C	CTS	Clear to send
6	C	DSR	Data set ready
7	E	–	Signal earth
8	C	DCD	Data carrier detect
9	–	–	Testing
10	–	–	Testing
11	–	–	Unassigned
12	S	SDCD	Secondary DCD
13	S	SCTS	Secondary CTS
14	S	STxD	Secondary TxD
15	T	–	Transmit timing (DCE source)
16	S	SRxD	Secondary RxD
17	T	–	Receive timing
18	–	–	Unassigned
19	S	SRTS	Secondary RTS
20	C	DTR	Data terminal ready
21	C	–	Signal quality detector
22	C	–	Ring indicator
23	C	–	Data Signal rate select
24	T	–	Transmit timing (DTE source)
25	–	–	Unassigned

TYPE CODES : C=CONTROL, D=DATA, E=EARTH, S=SECONDARY. T=TIMING

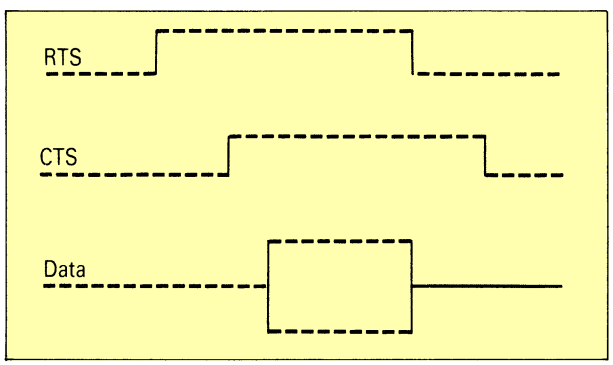
Fig. 4. RS232 pin-outs

Handshaking. Once communications have started by way of the RS-232 interface, you have to make sure that the data goes in at one end at the same rate it is removed at the other. Obviously the baud rate (*qv*) has to be the same in both pieces of equipment. Typical baud rates are 300 or 1,200 baud for a micro connected to a modem, and

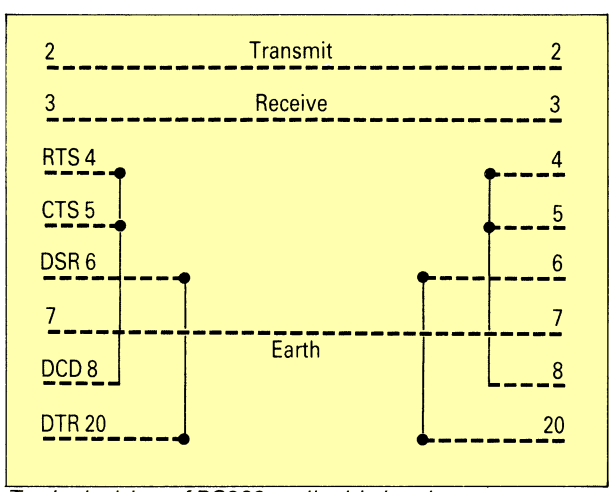
up to 9,600 baud for a micro connected to a printer.

A receiving device will usually have an area of memory to act as a 'buffer' between the computer and the communications line. Think of the buffer as a reservoir that fills up as the rain comes down from the hills on wet days, and drops its level on dry days as the water is drawn off by the town below. Like a buffer, the reservoir supplies the town with a steady pressure of water, isolating it from fluctuations in the natural supply.

Obviously there must be taps somewhere between the town and the reservoir if the inhabitants are not to be drowned. An RS-232 communications line needs a similar arrangement, so that the printer can give itself time to print by sending back some sort of signal to the computer to turn off the data stream. The signal will be like a simplified tap, having an on position meaning 'send data' and an off position indicating 'hold back'.



RS232 two control line hardware handshaking



Typical wiring of RS232 to disable hardware handshaking

This process, called 'handshaking', can be done in one of two ways. In so-called **hard handshaking** a separate wire on one of the flow control lines is reserved for the purpose. The receiving device changes the voltage on this wire to let the sending device know when it is time to pause. The sending device of course has to monitor this wire to detect the message. **Soft handshaking** needs only the Rx and Tx lines. Whichever of these is not being used by the sending device is commandeered by the recipient for use as the 'tap'. But instead of the on/off voltage used on the flow control lines, special characters are send back, to be read as flow on and flow off codes. If the system is using hardware handshaking the state of the tap can be represented by any of the flow control lines – usually RTS or DTR. The principle is identical, whichever line is chosen.

Soft handshaking offers a choice of two distinctly different methods:

Xon and **Xoff** are the names given to a pair of ASCII (*qv*) codes, transmitted in this case by the recipient. When the sender receives Xoff it knows to stops sending, and won't resume until it gets an Xon. Obviously in this case it is up to the recipient to make judgements about when its buffer is about to overflow, and when it will need more data.

ETX/ACK is an older method of software handshaking suitable for less intelligent terminals. The sender effectively transmits the data in blocks calculated to fit in the recipient's buffer. At the end of each block the sender introduces into the data stream the special non-printing code ETX, which stands for End of TeXt. It then stops sending, and waits, listening to the recipient's transmission line.

Meanwhile the recipient is working its way through the block of text. The last character it finds in the buffer is the ETX code transmitted by the sender. This discovery triggers the only rule it has to obey, and it responds by sending the special ACK (for ACKnowledge) character on its own transmission line. On receiving this character the sender restarts transmission, and the cycle begins again.

Checking the data – When a message is being sent down a serial line it is vulnerable to corruption from things like stray magnetic fields. Various methods are used to check for errors, in order to minimise the risk of garbled data.

The streams of bits are sent in separate chunks, usually thought of as bytes, but in this context preferably called 'bit fields'. Each bit field is usually seven or eight bits long, and is fenced at beginning and end by a number of extra bits called 'framing bits', which are used for timing and error checking. The timing bits are called start and stop bits, and there is often a further bit, called parity bit, for error checking.

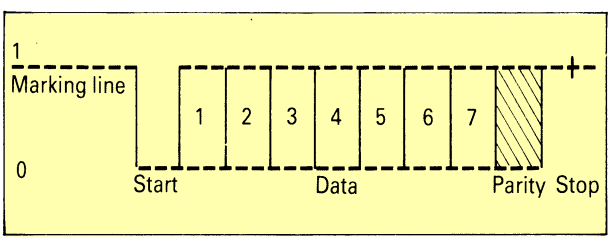
Parity checking is a very simple concept that doesn't

Character	ASCII	Databits	odd	Parity bit No '1's	even	No '1's
A	65	1000001	1	3	0	2
B	66	1000010	1	3	0	2
C	67	1000011	0	3	1	4
X	88	1011000	0	3	1	4
Y	89	1011001	1	5	0	4
Z	90	1011010	1	5	0	4

Parity checking

pretend to spot all errors, but will catch most of them with the need for very little additional calculation. The number of set bits (ones) in any bit field will either be odd or even, and corruption of a single bit – the most usual kind in serial transmission – will turn an odd number of set bits to an even number, and vice versa.

If the sender includes an extra bit reflecting the 'parity', or evenness of the set bit count, the recipient can 'parity check' the received bit field and make a good guess as to whether it is corrupted. One of two parity rules are used: 'even parity' adjusts the parity bit to make the total number of set bits even, and 'odd parity' ensures that the number of set bits is odd. Parity checking will only highlight errors of a single, or an odd number of bit changes. An even number of errors will preserve the original parity.



Structure of serially transmitted character

Matching protocols. When setting up serial communications you have to make sure that the equipment at each end of the link is acting according to the same rules, or 'protocol'. The parameters that have to match are:

The number of data bits per bit field, typically between seven and ten.

The number of start bits (hardly ever anything other than one).

Number of stop bits (usually one or two, but sometimes one and a half).

Whether there's parity checking, and if so whether it's even or odd.

You may be puzzling about the one and a half stop bits. In fact these stop and start bits aren't really bits at all, just lulls of known lengths which help to keep the receiving device in step with the transmitter.

Sometimes a more sophisticated error checking method is used, especially for communicating large files of data between computers. The most common method is called the **Christensen protocol**, or XMODEM. Some packages use their own error checking method, in which case you will need to run the same software in both the sending and receiving computer.

The physical connection. When wiring up, first work out the pin numbers on the RS-232 plug (be it 'D' type or, as on the Beeb, a DIN plug). These are usually marked on the plastic block that holds the pins. Having done this, and decided which pins to connect, use multicore cable with the appropriate number of wires.

Bare about five millimetres of each wire, cover it with a coating of a solder using a small soldering iron (15 watt will do). Place the wire either into or onto the connecting pins (depending on their design), and check that there's no excess bare wire (which could cause the pins to short out). Then place the tip of the iron on the connector pin until the solder on the wire has melted. Once cooled, check that the joint is secure (with a light tug); apply more solder only if necessary.

Computer Crime

Computer crime first gained international attention in 1973, when a computer was used to crank out phoney life insurance policies for sale to reinsurers. Investigators probing the Equity Funding Scandal, as it was known, discovered that $143 million of the firm's assets were non-existent and that 19 per cent of then resulted directly from computer fraud.

The Cost of Crime

Computer crime is a flourishing business at local, national, and international levels. Just how much money it is costing is unclear: the banks and businesses are reluctant to admit how much they've lost, fearing that publicity will heighten customer doubts about their stability – and the police do not keep separate statistics.

It is estimated that computer criminals in the US take about one hundred million dollars a year, and the figure of one billion dollars a year is expected soon. One reason why it is so difficult to pin a figure on the amount that is taken is that the very instrument used to commit the crime can be programmed to erase all traces of the act. Used like this, the computer is the biggest boost to the criminal career since the invention of the glove!

The proliferation of remote terminals, including the micros now appearing in living rooms all over the western world, and the way in which smaller computer operators share time on a larger single computer all makes it even easier for people to break into computer systems. All that is necessary, besides the computer, is a modem (*qv*) and the right access code.

Many computer networks publicise their telephone numbers as the means of soliciting customers. Once inside the network, the user needs only patience and some shrewd guess work in order to link up with the system's software. From there on it is as easy to steal a million pounds as it is to steal fifty pence.

Obviously financial institutions are the most vulnerable. Banks, with their ever growing reliance on automatic tellers and home banking systems, use electronic networks controlled by computers to transfer hundreds of billions of pounds daily. Most computer crime seems to occur in organisations where there is a very significant lack of control, but the explosive growth in personal computers and increased access to data-banks have combined to send shivers down the spines of corporate computer experts whose job it is to protect their systems from outsiders. In the past the culprits have usually been insiders, persons who have earned a degree of trust and are in a position to violate it.

Pranks, traps and blackmail

One of the many tricks that computer criminals and errant computer nuts have devised as a way of wreaking havoc on electronic America is known as the 'Trojan Horse'. Like the Ancient Greeks penetrating the defences of Troy, these modern invaders are able to slip extra commands into a computer program. Then when a user with a suitably high clearance runs the program, the action unknowingly triggers the altered command – possibly to tell the computer to transfer money to the culprits account.

Other tricks of the trade include 'data-diddling',the mischievous altering of information stored inside the computer, and 'super zapping', activation of a computer's emergency master program.

Computer crime is being practised by younger people, the result of computer science classes and the increased accessibilty of home and personal computer power. In Chicago two years ago, two high school students used a home-made terminal to gain access to the main computer at De Paul University. They shut it down for an entire week during enrolment and threatened more of the same unless their demand for a computer program worth several hundred dollars was met. They were caught and placed on probation.

Security Techniques

Besides insurance, many companies have opted for sophisticated 'encryption' devices which scramble computer messages so that they cannot be intercepted by outsiders. Although specially secured computer rooms complete with guards are likely to scare off interlopers from outside the company, it is still the disgruntled or dishonest employee who represents the greatest threat.

American Express meets this threat with an elaborate security system which begins at the moment of entry to its credit-card operation center in Plantation, Florida. One of the largest users of computers in South Florida, American Express is likely to have 1,000 computer terminals in use on any given day, and most of these are working 24 hours a day. Each computer user has his own special sign-on number and password, which must be changed at least every 14 days. The transactions which individuals may to perform are also limited; tasks outside an employee's clearance category are met with the response `'USER NOT ALLOWED FOR THAT TRANSACTION'`. Each transaction is recorded on tape for future review, and the machine will automatically disconnect employees who fail to perform certain transactions within a preset period.

One of the things that banks do to increase security is to use temporary bank account files for daily work on customer's accounts. The work is kept on these temporary files until the end of the day when it is relayed electronically to the three main computers, so making sure that a customer's master file is never on line. Meanwhile, the programmers who deal directly with control computers are limited to specific work areas – a programmer handling deposit related transactions would not, for instance, be involved with loan applications. In America today, all new bank employees are expected to take a polygraph (lie detector) test as well as being finger-printed – with the prints being sent to the FBI.

Computer Graphics: Art For CRTs Sake

Cathode Ray Tubes (CRTs) first began to be used instead of teletype machines as a method of operator/computer communications in the early 1950s, but it wasn't until the early sixties that computers which could draw became feasible. In 1962 Dr. Ivan E. Southerland of MIT published a formal description of a graphics computer system, but the idea had to wait nearly another ten years for the technology to make it commercially viable.

Computer users have been intrigued by graphics since the very beginning. Before the industry took the subject seriously, many a computer installation would be decorated with drawings generated using characters on a line printer or teletype. These were not graphics programs in the strictest sense – all they did was to print and overprint characters derived from the printer's character set in such a way as to produce something pictorial.

Human beings can assimilate written information at a rate of about 1,200 words a minute. If that information comes in the form of pictures, the rate leaps up to the equivalent of 40 million words a minute. On this basis, a single picture paints 33,333.33 recurring words! This realisation has been the impetus that during the last 10 years has moved graphics on to being one of the most important methods of communication between humans and computers and between humans and humans. The emergence of cheap microcomputers has served to accelerate the growth in the use of graphics and the software required to produce them.

Growing up in a market oriented towards games, micros have acquired very sophisticated graphics facilities, and can manipulate colour and line with an abandon unknown on many a mainframe. But graphics applications extend well beyond Pacman. Micros can draw line graphs, pie charts and bar charts for business; can help architects with design, and even animate displays for industrial planning (*see* Computer-Aided Design).

How graphics work

Most of the graphics technologies rely on the well established principle of **raster scan** display, the same technique used to display television pictures. The electron beam which strikes the phospor surface on the inside of the CRT moves rapidly back and forth across the screen drawing horizontal lines, shifting slightly in a vertical plan after each line is finished, until it has filled a picture-sized rectangle, the raster. The image is formed by modulating the strength of the beam as it travels and re-travels its regular path.

Another form of graphics generation comes from the use of an oscilloscope-like screen. Called **vector graphics**, its images are built by specifying co-ordinates. Instead of being confined to regular scanning, the electron beam can dart about anywhere it is directed, effectively painting the image with a brush of electron beams. This gives very high quality results, but requires a lot of computational power because the display must receive a constant stream of co-ordinates and values indicating the beam intensities.

The logical resolution of a vector-based graphics system depends on the kind of maths it uses and is independent of the hardware, the physical resolution is dependent on the properties of the display devices. A system using integer maths can operate a co-ordinate grid of 65534 x 65534 pixels. With floating point maths the possible logical resolution rises to 10 x 10 pixels.

Resolution of this order is far beyond the capabilities of any physical devices, but is used in systems with the facility to zoom – that is, to take a section of the image and blow any amount of it up to the full size of the display device. Floating point vector graphics will permit a display representing something as big as a football pitch to hold one small element the size of this book (A4) with such detail that you could zoom in on it close enough to read every word of this page. You could also continue the zoom until one of the letters filled the screen.

But almost every micro-computer uses raster-scanning, either in the form of a television set, composite video (which in effect is television without any broadcast modulation) or RGB, a system feeding in data separately for each of the Red, Green and Blue electron guns in the CRT.

While vector graphics deals in lines, raster scan graphics handles **pixels** (PICture ELements). A pixel is the smallest element you can manipulate individually in a system.

Micros often have two types of screen display: text and graphics. The text equivalent of a pixel is a rectangular block displaying a single printable ASCII character. 40 of these (80 on some machines) can be displayed across the screen, which also has room for around 24 lines vertically. So a 40 × 24 screen will be able to display 960 characters.

A close look at one of the letters will show how it is composed of a matrix of, typically, 8 by 8 dots. In **high resolution** graphics mode (hi-res for short) the computer allows you to control any one of these 64 dots individually, so that instead of a mere 960 character positions, over the entire screen you will be able to manipulate something like 64,000 of these pixels.

Typically machines that use high resolution graphics can also operate in **medium resolution**. In medium resolution the screen will display something like 160 × 96 pixels, and the memory freed by this reduction in the number of pixels to be looked after can be used to handle more colours, different shades, and even to keep several separate screens in RAM ready for instant display.

Sometimes the high and medium resolution graphics can be superimposed on the screen by using two separate memory stores. Or a mixed screen can be used, with scrolling text in the bottom lines and high resolution graphics across the rest. Lower resolution tends to emphasise one disadvantage of raster scan display over vector graphics: the tendency for diagonal lines to appear as a jagged series of short steps – a phenomenon known as stepping or aliasing.

Other, usually older graphics systems use the large character positions as the smallest unit, but provide ready-built shapes which can be displayed like any other character. These usually include vertical and horizontal lines of different thicknesses, diagonal lines, chequered

squares, circles, large dots and the shapes of playing card suits. Using these elements you can create little animated figures, bouncing balls or space invaders which can be moved about the screen at will. Most microcomputers will allow you only to display one character on each space, but some will let you have two displays visible at the same time, one on top of the other. You can have a background screen and move characters over this, or use the second screen to build up more complex shapes.

Several micros have user-definable characters, so that you can create your own. If for instance you were creating a histogram to compare salaries in a company you could add a little car character to all the people who have a company car, or a pound sign to all those with expense accounts.

The software handle

Micros have different software methods of building a graphics picture, depending usually on the BASIC supplied. Some systems require a good deal of low level `POKE`ing and `PEEK`ing, others let you draw a line from any one point on the screen to any other with a command like `PLOT (0, 0) - (319,199)`, and versions of GWBASIC (*qv*) include a number of commands like `CIRCLE`, `PLOT` and `PAINT`. Another useful command is `FILL,` which allows a shape to be shaded to a certain intensity or colour by giving the position of any coordinate in the area, and defining the colour of the boundary line you are filling. Features like these make it easy to draw your biorhythm chart or to display the ups and downs of the stock market, although complicated subjects like real-life pictures will have to be meticulously worked out with graph paper and lots and lots of patience.

It's useful too to be able to save images on disk. This allows an image to be built up before it is displayed and then brought instantly back to the screen at some appropriate point in the program, avoiding the relatively slow picture drawing typical of graphics generated in BASIC. As a luxury some systems even include the facility for fading the display in and out.

Interpreted BASIC is a slow language, and the quest for speed and flexibility may take you into the dark realms of assembler (*qv*). This is an excellent and rewarding way of getting to know your micro, but most of what you will learn like this will be heavily machine dependent. Beware of coming away from a year of intensive programming with a head full of `PEEK`s, `POKE`s, operating system calls and super-secret bug fixes that have made you king of your own small micro but have left you a humble peasant in the wider world of computing.

Commercial packages

An alternative to the 'dirty hands' approach, if you can afford it, is to use commercial graphics packages to take care of the details. This is an option that business users often settle for with a sigh of relief after the initial burst of enthusiasm. Graphics packages are designed to meet two general requirements. First, to display business or statistical information in a way that indicates trends or proportions; and second to display mathematical plots. Some packages are even able to simulate three-dimensions.

Bar charts and graphs are little more than pretty pictures in a business context unless they are provided with headings and numbered scales on the X and Y axes. Being able to label the axes, 'Gross income' against 'months' for example, is also more or less essential.

Some figures are better presented as circles divided into differently coloured or shaded sectors. Market share and input cost categories are two aspects of business statistics that benefit from this approach, fancifully called **pie charts** from their resemblance to a pie sliced into portions. Each sector of the pie should be capable of being labelled separately, and the percentage of the pie represented by each sector printed alongside.

Colour is important for presentation graphics. Although for some applications four colours is sufficient, the effect tends to be garish with the sort of large filled areas you get in pie charts. A palette of at least eight colours will help you to avoid eye-aching clashes.

The limitations of memory on most micros mean that you have to perform a balancing trick between colour and resolution. Because more pixels are used in high-resolution graphics the computer is using up more memory, but each pixel has to be given a colour instruction. By using four colours instead of 16 it can store its colour codes in two bits instead of four, halving the amount of memory needed for the purpose.

When building chart displays some programs require you to re-enter the data by hand each time. It is better if the data can be re-created automatically by reading from a file on disk or cassette, perhaps in the form of a spreadsheet. Check to see if there is some way that the images can be stored directly, so that they can be brought back instantly without having to be recalculated by the BASIC plotting routines, a necessarily slow business. Without this facility the simpler two-dimensional plots may take a minute or two each time you want to display them, and complex three-dimensional plots can even take a matter of hours. Not what you want if you're trying to impress the boss with your rapid grasp of figures!

Quick on the draw

One method of storing graphics for quick retrieval is to use an array (*qv*). When you run your program to generate the original display, you can place each co-ordinate in a two dimensional array, each element in one dimension being an x-axis value, and each element in the other dimension being a y-axis value. When you want to display the picture, you simply write a routine that reads from the array the plots of each point.

This can, however, become complicated if you're using a variety of colours in the chart. In this case, you could create the display you want, and then do one of two things. A binary save of the screen area of memory (*see* Memory Map) to make a byte by byte copy onto backing store will allow you to recall the picture with a binary load

and display it almost instantly.

Alternatively you could use an array again, but this time PEEKing the value of each location of the screen area into the array. This will mean that the routine needed to redisplay the chart will be difficult to program without detailed knowledge of how your micro stores graphics. But it does mean you can edit the chart simply by changing certain values in the array. If you're very keen, you could write a number of routines to play around with the array and in effect create your own graphics editor.

Even at the low end of the computer graphics market, given enough memory, graphics can be a powerful and interesting instrument to explain ideas and forecasts. With a bit of clever editing and rejigging it is possible to make each graph look very different and attractive.

Before you buy software, make sure that the manual explains clearly what the package is supposed to do and how it is supposed to do it. You would be justified in expecting a manual about graphics to be copiously illustrated, colourful and well laid out. Look for a full and clear description of the hardware it needs (if you are going to have to spend money on an extra graphics board it is as well to know this before you get the package home) and how much memory is required. In the best packages of this kind the instructions should be explicit enough for someone who has never used the micro before to be able to get the program up and running without additional help.

Standards in graphics

Both the International Standards Organisation (ISO) and the American National Standards Institute (ANSI) are working on standards for graphics. One of the companies which claims a strong commitment to these standards is Digital Research (DRI), the progenitors of CP/M. In its own words, the company 'plans to provide the industry standard for graphics'. The DRI approach is very similar to the way that the CP/M standard works, using standard interfaces at the programmer and device level.

Two standards are emerging. The first is the ANSI and ISO Graphical Kernel System (GKS), a standard for the programmer interface which guarantees source code portability. The second is the ANSI Virtual Device Interface (VDI) standard, which by operating at the device level allows portability of object code.

This makes the graphics standards an interface between the operating system level and the language and application level. The heart of this concept lies in the Graphic System Extension (GSX) to the CP/M operating system, which allows graphic output using standard operating system instructions.

GSX is made up of three parts:

The first part, containing the device independent component of GSX, is called Graphics Device Operating System (GDOS) and is the graphics equivalent of the BASIC Disk Operating System(BDOS).

The second part, the machine dependent module, is called the Graphics Input Output System (GIOS). Like the BIOS in the CP/M system it provides a software connection between the graphics operating system and the machine.

The third part is a utility called GENGRAF. This is used to configure the graphics application to run in the GSX environment.

Spritely animation

One of the most rewarding, yet tedious tasks in programming, is trying to create graphics which will look convincing as they plod around the screen. Up until now, the only way to get really good graphics has been to use machine code, and even then a great deal of computation is needed to get the pictures to move. But with some new packages on the market, it's possible to have aliens swarm across the screen, or a plane dive into a spaceship – all driven from BASIC.

Sprites, also called 'players' on some systems, are hi-res programmable objects that can be made to look like almost anything, and can be moved across the screen quickly and efficiently, simply by changing a couple of bytes. They also have the ability to overlay one another without either being affected. Overlay is a feature of a computer's screen handling, and so is not available on all micros.

Because sprites are manipulated by the computer's hardware, not every machine can use them – in fact it's only a few that do. But now companies are seeing the possibility of using sprites on non-sprite machines, overcoming the hardware problems by emulating sprites with some very special programming.

The third dimension

Since we live in a three-dimensional world, the brain is very good at interpreting the two-dimensional pattern of lines and colours projected onto the retina of the eye as a view of the complex world of three dimensions. This ability of the brain to turn a 2D image into a 3D experience makes it quite easy to represent a three-dimensional object on a flat screen. Given a few simple visual clues, the brain will be deceived into perceiving an object with depth, rather than just the jumble of meaningless lines drawn by a dumb machine.

There is a large difference between a program which maintains an internal representation of a three-dimensional object, and one which can construct a pattern which looks like a view of that object. The programs below may lead the viewer to assume that the machine holds an internal model of the object it is drawing, and is therefore demonstrating enormous computational power by manipulating more information than is really the case.

If you imagine you are holding a coin as you rotate it vertically, the projected view of the coin is an ellipse with varying amounts of 'pointedness' on its ends, as in Figure 1. A circle is simply the special case where POINTEDNESS = 1.

This technique could be used to draw a bar chart consisting of solid pillars. Draw an ellipse, then draw two

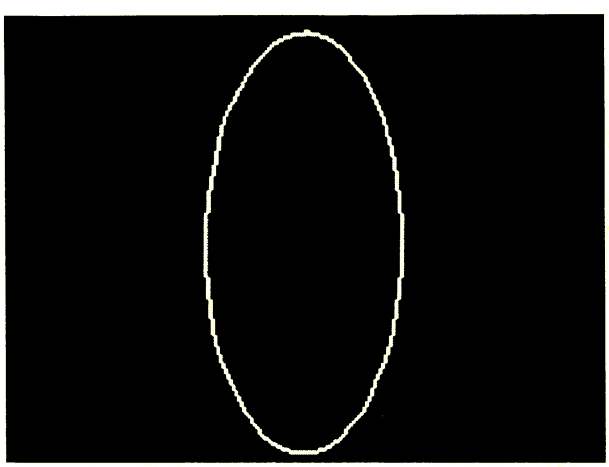
lines running down from the pointed ends. Since we live in a three-dimensional world our eyes naturally interpret this rather arbitrary patterns of lines as the outline of a solid cylinder.

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,
0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,
3 : NEXT
100 FOR ANGLE=0 TO 30  STEP 0.
1
110 X = 160+3 * ANGLE * SIN (A
NGLE)
120 Y = 100+ 2 * ANGLE * COS (
ANGLE)
130 GOSUB10000
140 NEXT
9000 GET A$:IF A$="" THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(
X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2^(7
-(XAND7))
10020 RETURN
```

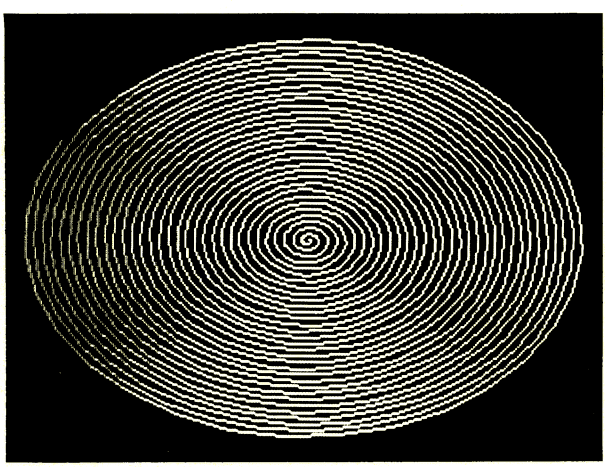
Fig. 1. 'Ellipse' effect

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,
0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,
3 : NEXT
100 SIZE = 100 : RATIO = 0.5
110 FOR ANGLE=0 TO 2 STEP/60
120 X = 160 + SIZE * SIN(ANGLE
)
130 Y = 100 + SIZE * RATIO * C
OS (ANGLE)
140 GOSUB10000
150 NEXT
9000 GET A$:IF A$="" THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(
X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2^(7
-(XAND7))
10020 RETURN
```

Fig. 2. Spiral



Ellipse created by Fig 1



Spiral created by Fig 2

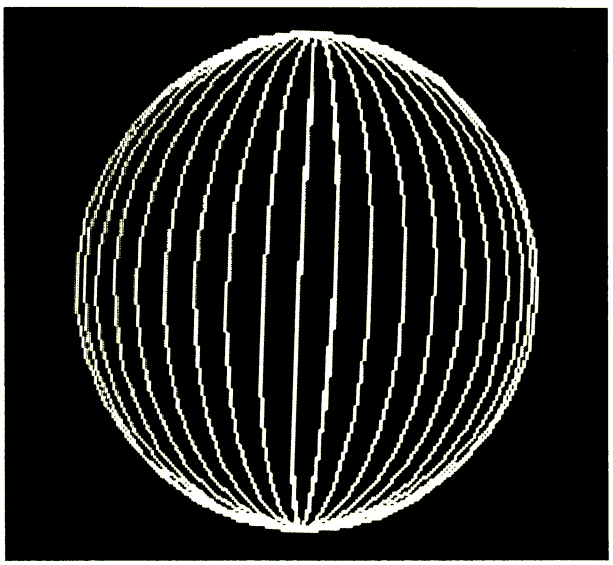
In the previous example, an ellipse is used to represent the edge of a flat disc. A more convincing image is produced if the surface of the disc is displayed. Figure 2 is a program which will produce a spiral.

The rotated coin approach can be extended to give the illusion of a three dimensional sphere, as in Figure 3. Here we draw a number of ellipses with a common origin, and gradually reduce the pointedness of each ellipse. The final result looks a bit like an orange with the ovals running around the gaps between the segments.

Another way to represent a sphere is to use a combination of **SIN** and **COS** functions to produce a pattern which appears to be tracing around a solid ball, as in Figure 4. Figure 5 is an example which will draw a view

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,
0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,
3 : NEXT
100 FOR ANGLE=0 TO 60.4  STEP
0.1
110 X = 160 + 50 * SIN (ANGLE)
 * COS (ANGLE/40)
120 Y = 100+ 50 * COS (ANGLE)
130 GOSUB10000
140 NEXT
9000 GETA$:IFA$=""THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(
X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2^(7
-(XAND7))
10020 RETURN
```

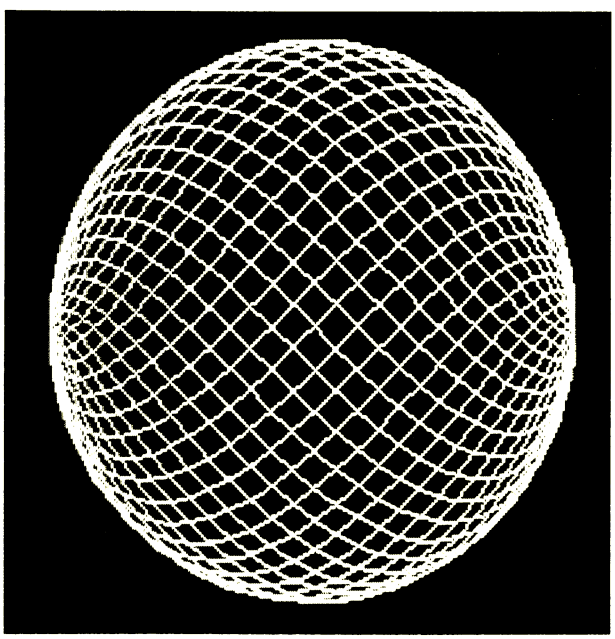
Fig. 3. Multiple ellipse



Multiple ellipse created by Fig 3

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,
0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,
3 : NEXT
100 FOR ANGLE=0 TO 125.7 STEP
0.1
110 X = 160 + 50 * SIN (ANGLE)
120 Y = 100+ 50 * COS (ANGLE)
* SIN (0.95*ANGLE)
130 GOSUB10000
140 NEXT
9000 GETA$:IFA$=""THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(
X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2^(7
 (XAND7))
10020 RETURN
```

Fig. 4. 'Solid ball' effect



Solid ball created by Fig 4

of a wire box as it moves towards you. **DRAWREL** draws a line relative to the current cursor position, and **MOVEREL** moves the cursor relative to its current position.

This program draws the edges at the back of the box. If we want the box to be solid we will have to change the program. **Hidden line removal** is the name given to removing the parts of a three-dimensional view obscured by objects in the foreground. If an internal model is being displayed using a generalised viewing routine this can be very difficult and expensive to achieve. The method described here is very simple, and relies on the ability to fill in areas on the screen quickly.

Draw your three-dimensional view starting with the most distant object, filling in its faces. The 'close' objects are simply drawn on top of more 'distant' objects, thus obscuring the parts which should not be visible. If you want to use this technique with line drawings, you can fill in the faces of each object in the same colour as the background.

As parallel lines on a flat plane get further away, they

```
10 V=53248
20 FOR I=0 TO 62 : READ J : PO
KE 832 + I,J : NEXT
100 PRINT""
110 POKE 2040,13
120 POKEV+21,1
130 POKEV+39,1
140 POKE V+1,100
150 POKE V+16,0 : POKE V,160
9000 GET A$ : IF A$="" THEN9000
9010 POKE V+21,0
9020 END
60000 DATA 0,0,0,0,0,0,0,254,1
28
60010 DATA 1,64,64,2,34,32,4,2
0,16
60020 DATA 8,8,8,16,20,4,63,22
7,254
60030 DATA 16,20,4,8,8,8,4,20,
16,2,34,32
60040 DATA 1,64,64,0,255,128,0
,0,0
60050 DATA 0,0,0,0,0,0,0,0,0
60060 DATA 0,0,0,0,0,0
```

Fig. 5. 'Advancing' wire box

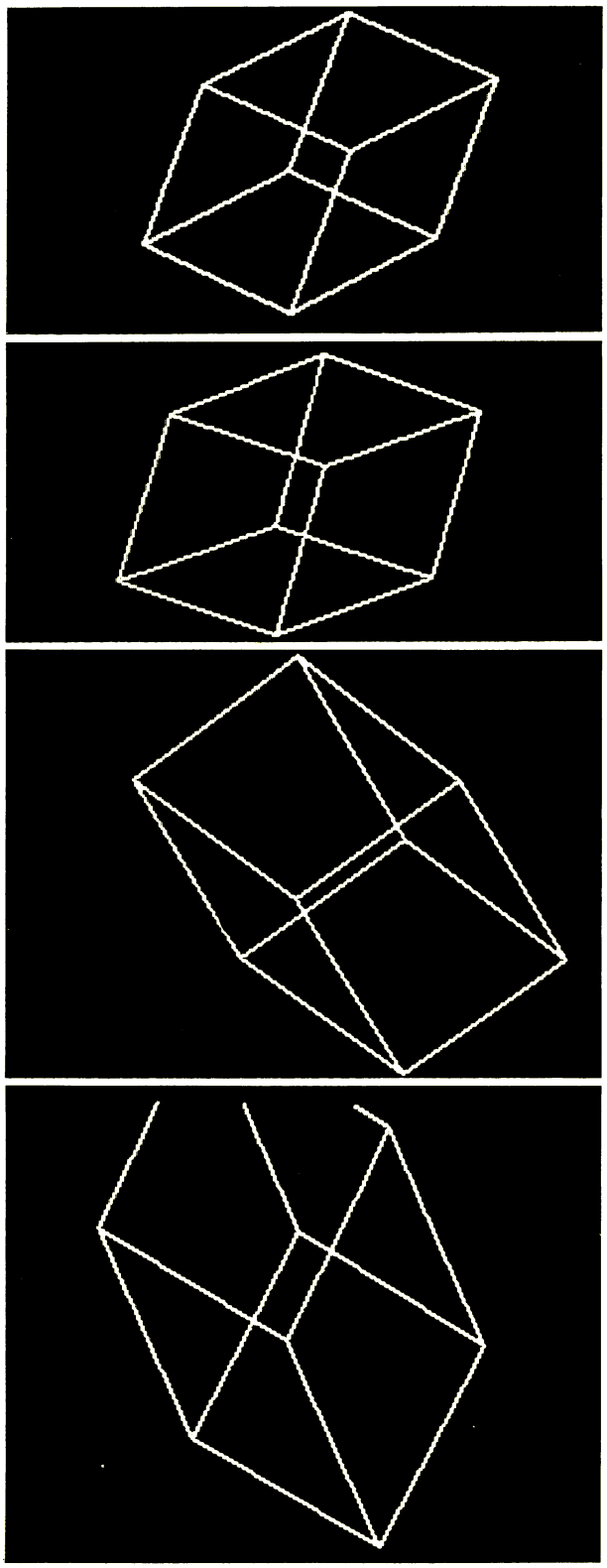
appear to converge. This fact can be used to give the illusion of depth. Figure 6 is a program to draw a view of a flat plane with parallel lines running off towards the horizon.

All the previous algorithms can be used to generate a single view of a three-dimensional object. Here is an informal description of how to generate a program that can create and draw a picture of a fractal landscape. **Fractal graphics** is a way of emulating rough surfaces by treating them as two dimensional, and leaving it to a random (*qv*) computation to supply the third dimension implied by the roughness. To do this, an internal representation of the landscape must be stored. This is then acted on by the view routine, which could be generalised to create a view of the internal model from any direction.

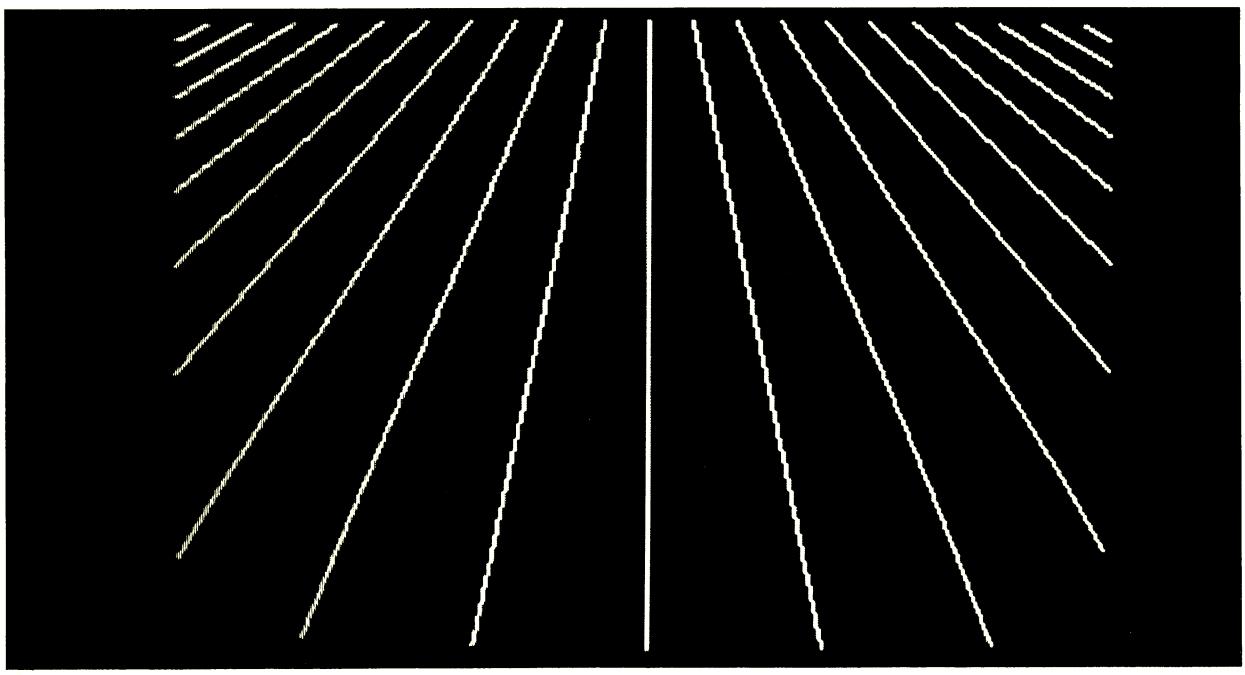
The internal model of the landscape we are using is a two dimensional array of values holding the height of the ground at each location, rather like a grid laid horizontally, with a value stored for the altitude at each point here the grid lines cross. In this way the random three-dimensional topography of the ground can be stored. To generate the landscape, start with all the values in the array initialised to zero (or a fixed value), which will be treated as sea level.

Choose at random a line running across the array in either direction.

Raise the ground level of all the points which fall on



Sequence showing moving wire box created by Fig 5



Converging lines created by Fig 6

one side of this line. The first time you do this the flat plane will be left with a small cliff running across it.

Continue steps one and two, ploughing up the landscape in two directions until it is sufficiently rugged.

The criss-crossing of random cliffs should make the grid points gradually grow up into an undulating landscape. Lakes or sea can be introduced by flattening out all points below a certain altitude. Snow and tree lines can be represented by changes in colour at different heights. There is endless scope for experimenting with different random lines and step sizes.

You could improve the program by replacing the absolute height values held in the array with values which represent the change in ground level relative to an adjacent grid point, to the left, say. Then the introduction of each cliff would be a simple matter of altering the array points lying along the edge of the cliff.

To draw the landscape, a three-dimensional point `(x,y,z)` can be represented on a flat plane by the point `(x+y/2,z+y/2)`. The surface is mapped out by drawing lines to connect the three-dimensional points described in the array.

This brief description of drawing landscapes should give you some foothold into the world of true three-dimensional graphics, in which the machine has an internal model containing detailed data about each of the dimensions. The programs in our examples are 'cheats', in the sense that they keep track of only two dimensions, using illusion or *ad hoc* randomness to create the third. This is not just to make the computations easier – although they will certainly become much more complicated in true 3D – but also to preserve the sanity of your micro, which lacks the RAM capacity needed for the job.

```
10 B=8192 : POKE53272,29
20 POKE53265,59
30 FOR I=B TO B+7999 : POKE I,
0 : NEXT
40 FOR I=1024 TO 2023 : POKEI,
3 : NEXT
100 Y=100 : FOR X=50 TO 250 :
GOSUB10000 : NEXT
110 FOR I=/4 TO/4 STEP 0.1
120 S = SIN(I) : C= COS(I)
130 FOR J=0 TO 80/C
140 X = 160 + J*S : Y = 100 +
J * C : GOSUB10000
150 NEXT : NEXT
9000 GETA$:IFA$=""THEN9000
9010 POKE53272,21
9020 POKE53265,27
9999 END
10000 BY=B+320*INT(Y/8)+8*INT(
X/8)+(YAND7)
10010 POKE BY,PEEK(BY) OR 2^(7
-(XAND7))
10020 RETURN
```

Video disks

The highest quality graphics is promised by the marriage of home computers and video disks. Whereas the average home computer game has to run in 16K to reach the widest market, by using a video disk to supply the images the equivalent of several megabytes of program and data become almost instantly available.

High quality graphics will obviously increase the appeal of games, and the greatly enchanced repertoire of events that can be stored on a large-capacity disk will enable games designers to multiply the number of predicaments that befall the players. The sound track for these hybrid games can be drawn directly from the disk, and the visual component supplied by doctored extracts of existing movies, or graphics pre-generated on mainframe computers or a mixture of the two. 'Pre-digested' graphics are necessary if home computers are to look sophisticated – at present they take too long to generate their own pictures. A future generation of more powerful home computers will certainly be able to create their own graphics from program and data supplied by the disks.

Getting it on paper

The display of hi-res screen information in attractive colours is only half the story. At some point we need to commit our screen-based graphics to paper, and this is where the plotter comes in. The quality of the image produced on the screen is primarily dependent on its resolution – the number of points you have to play with. Resolution, usually expressed in X and Y values, can be a slightly confusing term because its effect on the image produced depends to some extent on the orientation of the lines you are drawing. A lowish resolution which is fine for horizontal and vertical lines will produce a stepped result in any other direction.

This is where the plotter scores. The 'step' is purely a shortcoming of the screen which has to build its lines up from individual points (called pixels). The computer knows all the points in between and is capable of representing them on the plotter, with its higher resolution.

There are other ways of getting hard copy from the screen of a graphics system. You can photograph it, but there are problems of distortion due to screen curve, reflection and camera lens effects. There are more sophisticated photographic methods available to remove most of the distortion, but they cost many thousands of pounds.

An alternative is to use a matrix printer capable of producing a one for one match of the points on the screen for dots on the paper, creating an exact copy of everything on display in what is usually referred to as a **screen dump**. Because matrix printers build their images from dots this does not bypass the 'step' representation of all curves and sloping lines. However, it does allow a mix of text with the graphics.

Daisy wheel printers too can produce graphics, drawing by using the full stop and moving in increments. The result produced can be very good and have a resolution higher than many VDU's, but daisy wheel printers are very slow at this sort of work, which tends to rule them out for serious use. You're also liable to end up with a pile of duff wheels from which the full stop has broken off with over-use.

The plotter is an electromechanical device capable of putting pen to paper and drawing. There are two basic types: flat bed and drum. A **flat bed** plotter, as its name suggests, is a plotter on which the whole sheet is laid out flat and the pen is tracked across it in X and Y directions. A **drum** plotter has the paper fixed or supported on a drum. The pen moves in only the X axis, Y axis movement being produced by the drum moving the paper relative to the pen.

Both types of plotter can handle multiple and therefore multi-coloured pens. The flat bed plotters tend to produce better quality images but are not quite as fast as the drum. Although the plotters can draw on paper in X and Y directions, it is the computer that controls exactly where the pen should move. The plotter interprets when the computer sends it 1362 up and 534 left, that the pen moves up three inches and left two inches. The plotter

The graphics on the 64 are provided by the VIC II chip. High resolution or dot graphics are provided, with a resolution of 320 dots horizontally and 200 vertically. The 64 also has a powerful 'sprite' facility, which is the main source of animation of the 64. Unfortunately no commands are provided in the 64 BASIC to allow you to use these features easily, so you have to revert to the POKE command, as shown in the example programs. For dot graphics you can only plot individual points, so that drawing a line is quite slow. Exploring the possibilities of animation is something that really has to be done in assembler, and the moving wire box example is difficult to do in 64 BASIC for this reason, and the display you get will reflect this.

All the example programs have a standard subroutine which shows how to use an XY coordinate pair to find the bit corresponding to that point on the screen, which is then turned on. It is this calculation which takes the time; life would be much simpler if you could tell the 64 to plot a point or draw a line using normal screen coordinates:

```
PLOT 100,20 : LINE 20,0 TO 20,120
```

Of course there are many cartridges available which include similar commands, and adding one of these to your computer makes writing graphics programs much easier.

Example 5 does show how to draw a wire cube using a sprite, but it is still quite hard work. The power of the VIC II chip means that this sprite can be moved about the screen very easily, without the time penalty involved when re-drawing it in BASIC. Unfortunately, you still cannot vary its size (other than doubling it) or its orientation.

A 64 sprite is a block of 24 horizontal by 21 vertical dots which is defined by POKEing 63 bytes of data as shown in the example. V, V+1 and V+16 define the sprite's position (this is Sprite 0; other sprites – you can have up to 8 – use V+2 to V+15). V+21 turns on the sprite and V+39 sets its colour.

When the programs run you will have to wait a few seconds for the screen to clear.

itself has a great deal of intelligence and can scale its axes and interpret the numbers sent from the micro into pen positions.

At any one time the plotter needs to be aware of the exact position of the pen, and it achieves this by driving the pen with a **stepper motor**. The conventional motor which spins at its own speed until the current is removed is practically impossible to use for such applications. But when you apply the current to a stepper motor it only turns part of a revolution and to get it to move again you must re-apply the current. This creates a direct relationship between the number of pulses of current and the rotation of the motor shaft.

So when the stepper motor is installed with a plotter, movements of absolute size and direction are possible and the micro always knows precisely where the pen is. The motor's angular rotation step is fixed and they come in a variety of standard sizes with 'small' steppers of 1.8° and 5.0° and 'large' steppers 7.5°, 15.45° and 90°.

Most plotters have the ability to lift the pens off the paper – useful if you want to draw something in the middle of a page without a line crossing to the starting point. This poses problems to the writers of graphic software, as the pen lift command may be different for each plotter.

In **multi-coloured plotters** the different colours are produced by using one of two techniques. There may be multiple pens (up to eight are common), which the mobile pen holder picks up and uses, then swaps for another as required. Alternatively the pen head may rotate, and have all the colours mounted at one time.

As well as being able to change colour and lift the pen, some plotters can, on receipt of an ASCII character, reproduce it in a chosen typeface, scaled and rotated in the correct position. Another feature is to use the plotter in reverse - not as an output device but for input as a digitizer. This gives you two essential graphics peripherals for the price of one.

Copyright: Who Owns Software

The laws of copyright affect everybody who writes even the most elementary Basic program, as well as all those non-programmers who run other people's software on their machines. Copyright is the almost universally recognised right which the creator of a 'work' has in the subsequent use of that work, and it restricts the liberties that other people are allowed to take with it. In particular, in the United Kingdom, the law of copyright states that original literary works written by qualified persons are protected as copyright works and that reproducing or publishing that work in any material form is a restricted act.

The question for those concerned with protecting the efforts of hardworking programmers lies in the meaning of the words 'a work'. The relevant definitions in the Act are not restrictive but instead add extra meaning to the ordinary meaning of the words. A literary work is defined as including any written table or compilation, and 'writing' is defined so as to include typewriting, printing or any similar process. It is therefore possible for the courts to decide that a program listing printed by a line printer and reproduced in the pages of a magazine is a literary work. But it is not enough for a program to be declared a literary work for it to have copyright protection. It must be an *original* literary work – and the word is used in an unusual manner in the law of copyright. It does not mean unique or even first; the term 'original work' means original labour in that the author has to have expended a substantial amount of his or her own skills, knowledge, creativity, taste or judgement in producing the work. The author of a work is the person who originates it, from whom it comes.

The requirement that an original literary work must be written by a *qualified* person isn't a reference to an author's school certificates, but instead concerns the place where the person lives or is resident. All UK and Irish citizens wherever they may live in the world are qualified persons, so too are all people and companies resident in the UK. Foreigners can also gain protection under the Act if the country in which they are domiciled or resident is a party to the convention by which Britain gives copyright protection to the works produced in that country. All Western European countries, the Commonwealth and the United States have ratified the particular conventions and hence works written in those countries by non-UK citizens are protected under the Act.

If a computer listing is an original literary work written by a qualified person, then it will be protected under the copyright Act. This extends the protection of the Act to any process of reproducing a program or publishing it in any of the usual 'literary' ways, but what about the usual 'computer' ways of publishing the software? Unfortunately the Act does not explicitly cover the business of loading listings into a microcomputer, or of distributing copies on floppy disks, because microcomputers and floppy disks did not exist when it was being drafted.

These more complicated questions have to be decided by working from the basic principles of copyright law and equity. When programmers send a computer listing into a magazine for publication, he or she clearly intends for it to be read. The original ideas contained in the program can no longer be said to be confidential. But unless the original programmer has specifically waived copyright in the listing, it remains open to bring an action for infringement of copyright should the listing be reproduced or republished – in the same way that the author of a poem can also stop unauthorized reproduction and republishing. So if the listing was systematically photocopied from a copy of a magazine, the programmer could well have a case against anyone distributing the photocopies.

But the programmer would immediately be faced with a number of problems. The copier would first be able to say that he thought that by publishing the program in a magazine the programmer was placing the program listing in the public domain. This is often a reasonable assumption in view of the fact that many programmers who have programs published intend at the time to abandon copyright in the program. For this reason it is possible that the court would find that any reproduction which occurred before the service of legal proceedings on the copier were done innocently. In consequence the programmer would not be able to recover damages for the marketing and sale of the reproductions prior to service of proceedings, but instead would have to accept merely an account of profits. If the copier had given the reproductions away the programmer could recover nothing.

Several other questions of law would have to be decided by the court. The program listing in a computer or computer-related product (such as cassette tape or ROM chip) would be recorded in a form of notation not discernable to the human eye, and the court would have to decide whether this was a reproduction in a material form within the meaning of the Act. It would also have to reach a conclusion as to whether reproducing a digital electronic recording of a literary work was publishing within the meaning of the Act. Even in the age of Information Technology, these legal questions are both novel and serious.

These are matters which could occupy days of argument in expensive court proceedings, and the case could take something like two years to come up for hearing, because of the length of the queue. For this reason many prospective plaintiffs take a practical view of these affairs and never bring cases to trial. The alternative is to obtain interim relief: immediately upon issuing and serving the writ an opportunity arises for a plaintiff to seek an injunction pending the trial of the action. In practice if an injunction is obtained at this stage, the parties usually agree to let it continue indefinitely and never actually bring the case to court.

For this reason it would be unlikely that you would have to pay damages for typing in published programs and passing copies on to friends, but if you are going to copy a listing and sell recordings of it for profit,you are on shakier ground. If you merely took the ideas from the program – as opposed to copying the code – and developed them into a commercial package, you would have created a new work in which you would have the copyright. A great deal of commercial software has been developed like this.

CP/M: The Classic

Historically speaking, CP/M is the bridge between the world of the mainframe and that of the microcomputer, carrying across many of the old tried and tested concepts and introducing a few of its own. CP/M begin its career in 1973 as a very simple software routine to breathe life into one of the earliest systems built around a microprocessor, the Intel 8080, forerunner of the Z-80 used in computers like the ZX-81. Gary Kildall, its creator, developed it while a software consultant at Intel and it first ran on his own prototype 8" disk drive system.

At that time, disk-based micros were just appearing in the market place as an evolution from the paper tape and cassette reading minis. CP/M's strength, as hobbyists and manufacturers quickly found out, was that it enabled the wide ranging 'mish-mash' of hardware available in the mid-1970s to be assembled into working disk-based computers.

Originally the initials are reputed to have stood for Control Printer/Monitor, but as Gary Kildall moved the centre of operations from the shed at the bottom of his garden to found the grand-sounding Digital Research Inc (DRI), and as the software metamorphosed into a fully-fledged operating system, the acronym was revised to stand for **Control Program for Microcomputers**. Under any name, CP/M is an extremely important piece of software in the history of the micro.

Two features made CP/M a world-wide success in the second half of the 1970's. Firstly it was cheap – at less than $100 a copy it was nicely priced for the US hobbyist market. Secondly it was portable. As long as the processor was an 8080 or equivalent, the operating system could be set up relatively easily for any disk, console and printer configuration. This was achieved by keeping all the code involved in talking directly to the hardware in a separate module.

Few microcomputer manufacturers use the 8080 chip these days. The Zilog Z80 chip, which is an enhanced 8080 designed by ex-Intel employees who left to make some money for themselves, is much more popular. Because the Z80 is 8080 compatible, it can be used with CP/M, and this is the combination which became such a powerful force in microcomputing at the end of the seventies.

The list of Z80-based machines using CP/M is vast, and they all share the common feature of using disks rather than cassettes for backing store. Indeed one of CP/M's principle functions is to organise the contents of disks. Cassettes are only able to cope with a minimal level of control, usually no more than turning the motor on and off. Disks require a great deal more administration. CP/M has that job, and includes a number of routines, called utilities, to help users along the way.

The structure of CP/M

CP/M is made up of three modules:

The Console Command Processor (**CCP**).
The Basic Input/Output System (**BIOS**).
The Basic Disk Operating System (**BDOS**).

The 'basics' here are nothing to do with the language of that name, but are intended to indicate that the routines are dealing with the hardware at a very low level.

The **CCP** is the outer 'shell' of the operating system, with the job of testing the input at the command line, and turning it – where possible – into instructions which can be passed on to the code doing the work inside CP/M. This is sometimes known as 'parsing'. It can be viewed as the 'master' of the system, with the BIOS and BDOS as 'slaves'. The CCP might for example pass on to the slaves the high level command to rename a file (the command `REN`), and the BDOS in this case will do all the donkey work needed to convert this into the series of low-level operations required to achieve the required objective.

The **BDOS** is the logical core of CP/M which accepts the low-level commands that have been parsed by the CCP, and which executes them on an *imaginary machine*, standard across all CP/M systems.

The **BIOS** is the code needed to relate this imaginary machine to the real hardware.

Booting the system

Unlike machines which carry the operating system in ROM, CP/M systems may have no more than a tiny amount of ROM, just sufficient to trigger the loading of the operating system. There are several ways that this can be done, but it usually goes something like this:

Powering up the machine causes an automatic jump by the CPU's program counter to the beginning of the ROM area.
A short piece of code at the ROM jump address is executed, this is just enough to reach out to the first sector of the disk (usually in drive A), pull its contents into memory and execute them.
The code loaded from the disk contains a routine to pull in the entire operating system from one of the disk sectors – usually the whole of track 0.
The program counter is set to the beginning of the CCP, making the operating system 'go live'.

This rather magical process of breathing life into an inert box of chips is called **booting the system**, or 'bootstrapping', from its uncanny resemblance to the impossible task of pulling yourself up by your own bootlaces.

Once the disk has loaded the operating system, you do not immediately start programming in BASIC, as you would with many cassette-based machines. Instead, a message something like this will appear, indicating that CP/M is resident:

```
64K CP/M Version 2.2
Copyright 1982 by Digital Research
A>
```

The number, 2.2 in this case, is the release of the CP/M operating system currently loaded in the system. The version number may be three digits, or two digits and a letter, the last character being added by your computer supplier as the individual system implementation version

number. The first number is the version and the second a revision within that version. **Versions** are overhauls incorporating major changes of features, while **revisions** correct bugs or provide small improvements.

The `A>` is a prompt is inviting you to enter an instruction to CP/M. It might be to load another program from the disk, or to display a directory of all the files held on disk.

On some systems it is possible that this prompt will never appear, because CP/M allows the implementor to customise a system by setting it up to **autoload** the package required immediately upon loading the disk, without intervention from the user.

Resident and transient

CP/M has two levels of commands. The built-in or **resident commands** are held in RAM during the whole time the operating system is working, whereas the **transient commands** are called up from the disk as and when necessary, being discarded when their work is done.

When the CCP receives an order in response to the prompt, this will be executed immediately if it is a built-in command, otherwise the system assumes it to be a *transient*, and asks the BDOS to have a look on the disk for a file of that name (with the extension `.COM` identifying it as a command). It is then loaded into the area of memory kept available for programs in general, the Transient Program Area, or TPA.

The command `DIR`, for example, is a resident command which produces a directory of the files held on disk. Amongst these disk files you will usually find other utilities like `PIP.COM` and `STAT.COM`, whose functions include copying files from one disk to another, and supplying statistical information on system usage. `PIP` and `STAT` are *transient* commands.

The success of CP/M

Under CP/M the writer of applications software doesn't have to worry about the peculiarities of the hardware; the BIOS takes care of that. And the hardware manufacturers can put CP/M onto their own machines simply by concentrating on the BIOS, without having to worry about the rest of the operating system code. These two factors created the synergy – to use one of DRI's favorite words – that made CP/M the *de facto* operating system in spite of criticisms about 'machine-mindedness' and general lack of friendliness. That synergy has launched a vast number of CP/M systems, and an enormous variety of software – and having a choice of software is one of the most important factors when buying a system.

MP/M and CP/M plus

Since its first development CP/M has undergone a number of changes, in particular a not altogether successful attempt to put a multi-user system (MP/M) onto the Z-80. CP/M was designed as an operating system of 'ones' – one operator, one program, one printer, one computer. The idea of MP/M was to share the resources of the computer between a number of users. The only extras the system required over and above the obvious extra terminals and their ports, was extra RAM.

Unfortunately the first version of MP/M was treacherously inclined towards allowing users to crash each other's files. MP/M 2 followed, with file and record locking to prevent this, but although a creditable attempt to provide a multi-terminal, multi-tasking micro at a reasonable price, it made great demands on the processor and tended to be sluggish under load.

However the experience with MP/M proved invaluable when Digital Research came to tackle the 16-bit world, and as a spin-off inspired the creation of CP/M version 3, commonly known as CP/M-Plus, which embodies many MP/M characteristics in a single user system.

Like MP/M, CP/M-plus offers facilities for time stamping and archiving. This involves using a specially extended directory for keeping tabs on when files were created and updated. The command `INITDIR` reorganises the disk directory so that there is room to include the date and time stamps, `SET` is used to start up time stamping, and DATE is used to input the date and time.

The `SET` utility is also used for password protection, another new CP/M-plus facility inherited from MP/M. Each file can have its own password, which is set using the `SET` command. CP/M-plus even allows for a whole disk to be password protected.

CP/M-80 versions

The versions of CP/M-80 released to date are as follows:

1.3 - *First release*: Sequential files only.
1.4 - Bug correction revision.
2.0 - *Second release*: Random access files supported.
2.1 - Bug correction revision.
2.2 - Most commonly used version.
3.0 - Advanced MP/M-like version, mostly used on banked-switched machines.

The main advance in the transition from 1 to 2 was the ability to use random files and hard disks. Properly implemented, version 3 permits great improvements in disk access times, and improved directory listings.

16-bit CP/M

The redesign of CP/M to run on the new generation of 16-bit processors began inauspiciously with a version called CP/M-86, which was intended to be as like the 8-bit version as possible. It took little advantage of the possibilities offered by 16-bit hardware, and was overshadowed by the arrival of MSDOS (*qv*), which rode on the coat-tails of the best-selling IBM-PC.

The crucial loss of the IBM PC contract spurred Kildall and his team to produce something better than MSDOS. The result was Concurrent CP/M, a powerful multi-

tasking, single user system written to run on the 8086/8088 family of processors.

Concurrent CP/M: Four in one

Concurrent CP/M is a multitasking sixteen bit version of the old DRI workhorse built around the idea of 'virtual screens'. The concept requires some explanations. Virtual screens don't really exist, they subsist, so to speak, in a computer's imagination. Actually it isn't all that hard to grasp – most of us have already become familiar with something like it through television. While we watch one channel, we're aware that there are other channels available for display at the touch of a button. We have a sense that the physical screen is somehow separate from these channels, simply a vehicle through which the 'virtual channels' become apparent when we choose.

On these imaginary but immediately realisable screens a number of quite separate programs can be running at the same time (concurrently), each generating its own screen output. To make any particular screen appear, all you have to do is press a couple of keys.

You might, for example, want to edit one program while compiling another. You could start off the compilation, and then swop over to another virtual screen to start editing. On the IBM PC, for instance, each screen is accessed by pressing the control key and then 0,1,2 or 3. Compilation could be taking place on screen 0, and editing on screen 1, and so on. Every so often you could press control 0 and see how the compilation's going, and then return to the editing by pressing control 1.

Being able to run more than one program at once is known as multi-tasking. Don't confuse this with multi-user systems like MP/M, which are designed to allow a number of real screens, and thus users, to be connected up to a single processor.

Concurrency offers the freedom to use the micro's resources as and when you need them. But you don't get something for nothing. Because there's still just a single processor at work behind all this concurrency, switching about between the processes is bound to produce degradation in performance.

Two factors help to hold this degradation at bay. For a start. Concurrent CP/M is designed to run on the powerful 16-bit 8088 and 8086 processors. Most software doesn't take advantage of the power offered by 16-bits anyway, so there is a lot of slack to be absorbed by the processor-hungry concurrent system.

Secondly, Concurrent CP/M uses a RAM disk, called `MDISK`. This is another virtual device, this time a disk drive. The operating system sets up an area of memory, anything between 64K and 192K, to behave like a disk drive. It has the advantage over real disk drives of being very fast. One of the big hold-ups, often the biggest, when you run a piece of software is disk access. It takes several seconds for the disk to be started up, searched and then read to or written from. A RAMdisk (so-called because it uses Random Access Memory) offers almost instantaneous response.

Concurrency works best when the concurrent tasks are complementary. For example, it would be best if the program running on virtual screen 0, say, involved a lot of console work (a word processor, say), whilst the program on screen 1 was number crunching and writing the results to `MDISK`, and the program on screen 2 was simply printing a large text file from the real disk, using spare processor time when it was available and pausing when it wasn't. If all three programs were making demands on the real disk drives, the system would slow down appreciably.

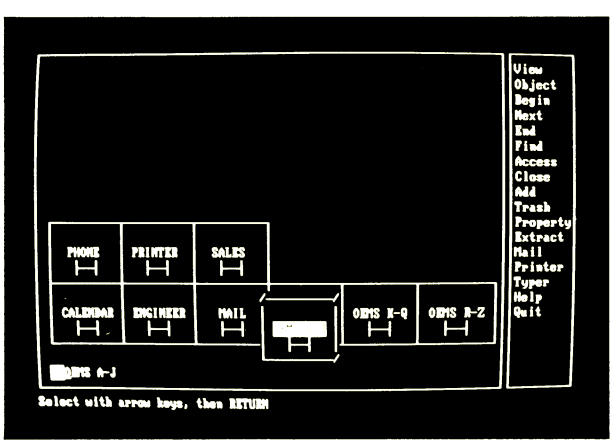
Sometimes a task produces a lot of console output which spills off a virtual screen and so is lost before you can switch over to see it. Concurrent CP/M provides facility called **buffered screen mode** for dealing with tasks like this. Supposing you were editing a file on screen 0, whilst a database program was generating a large report for display on screen 1. By using a command called `VCMODE`, the screens can be changed from dynamic into buffered mode. In dynamic mode screen data scrolls away into oblivion in the usual way. But buffered mode puts all displayed data into an area of memory (RAM, `MDISK` or a real disk), ready to scroll past again automatically next time you switch to that screen.

As with CP/M-plus each file can have a password and a time stamp to keep your files secure. The system also allows the implementor to include a status line, usually the 25th line (at the bottom of the screen), which supplies information on the state of the system and the time.

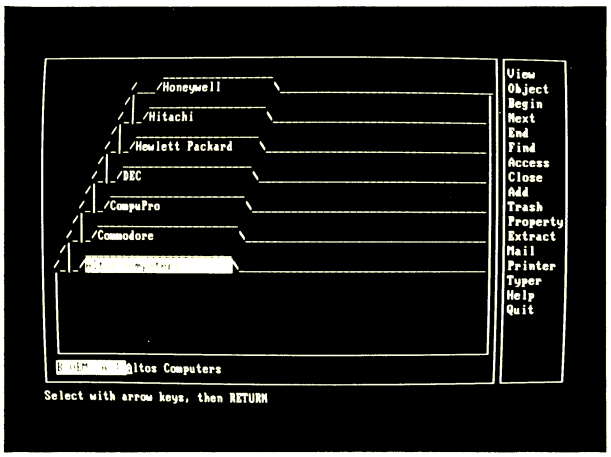
The price you pay for all this is memory – Concurrent CP/M needs lots of it. To supply all its concurrent facilities and the `MDISK` needs a minimum of 256 Kbytes of memory.

Personal-CP/M and VIP

A standard in personal computer operating systems may at last be approaching with the introduction of Personal-CP/M, the little brother of CP/M 2.2. It promises access to a whole warehouse of existing CP/M software at home computer prices. One of the main differences from 'ordinary' home computer operating systems is that there is no BASIC interpreter doubling as the user interface. Of course BASIC can still reside on disk or in ROM, but would



The VIP facility enables a set of 'filing drawers' containing data to be 'pulled out' by the user.



This shows a detail of the 'opened' file-drawer.

simply be another program to run, Similarly PASCAL, FORTH or whatever can all use the same consistent operating environment.

This new version of CP/M is 'burned' into a ROM and integrated into the initial design of a computer, so that manufacturers can put together CP/M machines that don't rely on disk drives. Theoretically even the smallest computers can be designed to take Personal-CP/M. The system requires 4 to 6.5K of ROM, and if the disk I/O functions are removed, this drops to between 2K and 4K, a minuscule memory requirement making it easy for ROM packs and game cartridges to be slotted into the 64K memory space of the Z80.

Digital Research envisage Personal-CP/M being used with another of their packages called VIP (Visual Information Processor). VIP has been called the 'downmarket Liza' because it offers a very simple pictorial interface which can be based on ordinary ASCII characters for systems lacking the refinements of graphics. Software houses will use it in conjunction with their own software packages, presenting the user with a consistent interface and removing the tedium of having to learn a complete new set of instructions every time a new package is used.

Personal-CP/M is not restricted to the Z80, it can be extended to work on the 8086 and 8088 16-bit chips. It will also support communications for data I/O. The device-dependent module of CP/M, the BIOS, could support communication protocols that would allow connection to other computers, modems and even cable TV.

Is it good enough?

Critics of CP/M say that the interface used by all the Digital Research operating systems is very largely inhuman, and ignores much of the research that has gone into the way people interact with machines. This is the new thinking which has inspired machines like Apple's Liza and Macintosh. But there is strength in the argument, put forward by Gary Kildall the designer of CP/M, that the job of an operating system is no more and no less than to operate the system.

If a well-upholstered front-end bristling with user-friendliness is what you want, says Gary Kildall, the applications packages can provide it. Alternatively CP/M makes provision for the CCP to be replaced by a more sophisticated user interface which you can design yourself if you're prepared to roll your sleeves up and get stuck into assembler.

Commercially available software to replace the CCP includes the UNIX-inspired MICROSHELL from New Generations Systems Inc, and Borland International's MENUMASTER, a very sophisticated user interface which is also a computer programming language in its own right.

Without aids like these the behavior of standard CP/M is certainly reminiscent of the days when professional low-level programming dominated microcomputing. CP/M-plus is undoubtedly an improvement on previous versions, but it still follows the laconic tradition which many commentators have found inappropriate to modern developments in microcomputing.

Data Compaction: The Squeeze On Text

There is a cynical saying among old micro hands that data always expands to fill the space available. The corollary is that as long as the capacity of your cassettes or disk drives remains finite it will never be enough. The alternative is to squash more data into the same space.

Compaction techniques take advantage of certain characteristics of data which enable it to be shortened without removing any of the meaning. The difficulty lies in devising a technique which can be used generally and not just for particular types of file.

One example of data compaction is in common use on micros. It is normal practise for BASIC interpreters to compact their source code by replacing all the reserved words with single characters not otherwise used in BASIC. For every occurrence of the reserved word, a unique single character called a **token** is substituted when saving to the disk. On loading again the reserved word is inserted whenever the token occurs.

The overhead in this system is the need to store a table of reserved words and their corresponding token symbols. This is unlikely to take up significantly more space as tables of reserved words are required by the BASIC interpreter anyway. Unfortunately this technique only works with languages with relatively few words. There are 300,000 words in English, not including 400,000 technical terms – far too many to give each one a symbol of its own, and there would be no room in a micro for the table!

If you could be certain the file was a text file written in English you could save some space by replacing common words like 'and', 'the', and 'there' with shorter symbols. But as the list of candidate words for compression is extended assumptions have to be made about the application. Common words used in letter writing (such as 'thank you', 'pleasure', 'enclosing'), are anything but common in, say, the writing of reports.

Obviously tokens allocated in place of words must not occur elsewhere in the text. A text file may contain any one of the ASCII character set, so none of these letters, or even the codes like Carriage Return, can be used as symbols. However, only 128 characters are defined in ASCII (using just seven bits of a byte), and a byte can hold up to 256 distinct values. The eighth bit is not used for characters in normal plain text systems (WORDSTAR is an awkward exception), so it can be pressed into service as a special symbol. On this basis a byte can either hold a single character (eighth bit off), or one of 128 possible tokens (eighth bit on).

Digraphs and trigraphs

The vocabulary will vary with the subject, so perhaps we need to go deeper into the text than the words if we are to devise a general method of compaction. A word is constructed from various combinations of letters: a combination of two letters is called a digraph, and a three-letter combination is known as a trigraph. So 'word' is formed from three digraphs 'wo', 'or' and 'rd', or two trigraphs 'wor' and 'ord'. It turns out – at least in English – that letter combinations like these occur generally across a wide variety of subject vocabularies. The most common digraphs include 'en', 'in', 'at', 'th', 'st', among many others. Common trigraphs include 'ing', 'ion', 'the', 'and'.

As far as the computer is concerned, the space between words is also a character; and as such it is obviously very common and should be included in any list of digraphs (for example, the digraphs 's plus space', 'e plus space', 't plus space'). There are very many other digraphs and trigraphs which *never* occur (such as qo, jj, quu, nnn), in fact about 40 per cent of possible digraphs do not occur in English, and something like 70 per cent of trigraphs never occur. This is called the redundancy of a language – combinations of letters with no function.

Only a small proportion of all the possible combinations are very common, and the most frequently occurring of these turn up in all sorts of words, quite independently of the subject matter. For example, 'and' occurs in such disparate words as 'and', 'sand', 'mandate', 'operand', 'standing', 'candid', 'andy pandy' and so on. The 128 available tokens can be used to represent similarly common combinations of letters.

This means that data can be compressed and subsequently expanded with the aid of a table of common digraphs and trigraphs and their representative tokens. An expansion/contraction program module can be added to the load and save commands of many editors and word processors. Alternatively, a separate program may be used to compress data once it has been saved.

The degree of compaction achieved depends on the care with which the 128 digraphs and trigraphs are chosen. For reasons of clarity we'll use the easily printed characters `*#!$%&+/` as our tokens; ignoring the fact that they are ASCII codes which would not, in real life, be available for this job which is normally done with the undefined characters in the ASCII range 129 - 255. Our tokens represent the digraphs and trigraphs as follows:

```
* = S followed by a space      % = RE
# = IN                         & = IVE
! = IT                         + = AND
$ = AT                         / = THE
```

The opening paragraph at the top of this page contains 287 characters. Using the above simple table the section changes to:

```
/% i*a cynical say#g among old micro h+*th$
 d$a alway*exp+*to fill / space available.
/ corollary i*th$ a*long a*/ capac!y of you
r cassette*or disk dr&*%ma#*f#!e ! will nev
er be enough. / altern$& i*to squash mo% d$
a #to / same space.
```

This represents a saving of 46 bytes, or 16 per cent. But with a careful choice of digraphs and trigraphs over a more representative cross-section of text, savings of more that 50 per cent are possible. One commercial package, CLIP, available under most of the common micro operating systems and running on some mainframes as well, regularly delivers a text compression of 60%, thus more than doubling the effective capacity of a disk. Considerable thought and ingenuity is necessary to achieve this level of efficiency, but with a little research and programming, it shouldn't be hard to come close to 50%.

```
READ FILE INTO BUFFER;

TP=0;
CP=0;
REPEAT
   TRIG.TABLE.PSN = 1;
   FOUND = FALSE;
   REPEAT
      IF TRIG.TABLE(TRIG.TABLE.PSN,TRIGRAPH)=BUFFER(TP,TP+1,TP+2) THEN
      BEGIN
         FOUND = TRUE;
         BUFFER(CP) = TRIG.TABLE(TRIG.TABLE.PSN,TOKEN);
         TP = TP + 3;
         CP = CP + 1;
      END;
      TRIG.TABLE.PSN = TRIG.TABLE.PSN + 1;
   UNTIL FOUND OR TRIG.TABLE.PSN > TRIG.TABLE.LENGTH;
   IF NOT FOUND THEN
   BEGIN
      DIG.TABLE.PSN = 1;
      REPEAT
         IF DIG.TABLE(DIG.TABLE.PSN,DIGRAPH)=BUFFER(TP,TP+1) THEN
         BEGIN
            FOUND = TRUE;
            BUFFER(CP) = DIG.TABLE(DIG.TABLE.PSN,TOKEN);
            TP = TP + 2;
            CP = CP + 1;
         END;
         DIG.TABLE.PSN = DIG.TABLE.PSN + 1;
      UNTIL FOUND OR DIG.TABLE.PSN > DIG.TABLE.LENGTH;
   END
   IF NOT FOUND THEN
   BEGIN
      BUFFER(CP) = BUFFER(TP);
      TP = TP + 1;
      CP = CP + 1;
   END
UNTIL TP = END OF FILE;

WRITE OUT TP;
WRITE OUT CP;
WRITE OUT COMPACTED FILE;

END OF COMPACTION PROGRAM
```

Fig. 1. Data compaction algorithm

```
READ IN TP,CP AND COMPACTED FILE;

REPEAT
   IF BUFFER(CP) = A CHARACTER THEN
   BEGIN
      BUFFER(TP) = THE CHARACTER;
      TP = TP - 1;
      CP = CP - 1;
   END ELSE
   BEGIN
      FOUND = FALSE;
      DIG.TABLE.PSN = 1;
      REPEAT
         IF DIG.TABLE(DIG.TABLE.PSN,TOKEN) = BUFFER(CP) THEN
         BEGIN
            FOUND = TRUE;
            BUFFER(TP,TP-1) = DIG.TABLE(DIG.TABLE.PSN,DIGRAPH);
            TP = TP - 2;
            CP = CP - 1;
         END;
         DIG.TABLE.PSN = DIG.TABLE.PSN + 1;
      UNTIL FOUND OR DIG.TABLE.PSN > DIG.TABLE LENGTH;
      IF NOT FOUND THEN
      BEGIN
         TRIG.TABLE.PSN = 1;
         REPEAT
            IF TRIG.TABLE(TRIG.TABLE.PSN,TOKEN) = BUFFER(CP) THEN
            BEGIN
               FOUND = TRUE;

BUFFER(TP,TP-1,TP-2)=TRIG.TABLE(TRIG.TABLE.PSN,TRIGRAPH);
               TP = TP - 3;
               CP = CP - 1;
            END;
            TRIG.TABLE.PSN = TRIG.TABLE.PSN + 1;
         UNTIL FOUND;
      END
   END
UNTIL CP = -1;

WRITE OUT TEXT FILE;

END OF EXPANSION PROGRAM
```

Fig. 2. Data expansion algorithm

Compression algorithms

The algorithms in Figures 1 and 2 are given in PDL – Program Design Language, to show one way of using this method of compacting and expanding text files. The algorithms require two tables, one of trigraphs and their corresponding tokens, and one for digraphs with their tokens.

The text file is read into a buffer in main memory, compacted and written out again. Two pointers are used: a text pointer (TP) which points to the next character to be examined; and a compaction pointer (CP) pointing to the position in the compacted file where the next character or token is to go. The compacted file over-writes the text file in memory (*see* Figure 1).

The expansion algorithm reads in a compacted file (along with the TP and CP values), expands all the tokens and then writes out a text file. In order to expand the file in the same buffer as the compacted file, the expansion must occur in reverse, that is, the last character compacted must be the first expanded – otherwise some of the compacted file will be overwritten every time a token is expanded. This is why the variables TP and CP are included in the compacted file. They are needed for an efficient expansion (*see* Figure 2).

Data Processing: A Personal Software System

Using a rough calculation your cheque book will contain something like 4,000 critical 'bits' of data by the time it is dispensed with. Include your bank statement, telephone and address book, car and insurance documents, receipts, bills, rent book and diary, and you arrive at a total of personal data amounting to many times that figure. Then you might add all the information which you, not always reliably, commit to your memory: a list of the books, records and videos you own, and so forth.

Looked at this way your whole way of life can come down to a series of Yes and No representations, a lot of ons and offs represented by those millions of bits of data. Co-ordinated by an information centre of some kind capable of supplying the data when you call it up, there's a chance that all this chaos of data could be shaped into some sort of order.

Perhaps this is the miracle of organisation that people hope they are getting when they buy a microcomputer. If it is, then disappointment usually follows. A microcomputer has the *potential* do all this, but two fundamental requirements have to be met first – and they seldom are:

> There has to be the right software on the system, exactly tailored to the data you need to collect and to the answers you are looking for.
> Somebody has to maintain the system, making sure that old data is deleted and new data is constantly fed in.

The second requirement obviously depends on you, and is there as a reminder that once the system is up and running the hard work has only just begun. But if the design of the system is right in the first place, keeping it up to date should be no more than a slog – it is the system *design* that needs genius! The impetus towards domestic filing and financial software offers BASIC beginners the opportunity to start developing their own programs, tailored to their specific needs. At the start these programs can be fairly simple.

There are a number of stages to developing a system like this:

> First set your objectives. What data are you going to be collecting, and how are you going to store it?
> Devise a means of entering the data, allowing for the fact that some of it will be text and some of it will be numbers. Think about what items of data relate to what other items.
> Lastly there will have to be some way of getting data out of the system in a useable format.

Having split the information centre into three main functions, the next stage in your **systems analysis** (as this operation is called) should be an examination of each function.

Data structure – There's no way of finding a means of entering data before we know what sort of data is to be entered. It could be financial: daily expenditure, regular outgoings, income and savings. What form will the daily outgoings data take? Probably the date (for example 'Mon, 13 July 1984'), a number indicating amount, and possibly another entry indicating where the money went.

The same procedure should be followed with other data. Perhaps you want to record all the telephone numbers of friends. You would need to enter a the name, and perhaps address, and a number.

BASIC has relatively simple ways of distinguishing between alphabetic, integer, floating point and other kinds of data – known, unsurprisingly, as **data types**. There are two main data types: numbers and strings, so called because they are literally a collection of alphanumeric characters strung together, like `JONES` or `A25FG` – anything you would put between quotation marks. Variables that will hold strings are nearly always suffixed by a dollar (`$`) sign, to distinguish them from numeric variables. Some versions of BASIC follow the tradition set by Microsoft BASIC (*qv*) of distinguishing between different types of number, particularly integers (whole numbers, no fractions or decimals, as opposed to 'real' numbers, the fractional quantities we have to deal with in real life), which are suffixed with the percent (`%`) sign.

Data entry – Having itemised the types of data that will be entered into the system, the next thing to establish is what you intend to do with it once it's there. One item of data, for example a friend's telephone number, might be best left well alone. Another, perhaps your daily outgoings, needs to be integrated with other financial data.

Reporting – Finally you want to present all the data you have collected and processed in a manner to which you are accustomed. This last stage is as crucial as the previous two. Though displaying information in an orderly and neat way may at first seem cosmetic, it determines how successful your information system has been.

This last stage is sometimes called report generation. When the system produces reports on, say your monthly financial status, it must do so in a way you understand. You can't just run the program and chuck a group of characters on to the screen. First they must be ordered and printed alongside suitable headings and labels.

All this speculation is leading towards a system specification. It supplies you with a good foundation for subsequent work; you can use it to plan a project, outlining the work that needs to be done and how long it should take.

The system specification is gradually honed and added to until it is time to turn it into a program listing for your microcomputer. Sadly this can't be done by magic – a lot of thought is needed. But at all times the specification should supply enough information to enable you to go on to the next stage of the project.

Having established the data that will be entered, processed and eventually displayed, you should be in a position to identify the programs that will need writing. One of the most obvious is a program to enable you to enter names, addresses and telephone numbers; then to recall the numbers according to name. A more sophisticated program might allow you to specify part of a name, or even an incorrect spelling (perhaps 'Browne' instead of 'Brown'), and still find the right number.

But this program could do more. It could be used in conjunction with a word processing package, to supply addresses when you are writing letters. It could be used in conjunction with a diary program, to supply a telephone number alongside a reminder that you have to call somebody. It could even be combined with a letters file *and* the diary to print an apologetic letter to your bank manager automatically every month.

It is this use of a combination, or **suite** of programs that turns your micro into an information centre, rather than just an expensive electronic note book. On the larger disk-based systems the commercial software houses are rapidly moving towards the idea of 'integrated software', but although there are many domestic filing and finance programs for home micros, there's little evidence of any integrated cassette-based systems on offer.

So, at the moment it is up to you to develop programs, and to integrate them, perhaps with a few packages that you have had to buy. These packages will provide programs (like a word processor, say) that are beyond the development expertise of all but the most experienced programmer.

Another approach to the whole problem, and one that is gaining popularity in professional computing circles, is to use one big database that contains all the information you use, rather than a set of individual files. You would probably be hard pushed to develop something like this for yourself, because the means used to enter data and retrieve information are extremely complicated. In effect, these sorts of systems integrate all data into a homogeneous mass that can be viewed in a number of different ways.

But whatever your filing problem, it is worth a serious search to find a commercial package to do your application. If you can find one, it will have numerous advantages. For instance, it will probably be written in assembly language and therefore will leave more memory for data and run faster than a BASIC program.

Most established home computers have one or two such programs available. They may only be 'toy' database and spreadsheet programs, but you may only have a 'toy' application. At only a few pounds for a cassette, they are among the best value buys in the software market.

If you're thinking of it, don't underestimate the difficulty of writing your own. It is essential that you understand the limitations of your hardware. For one thing, there has to be a balance between the features and facilities you want and need and the amount of memory you leave for data storage. Remember rule 50: if you can't store more than 50 items then stick to your old paper-based filing system.

30K of memory is pretty much the minimum memory for a useful system. A name, address and phone list might allow 200 bytes for each entry. If the program to handle the list took up 6K of a 30K machine, there's a maximum of 120 entries. This might be fine for your application. Try it on a 16K machine, though, and you'll get less than 50.

The other storage problem is that most home computers still rely on cassettes. Waiting for even a 1200 baud cassette to load can spoil many reasonable applications. Even if you have the patience, most cassette systems limit you to the machine's memory capacity, because the easiest way for a database program to work is to read all the data in to RAM, process it, and then write it all back out again to the tape.

Many BASICs are inadequate for processing data. They don't include simple statements to look through list of names and addresses to find particular words, and often get confused between capital and small letters. Many can't format output easily or allow enough control over what the user enters.

The flexibility of the language allows you to overcome any of these problems by writing your own BASIC routines – but then you run up against the fact that BASIC is a slow, interpreted language. Unless you plan for it, BASIC 'sort and search' routines are going to take longer than doing it by hand. Sorting (*qv*) is a great help, and avoids having to search by looking at every item in turn from the first one onwards. Imagine looking up a word in an unsorted edition of the Oxford English Dictionary!

Using a home computer for a serious application is pushing the machine to its limits. But it can be done. Provided you can think of a suitable application and you can buy or write the software to handle it, it could be an absorbing and successful venture.

The INPUT statement is not really suitable for inputting data in a serious software system; for example if an address contains a comma (92, High Street) then the INPUT statement will stop after it and give the error message EXTRA IGNORED. If you just press RETURN without entering any data, the program will stop! There is a way round this though:

```
100 OPEN 1,0
110 INPUT#1,A$
```

Doing this, though, requires writing your prompts using a separate PRINT statement; INPUT# does not even give a question mark.

You will find it helpful to write a general-purpose input routine using the GET statement. This involves quite a bit of work, but you can make it an all-singing-all-dancing routine, ignoring letters when you are inputting a number, only accepting a certain number of characters, and disabling the CLR key but still accepting cursor left and right, insert and delete.

On the output side, 64 BASIC does not have any statements to format the data printed on the screen or printer. If you have some numbers to print in a table you have to round them to the correct number of decimal places and pad them to the right length before you print them, as we suggest in the chapter on BASIC. This can make the business of producing a nicely formatted printout or screen display very awkward.

The disk drives support random access files (called relative files) but these only work on numeric key fields (the field used to identify each record). There is no indexed file structure such as ISAM; if you want to use an alphanumeric key field such as a person's name, you have to write some extra code to achieve it.

Databases: Structuring Information

The word 'database' is used in a baffling variety of ways in the computer business, but we all roughly know what it means – a collection of information arranged in such a way that we can easily recall specific parts of it as and when we need. This simple statement implies that there are three elements to any database system:

A method of storing the data.

A method of getting it out again.

Some kind of interface with the user that translates English-like requests into actions using the other two parts of the system.

Setting up a database will need some careful thought about what it is you want to store, and the sort of questions you're likely to ask. One of your first efforts might be a simple name and address file, where each entry looks something like this:

```
Name: Cynthia Muldragon
Address: The Embers, Smoke Rd, Northampton
Phone: 0896-33214
```

Each line of this entry describes a different attribute of a particular case, and is usually known as a 'field'. This and a collection of similar entries, or 'records' as they are usually called, make up a file. Every record in the file will have the same set of fields – but the data will differ from record to record.

We'll be entitled to give this file the grand name of 'database' – albeit a very simple one – once we've included the software to add new records, alter existing records, and find and display any particular record. There will have to be some method of letting the system know which record we want to look at, and a simple way of doing this might be to enter what we know to be the contents of one of the fields. In this case we would put in a command something like this:

```
DISPLAY FOR NAME = 'Cynthia Muldragon'
```

and expect the system to show us the rest of the record. Used like this the name field is more than pure data, it is the 'handle' we use to get at the record. Fields treated like this are called 'key fields' or just 'keys'.

This simple form of database is known as a 'flat file', because the string of records is thought of as being strictly two-dimensional. More complicated database systems may be multi-dimensional, seeking data across several files and bringing it together to answer your queries.

Storing data

A file is similar to an array, with each entry, or 'record' as it is usually called, representing an element. The essential difference is not that files are kept on backing store while arrays (usually) stay in core memory, but that the length of the file is undefined. In theory there is no limit to the size of a file, other than that imposed by the operating system and the physical limitations of the cassette or disk on which it is kept.

There are four main types of file:

Serial files store data records like beads on a string. To get to any particular record you (or the program) have to start at the beginning of the string and work your way through until you find what you are looking for.

Sequential files are serial files sorted into a recognisable order for faster access.

Fixed length record files store their records in such a way that the application program can reach for them directly without reference to the other records.

Random access files are files in which the position of any record can be determined instantly by the operating system.

Only the first two types are usually available on cassette-based computer systems. In fixed length record files, as the name implies, all entries are guaranteed to be the same size. Files of this kind – and random access files, which work on a very similar principle but are controlled directly by the operating system – are a faster way of getting at records, although they can take up a lot of space because of the way data is 'padded' to fit the record size, usually by blank spaces. Calling them 'random access files' is really a misnomer, as you seldom want to treat a file like a bran tub and draw out records at random!

We speak of the file as containing 'records' and 'fields', but in reality all we have is a string of bit patterns arranged into bytes. The system needs to know how to interpret these bytes to reclaim them as data, and in a practical system it would also be useful to know how to display the data on the screen.

In some database managers the rules defining the shape of the records and the kind of data they contain are held in a special header record at the beginning of each data file. This keeps everything neatly together, but there are advantages in using a separate file for data definition.

In some of the most flexible systems a single logical flat file is physically implemented as three files:

The data file which contains the data.

The data definition file which defines how the data is to be interpreted.

The format file which defines how each record is to be displayed on the screen.

Data types recognised by the database system may be alphanumeric, alpha only, numeric only, money and date. The forms file may also keep track of maximum and minimum limits, so that the system can query any attempt to enter data outside this range. It is useful to be able to set default values that the system will enter automatically if you leave a field blank. One common use for ranging and defaults is in a date field, where you might set up a query covering years outside the range 1970 to 2000, setting the default to the current year if no figure is entered.

Where several files are used to define a single set of

data there is often a fourth all-embracing file called the **data dictionary**, which keeps track of the relationship between all the format, description and data files you are using.

As an alternative to this sort of structure, a system may keep all its data as coded ASCII strings. Fields are not fixed length, with the great advantage that no trailing spaces unnecessarily pad out the disk space. There are disadvantages to this approach, though. Individual records can never be revised, so that amendment becomes a two-stage process of marking the record as no longer valid and then appending the rewritten version to an overflow file, which will be later incorporated into the main file by an explicit 'tidying' process.

Fixed fields are conceptually simpler, although much space tends to be wasted padding out fields with blanks, with consequent increase in the search times.

Talking to the system

The simplest task a database performs is to display a particular record or set of records at the request of the user. Most database systems invite the user to define a 'model' which the software then tries to match against all the records, masking out the ones that don't fit and finding the ones that do. The model entered into the command line might look something like this:

```
'DISPLAY LEDGER WHERE ACCOUNT = "SMITH01"'.
```

Here `SMITH01` is an exact model, but logical relations are normally also allowed – for example:

```
'DISPLAY DEBTORS WHERE OWING > 450'.
```

Usually database systems understand ambiguous models, where a 'wildcard' asterisk means 'and any other trailing characters'. So 'BUS*' will find 'BUS-STOP', 'BUSTARD' and 'BUSINESS & OFFICE MANAGEMENT WEEKLY'. This very useful facility can be extended in some systems to permit searches for internal substrings. The command:

```
LOCATE FOR 'BUS' $ name
```

might search the database for any record where the field called 'name' contains those three letters juxtaposed in that order. One particularly sophisticated British database system goes even further: not only can you search for internal substrings with the intuitive model form '*arch*' to match 'Middlemarch', 'Monarch', 'Barchester' and so forth; a search is also possible in which the system looks for a lead-in consonant pattern (taking into account any initial vowel). To achieve this the model is prefixed with `@@`, so that `@@Smith` will conveniently track down any Smithes, Smythes and Smithers lurking around your files.

Some packages have their own built-in enquiry languages, possibly supplemented by code generators. A sort facility is often included that may completely rearrange the records, or possibly only create a file of sorted file pointers, leaving the original database untouched. This has the advantage that several notional versions of the same database may be stored without having to duplicate all the data.

Producing reports

So far we have been discussing the business of interrogating the database as a piecemeal interactive operation. But database systems can usually be persuaded to produce regular reports to the screen or to the printer on all or part of their contents, possibly performing calculations on the data at the same time.

The reports can be formatted to the user's requirements – often quite elaborately, with page headings and footings, and titles for the columns and rows. A package with a particularly fancy report generator may use its own built-in screen editor to draw an approximation of the final layout before taking you through a series of question and answer to fill in the precise details of how you want the report to look.

Changing your mind

Only the very experienced design a database correctly first time, but even so your needs are bound to change, and there will come a time when you wish your files were arranged differently. All good packages acknowledge this. Typically a command called something like REOG or MODIFY STRUCTURE will take you back into the screen editor and lead you through the process of reordering, adding or deleting fields. Altering field descriptions may be more complicated, involving writing out the file in standard ASCII format and reading it in again.

Multi-dimensional views

When sketching the outline for your database you may begin with a diagram on a piece of paper. But paper is essentially a two-dimensional tool – in real life data is likely to be multi-dimensional, which is another way of saying that you can take several 'views' of it.

This idea of different views is actually a familiar one – you could think, for instance, of a group of wedding guests. The photographer will want to see the group as a whole, of course; he'll also want to take pictures of parts of the group, such as the bride's family, or the attendants. These 'views' just involve subsets of the original group, using two dimensions. But if, for instance, you were in charge of the parking space at the reception, you'd also want to know which guests had cars, their licence plate numbers, and in what order they'd be likely to leave.

Such information doesn't fit naturally into the structure of a guest list, yet it is clearly related to it. If you were drawing a data structure like this on paper you'd probably make two lists, perhaps with arrows from the name of each guest to his or her car on your car list.

Database philosophies

Different forms of database organisation are simply an attempt to represent these 'views' of the data in a form which can be manipulated by a computer system. The ease with which the data can be kept accurately, and the extent to which it can easily be retrieved in an appropriate

form, depend directly on the type of data model used. Speed of operation depends on the kind of model and on the way in which the model is implemented.

People use several kinds of data model when implementing database management systems. Apart from the flat file, the most common are: the *hierarchical*, the *network*, and the *relational* models.

The **hierarchical model** views data as stored in a tree-like structure, rather like a family tree or pedigree. It works quite well for information which has that kind of structure, but not so well for less structured information, or where you want to get views of the information which cut across the 'branches' of a tree .

The **network model**, as its name implies (cynics call it the spider's web approach), regards all information as potentially inter-related. This model can be extremely powerful, if complex.

The **relational model** uses a simple tabular structure, with each record – person, car – forming a row, and the characteristics – name, car number – forming the columns. Each distinct group of information – guests and car-owners in our example – forms a separate relation.

Both hierarchy and network models, unlike the relational model, include 'navigational' information about the relationships between different parts of the structure as part of the model. So it can be quite easy to get information updated incorrectly – a situation which is sometimes called 'losing data integrity'.

Relational databases

The data in a relational database is 'pure' – unmixed with any structuring information, and the breakdown into simple relations can be quite well represented on a micro by a set of flat files. For these two reasons the relational approach has become very popular on the larger micros, and is worth exploring in more depth here.

The concept of a relational database was developed by E.F Codd of IBM, and came to the public notice in the early 1970s. It has all the classic symptoms of a 'brilliant idea': it is essentially simple to the point of common sense, but actually very tricky to explain fully – its formalisation required the invention of new branch of mathematics called relational calculus.

Codd's insight was that there are only a finite number of sensible questions that can be asked of any given body of information. One implication of this is that there is something about the data that defines the set of askable questions. In instituting a database we tend to set out data in a way that anticipates particular questions we may want to ask, but that obscures the validity of other quite sensible questions we may want to ask later.

Codd showed that there was a universal set of rules we can apply to any body of information that tells us how to lay it out as a set of tables, or 'relations' in a way that crystallises rather than obscures the relationship between the data and the set of all the sensible questions we can ask.

The tables must be strictly rectangular – each row must have the same number and size of column as every other, so you can't include varying amounts of information in a single relation. For instance, if you wanted to keep information about a family (such as their address) and about each member (say the sex, age, and school of each child) you would need one relation containing the family records, referencing another relation containing the records for each child.

The relation must not include any redundant data – in its purest form this means that the relation must have a single unique key, and the non-key items in the record must depend on 'the key, the whole key, and nothing but the key'. This lack of redundancy (arrived at by a process Codd called 'normalisation') is what gives the relational model its bias towards accurate data, since the absence of repeated items means that any one data value needs to be updated only once.

In addition to the usual 'flat file' manipulation commands such as sort, append and copy, relationally inspired systems use several powerful commands to manipulate multiple data files.

Compare allows you to make 'matching' or 'not matching' comparisons between individual fields in a pair of data files. Records that meet the criteria are accumulated in a third 'result' file.

Project creates a new data file from the current one containing only certain specified fields. Normally used prior to the JOIN command.

Join produces a result file comprising fields items from two separate data files.

Post updates the values contained in specific fields in data file A depending on values found in comparable fields in data file B. A simple and direct method of, for example, updating balances to a sales ledger after a payment run.

These operations are extremely powerful, but they can consume large amounts of processing power – and this is the catch! Applied to micro-software the description 'relational' tells you negative things about the product. There will be no pointers joining fields across files and nothing in the database except the data itself and rules for coding and decoding it.

Several micro database systems manage this much, but getting at the records requires some non-relational crutches like indexing and sorting. In a true relational database system there will be no need to sort files because the records are never stored in any particular order. In fact they aren't strictly speaking filed, but rather 'tabled', with any given set of data being accessible to inquiring software procedures by the simple process of inspecting the contents.

Simple conceptually, that is. Implementing a true contents-accessible file system is hard enough on a mainframe, and on a micro is out of the question. Several packages allow sub-string searches of the records, but only in ways that draw the user's attention forcibly to the sequential (ie very non-relational) ordering of the records.

Search times become lengthy as the file become large.

But as long as a database system does what you need, it is probably best not to worry about the philosophy of the underlying model. While the relational approach has a lot to offer in ensuring that your data is accurate and that you can get at it as you wish, it is very difficult to implement, and there are also circumstances in which removing redundancy from data distorts it to such a degree as to make the cure worse than the disease, in terms of ease of use, speed of response, or both.

Buying a database

We have given a lot of space to the relational model because of its theoretical interest and its potential on future machines. Don't worry if the system you settle for uses a different approach – as indeed it will certainly have to if you are relying on cassettes for storage. Some experts think that the most appropriate use for the relational model is as a tool for analysing the data structures before implementing them. After analysis the particular database is set up using hierarchical, network or whatever model is the best for the circumstances.

The important thing when buying a database system is to make sure that you and the dealer understand what you need it for. Preferably see it in action on a machine like your own, and check the following items:

- How many records, and what size of records, it can store?
- How fast it is at finding records?
- Is it capable of multi-file accessing, as in a relational system?
- Can a database can be reorganised easily once it has been set up?

Don't forget when costing your database system to include the time and trouble involved in getting to understand it, and in typing data into it. If you have a database system already and are upgrading to something more sophisticated, do make sure that you can carry your existing data across without having to type it all in again. A surprising number of typists can tell stories of long weeks spent entering data into a computer – copy-typed from a computer print-out!

Going it alone

When you see the prices charged for database systems you may be tempted to think of programming your own in BASIC. It can be done, but if you need something more than a very simple flat file system, don't be too optimistic about the time it will take. File handling is not BASIC's strong suit, and much programming effort will have to go into getting round its limitations.

Editors: Tools To Handle Text

With the proper software, micros are very good at storing information in the form of plain text and subsequently recalling it for editing. This class of software forms the basis of word processors, but is also common in the more humble text editors – text handling tools of modest talents normally used for creating and modifying program source code. Two kinds of text editor are available on modern micros: line editors and screen editors – fully-fledged word processing packages are a simply a development and enhancement of screen editors.

Line editors

A line editor only lets you have access to one line of text at a time, while a screen editor lets you roam more or less freely across the whole (or part of) the screen. The typical home micro does not usually include a true screen editor, but will almost certainly have a line editing system built into its ROM for programming in BASIC.

The fact that line editors appear on so many machines has much to do with the way BASIC (*qv*) was first offered on a timesharing basis at Dartmouth College in the US. Line editing programs are quite small and fairly simple in construction: they met the early constraints of both having to fit into a small amount of memory and being able to operate reliably when used by many people writing BASIC programs on a teletypewriter. Despite technological advances such as the widespread use of television and monitor screens, this tradition has been carried on into the low-cost, BASIC-oriented machines available today.

A common feature of line editors is the way they label each line with a unique identifier, typically a line number – 10, 20, 30, 40 and so on. So, the command `LIST 30` would cause the line editor to look for a line preceded by the number 30, and then to display it on the computer screen. Of course, you may `LIST` as many lines as you like, but only one may be worked on at any given moment. This single line rule also applies to pseudo-screen editors like the BBC micro and Sinclair systems which allow you to list a set of lines and then to use the cursor control keys to select a line for editing. For all this complex screen manoeuvering you still only get a single line to work on.

Screen editors

Because BASIC is a line oriented programming language, a line editor will usually provide adequate editing facilities. The situation changes radically with languages such as PASCAL, where lines are not normally numbered and layout is free-form, depending on arbitrary indentation conventions to make the structure clear.

Screen editors give the user the opportunity to treat the whole screen as an erasable slate. Everything that appears on the display (with the possible exception of status lines and menus) can be edited in situ. Whereas a line editor of some sort of is normally bundled in with the system, screen editors usually have to be bought separately. To help you choose one, here are some features to look for.

There should be easy **insertion** and **deletion** of text without having to memorise 20 or more commands and endless sequences of menu selections.

You must be able to **move around the screen** to any point as easily as possible.

Many screen editors explicitly display otherwise invisible 'characters' like carriage returns, and this can be helpful as long as the screen isn't unnecessarily cluttered with control characters.

Although it may seem like a luxury at first, once you become used to screen-based text editing you will start wanting to move blocks of text around – **block manipulation**. A screen editor must provide the user with some way of indicating the beginning and end of a block that is to be moved, some packages have confusing procedures which can lead you to think that you have 'lost' the paragraph if you don't follow the rules exactly.

An ideal editor should also be able to include one file inside another, making it easy to manipulate libraries of subroutines (*see Programming*).

A **global search and replace** command is essential. It allows the user to replace any specified word with another *throughout the text file*. This is invaluable for translating programs for different machines requiring simple but seemingly endless syntax changes.

The UCSD p-system (*qv*) has one of the most feature-full screen editors of all, and is rated highly enough by some enthusiasts to be used not only for writing program source code (its intended purpose) but also as a word processor. There can be little doubt that specially written word processing packages offer a more tailored environment for the extensive entry and manipulation of text files.

Like all Commodore machines, the 64 has a very good full-screen editor. When keying in a line, you can move the cursor backwards over the line as well as inserting and deleting characters. If you have a number of similar lines to enter, you can type in one line, then move the cursor up the screen, change the line number, and edit the line as necessary. Unlike some other machines, there is no need to move the cursor to the end of the line to enter the edited text.

Education: Learning With Computers

There are several reasons why a microcomputer makes a good home-teacher of both programming and other skills. It has endless patience, it won't make the user feel stupid, and will cater for people across a wide range of intelligence and experience. The pupil always has the computer's undivided attention – but unlike books and television, learning from a computer is a constantly active experience since the user must be continually responding to keep computer moving. Properly programmed, a computer allows you to tap into the knowledge of experts in whatever field you are studying (*see Expert systems*).

In spite of this, good educational software is taking its time filtering through to the domestic user. The computer programmers who have traditionally written software lack experience in teaching, and as a result much of the early educational software was based on simple question-and-answer responses: more akin to the training of laboratory rats than of humans! Software manufacturers, however, know that there is a very profitable market in this area, and interesting educational software is of considerable commercial importance.

Where to start

The golden rule is to avoid buying software 'blind' whenever you possibly can; this applies particularly to educational software because of the predominance of ill-conceived and poorly designed packages. Go along to the shop with a clear idea of what you want and, since you and your family will be using the software together, don't just stand back and allow the dealer to spoon-feed you with a canned demonstration – make sure you can have a go with it together!

Is the program clearly simple to use? If not, will you have the time to sit and work through it with with your family?

Is the program repetitive? You obviously want to use the program more than once, but the usual yes/no multiple choice option programs soon become boring (hopefully because you have learned all the answers!). Look for programs demanding some degree of creativity in use, so that you and your family can develop your own ideas.

If there are several people in the family will they all be able to use the program as a group, or will they have to take turns? This is important if you want to keep the coefficient of friction to a minimum.

How is the package is presented, what sort of backup material comes with it or is available?

If you are not happy with the idea of buying software in a shop or by mail order, or if you just want advice, then there are several things you can do. A good place to start is at your local school. There are now microcomputers in almost all secondary and many primary schools, and there should be a least a couple of teachers who can advise you – especially if you happen to have the same type of machine as the school.

The most common micros in schools in the United Kingdom are Research Machines, Commodore, Sinclair, Tandy, Apple and the BBC. There are two main user groups in education. MUSE, Microcomputer Users in Secondary Education, and MAPE, Microcomputers and Primary Education. Both of these groups have a mixed membership of parents, teachers and manufacturers, and will be able to give you useful advice. Several parent/teacher associations have set up their own user groups and you will probably be able to exchange programs with fellow users. You might also contact the Computer Education Group, affiliated to the British Computer Society and open to anyone interested in computer education.

The main problem with using schools as a source of educational software is that a lot of the software will be tied to school curricula, and will probably be designed for use in groups with a teacher. Because of this you may also find the software expensive, as it may include a lot of background material and may itself be fairly sophisticated.

Several reputable educational publishers are now producing educational software for schools and will send packages directly to you by mail order. Programs from this source are usually worth looking at since they are usually written by teachers. Most educational publishers are now considering the home market as a commercial proposition.

A final source of useful programs will be the user group (*qv*) for your machine. Your local group may well have members who share your interests.

CAL and training

Training using computers already has an established history in the commercial world, and even a nickname. CAL stands for 'Computer-Aided Learning', a label for the general principle of making interaction with a computer an integral part of a training. CAL (sometimes known as CAI – Computer Aided Instruction) has its origins in training computer programmers, and was extended – particularly in the military services – to training the operators of complex and expensive equipment concerning tanks and radar systems.

Writing training software is expensive and time consuming. Traditionally, CAL has been almost entirely based on mainframes but computer aided learning packages are becoming available for micros, and with the growing population of home micro owners, course development is showing signs of becoming rapidly cheaper.

Silicon teaching scores

There are many advantages in using computers in training:

Computers are becoming more readily available than human teachers, allowing pupils to learn when they want to learn, rather than in a class at times arranged to suit the teachers.

The lessons taught by computers have a readily monitored consistency, according no special advantages to favoured pupils, and never having an off

day. By removing these intangibles the quality of the course can be far more easily assessed.

Conventional education tends to put the student in a passive role in relation to the teacher whereas CAL depends on active participation, inviting constant response from the student as an individual.

Because replies are typed in, the computer can very easily keep a record of how the student is doing, as well as being able quickly to compute overall performance and particular strengths and weaknesses.

CAL is 'distance learning'. The course can come to the student rather than the other way round, so cutting travel expenses and time.

One thing that has helped to keep down the cost of this kind of software is the advent of authoring languages. Designed for educators who want to write teaching programs, authoring languages put the emphasis on the teaching rather than the programming.

PLATO

Many of the authoring languages are based on PLATO, the result of a project started at the University of Illinois in 1962 with backing from Control Data and the National Science Foundation. PLATO was a mainframe product until 1981, when MICROPLATO was launched.

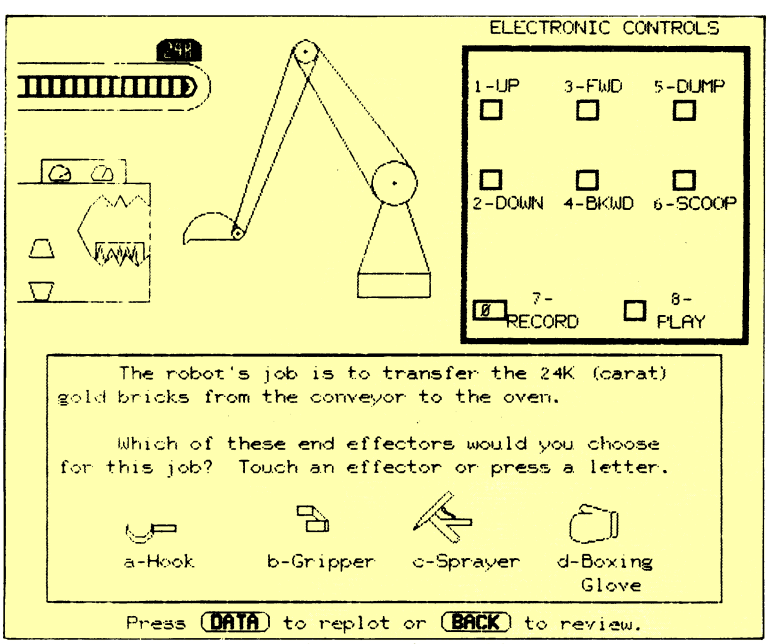
A touch-sensitive screen to input information and answer questions makes PLATO fun and easy to use. It's not one package but several, each for a different application and in creating a training course, the course designer can use one or all of the following application models:

The tutorial lesson model is used to write simple lessons using text and graphics. The author can choose from four different types of questions to set the student: multiple choice, true or false, short answers, or touch choice. This package also contains options for indexes and branching.

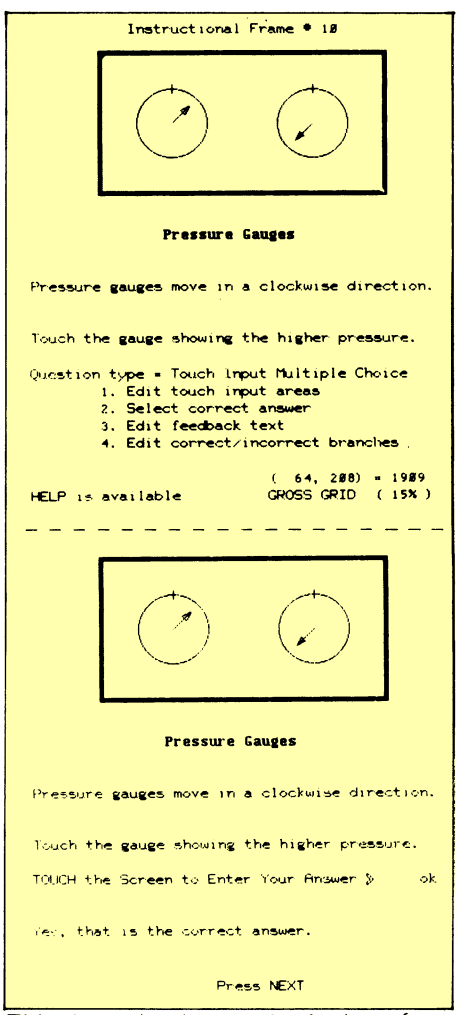
The situation simulation model really makes the most of the touch sensitive screen. This model also allows the author to give points according to the choices made by the student.

The drill and practice model is used to test the experience gained on the other models.

The interactive training model is designed to cope with a wider variety of responses from the student, who can fill in blanks or write essay-style answers, as well as making the sort of short answers PLATO normally caters for.



A simple plato program, using a touch-sensitive screen, which teaches users how to train robots for difficult tasks.



This shows (top) the author's view of a training program, and (bottom) the students final view.

PILOT

Unlike PLATO, PILOT is available on a variety of different hardware, and can be found at prices that individual micro-owners can afford.

PILOT has features which:

- Check the student's input against pre-defined answers (answer matching).
- Control branching as a result of the user's response.
- Keep records of student scores, time taken to answer input, and so on.
- Tackle arithmetic computation
- Manage disk or cassette files.
- Operate auxiliary devices such as video-disk or audio tape machines.

PILOT is an acronym for Programmed Inquiry, Learning or Teaching, and came out of the University of California in San Francisco in the early seventies. Its original eight single-letter commands have been greatly extended in the various versions developed to run on a large variety of micros, and today it is acknowledged as one of the leading authoring languages.

Inspired by the hardware facilities available on today's low-cost micros, PILOT extensions now include editing routines which not only make program writing simpler, but which are also able to incorporate attention-getting special effects of presentation (sound, graphics and so forth) to help to keep up the student's interest.

The M command (standing for MATCH) is central to the idea of PILOT. It allows the program author to define how the program will respond to the students answers, and can anticipate synonyms, inspect the sequence of responses and cope with negatives (*eg.* Any answer except 'butterfly' means repeat the first sequence). Special controlling characters extend the use of M when a complex match is required, as in these examples:

```
m:elephant          (match against 'elephant')
m:elephant & monkey (match against 'elephant' followed by 'monkey')
m:elephant!monkey   (match against 'elephant' or 'monkey')
ms:elephant         (allows a match against a mis-spelling even if
                     a character of the response doesn't fit)
m:ele**ant          (matches with any character in the * position)
mj:                 (used to jump to the next match instruction if
                     a match fails)
m,%elephant%        (ensures that 'not an elephant' is not accepted)
```

The match facility works by studying the roots of the words, it can throw up some unexpected results in use and experimentation is vital.

MICROTEXT

MICROTEXT is easier to use than PILOT, because it does away with the statement style which languages like PILOT have inherited from BASIC. It includes sophisticated functions using variables, branching and flow charts to help the experienced programmer create a more elegant course.

In structure it has a lot in common with frame-based systems like PRESTEL and TELETEXT, and MICROTEXT graphics are of the relatively unsophisticated teletext type. To get round this deficiency the NPL (who created the language) has written a version which will allow the package to interface with a video disk or tape. Whenever you need an especially clear picture for training, the 'courseware' will automatically switch on the video. The package can also be used to control a slide projector.

To insert a video section into the course, the author would type in a simple command such as `$play`, followed by the appropriate frame numbers. This would, of course, be invisible to the student who would just see the video picture on the screen.

This combination of computer-based training and video training opens up an enormous number of applications previously closed to computer software – for instance, a comprehensive home doctor package which would not only describe symptoms with text, but which could also show visual examples may well have some application in the future.

Other authoring languages

Other authoring systems of a similar degree of sophistication as PLATO include WISE produced by Wicat, Plessey's TICCIT and the REGENCY system designed by Redifusion Simulation. Further down the price range with PILOT and MICROTEXT are EASYWRITER from the National Computing Centre, STAF from Leeds University, and the Tandy AUTHOR I system. IBM has announced an authoring package called PRIVATE TUTOR for the PC, and this has already been used to produce commercial training programs including a course on the the DOS 2 operating system.

The future

Authoring Languages such as PILOT look like playing an important part in building the bridge between computers and the new video disk technology. Industrial videodisk players have been developed with interfaces that allow the computer and the video to share the same screen, and which even allow both signals to appear simultaneously, superimposing text on pictures, or overlaying diagrams and photographs. But at present the cost of creating master disks is measured in thousands of pounds, giving the system little scope for development except by companies with large R and D budgets who clearly hope to scoop the market for commercial training packages.

Much of the excitement about CAL has been stirred up by teachers and technicians fascinated by the technical possibilities. But some authorities argue that situations where a computer is genuinely useful as a teaching tool are fewer than teachers tend to think. It might be tempting, for example, to use PILOT to drill a class in French grammar, using a single computer as an automated blackboard, but experience shows that 'pack-drill' teaching of this sort produces generally poor results.

Floppy Disks: Drive-In Data

A gramophone record allows you to drop the pickup on any track by sight; with a tape you have to wind through sequentially to get to the next tune you want to play. The idea of using a removable disk with just this kind of 'random access' facility for recording and reproducing digital data was rejected by IBM in 1960, and mainframe technology grew up through the sixties using high-speed digital tape recorders as backing store.

But the appeal of random access was not lost on mainframe manufacturers, and towards the end of the 60's they developed large magnetic disk subsystems consisting of a number of flat circular plates called platters, coated on both sides with a magnetically sensitive material. A set of read/write heads rather like multiple gramophone pickup arms was interleaved with the platters and disengaged when the power was removed so that the whole stack (called a disk pack) could be detached and replaced.

Unlike a gramophone record there was no groove. The data was recorded on concentric rings of the medium called tracks, each track being divided into sectors that could be treated by the system as so many snippets of magnetic tape. Normally there would be no physical delineation of these tracks and sectors: the system would create the tracks by moving the heads in pre-defined steps, and divide each track into sectors by a preliminary writing process called 'formatting' which marked them out magnetically on the disk's surface.

Apart from the cost, one great disadvantage of this kind of storage was that dust particles between the head and the magnetic surface were likely to send the head ploughing into the platter. When this happened the head, the platter and the data on it would have to be replaced, so devices of this type had to be operated in a scrupulously clean and monitored atmosphere.

The technology developed in two directions. By settling for slower access speed and fitting less data onto the medium it was possible to produce small units using single, cheap replaceable platters that needed no air conditioning.

Because they did not have to meet the (literally) rigid requirements of multi-platter disk packs these replaceable disks of coated plastic protected by a fibre jacket came to be known as floppies. Floppies began to appear as a method of transferring data between IBM mainframes in the early 1970s, and the IBM single-sided, single-density 8" floppy is still the only universal standard for data exchange.

An alternative development was to tighten up the specifications, pack more data onto the medium, make all the components smaller and – the final elegant touch – build the air conditioning into the device. Once again it was IBM who pioneered this technology, to give us the Winchester hard disk (*qv*).

Shugart, the company that claims to have offered the floppy disk idea to IBM in the first place, took the bulky 8" disk and used it as a basis for its 5¼" system. This development coincided with the rise of microcomputers in the late seventies, and the new disks – small, and above all non-IBM – matched the new mood of freedom that micros brought with them.

Disk manufacture

The floppy consists of a round mylar disk onto which has been laid metal oxide (the magnetic medium). In the centre of the disk is a hub socket which allows the drive to both centre and rotate the disk. If the hub socket wears there is a danger of misalignment, so disk manufacturers often add a protective 'hub ring'.

Disk coating technology is a high-technology business, and a great deal of R & D effort and finance is required to set up as a disk manufacturer. Two megabytes are now quite practical on a 5¼" floppy, and by using special vertical recording techniques on nickel-cobalt, coating capacities in excess of 3 megabytes have been obtained. As data densities increase to a capacity that would have belonged exclusively hard disks a few years ago, so the quality of medium becomes crucial.

The original manufacturers of disk base material are few enough world wide to be counted on the fingers of one hand. There are probably less than twenty producers of the coated material – none of them in the UK. The majority of floppy disk vendors are, for the most part, buying in the raw material and packaging it.

There is a huge variation in price of diskettes to the end user, in part because there are two distinct methods of manufacture. The cheap way is to coat the base material in strips using rollers, with the risk of creating ridges on the surface. To obtain a smoother surface the magnetic media can be sprayed onto the base material while it is being rotated. This method is considerably more expensive.

After either method of manufacture the disk will undergo a finishing process that polishes and burnishes the surface. But building a disk takes more than getting the magnetic medium right; it has to be correctly mounted in the diskette jacket – a component that is quite a feat of technology in its own right.

Even with the protective jacket, floppy disks and their associated drives are far more vulnerable to dirt than audio record discs. The jacket must have an oblong hole left open to allow the read/write head to access the disk. For this reason the jacket is usually made from static-resistant vinyl material, and has a cloth lining which helps to lubricate the disk as it rotates as well as trapping any dust.

As a further refinement the jacket usually includes a write protect notch. On 5¼" disks, this notch (which resembles the handiwork of a ticket inspector) must be covered to prevent accidental over-writing. On 8" disks it is the reverse – the notch must be uncovered to 'write protect'.

Reading and writing

The read/write heads each consist of a high permeability split-ring core wound with a fine wire coil. Permeability is a measure of the reciprocal of the reluctance of a material to be magnetised, and 'high permeability' broadly speaking means 'easy to magnetise'. When a current is passed through the fine wire coil, a magnetic field is

generated around the core gap that almost instantaneously magnetises the area of ferric oxide beneath it. The current can be made to pass through the coil in one of two directions, magnetising the surface in one of two senses, so that the pattern of magnetisation on the surface can be used to represent digital data.

When the head is used to read from a disk, the same process works in reverse. Instead of passing a current through the wire, the flux change created as the disk surface passes under the head generates a current in the coil, again in one of two directions. Depending on the direction of the current the disk circuitry can detect whether it is reading a 1 or a 0.

Formats and densities

The way information is arranged on a disk, the format, and the amount of information packed into any particular area, the density, varies from machine to machine. Broadly speaking disks are either single density or double density and single or double sided. As a rule a machine can only read disks produced by a machine of its own type.

Recording density is usually expressed in tracks per inch (TPI); the more tracks on the disk, the more data can be stored. The number of tracks corresponds to the number of concentric rings which can be squeezed onto the disk. With 8" disks life was relatively simple because manufacturers kept to IBM's standard format of 77 tracks. No such standard applies to 5¼" disks, so 35, 40 and 80 track disks can be found. To make life more complicated, the search for ever increasing storage capacities has led to a different recording technique being used. Rather than trying to squeeze more tracks onto the surface of a disk, a new recording method was developed to double the amount of data that can be held. The standard method of Frequency Modulation (FM) allows single densities to be recorded, whereas using Modified Frequency Modulation (MFM) allows double densities. Disks suitable for MFM recording are thus referred to as double density disks.

Another way has been found to fit more tracks onto a disk – originally, disk manufacturers were working to 48 tracks per inch (TPI), but they can now produce disks which take double the tracks at 96 tracks per inch. On a 5¼" disk where the working surface is less than one inch deep, using 96 TPI gives a maximum of 80 tracks. Double sided double density, 96 TPI disks are called quad density. The accuracy needed to operate this system makes them very sensitive to alignment.

To make a disk useful for data storage, the computer has first to organise it into into tracks and sectors (*see the entry on Hard disks*). The operating system is responsible for putting files into these sectors and tracks, as well as cataloguing their position in a special area of the disk called a directory.

When you ask for a file to be deleted from the disk, the directory is amended accordingly, though the file itself will usually stay where it is until overwritten. Often when the operating system refuses to give you access to disk files you want, it is because the directory has become corrupted in some way. Products are available which allow you to scan through a disk's contents, to amend a corrupted directory and to recover files which have been deleted.

Hard or soft?

In order to find data stored on a disk, the drive head searches for the right track. Once it has found it, it has to search for the right part (or sector) of that track. The way in which tracks are divided up into sectors varies enormously – the most common method for governing the search for sectors is software controlled, and this type of disk is known as soft-sectored. All that a software controlled disk drive needs in order to find the right sector is a reference (or index) hole situated near to the hub ring.

The alternative, rarer method, is to use hard-sectors. In addition to the index hole, the disk is punched with a further series of holes, each marking the beginning of a sector.

Buying floppy disks

With at least twenty-five floppy disk brand names on the market, it isn't a simply matter of buying the cheapest available. You need to know if the disk is compatible with your machine, and if its specification suit your needs.

To pick the right floppy disk you need to know if you want double or single sided, double or single density, the number of sectors, and whether your system uses soft or hard sectored. Some manufacturers mark this information on the disk label or use a code with the figure 1 standing for single sided, and 2 for double sided. A large D usually indicates double density.

The way in which disks are manufactured helps to cut down the number of disk variants that need to be produced, as well as helping to hold down costs. A common method is to aim to produce all disks as double sided and double density. Subjected to rigorous testing, only a limited number will eventually be sold as double sided and double density. Those that fail but are considered good enough to be single density are then tested for 35 or 40 track operation. The process of elimination goes on until you reach a single sided, single density, 35 track disk.

This method of manufacture explains why certain tricks for formatting supposedly unsuitable disks will work in practice. Manufacturers with an eye to keeping a good name for their products in the marketplace allow fairly wide margins of error. Thus it is not unusual for a disk sold as single sided to 'magically' work on both sides. Likewise, disks sold as having 35 sectors can work in a 40 sector format. On the other hand it is not something that you can positively bank on.

It is perfectly possible to use a double sided disk in a single side drive, and a double density disk in a single density drive – but don't blame the manufacturer if you lose all your data!

Having decided the type of disk you want, the hard task

of selecting a brand of disk begins. One of the most obvious courses is to go for a disk made to a recognised standard. Unfortunately, with floppy disks the only widely accepted standard is IBM's 3740 for 8" disks. There are quality standards for disks in existence: ANSI (USA standard); ECMA (European); DIN (German); and ISO (International Standard, mostly used by the Japanese). However, most manufacturers claim that their own test procedures are more stringent than these standards.

Individual manufacturers also make great claims for the type of lubricant they use to prevent surface wear under prolonged use. These claims have to be taken with a pinch of salt, but it is well worth checking if the disks have been tested in batches or individually. The latter, often referred to as individually certified, is the more preferable.

Some manufacturers are prepared to offer lifetime guarantees on their products. But is this really much better than a five year guarantee from others? After all, what are the chances of your micro being anything but obsolete in five year's time?

No-one can really guarantee that every disk they supply will be perfect, but if you buy a faulty floppy you will probably know about it as soon as you try to format. It can happen to the most expensive as well as the cheaper disks. Disks which fail from continuous use are thankfully rare.

Carefully consider your own needs before buying. Is the data you intend to store so valuable that you can afford to spend an extra #3 per disk? Do you want to buy in bulk? If not the price you pay per disk is going to be higher. Some disks are only sold in tens. Does mail order suit you, or do you live just around the corner form several discount houses? The easier a disk can be returned if it proves faulty, the more you can save in the long run.

Part of the reason for high prices is the speed with which some suppliers can deliver. Some offer immediate delivery ex-stock. A customised service such as formatting will always cost extra.

Disk standards

5¼" disks aren't really standardised, but the situation is improving. Many machines now have drives which can, with the appropriate software, adapt to write and read disks in different formats. It can be a sensible idea if you work for a company with many different sorts of machines.

Also, any machine which follows the standard MS-DOS disk format, belongs to a club that can read and write each others' disks. So the IBM PC, its lookalikes and many other 16-bit machines can merrily swap disks (though not necessarily working programs).

The advent of sub-4" micro floppies has introduced a new dimension to the problem. But in microcomputing theoretical 'official standards' tend to give way to the de facto standards that everybody actually uses in practice. The de facto sub-4" diskette standard is the Sony, a lead that was confirmed when IBM withdrew their rumoured 100mm microfloppy.

Microfloppies

The Sony microfloppy drives have no door: the disk is simply pushed into the slot until it is automatically clamped into position. A release button neatly ejects the disk when you want to retrieve it. Measuring around two by four by five inches, the drives are considerably smaller than the traditional 5¼" inch arrangement, although there is no great space-saving over some of the newer 'half-height' 5¼" inch systems.

The biggest difference is in the disks themselves, each measuring around 3½" inches. A metal hub and a strong plastic 'shell' allow the disk to spin faster than normal floppies, around 400 rpm. Together with the smaller distance the head has to cover, faster rotation improves the speed of disk access.

Unlike standard floppies, the casing is rigid enough to stand being labelled in pencil or biro without risk of damaging the magnetic medium inside. A sliding metal guard (which automatically opens when you insert the disk) covers this delicate internal surface when not in use, so you can carry it in your pocket (or drop it on the floor even) without harming the data. With a double-density system that allows around 280K of data to be stored on each disk, the Sony system is better than many 5¼" floppies though it doesn't compare with some of the spectacular capacities (2+ megabytes) which have been achieved on the older system.

The advantages of microdisks over 5¼" systems boil down to the size of the drives and the robustness of the disks themselves. Gone is the floppy paper sleeve and the exposed section for the read/write heads, so easily corrupted with finger marks. Gone is the slot on the side that has to be covered with sticky tape in order to write protect the disk. They can be protected from erasure (write protected) by flipping a switch. This makes them much easier to handle, store and protect.

Game Of Life: The Archetypal Model

Conway's game of Life applies four simple rules to govern births and deaths in an imaginary two-dimensional population spread out over an extended grid. The player does nothing apart from setting up the starting conditions and watching as the population patterns and numbers change from generation to generation. What makes even the most unexpected people into addicts is the way that slightly different starting conditions can have such a profound and extensive effect on the outcome of the game. There ar populations which just up and move away, others which rotate, float and cycle through a fixed set of patterns. *Life* followers spend their time discovering and naming new patterns.

A population is made up of individual cells. These cells can represent any culture of living creatures (like amoebas, plankton – even programmers), and the game models the population's history according to how the environment of each individual cell changes. Whether a cell lives or dies depends only on the number of neighbours it has, so it is an asexual model. Altering the basic ideas to incorporate such variations as two different types of cell is left to the imaginative reader.

The inherent simplicity of the *Game of Life* makes it very easy to program. Because of this simplicity, designing such a program can be useful for illustrating one of the main contentions in software design – that of **memory requirement versus speed**. There is a trade-off between how much information is stored and how much is calculated when needed. A bias toward storing more information will require more memory, but can decrease the time taken to perform a search of the structure. A bias toward simpler data structures can save memory, but it may take longer to analyse.

John Conway, a Cambridge mathematician, developed the principles of the *Game of Life*. The model universe consists only of cells and emptiness. The rules are very simple:

> If a cell has *two* or *three* neighbours then it lives on.
> If it has *one* or *no* neighbours it dies of starvation – or loneliness, or hypothermia, or sexual deprivation or whatever you like!
> If it has *more than three* neighbours it dies of suffocation.
> However, if any empty cell-sized space has *exactly three* neighbours, a new cell is spawned and grows in that space.

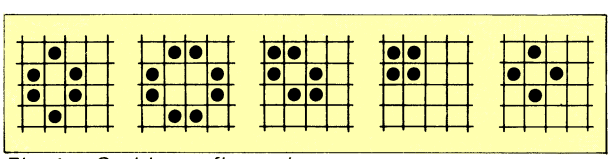


Fig. 1. Stable configurations

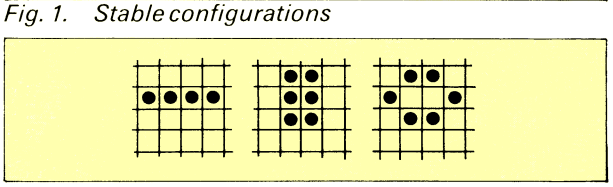


Fig. 2. Life history of a string of four cells

And that's it. The only missing link is how it all starts in the first place. For this example, we'll just say that there is a random spread of cells across our universe.

Many interesting and strange cell structures form as time progresses. Some of them are stable, and others die off. Figure 2 shows the Life history of a row of four spaces. The stable result, reached after only two time cycles, is called a beehive. Figure 1 shows a number of small stable configurations. Other shapes oscillate. Figure 3 shows the simplest oscillating structure, called the *Blinker*.

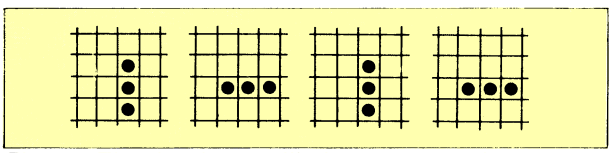


Fig. 3. The simplest oscillating structure, the 'blinker'

Designing the program

Given the simple specification, how would you design a program to model it? The obvious method would be to have a two-dimensional array of characters representing the universe, each character being either an empty space or full one. The updating procedure can then take each space in turn, check its neighbours, and alter it accordingly. However, there is a problem here. All spaces must react simultaneously to their environment, so changes cannot be made to the universe until all the necessary alterations have been worked out. So, really, two universe arrays are needed: one to represent the current state, and one the future.

The current universe should appear on the screen, while the future one is calculated from it. Then the future universe becomes the current one, and the current one can be used to calculate the next future state.

Each space has eight neighbouring spaces. To see what happens to any one, all that is required is a check of all the neighbouring spaces. Assume that the universe has wrap-round edges so that the furthest point on the left is a neighbour to the furthest point on the right. This means that the total number of checks needed to produce an updated universe is eight times the size of the universe. That is, 8 times number spaces vertically times the number of spaces horizontally. This gives an indication of the speed of the updating procedure in relation to any algorithm we might devise.

A little thought will reveal that the model universe will contain some areas of pure emptiness. The simple procedure described will methodically test all eight neighbouring spaces of each space. In other words, a void of 10 locations by 15 locations will require 8 x 10 x 15 checks, to produce a result telling it that it is still a void!

The important information about any space is how many neighbouring spaces it has. In the first algorithm, this number is constantly recalculated. Instead of this, the universe arrays can be extended to hold more information about each space, more than just whether it contains a live cell or not. To each location in both of the universes

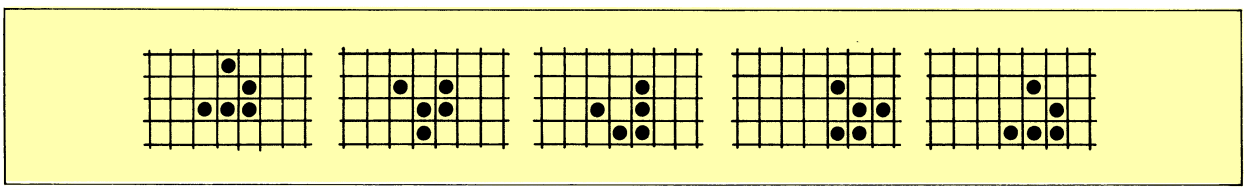


Fig. 4. A moving structure, the 'glider'

can be added a count which records the number of neighbouring cells. These numbers will need to be calculated when the universe is created, but will not need to change unless something happens in the near vicinity – that is, when it or a neighbouring cell 'dies', or a new one is spawned.

If a cell dies, all adjacent 'neighbour counts' need to be reduced by one. Likewise, if a cell is spawned, all counts need to be increased by one. When building up a future universe, it is not necessary to count up the neighbours of any entity. All that needs to be done is that the 'neighbour count' is referred to. Adjusting the counts when a cell is spawned or dies will affect the creation of the next universe.

So the procedure to create a future universe is as follows. First, all the current neighbour counts are transferred to become future neighbour counts. Then the current universe is scanned, and if any space changes status, the corresponding changes are made to the future neighbour counts and spaces. If nothing happens, the current space becomes the future space.

The first algorithm will consistently take the same length of time to create a new universe, whereas the second one will vary depending on the number of cells living and dying. This second time will only approach the first if there are changes to nearly all the spaces in the universe; otherwise there should be a considerable improvement.

Extraordinary results

There are many fascinating aspects to the *Game of Life*, such as small cell configurations that produce large populations, and other configurations that move as a body across the universe. The most famous (because it's the smallest) cell structure that moves as a unit is called the *Glider*. Figure 4 shows this most interesting animal. The Glider moves one square diagonally every four time cycles. Other moving shapes have been found.

Some *Game of Life* fanatics have found *Glider Producers* and *Destroyers*. A Glider Producer (called a glider-gun) is a cell structure that periodically fires off Gliders into space. A Glider Destroyer will swallow up Gliders that are fired at it. These cell structures have been found largely by accident, and involved firing large numbers of these Gliders at each other, simply to see what happened.

In some of the larger structures, reproduction can take many hundreds of time intervals. In fact some of the larger computer models of the *Game of Life* produce a new screen only every hundred or so updates to the universe.

A word of warning before you rush off and program it. You will probably be sadly disappointed with the results. Interpreted BASIC is simply too slow to recreate a universe of interesting size quick enough. What is required is assembler (The universal computer answer is, after all, 'You write it in machine code.').

Games: The Micro At Play

Even in the very early days when computing cost billions, the temptation just to play around was never very far away. Of course the games those first computer operators played, once the serious work of calculating quadratic polynomials was over for the day, were a long way from the latest multi-colour, hi-res bloodbaths most people associate with computer games. The games in those days involved pure mathematics, such as printing π to a thousand decimal places or colouring in maps with as few or as many colours as possible.

It was a logical step from the playing of 'intelligence' games to making the computer play 'intelligent' games. The oldest computer game must be *Guess my number*. Based on the binary search (*see* Sorting), and really too trivial to interest anything but a sleeping human being, this charade of the computer 'thinking of a number' promoted the machine to the role of a player and opponent.

Today many of the traditional games have been computerised. First came the pencil-and-paper games like *Noughts and crosses*, then *Bulls and cows* (sold to the public in boxes marked *Mastermind*). Then it was the turn of the board games like *Scrabble* and *Monopoly*, wooden pieces being turned into binary data.

Of course it is unfair to expect microcomputers consistently to win against the best of their human opponents. Until recently, any club player could demolish a micro chess program, and even specially designed machines with chess-playing hardware don't match the chess masters. One or two other games like the Japanese game *Go* also continue to present problems for the computer.

But the competition has only just started. Computers get better and better – and they have already ruined some games. There's something very depressing about a game that machines can play better than you. Spear's *Reversi* (more recently marketed as *Othello*) is an obvious case. Even with its strictly limited combination of moves, even a modest micro can become an invincible opponent.

Simulation

Some of the best 'games' come from using a computer for real-world simulations. With a computer anything that takes too much time money or effort to do in the real-life world – wars, traffic jams, pilot training – can be done as 'pretend'.

The world of business uses simulation games, often played in teams, for training in the internal working of company finance, and to develop awareness of the effect of external forces like government policy and economic climates. Several respected business simulation packages are widely used in industry and education, and some of these are now becoming available for home computer owners.

Although it doesn't directly relate to anything in the real world, the classic computer simulation is mathematician John Conway's *Game of Life* (*qv*), and it has been used as a basis for models of biological cells, demographic changes – even the birth of the universe. *Lunar lander* is the other entertainment simulation that predates microcomputers. In the late 1960s, when the computer company DEC wrote its original animated version – which involved landing a lunar excursion module near, but not on, a hamburger stand – it was reflecting a genuine and contemporary problem. Some of the simulations available today show a similar awareness of the world. *Three Mile Island* gives you an animated chance either to panic or save a nuclear reactor from a meltdown.

Arcade games have become considerably more sophisticated since the days when queues formed in pubs to play *TV tennis*. The struggle for higher profits and against players' increasing sophistication has forced manufacturers to develop ever more-complex technology. Arcade games have certainly provided the incentive for improved graphics hardware on home micros – although they don't exercise the imagination one jot.

Adventure games

But there is one mind-stimulating species of game, which like *Lunar lander* and *Life* dates back to the very beginnings of computing. It creates a whole world in the mind, supplying escape from the real world's troubles by substituting fantastic imaginary characters and disasters.

The game is called 'Adventure', and is a derivative of *Dungeons and dragons*, (*D&D* for short). In *D&D*, which involves using a book, a 'dungeon master' (a player) and a posse of explorers make their way around a network of dungeons, avoiding assorted dragons, trolls, warlocks and the like.

The game was first computerised at an American university, and a version called *Adventure* was developed, sufficiently slim to fit into a micro's memory. The game is simple. You are given descriptions of your position in a subterranean world, and can move about by using direction instructions (usually compass points) and up and down. Every description evokes a fantasy world of yet more dragons, dwarves, pirates, spectacular caverns and claustrophobic passages. The aim of the game is to avoid death and find treasure.

There are now a number of variations on the adventure theme. Some include graphics, rather than just text, which purists feel degrades the richness of the game.

One difficulty in writing adventure games has been finding a command language for the player to use which is sufficiently English-like to be unobtrusive. Probably the most successful adventure game to date, *The hobbit*, based on J.R.R. Tolkien's book of the same name, partly overcame this problem by allowing the user to type in sentences containing a number of commands. Nevertheless, the player is still confined to a specialised vocabulary.

Products are also becoming available which enable users to write adventure games themselves. A package like this will typically split an adventure game into the five basic components common to nearly all adventure games:

A **location database**, which holds text describing

each location on the adventure.
A **movement table**, which links the locations together.
An **object table**, which holds descriptions of objects that can move between locations.
A **dictionary**, containing all the words in descriptions and commands used by the player.
A **programming language**, which is used to program the effect of the player's actions.

Arcade games

Nobody quite knows how high-street fun palaces came to be known as arcades. The word means literally a row of arches, and it may be that street games first sprang up under ancient aqueducts. It may also refer to Arcadia, the Greek mythological world of dance and music.

But now it is firmly associated with colourful, fast-moving, interactive video games, either in their original habitat, the amusement parlour, or transferred to the domestic television set by way of a home computer.

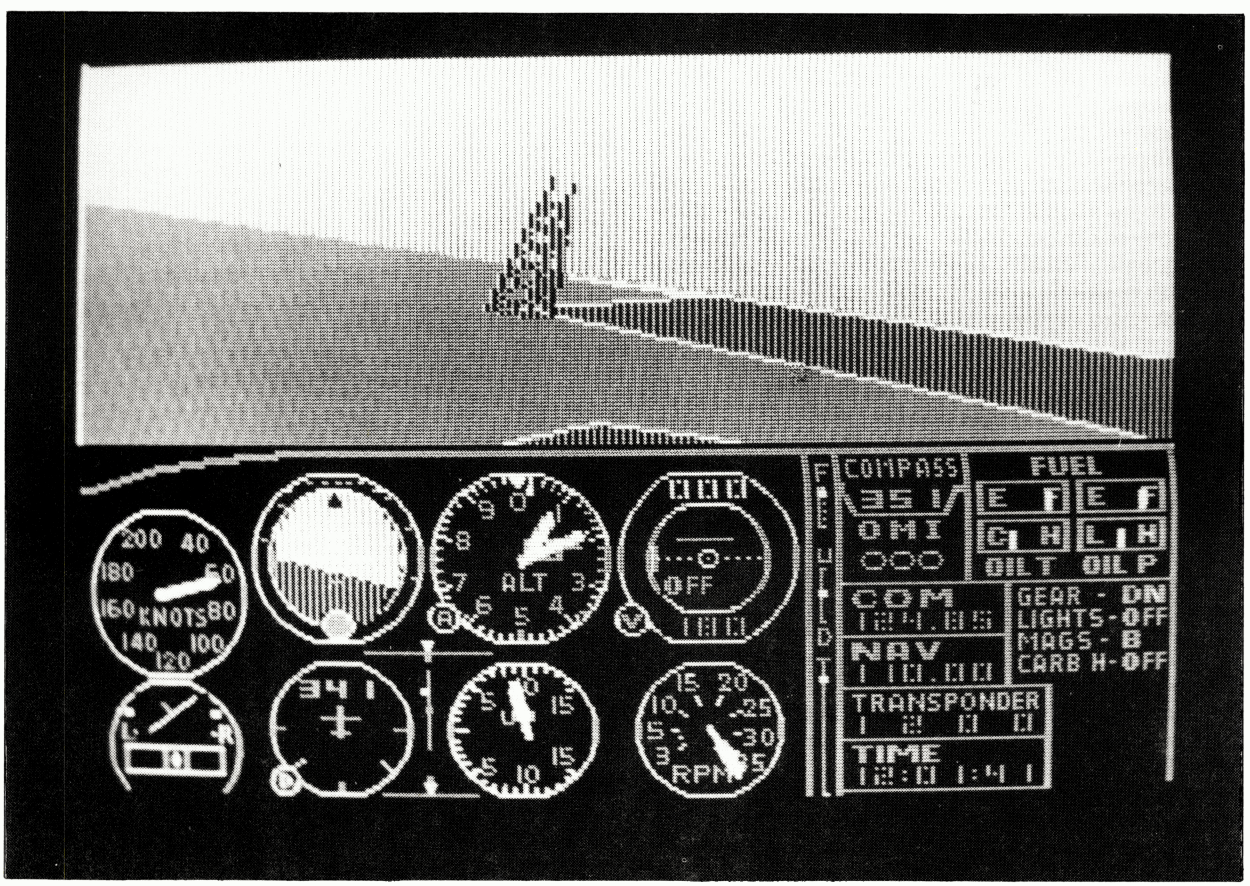
Arcade games fall into a number of categories, named after their prototypes:

Space invaders make you the defender your territory, usually a number of 'bases' at the bottom of the screen. Your weapon is a gun controlled by keys or a joystick, which you use to zap creatures advancing from the top of the screen.
Pac man drops you into a maze where you have to move around consuming or picking up various objects while avoiding killer creatures that chase you down the narrow alleyways.
Defender sends you jetting over a mountain range, shooting down or avoiding oncoming objects.
Asteroids challenges you to make your way unscathed through a meteor shower.
Donky Kong, a derivative of snakes and ladders, sends you clambering perilously between levels of a world structured like a multi-storey carpark.

Flight simulator

War games have always been popular, and you can buy several versions for any popular micro. Most of these rely on *macho* fantasies, making you the general or commander in charge – never the private with the hand grenade.

As a rule, the more sophisticated packages are the least bloody. There are now quite a few flight-simulation programs with no carnage at all. Some software companies are taking flight simulation very seriously, to



The MicroSoft Flight Simulator instrument panel as it appears on the screen.

the point of employing pilots as advisers – and even getting professional flight-simulation programmers to write for them.

Probably the best example of this is the *Microsoft flight simulator* for the IBM Personal Computer. It was written by Bruce Artwick, the president of a American company experienced in high-performance graphics. He gained his experience in flight simulation through working for the Aviation Research Laboratory at the University of Illinois and with Hughes Aircraft in the US.

Although Microsoft doesn't claim that the program will teach you to fly, it is intended to be as close to the real thing as a micro will allow. The aircraft is precisely modelled on the Cessna, even including the turbulent air penetration speed, and the maximum rate of climb at sea level. It is one of the few simulation programs which really make use of the keyboard, with about 30 keys used to control all the aircraft functions.

The instrument panel, displaying a daunting array of dials and controls, takes up the bottom half of the screen, leaving the top half for the pilot's view of the world. With impressive accuracy the graphics map an area of some 10,000 miles square, covering the United States and parts of Canada, Mexico and the Caribbean, taking in four populated areas, and over 20 airports and an wide range of radio navigation aids (operating on their correct frequencies).

The program takes account of variables like wind speed, clouds and the time of day. For beginners the package includes an 'easy flight mode', so that you can learn how to control the plane without having to fight the elements as well. For those who get bored with flying, take-off and landing, there is a section of the program called British Ace, where you pit your flying skills against the Luftwaffe.

Unlike the one-way brain-candy entertainment of television, the computer game stimulates and educates. But although the concept is as old as computers themselves, computer games are still very much in their infancy. They are bound to continue developing with the technology.

In particular, as communications networks become commonplace, the whole games arena may well be transferred to networks used to support multi-player adventure games. Today's computer games are accused of cutting the players off from ordinary social life. But they could become an important supplement to it.

It's going to be an adventure for all of us.

Writing commercial games

There has been a rapid evolution of fashion in computer games over the past few years. Only one rule has emerged: that there are no set rules for writing a successful game. But for aspiring programmers here are some tips based on experience that may help create that world-wide blockbuster!

Although many of the games you can buy for your micro are backed by big-name companies, most of them are still created by a single person working solidly on the project – sometimes for as long as eight months.

Usually it's the programmers themselves who originate the ideas. At an early stage an outline is passed on to a graphics programmer, who uses a design program to build up the game's images. Then a set of polaroid photos of the screen are mounted on sheets of paper with a text description of the game's specification. The package is then presented to an editorial board, typically made up of senior staff. If the idea is approved, the games programmer gets down to the serious business of coding, using the polaroids as a guide.

With so much money to be made out of commercial games writing, you might expect a heavy investment in market research. But in a field as new as this, customers tend not to know what they want until they get it, which gives free rein to the creators.

Nowadays the most popular games depend on reaction time; games involving the intellect sell less well. Games designers assume that people like aggressive games; the more graphics and sound the better. Aim to make your game addictive – and that doesn't just mean mindlessly attractive.

Give your players the sense that they are achieving something, such as increasing their scores. What helps to make arcade games popular is the running total at the top of the screen. Build in higher levels of difficulty so that a player is as interested in the hundredth game as in the first.

The volume of computers sold obviously determines, within a range, the volume of software sold for it. A reasonably successful game will sell to 2% of the installed base of machines; a very successful game will sell to 10%. It is very rare for a program to achieve market penetration higher than 10%. Given a target of 5% penetration, which is typical, it is obvious that a version for a computer that has only sold 20,000 units is unlikely to make much cash.

A successful program can sell 30,000 copies in the first six weeks. A games designer and programmer can each expect more than an average month's salary as an advance on sales, from a royalty of anything from 5% to 20% of the retail price.

How difficult is it to write a commercial game program? In a word, very. What takes the time isn't the programming, coding and debugging necessary to make the game work, it's the refining and polishing to add that final touch of glamour and make the software ruggedly uncrashable. One company receives fifty games a week, but finds that only one in a hundred is worth developing. The development process takes about three months between acceptance and distribution to retailers.

Because of the immense hard work that must go into any serious project, most programmers are throughly bored by their own games by the time the packaging and marketing stage is reached. So patience and determination rank high in the job description. A pretty thorough knowledge of assembler is also a requirement, because BASIC, though useful for developing game ideas in the early stages, just doesn't give the necessary speed that most commercial games must have.

Home Control: Domestic Systems Engineering

One of the most exciting aspects of owning a computer is the prospect of using it to control the home environment and domestic machinery. But it's not as easy as it sounds. In the home, there are many subsystems sufficiently dispersed to make any sort of centralised control and monitoring difficult if not impossible. Every part has its own little brain: the toaster, the washing machine, the central heating, the gas and electricity supply, the telephone, the television, video recorder and hi-fi. The promise of integrated home electronics held out by systems like MSX (*qv*) is not yet a fact of everyday life.

Everything under control

To the human who has to act as the central monitoring system, the gas, electricity and phone bills are usually an unpleasant surprise. Even if they aren't, only the most obsessive monitor is able to account for every therm, watt and unit. Control is no easier. Central heating control systems (the clock with two or four time switches) were designed for a peculiar creature which either never deviates from a rigid timetable, or likes to leave its environment as it warms up, and return when it cools down.

While we wait for manufacturers to provide us with an integrated system, what is needed is a method of centralising all the information so that we can control it as efficiently as possible. For the adventurous, prepared to program and make a few hardware changes, the micro offers an answer. A computer is entirely suited to taking information from a number of discrete sources, and processing it so it makes sense. For example, it can be used to take in all the temperature measurements over a period of time collected from strategic points around your house. These measurements could then be used to calculate the efficiency of your heating. In conjunction with information about the movements of all inhabitants, it could show whether the heat is being delivered to the right place at the right time.

It can also monitor the use of the phone, electricity and gas and keep a file on your income and your weekly expenditure (which you would have to update at regular intervals). Certainly all this requires fairly sophisticated software and hardware, but it may prove rewarding and fruitful. The technology exists, and it's fair to say that almost anything can be accomplished with your home computer given a little thought, a lot of work – and a deepish pocket.

The components

One thing has to be faced up to in a practical experiment of this kind: despite the fact that you bought an incredibly powerful computer for very little money, that computer is only a part of the system needed for monitoring or controlling your home. The other components are expensive, especially if they haven't been subject to the same explosion of technology and volume sales.

Figures 1 and 2 show in simple block form the elements of monitoring and controlling respectively:

Transducers are devices which convert one form of energy (mechanical, thermal, light etc.) into another form (electrical for our purposes). A thermocouple, for example, converts heat into electricity and can therefore act as the temperature sensor in your central heating controller. The difficulty is that this device typically gives

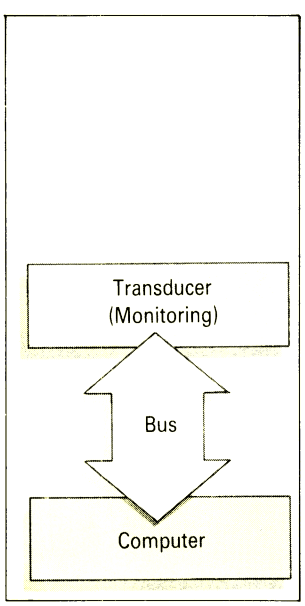


Fig. 1. Elements of a monitoring system

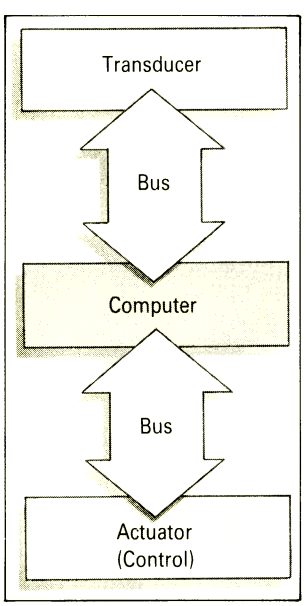


Fig. 2. A control monitoring system

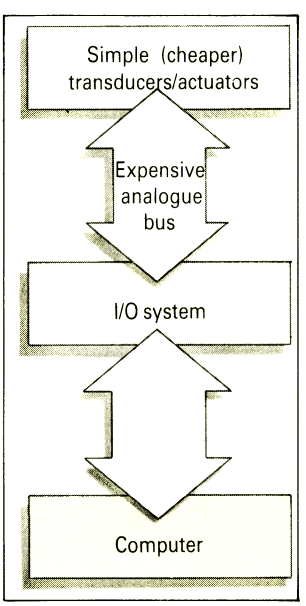


Fig. 3. A control monitoring system using an expensive analogue bus to read and send signals

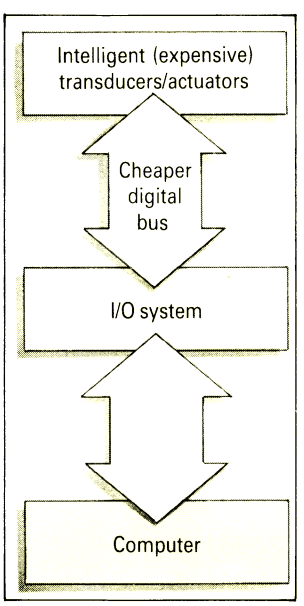


Fig. 4. A control monitoring system using intelligent transducers and actuators

out a signal of only a few millivolts – and this as analogue information which the computer cannot read directly. The same problems, more or less, apply to other transducers such as flowmeters, level detectors, load cells, and strain gauges.

Analogue-to-digital converters are needed to interface the micro with the world outside, bridging the gap between the more or less continuous varying signals put out by transducers, and the series of 'ons' and 'offs' handled by the computer.

The **Bus** is the means by which information is carried to the computer from the transducers and from the computer to the actuators of your system. The bus must be able to carry information accurately from an identifiable source to the correct destination at an acceptable speed. For accuracy the bus must transmit and receive information without the introduction of noise or interference from other systems or messages, and must also have enough capacity to carry the data with the level of precision needed.

As there's not a lot of point in capturing a value of 39.7 and not knowing whether it's the phone bill or the temperature of your coffee, the computer must be able to tell which transducer is responsible for the data presently on the bus; equally, any response from the computer must be sent to the correct destination. A signal to open the cat flap should not empty the bath.

The physical nature of the bus is dependent upon the speed of transmission needed. The designer will have to make a choice between serial or parallel routing, between screened and unscreened cables; as well as taking account of factors like the acceptable amount of signal attenuation over the bus length. In a typical home application, technical considerations of bus type will be secondary to aesthetics and cost – you would soon think twice about running a cabling system of 25 screened conductors from your computer to several points in every room in the house, once you saw what the material alone was going to cost. Laying enough cables to wire each transducer separately for analogue signals, as in Figure 3, would be cheaper, but the intelligent transducers and actuators in Figure 4 can identify themselves on an even simpler digital bus.

Actuators are are used to carry out the actions your program decides are necessary, working from the information your computer has received from its transducers. Typical actuators such as motorised valves, solenoid valves, motors and pumps are triggered (though rarely powered by) signals at digital level from the computer and act by changing state appropriately. The simplest forms of actuators are on/off types such as solenoid valves, but multi-level conditions, as when changing the volume level on a TV set, are also necessary. One possible source of actuators and transducers is the automation industry. A number of systems have been set up to use microprocessors to monitor and control manufacturing plant. Most actuators need energy to act, and therefore need a power supply and wiring of their own in addition to the control signal wiring.

The art of fail-safe

Crashes occur from time to time due to software or hardware failure. While it is annoying to lose a high score in a game of Invaders, and infuriating to lose a tediously keyed-in address book file, a crash in a domestic control system which has control over fuel, ventilation and hot water, is positively dangerous.

It is essential to ensure that actuators fail safe, in other words fuel supply valves close when they fail, while exhausts should fail in an open state – it is common practice to fit some kind of transducer to the actuator to feed information back to the processor indicating that the system *is* functioning correctly – safety precautions which drive your costs up even more.

So here is the control enthusiast's safety code:

> Actuators must be chosen so the systems are intrinsically fail safe, for example with a fuel cock closing on fail.
> The computer itself must be monitored for crashes, and the system must fall into a safe state should it fail.
> Partial or total power failures must also cause the system fail safe.

Using what you've got

There are already a number of systems around the home which can be relatively cheaply and usefully connected up to your computer.

Telephone call monitoring is a very simple, cheap and safe project for home computing. With an approved socket installed at the telephone junction box, and armed with the necessary permission from the phone company, you should be able to monitor the signals, which in turn makes it relatively easy to keep track of incoming and outgoing calls. Every time the phone handset is lifted, the polarity of the line is changed, which means a micro could be alerted to the possibility of a call.

The telephone system uses a series of pulses to represent numbers. These can be counted by the computer, and the number being dialled calculated. It is then a relatively simple programming task to compare the number against the telephone company's rates (held in a specially created file) or even against the called party's name and address. A real time clock is built into some micros, and available as an extra for others, and this can be used to measure the duration of the call, and to note whether it has been made during peak or off-peak hours.

This data would complete the phone log, and you could generate monthly reports on the use of the phone, giving the numbers called, their individual cost, the times at which they were made and the phone bill for the month. You could also develop the software for measuring the use of the phone in standard units of measurement, which would make a straight comparison between your reports and the phone bill easy.

Autodialling is equally straightforward. The system could look up the personal number in your private

telephone directory (held as a file on the computer), operate the hand set switch to get a dialling tone, and generate the appropriate series of pulses down the line.

Monitoring the electricity is even easier. Don't, whatever you do, improvise unorthodox connections between the mains and your computer – all you need is one of the cheap clip-on ammeters now readily available. These measure the magnetic field around a cable, which rises and falls in relation to the amount of current flowing. They can be fastened onto the incoming supply cables without electrical connection, and the current can be read off a digital display.

Getting the measurements into your system is a little more complicated, but it can be done by tapping the digital code from a binary coded decimal display driver in the ammeter. If the system then multiplies the current reading by time and voltage (which should be a constant 240 volts), you will end up with the units which form the basis for electricity billing: kilowatt hours. These ammeters could also be strategically placed around the house to monitor the use of electricity.

Thermometers can be used to monitor heat levels and to calculate the use of heat around the house. Controlling the central heating would need an on/off switch timed by the computer. To monitor the consumpion of gas or oil fired central heating will need some sort of in-line flow meter, but you could experiment with a sensitive microphone, since the flow of liquid through a pipe tends to generate noise in proportion to its flow.

The Mains as a Bus. An alternative to laying new wires is to use the existing mains supply as a data bus, *don't attempt this unless you know what you are doing.* The mains supply is an alternating current (AC), which means the current flows one way and then reverses to flow the other way fifty times per second. As the current reverses it passes through zero volts, and it is a this point that the mains wires are free to carry information – see Figure 5.

The point when the message is sent is called the zero crossing point, and the system is already used in a number of domestic systems, for example in baby alarms. It is possible to make up nodes using the dedicated chips available for the purpose, but it is worth stressing again that this isn't a job for the casual dabbler.

The micro in the greenhouse. Because the micro is a comparatively recent addition to the household furniture, its use in domestic control and monitoring is untried. A greenhouse, if you're lucky enough to have one, makes a good environment in which to experiment – as long as it isn't stocked with expensive plants.

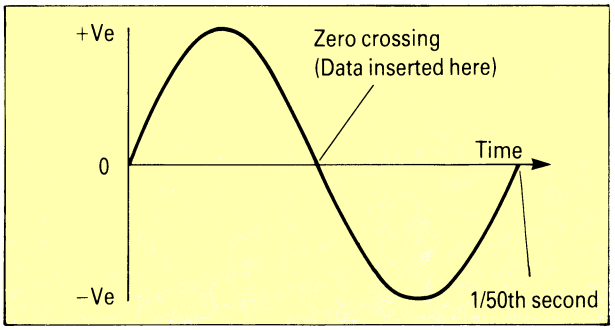


Fig. 5. Cross-over point in AC mains supply

For plants, you generally only want to ensure that the temperature is kept within certain limits, and this can be done with simple measuring devices, preferably home-made or adapted from existing ones. Two conventional thermostats could be used, set to the upper and lower temperatures required. You could also try monitoring humidity with a device generating a signal when the environment becomes too dry. Two contacts placed a short distance apart in the soil could be used to monitor soil conductivity and hence the moisture content. Control could be in the form of a low-wattage electrical heater, turned on and off by a well insulated relay. A solenoid and a few levers would be all you would need to regulate ventilation.

A telephone answering machine, if you have one, offers the possibility of initiating processes in the home by putting through a call while out at work or away on holiday. The phone answering circuitry responding to your call would in effect turn you into an additional monitoring station on the bus. You could then use the remote phone dial to send a number down the line to the computer, which would activate the process you want. For example, the first number could identify the process, and the subsequent numbers a series of start and stop times.

A standard interface

One practical problem we have skated over so far is that the binary logic levels used by computers are rarely compatible with the on/off signals of objects like central heating pumps and vacuum cleaners. To bridge the disparity you need a conversion box, or interface and there are two standard techniques for doing this:

Memory mapping makes the interface appear to the micro as a standard memory location, data written to or found in these particular bytes is decoded by special electronics, and is thus connected to the outside world.

I/O mapping. Most micros have a special set of instructions for accessing input/output devices through logical connections known as 'ports'. These are handled by the processor in a way resembling memory mapping, but the ports are generally separate from the true RAM array.

Either of these powerful techniques requires direct access to the logic signals of the computer bus, and extensive decoding circuitry. Both techniques also need specialised driving programs, which may well have to be in machine code, as many BASICs will not include commands for reading from and writing to these special locations.

Because of these problems, a functionally identical interface would require a different circuit and program for each computer it was to be used on. When you upgrade to your next computer, you have to do the work all over

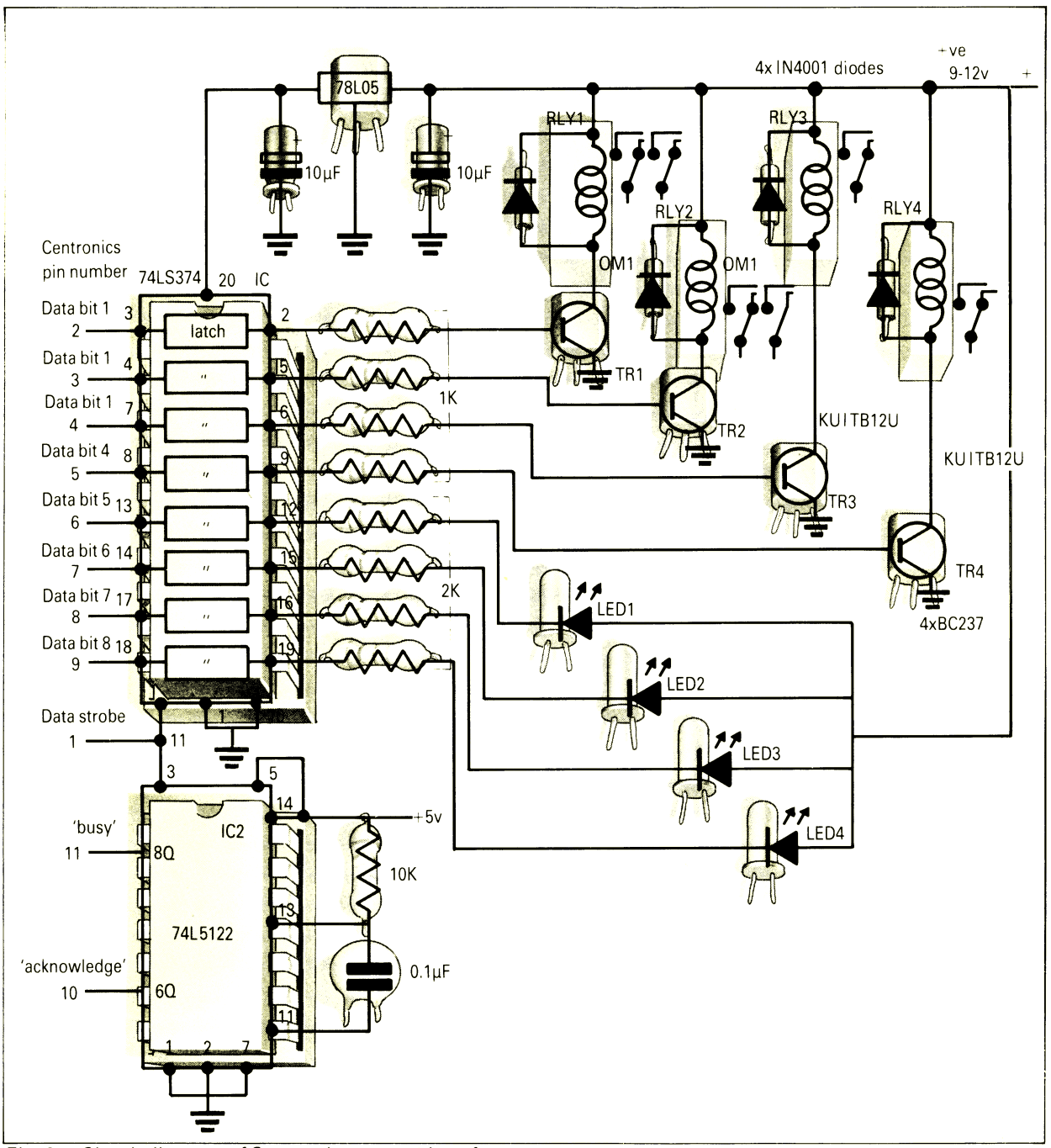


Fig. 6. Circuit diagram of Centronics control interface

again. But there is a way round this. For driving actuators, at least, where we limit our demands to output only, it is possible to build an interface that will be almost universal, and extremely easy to drive from software.

Nearly all home computers can drive a parallel printer though an output port also known as a Centronics interface (*qv*). We can exploit this fact to build an (almost) universal interface by making our interface box look like a printer to the computer. If this is done properly the computer's built-in commands for sending data to the printer will work just as well for sending data to the interface. The result is a simple universal interface, with no modifications to existing hardware and zero software overhead!

In the rapidly changing world of microcomputers, the parallel printer interface is one of the few reliable standards. Its main job is to transmit a succession of 8-bit data words to the outside world, accompanied by some

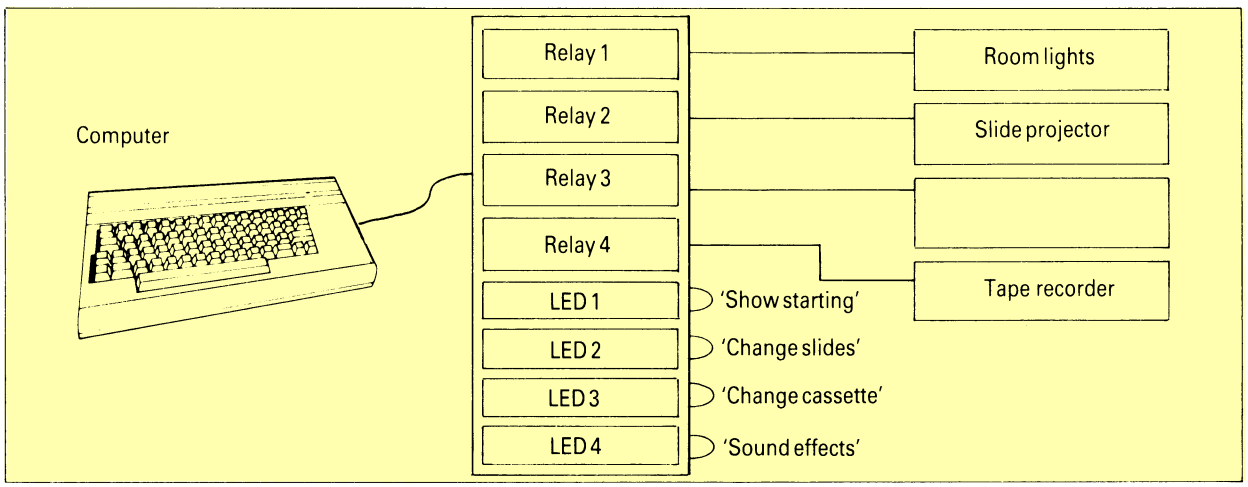


Fig. 7. Configuration of interface to control a slide show

simple handshaking, and you'll find a full description of its workings in the entry on Communications. To fool the computer into thinking that our special interface on the end of its printer cable is really a printer, is simple. The circuit diagram of the interface is shown in Figure 6. All it consists of is an 8-bit data latch to grab and hold on to the data on the interface pins of the connector, and a monostable (a device to generate pulses) to feed back the acknowledge and busy signals.

Thus, every time the computer strobes data-strobe low, the latch (IC1, 74LS374) latches the data on its input pins, which are connected to the computer's output pins. 'Latching' means that the data appearing (however briefly – within reason) on the output pins of the device will stay there – even when the data supplying the input pins has disappeared.

The rising edge of the data strobe signal triggers the monostable (74LS22) which generates a busy and acknowledge signal back to the computer. IC3 is a 5-volt regulator to provide a steady supply for the logic gates from a higher unregulated input.

The interface has two sorts of output, relay and LED. The relay outputs are for switching external signals and the LEDs are used as simple prompts and so on. The low-consumption LEDs can be driven directly from the outputs of the latch, but the relays need a buffer transistor to boost the power, and this is the task of Tr1-4. The diodes across the relays are there to stop the back induced voltage from the de-energising relay coil from blowing up the transistors.

In Figure 6 you will see six sets of three pins down one edge, these are the outputs from the relays. There are six because the first two relays have a *pair* of outputs each. The middle pin is the 'common' line. With the relay set in one position the common will form a connection with one of the outer pins; in the other position, it will form a connection with the other outer pin. This 'flip-flop' between the two outer pins could be used to turn one piece of equipment on or off by connecting two of the pins, or to swap the current between the two pieces of equipment by connecting all three pins.

The relays are turned on by outputting a 1 to the latch. Thus, outputting a character with a bit 0 set high will turn on relay 1, while outputting with bit 2 set high will turn on relay 2. The LEDs are turned off by outputting a high bit. So outputting 10000_2 (binary – equal to 10_{16} in hex) will turn the first LED off. It is easy to see that in order to arrange different controls, the character sent to the printer must be the value of the required bit pattern. For example, suppose we wanted to turn relays 1, 2, 4 on, turn relay 3 off, turn LEDs 1, 4 on, and LEDs 3, 2 off. The required bit pattern is 01100111, which is the same as 103_{10} 67_{16}. Or to use the data type more familiar to printers, this is the same as the lowercase 'g' in ASCII.

The interface is driven simply by using the printer output commands, thus `LPRINT CHR$(103);` would achieve the desired result. Don't forget though to use the semicolons to suppress the CR/LF (Carriage Return/Line Feed) sent by most BASIC interpreters at the end of a print statement. If you do forget them the interface will appear not to work, as it is responding to the last character the BASIC sent it (either CR or LF)

By driving relays directly from your computer like this you can control almost any device. One use could be as a slide projector controller and Figure 7 shows how the interface might be used in this sort of application. The relays can be used to turn the room lights off, the projector on, and to advance the projector slide by slide. The last relay can be used to control an audio cassette deck with a sound commentary.

The LEDs can be used for prompts such as 'show about to start', 'change slide rack', 'change audio tape', 'produce special sound effects' and so on. The computer is used to produce variable length delays for each frame and to keep the slide and tape machine in synchronisation.

This is just one example of the almost limitless uses for this quite simple interface device. But again, a word of warning. If you are going to switch mains voltages, you must take every possible precaution. Only the first two of the recommended relays are suitable for switching mains: the other two will switch direct current voltages up to a maximum of 50 volts.

Keyboards: The Micro At Your Fingertips

Engineers have been designing keyboards to interface with electronics for more than a quarter of a century. The best modern keyboards are:

Fully debounced, registering only one character no matter how hard a key is struck.

Buffered to store multiple keystrokes.

provided with *n*-**key rollover** so that all keys depressed will register even though previous key actions may not yet have been completed.

Movement, inevitable in any device translating mechanical action into electronics, is kept to a minimum in the best designs, and a working life of more than a quarter of a billion key strokes is possible. This is pretty impressive when you realise that even typing 40 words a minute for six hours a day gives only half a million key strokes a week – on that basis a really good keyboard going flat out should give little or no trouble until sometime around the end of its tenth year!

Keyboards of this quality are seldom found on standard dual-floppy machines, and never on home micros, but of course no home micro is not going to have to stand up to this sort of pounding. Nevertheless it is worth taking a critical look at the keyboard of any micro you are thinking of buying, bearing in mind that it is one of the most vulnerable components. Even if you are not going to use it for word processing, it is worth measuring it against word processing criteria to get some idea of how it will fare as your relationship with your micro develops.

Unions and ergonomists are beginning to insist on keyboards that are attached to the main body of the terminal by no more than a flexible umbilical cord. Home micros emulate this configuration by putting the computing components inside the keyboard and using the TV set as a screen. Either way the advantage over the integral console/keyboards which used to characterise minicomputers and the early micros is that the user's relationship to the keyboard doesn't define the distance from the screen, so that each can be adjusted for maximum comfort.

Traditionally computer keyboards have been fairly substantial pieces of equipment, but trends and technology are slimming this down to keyboards where the central 'H' key is little more than 3 cms above the work surface. One advantage gained is that a surface adjusted for correct work height without the keyboard won't have to be readjusted to keep the forearms parallel to the floor in the manner approved by the typing textbooks. Typists used to traditional typewriters may find the new low-profile keyboards rather too flat, and for this reason companies such as Olivetti and IBM have put adjustable feet at the back to increase the slope.

Other ergonomic points worth checking on include the keyboard colour scheme which should, ideally, distinguish between the standard typewriter keys and the other special computer keys. Computer keyboards differ from typewriter keyboards in having a selection of keys performing special functions. The most notable is the CONTROL or ALT key. This is much like a SHIFT key, in that it enables the existing keys to generate a completely different set of characters (or actions). However, these characters, instead of being upper case letters or punctuation marks, are used for control purposes: to send output to the printer, to scroll one screen at a time, or whatever.

However, too many keys can be confusing, particularly for the newcomer. An intelligently organised colour scheme makes the logic of the layout clear as long as the colours are not so garish as to be distracting. In the early learning stage with an unfamiliar keyboard it helps if the key tops to have a matt finish to avoid reflections.

The typewriter keys should really be laid out where a conventional typist would expect to find them. The Shift and Shift Lock keys may or may not emulate an ordinary typewriter, there are some weird and wonderful systems for producing odd-looking greek characters, beeping sounds and even a total keyboard lockup every time you try to unlatch the capitals – watch out!

Similar remarks apply to special function keys that might be accidentally hit during the course of normal data input. The RESET key on the Apple II is a notorious example. Often the keyboard itself is not to blame, and the solution lies with the manufacturer or programmer making adjustments to the software.

The position of the carriage return (or Enter) key, and of other frequently used keys, is also important. On the IBM Selectric keyboard the Enter key is large and L-shaped and within comfortable reach of what the touch-typist knows as the 'home keys'. The right little finger homes on a punctuation key, there is another punctuation key to the right of it, and to the right of that is the carriage return. Happily this comfortable 'standard' design has in the past been copied widely by other micro-computer and terminal manufacturers although, for no accountable reason, the IBM PC crams in an extra key, moving the carriage return key to the right and out of easy reach.

The overall number of keys can be relevant, too few and you find yourself resorting to six-finger keystrokes, too many and you find yourself paying for hardware which will never be used. A numeric keypad is handy for feeding numbers into the machine, but if all you are doing is wordprocessing, the mere presence of the extra keys adds to the sense of clutter (*but see* soft keys *below*).

On small business machines, particularly those with a separate console, it is commonplace for a keyboard to sport cursor arrows which don't do what they indicate. Don't automatically blame the keyboard – it is more often a sign that the software hasn't been properly tailored for the hardware.

Some keyboards emit an electronic beep but, whether yours does or not, the mechanical switch underneath each key should give enough tactile indication that the character has been generated. Without this subtle 'feel' characters may well be missed, or in the case of automatically repeating keys, incorrectly duplicated.

Soft keys

Some micro keyboards have keys that can be programmed by the user to perform special functions. How useful these 'function keys' are depends on the

The keyboard on the 64, as on all Commodore machines since the early days, uses standard typewriter pitch keys which are full travel, rather than the 'touch sensitive' type.

The keyboard is scanned every one-sixtieth of a second by the interrupt service routine and, if a new key has been pressed, the ASCII code for that key is pushed into a keyboard queue which occupies locations 631 to 640. You can find out the number of characters in the queue by looking at location 198. The maximum number of characters which can be held in the queue is normally 10, but you can lower (though never raise) the limit by changing the value in 649. If you want to (temporarily) stop somebody typing ahead before an INPUT or GET statement, then a POKE 198,0 just before the statement will flush out all the characters in the queue. Location 650 normally contains zero, which means that the cursor control keys, the INST/DEL keys and the space bar, repeat if held down. POKE 650,64 stops this happening, and POKE 650,128 makes all the keys 'auto-repeat'. The delay before the repeat starts is set by the value in 652, and the repeat speed is set in 651.

The function keys aroused a lot of interest when they first appeared with the VIC-20, although there are no commands in the BASIC to support them. If you want to make these keys produce text directly, you have to write a routine in assembly language which tests for one of the keys being pressed, and pushes the string into the keyboard queue. In a program though, you can make these keys (or any keys for that matter) appear to produce text on the screen.

```
100 GET A$="" THEN 100
110 IF ASC (A$) = 133 THEN PRINT "FUNCTION KEY 1"
120 GOTO 100
```

It is a great nuisance if you accidentally hit the STOP key while running an important program. Fortunately it is very easy to disable it. POKE 788,52 to disable STOP key and POKE 788,49 to enable STOP key. Turning off the STOP key also stops the clock and you should re-enable the STOP key before using the cassette system.

supporting software. Keyboards with this facility are called 'programmable', or 'soft' keyboards.

Some keyboard software can store a table of actions for every key while others limit this feature to certain keys (or don't offer it at all). Where this applies, the user can usually amend the table allowing the keyboard to be 'tailored' to a particular application package.

On many keyboards the numeric keypad will be 'soft'. That is to say, with the proper software it can be 'programmed' from a stored pattern on disk (or cassette) to carry out dedicated functions or even to write commonly occurring strings of characters otherwise requiring several keystrokes. There may be some initial confusion in typing '5' to save a file and '7' to move a block of text, but every user gets to know the layout of his keyboard after a few hours (even when it is totally illogical in layout!) and features like this are rarely a problem.

Languages: Alternatives To Basic

Because it is so much a part of the micro world, BASIC is stretched to its limits in applications where its use isn't always appropriate. It was designed to enable beginners to write short programs, but tends to get used for everything from real-time machine control to hospital payroll packages. The confusion about BASIC is heightened by the way it is used in many most home computers as the operating system interface, giving the impression that it is somehow more than just a language.

This is unfortunate, because there are many other languages, overshadowed by the ubiquitous BASIC. Admittedly some are not worth knowing about, and others (like B perhaps) have only historical significance; but most of them offer a micro user something different and worthwhile, even it it's only a new slant on how to think about programming. The snag with sticking with any one language, particularly if you work with a micro on your own, is that you become blinkered into the particular way of thinking about problems dictated by the repertoire of commands and the syntax. As the academics warn: 'program *into* a programming language, not in it'.

Usually all that is needed to put a new language to use is a software package and enough investigative enthusiasm and time to learn a new syntax. When choosing a language you'll obviously want to take into account factors like the speed of processing, memory requirements, data structures required, and whether anybody else is going to have to use or modify the programs you write.

Why not BASIC?

One of the main arguments against BASIC is that it positively *encourages* bad programming. Ease of understanding, debugging and modification are three important considerations when writing a program. Good programming practice includes adhering to the following general principles:

Use `GOTO`s as little as possible, preferably never.
Put only one statement on each line.
Thoroughly comment the program (with `REM`s in BASIC)
Don't jump out of loops with `GOTO`.
Structure the program and data to match the problem.

It is easy to see why BASIC is criticised: multiple statement lines speed up the program; `REM` statements slow it down; it is difficult not to use `GOTO`s from a within loop; and there are to few programming or data structures to enable a neat solution to many problems.

PASCAL and ALGOL

These same arguments are used in reverse to support the very structured and rigorous language, PASCAL (*qv*). There are strict rules as to the order in which a program is constructed, and it is very heavily 'data-typed'. A data type is simply the definition format of a lump of data; whether, for example, it will be an integer, a string, an array of real numbers, a boolean, an array of records, or whatever.

It is possible in PASCAL, for example, to have an array of records, each of different lengths, with each record holding a different set of data types, and so on. Two pieces of data can be compared only if they are of the same type. This certainly makes sure that the programmer observes the principles of structured programming, but can be very restrictive. For instance in strict PASCAL you can't easily assign or compare a string of length 9 with a string of length 10. String handling is where BASIC scores over PASCAL.

However, PASCAL does have a lot going for it other than the fact it promotes well designed programs: it is compiled, so programs run fairly fast, and is also quite efficient on memory space. The language itself was devised by Nicklaus Wirth, and is quite similar to ALGOL.

ALGOL is another well structured language, and has motivated the design of many other languages as well as variants of itself. However, it is not often seen on micros, and PASCAL seems to be about as close as you can get.

Assembler and C

The stringent data typing rules of PASCAL make it unsuitable for low level programming (low in the sense of systems programming or applications requiring real processing speed). Assembler programmers have long been crying out for a high level assembler, a language with the constructs of PASCAL (loops, assignments, procedures, and so on), but with the almost total flexibility of assembler (there is no such concept of data type in assembler, only bits and bytes).

This demand has been met (although on few home micros) with the development of the language C. It was invented by Denis Ritchie in the early 1970s as part of the development of the portable operating system Unix. It derives from a language with very similar flexibility called BCPL. All these languages, C, BCPL, BASIC, PASCAL, ALGOL and others like FORTRAN, are based on a similar idea (assignments, loops, integers, reals, arrays, and what-have-you).

LISP

A few imaginative computer scientists have taken different approaches, and given birth to such things as LISP. LISP officially stands for LISt Processing language, but because of all the brackets needed to write programs in the language the acronym unofficially stands for Lots of Infuriating Stupid Parentheses.

Its basic data types consist of 'atoms' and 'lists'. The letter `B` is an atom, the number `2` is an atom, the collection `B2` is a list, lists being collections of atoms or smaller lists. A simple list is defined by a number of objects in brackets; for example `(A B C)`.

A complex list can be constructed from smaller lists and atoms; for example `(((Ab)C) D E (F G) ((H I J)K)L M)` is a list. The main elements in this list are the lists `((A B)C)`, `(F G)` and `((H I J)K)` and the atoms `D`, `E`, `L` and `M`. Notice that the number of right brackets equals the

number of left brackets in the list. Get used to counting brackets, you'll be doing lots more of this if you take on LISP!

The idea of a function is also fundamental to LISP - for example, `(DIVIDE(24 6))` returns the answer 4. There are a lot of functions in LISP which manipulate lists, but one of the main attractions of this language is that you can define your own functions. A LISP interpreter needs only a small core of fundamental functions, all others can themselves be written in LISP.

Recursion is another fundamental concept in LISP, along with atoms, lists, functions and those blasted brackets. Despite the necessary bracket-counting talent, LISP is actually quite fun to use, being very different from BASIC in structure, but interpreted, so there's no delay in getting your first function into action.

For a fuller discussion on LISP, see the entry under that name.

The logic languages

Not surprisingly, perhaps, there have been a number of offshoots of LISP, using its exciting concept of list processing but with a less tedious syntax. PROLOG is one such language, as are LOGO and POP-2. There are two main ideas in PROLOG: inferences and facts. An inference is a statement like `x nearer-than y if dx less-than dy`. Given the facts 'Leeds, Glasgow, 200 and 400', PROLOG can be pursuaded to draw the conclusion that it is true that Leeds is nearer than Glasgow.

PROLOG concentrates on manipulating symbolic expressions, which makes it suitable for writing expert systems (*qv*) and the like. Its supposed English-like syntax and logical way of working has also suggested the language to educationalists as being an ideal introduction to computers for children.

But in this field it is undoubtedly LOGO which is making all the running. LOGO is much closer to LISP than PROLOG, its main claim to fame being that it is much easier to use. The interactive Turtle graphics associated with LOGO have also helped to make it a favorite in education.

FORTH is another language that allows programmers to define their own keywords. Once defined, any keyword can be used in the same way as any of the more fundamental functions. This makes it very flexible, and the code produced is more efficient than BASIC (that is, faster). It has, however, been criticised as a write-only language. Figure 1 shows a short example of a FORTH program. FORTH programs tend to be even harder to understand, especially if a lot of new keywords have been defined.

A few new languages are begining to appear, including the US defence department's attempt at a totally standard consistent language, ADA, and the British parallel processing language OCCAM.

Languages: APL

APL, A Programming Language, first emerged in the early 1960s when Ken Iverson devised a mathematical notation to teach the new subject of data processing. Iverson moved to IBM and his notation was used to define the working of their model 360, the classic mainframe which put IBM where it is today. Although APL spread through IBM like wildfire during the late 1960s, there were two reasons for its failure to catch on in the microcomputing community: it needed a great deal of processing power, and it was written using a unique and incomprehensible set of symbols.

The first objection has been met with the arrival of desk-top 16-bit machines, but the second objection is still a stumbling block. In spite of this, you only need to spend a few days playing with APL to realise the advantages of its compact and expressive 'funny characters'. There's a lot to be said for a system that can knock out a program in two minutes – a program which might otherwise take a team of COBOL programmers three days to write. For programmers using a more conventional language the symbols can be more than a little daunting, and it can be alarming to find that there are no control structures. If you're used to thinking in terms of the ALGOL-like `DO ... WHILE` loop, or the `IF ... THEN ... ELSE` and `FOR ... NEXT` constructions of BASIC, you will find yourself having to think again should you turn to APL.

APL's only point of similarity with BASIC is that it is interactive, being driven by an interpreter, step by step. It shares with PASCAL and C the capability of recursion (the ability of functions to call themselves). But essentially APL is more a mathematical notation than a computer language.

In particular, the concept of files is very different, and here again you will not find the familiar structure of a string of identical records to be thumbed through one by one by means of a program loop along the lines of `IF NOT EOF(1) THEN...`. Files in APL are tables and lists in which you store your variables, and the interpreter 'sees' the whole table or list all the time. This is an ideal way to handle complicated financial modelling, and also fits in very well with the design of relational databases (*qv*) – between them, two fields where APL has been particularly effective.

It should be said that this 'simultaneous overview' of lists and tables is a theoretical idea which can be made to work reasonably well in practice when the language runs on a mainframe (which it was designed to do) with perhaps as much as 360K of workspace at its disposal. A micro will have to do a lot of old-fashioned disk accessing behind the scenes in order to achieve the same results; but at least this housekeeping is taken care of by the interpreter, and the programmer doesn't have to worry about it.

If you have done a little mathematics, and are broad-minded enough to approach programming in what may well be a very different light, APL shouldn't be too difficult. It is very amenable to adaptation; if you don't like the funny symbols APL has a lovely feature that allows you to call them anything you want. For example, if you don't want to press just one button to sort a heap of numbers, you can write a one-line program to allow you to say, 'Please would you sort these numbers for me?' instead. If you really want to, you could spend a couple of hours and make the language look like BASIC or COBOL. But you might feel the time would be better spent familiarising yourself with the symbols and doing things which often prove to be well beyond the wildest dreams of the poor programmers working in other languages.

APL divides the available memory into partitions called workspaces containing the data, the relevant parts of the program, and room to manoeuver. VIZ:APL, one of the 8-bit implementations of the language running under CP/M, gives you a full megabyte of 'workspace'. It manages to offer such a large memory by being 'virtual' and 'overlaid'. 'Overlaid' means that only those parts of the interpreter that are needed are kept in memory. The rest is stored on the floppy disk and is only loaded when required. 'Virtual' means that all the data, arrays and functions held in the memory are broken up into 'pages'. When the interpreter runs out of space, it writes the least used page to the disk before loading the page it wants into the memory.

Since all this happens without you having to do anything, it appears as though you have a one megabyte memory. The only way you can even sense what is going on when you are running a really big program (or one using a lot of data) is by the way the disk drives seem to permanently on the go during execution.

Languages: BASIC

In 1769, in accordance with a provincial law requiring towns of 50 or more householders to set up schools, the small town of Hanover, sheltering among the maple trees between the White Mountains of New Hampshire and Green Mountains of Vermont, acquired a private institute of higher learning called Dartmouth College. Some time later the college acquired a mainframe computer and FORTRAN.

Two professors at the college, John Kemeny and Thomas Kurtz, thought FORTRAN unsuitable for introducing beginners to the world of programming. So in the early 1960s they introduced a new approach on the college's time-sharing terminals. They called it Beginners All-purpose Symbolic Instruction Code, a mouthful of words which boil down to the useful acronym BASIC.

BASIC quickly took hold on mainframes all across the United States, but because the root language was essentially limited to a handful of statements, each new implementation added new and different commands. But at that time mainframes and minicomputers were quite used to being 'islands entire unto themselves' and nobody minded very much about the problems of incompatibility.

BASIC dialects

In the micro world there are so many dialects of BASIC that somebody has suggested the language be renamed Babel. Historically the daddy of them all is MBASIC, from the software house of Microsoft, and that itself exists in a baffling variety of forms. MBASIC may not have started the whole microcomputer phenomenon, but it was certainly in right at the beginning, and the dialects used on early pioneer micros including the TRS-80 Level II, the Apple and the Pet are based on it.

The Microsoft story begins with two young students, Bill Gates and Paul Allen, at college together in Seattle. They had the use of a DEC minicomputer and managed to unearth some bugs in the operating system. DEC offered them free use of the machine for a year if they could find ten more. Legend records that they found 40, and during that time they thoroughly assimilated DEC BASIC.

Towards the end of 1974 a young firm with the grand name of Micro Instrumentation and Telemetry Systems (MITS) came out with a computer for under $500 – the first to use the new 8080 chip, hitherto only used for cash registers and pinball machines. It was called the Altair 8800, and in eight short months MITS had sold 4000 of them. Those sales were partly due to the highly praised Extended Disk BASIC available on the machine, a dialect bearing an uncanny resemblance to DEC BASIC. It had arrived on the machine by courtesy of Allen and Gates. At Allen's suggestion he and Gates had sold Altair the idea of putting the language onto the new machine – then they did it, in precisely three days!

They arrived at the Altair offices a day before the demonstration with their BASIC on paper tape. It was punched in the wrong code, and it took them all that night to convert by hand. Next day it ran first time, and MBASIC was born.

Defining your own functions

One advantage of BASIC is the way you can add your own functions to the language by using **DEF FN**. Next time you find yourself repeating a particular calculation inside a program, the chances are that you could make things quicker, shorter and clearer by defining it as a function.

Suppose you were a fanatical doubler of numbers and kept writing things using '2*', like this, all over your programs:

```
LET A=2*B
B=2*(5*7 + 6)
PRINT 2*LEN(A$)
```

What you would do is define the function at the start of the program by writing **DEF FN** and a name for the function, perhaps **D**. Next you would need an argument, or variable number, for the function **D** to work on. Let's call it **X**. Our definition then becomes **DEF FND(X)**. Whatever **X** is, the value of the function is equal to **2*X**. So a complete definition would be **DEF FND(X) = 2*X**.

Once this function-defining statement has been executed, which we make sure happens early in the program by giving it a low line number, we have a new function at our fingertips. Whenever you feel the urge to double a number, it is only necessary to use a statement like **LET A = FND(B)** or **B = FND(5*7 + 6)** or **PRINT FND(LEN(A$))** and so on.

You'll discover that different BASICs have different rules for defined functions. Some let you use long names like **FNDOUBLE**, or permit you to have more than one argument, in which case you could do something like return the largest of two numbers with:

```
DEF FNmax(A,B)=(A+B+ABS(A-B))/2.
```

Other versions of BASIC allow functions to return string values or – best of all – let the definition of a function run into any number of BASIC lines, bracketed with the statements **DEF FN**, and **FNEND**. Given all these enhancements and a bit of work, you could create a function FNmenu, to which you passed a list of options and which returned the number of the option chosen. But don't expect this ingenious construction to be portable. Most BASICs are limited to one argument, one line, and one numeric result – the constraints we observed in the doubling example.

Useful defined functions

But even within these limitations there is a lot you can do. The following examples should give you some ideas.

These functions provide sine, cosine and tangent in *degrees* rather than BASIC's MORE usual *radians*. They assume that the variables **HE=180** and **PI=3.1415927** have already been set up:

```
DEF FNSN(X)=SIN(X*PI/HE)
DEF FNCS(X)=COS(X*PI/HE)
DEF FNTN(X)=TAN(X*PI/HE)
```

Most BASICs also calculate logarithms to the 'natural'

base *e* rather than the usual base 10. You may well find that logs to the base 10 are more useful in real life, so here is the function to make the conversion:

```
DEF FNL(X)=LOG(X)/LOG(10)
```

The **RND** function on most BASICs generates a random number (*qv*) between 0 and 1. To generate a whole number between 1 and *N* use:

```
DEF FNR(N)=INT(RND(1)*N)+1
```

To round results to a set number of decimal places use:

```
DEF FNRO(X)=INT(X*P+.5)/P
```

where **P=10** for 1 decimal place, **P=100** for two decimal places and so on.

To round the nearest whole number this function becomes:

```
DEF FNN(X)=INT(X+.5)
```

Getting up speed

The biggest bore with programs written in interpreted BASIC is speed – they run s-l-o-w-l-y . . . On disk-based machines **compilers** are available to crunch BASIC **source code** into fast-running **machine code** ('object code'), but the price is often prohibitive, and they can be complicated to use. But even those of us stuck with interpreted BASIC can benefit from studying some of the techniques used by a good compiler.

Despite its speed improvement over an interpreter, a compiler is still a lot slower than a program written in assembler (*qv*). The advantage of a hand-coded program is that it can be carefully optimised to work as fast as possible, or it can be compressed into the minimum of memory space. But a good compiler will try to catch up on assemblers: while translating the BASIC source code down to machine level, it uses assembler techniques to improve the coding. A compiler such as this is called an **optimising compiler**.

By using some of the techniques incorporated in an optimising compiler and working backwards, it is possible to improve the performance of a program written to run under interpreted BASIC.

Probably the most important construct in all computer languages is the loop. It is certainly the most time-consuming, which makes it the best place to begin when attempting to improve efficiency.

An optimising compiler will try to improve your program in a number of ways, including:

Evaluating mathematical expressions once only where possible.
Removing invariants from loops so that the same code isn't uselessly re-run.
Simplifying common sub-expressions to speed up calculations.

Suppose you are in the process of writing a program which uses the mathematical expression **2.1498 * 18.999 * (133/3.14159)**. BASIC spares you the tedious business of having to work this out every time you need it in a program line, and you may well be tempted to insert it in the source code as it stands.

An optimising compiler would work out the value of this calculation at compile time (not at run time) and insert the result once and for all into the object code. But with an interpreter, when you enter the expression exactly as above into a BASIC program, will have to wade through the calculation each time it encounters it. If it is speed of execution you are after, here is a clear case for hand-optimising the program: work out the value yourself before typing it in. With a BASIC interpreter available, it is easy enough to use the direct mode with an un-numbered command:

```
PRINT 2.1498*18.999*(133/3.14159)
```

An **invariant** in a loop is an assignment or an expression that does not change during the execution of the loop. In other words, the assignment (or expression) could just as well be outside the loop. For example:

```
FOR L=1 TO 10
PI=3.14159
TT=TT+L*P
NEXT L
```

The assignment to **PI** is an invariant, and should not be within the **FOR ... NEXT** loop. This may seem obvious – not a mistake you would be likely to make when the program is first typed in. But after all the debugging, modifying and rehashing that tends to go on with BASIC source code it is surprising what silly constructs emerge.

The third optimisation procedure is slightly more complicated. Say, for example, a program contained the following two lines:

```
100    X=(Z*100*Y)/13+128.6
110    V=4*(Z*100*Y)/13-200
```

The expression **(Z*100*Y)/13** is known as a *common sub-expression*, because it appears identically in both lines. The program is made more efficient by calculating this once and assigning it to another temporary variable:

```
90     T=(Z*100*Y)/13
100    XC=T+128.6
110    V=4*T-200
```

Not only is the program more efficient, but it looks neater and is easier to understand.

There is, however, one major problem, and one of the reasons why **GOTO**s are regarded as an evil influence in structured programming. If the statement **GOTO 110** were present somewhere else in the program, then it would not be safe to make the substitution. This is because the variables **Z** and **Y** may well be different from what they were at line 90. In this event the expression we have assigned to **T** is not a common sub-expression. Optimising compilers, not surprisingly, are usually implemented for well-structured languages. In BASIC it is up to the programmer to make the necessary changes.

Figure 1 shows a program with all the three sins mentioned and Figure 2 shows an optimised version which runs something like 150 percent faster.

The program isn't guaranteed to make a great deal of sense but it does illustrate the concepts of optimisation.

```
5 REM LANGUAGES: BASIC
10 REM SLOW PROGRAM
15 TT=0
20 FOR L=1 TO 200
30 ME=2.1498*18.9999*(133/3.14159)
40 Z=ME+L*2
50 Y=ME
60 X=(Z*100*Y)/13+128.6
70 V=4*(Z*100*Y)/13-200
80 TT=TT+(X*V)
90 NEXT L
100 PRINT "MEANINGLESS RESULT=";TT
110 END
```

Fig. 1. A sinfully slow program

```
5 REM LANGUAGES: BASIC
10 REM OPTIMISED PROGRAM
15 TT=0
20 ME=1729.22501
30 FOR L=1 TO 200
50 Z=ME+L*2
65 T=(Z*100*ME)/13
70 X=T+128.6
80 V=4*T-200
90 TT=TT+(X*V)
100 NEXT L
110 PRINT "MEANINGLESS RESULT=";TT
120 END
```

Fig. 2. An optimised program

Screen-handling

Producing neat screen displays is an important part of good programming, but is not something BASIC is very good at. It takes hard work and a sizeable portion of a program to get good results. Many commands have been added to BASIC to make life easier but the enhancements add greatly to the confusion between different dialects of BASIC. Ironically the more neatly-presented a program is, the harder it will be to move it from one machine to another.

Two things to aim for when writing screen-handling routines are clarity and consistency. When preparing the screen, it is a good idea to sketch out your ideas first on a grid sheet. Clear the screen regularly and present only current and relevant information. Make all a program's displays consistent by allocating certain parts of the screen for titles, error reports, data entry and so on. This is easier if you use a standard set of routines for all display and entry work throughout the program. Write a routine to print titles, one to print error messages and so on. Centre titles using a statement like:

```
50   T$ = "THIS IS A TITLE":  PRINT TAB ((40-LEN(T$))/2);  T$
```

This works for a 40-column screen, so change the 40 if yours is different. Words should never be split at the ends of lines. It's not hard to construct a BASIC routine to 'word-wrap' your text by keeping a count of the characters since the last carriage-return as they go up on the screen. If the count exceeds the position where you want your margin, simply look for the next space and substitute a carriage-return and linefeed.

But this kind of smart text-handling may not be necessary for most applications, and in any case may slow up the program unacceptably.

A simpler solution, when keying in the program and beginning a **PRINT**" command, is to take the column occupied by the " as the left margin of the text. So if you enter text which runs over more than one line, check that none of the words in the second and following lines cross the inverted-comma column. If they do, you know they will be split when the running program prints the text. In order to avoid this, delete the offending word and enter spaces until the cursor is one space to the right of the " column, then retype the word and carry on entering the text.

Take care when displaying columns of numbers:

```
1
24
407
3
78
```

doesn't look right. Instead, it's better to have the units, tens and hundreds digits lined up:

```
  1
 24
407
  3
 78
```

Some BASICs have a **PRINT USING** command which makes this sort of thing easier, others need a bit of work. Suppose we want to print a column of positive whole numbers so that the units digit appears in column number 10 of the screen. If the number, let's say in the variable X, is one digit long, we need to start printing in column 10. If X is two digits long, we should start in column 9 and so on. So we need to tab to 11 minus the number of digits in X. One way to count the number of digits is to use **LEN(STR$(X))**. Here **STR$** makes X into a row of characters and the **LEN** counts how many there are. A word of warning – some BASICs add spaces when you use **STR$**. This method should yield a statement that reads like this:

```
100   PRINT TAB(11 - INT(LOG(X)));X
```

Here the statement returns the logarithm of X, knocks off

everything to the right of the decimal point to make it an integer, and uses this to calculate the tab. But before using this one, make sure that your **LOG** statement doesn't return a natural log; the routine needs a common log working to base 10.

String searching

Searching for one small string within another is known as a substring search. Many BASICs include an **INSTR** or similar command to do this, but for those that don't it is also possible to use a short routine along the following lines:

```
1000   P=1
1010   IF P+LEN(KS)>LEN(S$)+THEN P=0 : RETURN
1020   IF MID$(S$,P,LEN(K4))<>KS THEN P=P+1 : GOTO 1010
1030   RETURN
```

To use the routine set **S$** to the string to be searched, **K$** to the substring to search for, and then **GOSUB 1000**. If you want to start the search at a particular character position in **S$**, then set the value of **P** to the number of that character and then **GOSUB 1010**. The subroutine will return with **P=0** if **S$** does not contain **K$** at all. Or it will return with **P** set to the number of the character where **K$** first occurs in **S$**. If the strings contain a mix of upper and lower case, don't forget that the computer views **a** and **A** as different characters.

Interpreter or compiler?

All the BASICs used on smaller micros are interpretative, but many larger machines have the option of using BASICs which are either interpreters or compilers. The difference boils down to the way the machine translates the BASIC statements you type (the source code) into the language used by the microprocessor (the object code).

An interpreter takes each statement (for example **10 PRINT 2+3**) one at a time, translates it into object code, and then executes it. A compiler on the other hand swallows the whole of the source code in one go, translates it *all* into object code, and stores it in memory ready for processing.

The advantage of interpretative BASIC is that you can run a program immediately and see what happens. The interpreter throws your mistakes back at you when and where you make them, rather than obliging you to disentangle error messages later from a compiler printout. You can even try out instructions without having to write a program. Most versions of the language allow a **direct mode** in which single lines can be evaluated, so that if:

```
MID$("NEWHAMPSHIRE",3,4)
```

returns **WHAM** you'll know that you and your BASIC are in accord about the meaning of the **MID$** (pronounced 'mid-string') function.

But the process of translation has to take place line by line each time the program is run. In compiled BASIC the translation into machine code only has to be done once, although for a large program this process may take considerable time. Thereafter the program is stored as object code, and will run at optimum speed each time the program is executed.

For this reason running a compiled program is usually a lot quicker (a factor of ten times is typical). But compilers have the disadvantage of being complicated for the beginner to use. For this reason early experiments with a compiler can be discouraging, if not downright infuriating, involving repeated editing and re-running the source code through the mill of the compiler as bugs are slowly eliminated.

It would be easier if you could simply issue a command like 'COMPILE' and leave the machine to get on with it. But usually the compiler includes a library of routines, which are held in object-code form in a separate file, and this has to be somehow included in the compilation process. At compile time the compiler looks through the source code the programmer has written to see which functions are being called on, and then searches the library for the appropriate object code to perform that function. Once identified, this library routine has to be linked in to the complete program. In systems where backing store is at a premium this can involve much swapping of disks.

Compilers used with cassette-based micros have to fit everything into the main memory at once, because it would be impractical to link in routines from tape. Tape compilers also have to perform all the translation within the confines of memory. This can get very cramped, and compiler writers have to make their programs as simple as possible to fit. As a result, most compiled languages available on cassette (the most popular being PASCAL) are cut-down versions of their disk-based equivalents.

If the programs you write are likely to be used only occasionally, and timing isn't crucial, it is unlikely you will need a compiled BASIC. But for particular programs that are used frequently, and where speed is crucial – like the word-count utility used in writing this book – a good compiler could be essential.

The advantage of being able to run both an interpreter and compiler using the same source code (a particular strength of MBASIC of considerable appeal to writers of business programs) is that you can use the interpreter version during the development and testing stage, and then compile to produce the final version when you are ready. A few fine-tunings may be necessary between the interpreted and compiled version of the code to allow for minor incompatibilities.

GWBASIC

Microsoft BASIC has remained a sort of de-facto standard ever since its first implementation on the MITS micro in the mid-1970s. But in 1982 the arrival of IBM on the micro scene brought about an extensive reorganisation of the original interpreter into what is now known as GWBASIC. Like its forerunner, the GWBASIC philosophy is a sort of rallying point around which individual hardware manufacturers broadly base their own products, whether directly derived from Microsoft or not.

UNUSUAL MACHINE DEPENDENT FEATURES IN BASIC

MACHINE/ COMMAND	FUNCTION
ATARI	
ADR	RETURNS ADDRESS OF SPECIFIED STRING
CLOG	LOG BASE 10
DEG/RAD	ALTERNATE BETWEEN DEGREES AND RADIANS
STATUS	CALLS STATUS ROUTINE FOR A I/O PORT
BBC	
COUNT	COUNTS NUMBER OF CHARS TO SCREEN/PRINTER
DEG	CONVERTS RADIANS TO DEGREES
DIV	WHOLE NUMBER DIVISION
ENDPROC	END OF PROCEDURE DEFINITION
EVAL	ABILITY TO INPUT EQUATIONS INTO PGM
HIMEM	SETS HIGHEST MEMORY
LISTO	FORMATED INDENTED LIST OPTION
LOCAL	SETS LOCAL VARIABLES
LOMEM	SETS LOWEST MEMORY
MOD	REMAINDER AFTER DIVISION
OLD	OPPOSITE TO NEW
PAGE	MEMORY PAGING FACILITY
PROC	DEFINE PROCEDURE
RAD	DEGREES TO RADIANS
REPEAT/UNTIL	LOOP STRUCTURE
REPORT	CONVERT ERROR NO. TO DESCRIPTION
TIME	SET/READ INBUILT CLOCK
TOP	MEMORY SET
VPOS	VERTICAL POSITION OF CURSOR
CBM 64 AND VIC 20	
CMD	RE-ROUTES OUTPUT I.E. SCREEN TO PRINTER
SYS	START EXECUTION OF MACHINE LANGUAGE PGM
SPECTRUM	
BIN	BINARY NOTATION INPUT

Fig. 3. Unusual machine dependent features

SCREEN FACILITIES

MACHINE	MODE		RESOLUTION	SCREEN RAM (FROM TO)
ATARI	0	TEXT	40 X 40	
	1	"	20 X 24	
	2	"	20 X 14	
	3		40 X 24	
	4		80 X 48	
	5		80 X 48	
	6		160 X 96	
	7		160 X 96	
	8		320 X 192	
BBC	0		640 X 256	
	1		320 X 256	
	2		160 X 256	
	3	TEXT		
	4		320 X 256	
	5		160 X 256	
	6	TEXT		
	7	TELETEXT		
CBM 64			40 X 25	1024-2023 TEXT 55296-56295 COLOUR
CBM 20			22 X 23	7680-8185 TEXT 38400-38905 COLOUR
SPECTRUM			256 X 176	

Fig. 6. Screen facilities

COLOURS AVAILABLE

MACHINE/ MODE	RED	PINK	BROWN	ORANGE	LIGHT YELLOW	LIGHT GREEN	DARK GREEN	DARK CYAN	LIGHT BLUE	DARK BLUE	VIOLET	BLACK	GREY	AQUA	WHITE
ATARI 5 COLOUR	3	-	-	0	-	-	1	-	-	-	-	2	-	4	-	-	-
ATARI 16 COLOUR	-	4	3	2	1/15	13	12	14	11	9	7/8	10	6	-	0	-	-
BBC MODE 1/5	1	-	-	-	2	-	-	-	-	-	-	-	-	0	-	-	3
BBC MODE 2	1	-	-	-	3	-	2	-	6	-	4	-	5	0	-	-	7
CBM 64 (POKES)	2	10	9	8	7	13	5	-	3	14	6	-	4	0	11/12/15	-	1
CBM 64 (CHR$)	28	-	-	-	158	-	30	-	159	-	31	-	156	144	-	-	5
CBM 20 (POKES)	2	10	9	8	7	13	5	-	3	14	6	-	4	0	11/12/15	-	1
SPECTRUM	2	-	-	-	6	-	4	-	5	-	1	-	3	0	-	-	7

NOTES: ATARI Light Orange = yellow, red/orange = brown, turquiose = dark blue
BBC codes are forground - add 128 for background
BBC codes in mode 2 from 7-15 flash complimentary of codes 0-7
CBM 64 11,12 and 15 are grey levels

Fig. 4. Available colour

GENERALIZED PLOTTING AND GRAPHICS CONVERSIONS

FUNCTION	ATARI	BBC	SPECTRUM
BORDER - SET BORDER COLOUR OF THE SCREEN	-	-	BORDER N
CIRCLE - DRAW CIRCLE	-	-	CIRCLE X,Y,R
CLEAR - CLEAR SCREEN	-	CLS AND CLG	CLS
DRAW - DRAWS A LINE FROM PRESENT POSITION	DRAWTO X,Y	DRAW X,Y	DRAW X,Y
FILL - FILLS A SHAPE WITH COLOUR	XIO SEE LATER	-	-
LINE - DRAWS A LINE BETWEEN TWO POINT	-	-	-
MODE - SET GRAPHICS MODE	GRAPHIC 0-8	SEE LATER	-
MOVE - MOVE CURSOR WITH OUT DRAWING	POSITION X,Y	MOVE X,Y	PLOT C,X,Y
POINT - DRAWS A POINT ON THE SCREEN	PLOT X,Y	-	PLOT C,X,Y
SCAN - EXAMINS POINT ON THE SCREEN	LOCATE X,Y	POINT X,Y	POINT (X,Y)
INK - SETS COLOUR FOR NEXT PLOTS	COLOR=X	COLOUR=X SEE LATER	INK N

Fig. 5. Generalised plotting and graphics features

MACHINE SPECIFIC PLOTTING AND GRAPHIC COMMANDS

MACHINE/ COMMAND	FUNCTION
ATARI	
GET	GETS BYTE FROM SCREEN AFTER POSITION CMD
PUT	PUTS BYTE TO SCREEN AFTER POSITION CMD
SETCOLOR A,B,C	SET COLOUR REGISTER, A=REGISTER (0-4), B=HUE NUMBER (0-15) AND C=BRIGHTNESS 0-14
XIO	FILLS IN SHAPES IN HORIZONTAL LINES UNTIL NON ZERO PIXELS ARE FOUND
BBC	
GCOL A,B	COLOUR PLOT WITH VARYING EFFECT A = 0 PLOT COLOUR A = 1 "OR" PLOT WITH PRESENT COLOUR A = 2 "AND" PLOT WITH PRESENT COLOUR A = 3 "EXOR" PLOT WITH PRESENT COLOUR A = 4 "INVERT" PLOT WITH PRESENT COLOUR B > 127 BACKGROUND COLOUR B < 128 FOREGROUND COLOUR
PLOT K,X,Y	COMPLEX PLOT FACILITY K = 0 MOVE RELATIVE TO THE LAST POINT K = 1 DRAW LINE RELATIVE FROM LAST POINT IN CURRENT FOREGROUND COLOUR K = 2 AS ABOVE IN INVERSE COLOUR K = 3 AS ABOVE IN BACKGROUND COLOUR K = 4 MOVE WITHOUT DRAW K = 5 DRAW LINE ABSOLUTE FROM LAST POINT IN CURRENT FOREGROUND COLOUR K = 6 AS ABOVE IN INVERSE COLOUR K = 7 AS ABOVE IN BACKGROUND COLOUR K = 8-15 AS 0-7 BUT LAST POINT NORMAL K = 16-23 AS 0-7 BUT DOTTED LINE K = 24-31 AS 0-7 BUT NO LAST POINT K = 64-71 AS 0-7 BUT ONLY SINGLE POINT K = 80-87 AS 0-7 BUT FILL TRIANGLE
VDU	SCREEN WINDOWING FACILITY
CBM VIC 20	
POKE 36879,X	GIVES DIFFERENT SCREEN/BORDER COLOUR SCREEN POKE BORDER COLOUR(+) BLACK 8 0 WHITE 24 1 RED 40 2 CYAN 56 3 PURPLE 72 4 GREEN 88 5 BLUE 104 6 YELLOW 120 7 ORANGE 136 LT.ORANGE 152 PINK 168 LT.CYAN 184 LT.PURPLE 200 LT.GREEN 216 LT.BLUE 232 LT.YELLOW 248 I.E. POKE 90 FOR GREEN WITH RED BORDER (88+2)
SPECTRUM	
BRIGHT N	SET THE BRIGHTNESS LEVEL OF FOLLOWING CHARS 0 = NORMAL 1 = BRIGHT 8 = TRANSPARENT
DRAW X,Y,A	DRAW LINE AND ROTATE BY ANGLE "A"
FLASH	FLASH LEVEL OF FOLLOWING CHARS 0 = STEADY 1 = FLASH 8 = NO CHANGE
INK N	0-7 COLOURS 8 = TRANSPARENT 9 = CONTRAST
INVERSE N	INVERSE CHARACTER 0=NORMAL,1=INVERSE
OVER N	OVERPRINT AND MIX 0=COVER,1=MIX
PAPER N	SETS PAPER COLOUR
POINT (X,Y)	EXAMINE SCREEN 0=BACKGROUND,1=INK
SCREEN$ (X,Y)	RETURNS CHARACTER FROM SCREEN

Fig. 7. Machine dependent plotting and graphics features

At heart GWBASIC is the same old interpreter, rewritten to take advantage of new (primarily 16-bit) hardware. It supports high-resolution graphics, sound, joysticks, a light pen and programmable function keys. The most immediate improvement is in program editing. Instead of the line editor of MBASIC, GW now has a built-in full-screen editor.

Historically, the MBASIC line editor was based on the old idea of the terminal as a paper teletypewriter. Insertion or deletion of characters tended to leave an unkempt line full of editing symbols which then had to be tidied with a 'rewrite line' command. GWBASIC lets you move the cursor all over the screen, back and forth across words, characters and lines. As you insert and delete, the screen tidies itself automatically, rather like a word-processing program. Once you are satisfied with the edit, pressing the 'Return' key feeds the revised line to the BASIC interpreter.

Function keys are just one of the many different 'events' that GWBASIC can trap. With the keys set up properly, the **ON KEY** statement will branch to a particular subroutine whenever the corresponding function key is pressed. So you can use the function keys to interrupt a program from whatever it was doing, make it run a separate routine to deal with the key, and then resume where it left off. You can use the **KEY** statement to assign any sequence of characters to each function key. When you boot GWBASIC it automatically sets the keys to useful BASIC commands, such as **RUN**, **LIST** and **FILES**. The current key settings can be automatically displayed on the 25th line of the screen, where this is available.

In GWBASIC you have full control over the timing of interrupts so that delicate operations such as writing files are not disturbed. One obvious use for this feature is to provide **HELP** and **PAUSE** keys. Between them these two methods of function-key support make it really easy to be friendly to users. A similar system of interrupts is available for the 'Fire' button on joysticks, and to support a light pen. It is also used to give control over a communications port.

GWBASIC supports a whole new set of devices in its file-handling statements, including multiple printers, the keyboard and even a cassette recorder. More importantly, **COM1:** and **COM2:** are two communications channels. You can send and receive information using these, to and from modems, keyboards, printers and other computers.

All the necessary information, such as baud rates, parity bits and so on, can be set by the BASIC program. **ON COM** works like **ON KEY** – a program can be left to run normally until it detects the presence of information at one of the serial ports. Then it will branch to deal with the interruption, returning to its original job when the communication is over. This opens up possibilities for all sorts of sophisticated applications.

GWBASIC also supports graphics and sound. **SCREEN** is used to select graphics screen and resolution values. You can switch particular dots on and off (**SET** and **RESET**), see if a dot is set (**POINT**), draw lines (**LINE**), circles and curves (**CIRCLE**), and shapes (**DRAW**). Two very useful commands are **GET** and **PUT**, which (unlike the similarly named commands in other BASICs, where they deal with

STANDARD CONVERSTIONS FROM MBASIC

<SUPPLIED BY USER> $ = STRING F = FINISH VN = VARIABLE NAME
[] = OPTIONAL S = START LN = LINE NUMBER AL = ARGUMENT LIST
N = UMBER L = LENGTH IN = INCREMENT OF = OFFSET

FUNCTION	MBASIC	ATARI	BBC	CBM 64	CBM VIC 20	SPECTRUM
ABSOLUTE VAL (IE NO -VE)	ABS(N)	ABS(N)	ABS(N)	ABS(N)	ABS(N)	ABS(N)
ASCII VAL OF STRING	ASC($)	ASC($)	ASC($)	ASC($)	ASC($)	CODE($) NOT ASCII
ARCTAN IN RADIANS	ATN(N)	ATN(N)	ATN(N)	ATN(N)	ATN(N)	ATN(N)
AUTO LINE NUMBERING	AUTO [LN [,IN]]	-	AUTO [LN [,IN]]	-	-	-
CALL ASSEMBLY LANGUAGE ROUTINE	CALL VN,(AL)	-	CALL VN,(AL)	SYS(ADR)	SYS ADR	-
RETURNS CHARACTER OF ASCII CODE	CHR$(N)	CHR$(N)	CHR$(N)	CHR$(N)	CHR$(N)	CHR$(N)
NULL ALL NUMERIC VARIABLES	CLEAR [,N]	CLR	CLEAR	CLR(EXP)	CLR	CLEAR
CONTINUE PROGRAM EXECUTION	CONT	CONT	-	CONT	CONT	CONTINUE
COSINE IN RADIANS	COS(N)	COS(N)	COS(N)	COS(N)	COS(N)	COS(N)
STORE DATA FOR READ STATEMENT	DATA	DATA	DATA	DATA	DATA	DATA
DEFINE A FUNCTION	DEF FN VN[AL]	-	DEF FN VN[AL]	DEF FN VN=EXP	DEF FN VN=EXP	DEF FN VN VAR=EXP
START ADDRESS OF ASM ROUTINE	DEF USR[DIGIT]=N	-	-	-	-	-
DELETE RANGE OF LINE NO.	DELETE [LN][-LN]	-	DELETE [LN][-LN]	-	-	-
DEFINE MAXIMUM ARRAY SIZES	DIM <ARRAY LIST>	DIM <ARRAY LIST>	DIM <ARRAY LIST>	DIM <ARRAY LIST>	DIM <ARRAY LIST>	DIM <ARRAY LIST>
PROGRAM EDIT FACILITY	EDIT LN	SCREEN EDIT	SCREEN EDIT	SCREEN EDIT	SCREEN EDIT	EDIT LN
TERMINATE PROGRAM	END	END	END	END	END	-
REMOVES AN ARRAY	ERASE <LIST VAR>	-	-	-	-	-
RESERVED VARIABLE FOR ERROR CODE	ERR	-	ERR	-	-	-
RESERVED VARIABLE FOR ERROR LINE	ERL	-	ERL	-	-	-
SIMULATE OR DEFINE ERROR CODES	ERROR N	-	-	-	-	-
E TO THE POWER OF EXPRESSION	EXP(N)	EXP(N)	EXP(N)	EXP(N)	EXP(N)	EXP(N)
START OF LOOP ROUTINE	FOR<VN>TO<VN>STEP<IN>	FOR<VN>TO<VN>STEP<IN>	FOR<VN>TO<VN>STEP<IN>	FOR<VN>TO<VN>STEP<IN>	FOR<VN>TO<VN>STEP<IN>	FOR<VN>TO<VN>STEP<
FREE RAM	FRE(N)	FRE(N)	HIMEM-TOP	FRE(N)	FRE(N)	FRE(N)
BRANCH TO SUBROUTINE	GOSUB LN	GOSUB LN	GOSUB LN	GOSUB LN	GOSUB LN	GOSUB LN
BRANCH UNCONDITIONAL TO LINE NO	GOTO LN	GOTO LN	GOTO LN	GOTO LN	GOTO LN	GOTO LN
HEX VALUE	HEX$(N)	-	?	-	-	HEX$(N)
PERFORM ACTION ON RESULT OF IF	IF/THEN/ELSE	IF/THEN	IF/THEN/ELSE	IF/THEN	IF/THEN	IF/THEN
GETS ONE CHARACTER FROM TERMINAL	INKEY$	-	GET	GET VN	GET VN	INKEY$
READS BYTE FROM PORT	INP(N)	-	-	-	-	-
INPUT FROM TERMINAL	INPUT	INPUT	INPUT	INPUT	INPUT	INPUT
LARGEST INTEGER	INT(N)	INT(N)	INT(N)	INT(N)	INT(N)	INT(N)
IN STRING SEARCH	INSTR([OF,]$,Y$)	-	INSTR([OF,]$,Y$)	-	-	-
LEFT MOST NO OF CHARACTERS	LEFT$($,LN)	LEFT$($,LN)	LEFT$($,LN)	LEFT$($,LN)	LEFT$($,LN)	STRING (TO FINISH)
RETURNS LENGTH OF STRING	LEN($)	LEN($)	LEN($)	LEN($)	LEN($)	LEN($)
ASSIGN VALUE TO VARIABLE	LET	LET	LET	LET	LET	LET
LISTS ALL OR PART OF PROGRAM	LIST LN OR LN-LN	LIST LN OR LN-LN	LIST LN OR LN-LN	LIST LN OR LN-LN	LIST LN OR LN-LN	LIST LN
AS ABOVE BUT TO A PRINTER	LLIST AS ABOVE	LIST "P:"	CTRL B THEN ABOVE	OPEN 4,4:CMD4 FIRST	OPEN 3,4:CMD3 FIRST	LLIST AS ABOVE
RETURNS NATURAL LOG	LOG(N)	LOG(N)	LN(N),LOG(N)=BASE10	LOG(N)	LOG(N)	LN(N)
PRINT TO THE PRINTER	LPRINT	-	-	AS LLIST	AS LLIST	LPRINT
VIEW/REPLACE PORTION OF STRING	MID$ ($,S,L)	MID$ ($,S,L)	MID$ ($,S,L)	MID$ ($,S,L)	MID$ ($,S,L)	STRING (START TO
CLEARS OUT PROGRAM FROM RAM	NEW	NEW	NEW	NEW	NEW	NEW
REPEAT PART OF FOR LOOP	NEXT	NEXT	NEXT	NEXT	NEXT	NEXT
RETURNS OCTAL VALUE	OCT$	-	-	-	-	-
ERROR TRAP ROUTING	ON ERROR GOTO LN	TRAP LN/VAR/EXP	ON ERROR GOTO LN	ON ERROR GOTO LN	ON ERROR GOTO LN	-
BRANCH TO LINES ON VAL OF EXP	ON GOSUB LN,LN,..	ON GOSUB LN,LN,..	ON GOSUB LN,LN,..	ON GOSUB LN,LN,..	ON GOSUB LN,LN,..	-
AS ABOVE	ON GOTO LN,LN,...	ON GOTO LN,LN,...	ON GOTO LN,LN,...	ON GOTO LN,LN,...	ON GOTO LN,LN,...	-
SETS LOWEST ARRAY VAL (0 OR 1)	OPTION BASE	-	-	-	-	-
SENDS BYTE TO OUTPUT PORT	OUT PORT,BYTE	-	-	-	-	OUT PORT,BYTE
EXAMINE BYTE IN RAM	PEEK(N)	PEEK(N)	? ADR	PEEK(N)	PEEK(N)	PEEK(N)
INSERT BYTE INTO RAM	POKE ADR,BYTE	POKE ADR,BYTE	? ADR,BYTE	POKE ADR,BYTE	POKE ADR,BYTE	POKE ADR,BYTE
PRINT TO SCREEN	PRINT	PRINT	PRINT	PRINT	PRINT	PRINT
PRINT WITH FORMAT INSTRUCTION	PRINT USING	-	PRINT USING	-	-	-
RESEED RANDOM NUMBER GENERATOR	RANDOMIZE	RND(-EXP)	RND(-EXP)	RND(-TI)	RND(-TI)	RAND(EXP)
READ VALUES IN DATA STATMENTS	READ	READ	READ	READ	READ	READ
EXPLANATORY REMARK-NOT EXECUTED	REM	REM	REM	REM	REM	REM
RENUMBER PROGRAM LINES	RENUM NEW,OLD,INC	-	RENUM NEW,INC	-	-	-
ALLOWS REREAD OF DATA STATMENT	RESTORE [LN]	RESTORE [LN]	RESTORE [EXP]	RESTORE	RESTORE	RESTORE [LN]
RESUME AFTER ERROR HANDLING	RESUME	-	-	-	-	-
RETURN FROM SUBROUTINE	RETURN	RETURN	RETURN	RETURN	RETURN	RETURN
RIGHT MOST NO OF CHARACTERS	RIGHT$($,L)	STRING(S)	RIGHT$($,L)	RIGHT$($,L)	RIGHT$($,L)	STRING (START TO)
RANDOM NO BETWEEN 0 - 1	RND(N)	RND(N)	RND(N)	RND(N)	RND(N)	RND
RUN PROGRAM	RUN [LN]	RUN	RUN [LN]	RUN [LN]	RUN [LN]	RUN [LN]
SIGN OF NO. +VE, -VE OR 0	SGN(N)	SGN(N)	SGN(N)	SGN(N)	SGN(N)	SGN(N)
SINE IN RADIANS	SIN(N)	SIN(N)	SIN(N)	SIN(N)	SIN(N)	SIN(N)
PRODUCES STRING OF SPACES	SPACE$(N)	-	SPACE$(N)	-	-	-
PRINTS SPACES TO SCREEN	SPC(N)	-	SPC(N)	SPC(N)	SPC(N)	-
RETURNS SQUARE ROOT	SQR(N)	SQR(N)	SQR(N)	SQR(N)	SQR(N)	SQR(N)
INCREMENT IN FOR/NEXT LOOP	STEP	STEP	STEP	STEP	STEP	STEP
TERMINATE PROGRAM-BUT NOT FILES	STOP	STOP	STOP	STOP	STOP	STOP
STRING OF SET LEN AND CHARACTER	STRING$(L,$)	-	STRING$(L,$)	-	-	-
CONVERTS NO. TO STRING	STR$(N)	STR$(N)	STR$(N)	STR$(N)	STR$(N)	STR$(N)
EXCHANGE VALUE OF 2 VARIABLES	SWAP VAR,VAR	-	-	-	-	-
RETURNS TO OPERATING SYSTEM	SYSTEM	BYE	-	-	-	-
TABS ON TERMINAL	TAB(N)	POSITION	TAB(N)	TAB(N)	TAB(N)	?
TANGENT IN RADIANS	TAN(N)	-	TAN(N)	TAN(N)	TAN(N)	TAN(N)
TURNS LINE NUMBER TRACE OFF	TROFF	-	TRACE OFF	-	-	-
TURNS LINE NUMBER TRACE ON	TRON	-	TRACE ON	-	-	-
CALL ROUTINE SET IN DEF USR	USR ADR	USR ADR	USR ADR	USR PAR	USR PAR	USR ADR
NUMERIC EQUIVALENT OF STRING	VAL($)	VAL($)	VAL($)	VAL($)	VAL($)	VAL($)
WAIT FOR VALUE AT A PORT	WAIT PORT,N	-	-	WAIT ADR,EXP,EXP	WAIT ADR,EXP,EXP	-
LOOP UNTIL EXPRESSION NOT TRUE	WHILE/WEND	-	-	-	-	-
SETS WIDTH OF LINE	WIDTH(N)	SEE POKE	WIDTH(N)	-	-	-

Fig. 8. Main conversion table

ports and/or disk buffers) allow graphic images to be read into and from BASIC arrays. It's easy to move even the most complex shape, and with a little work you can add in some simple image processing including reflections, inversions and so on.

SOUND produces a single note and is supplemented by the **PLAY** command, which acts on a user-defined string containing a sequence of commands in a simple-to-learn music language. The commands support various octaves, tempos and voices and allow macros by combining strings together. You won't be able to write immediately portable programs – things like screen resolutions, the number of colours and the sorts of sound available still vary from machine to machine. But, if you're careful, GWBASIC can make moving a program between machines a much less daunting task.

MACHINE SPECIFIC SOUND COMMANDS

MACHINE/ COMMAND	FUNCTION
ATARI	
SOUND A,B,C,D	A=VOICE (0-3) B=PITCH (0-255 LARGER=LOWER PITCH) C=DISTORTION (0-14) D=VOLUME (1-15)
BBC	
SOUND A,B,C,D	A=CHANNEL (0-3) B=VOLUME (-15 - 0) 0 IS SILENT, RANGE FROM -15 TO 0 FIXED NOTE 0-4 ENVELOPE SHAPED C=PITCH (0-255) D=DURATION
ENVELOPE N,T,PI1,ETC	COMPLEX COMMAND WITH MANY PARAMETERS
N	ENVELOPE NUMBER (0-4)
T BITS 0-6	LENGTH OF EACH STEP IN .01 SEC (1-127)
BIT 7	0=AUTO REPEAT,1=NO REPEAT
PI1	PITCH CHANGE PER STEP SECTION 1 (-128-127)
PI2	PITCH CHANGE PER STEP SECTION 2 (-128-127)
PI3	PITCH CHANGE PER STEP SECTION 3 (-128-127)
PN1	NUMBER OF STEPS IN SECTION 1 (0-255)
PN2	NUMBER OF STEPS IN SECTION 2 (0-255)
PN3	NUMBER OF STEPS IN SECTION 3 (0-255)
AA	CHANGE IN AMPLITUDE PER STEP IN ATTACK PHASE (-127-127)
AD	CHANGE IN AMPLITUDE PER STEP IN DECAY PHASE (-127-127)
AS	CHANGE IN AMPLITUDE PER STEP IN SUSTAIN PHASE (-127-127)
AR	CHANGE IN AMPLITUDE PER STEP IN RELEASE PHASE (-127-127)
ALA	TARGET LEVEL AT END OF ATTACK PHASE (0-126)
ALD	TARGET LEVEL AT END OF DECAY PHASE (0-126)
CBM 64	
POKE 54296,N	VOLUME ALL THREE VOICES (N = 0-15)
POKE 54273,N	HIGH FREQUENCY (N = 34 FOR C TO 72)
POKE 54272,N	LOW FREQUENCY (N = 75 FOR C TO 169)
POKE 54276,N	WAVE 17=TRIANGLE, 33=SAWTOOTH, 65=PULSE AND 129=NOISE
POKE 54275,N	HI PULSE RATE (N = 0-15)
POKE 54274,N	LO PULSE RATE (N = 0-255)
POKE 54277,N	ATTACK/DECAY ATTACKS = 128,64,32 AND 16 DECAYS = 8,4,2 AND 1, ALL CAN BE SUMMED
POKE 54278,N	SUSTAIN/RELEASE SUS = 128,64,32 AND 16 REL = 8,4,2 AND 1, ALL CAN BE SUMMED
BOTH HIGH AND LOW ARE NEEDED FOR EACH NOTE, ALL POKES AFTER THE FIRST ARE FOR VOICE 1, ADD 7 FOR VOICE 2 AND 14 FOR VOICE 3	
CBM VIC 20	
POKE 36878,N	VOLUME (N = 0-15)
POKE 36874,N	PLAYS TONE (N = 128 - 255)
POKE 36875,N	PLAYS TONE (N = 128 - 255)
POKE 36876,N	PLAYS TONE (N = 128 - 255)
POKE 36877,N	PLAYS NOISE (N = 128 - 255)
SPECTRUM	
BEEP A,B	A = SECONDS, B = FREQUENCY

Fig. 9. Machine dependent sound commands

Crossing dialect boundaries

How many times have you seen a program listing that you have wanted to enter and use, only to find it is written in a BASIC not suitable for your machine? The problem is that the different dialects of BASIC are different languages in their own right, just like French and English.

For both computer languages and human languages, certain words can be directly converted: an English 'cat' becomes a French 'chat'; the trace-on command **TRON** in MBASIC becomes Applesoft's **TRACE**). Like human languages, computer languages often elude direct word-for-word translation. The work can't be done by rote, but require a knowledge and understanding of context and meaning. Without this understanding, the result will be a computer equivalent of the famous phrase 'out of sight, out of mind': on one occasion it was re-translated back into English from Russian as 'blind idiot'.

Some general concepts of the graphics, colour and sound features found on most home micros will be enough to enable quite complex conversions when no direct match is available. We've summarized this information in the form of tables. Many of the BASIC dialects have a considerable overlap: the majority of commands with the same meaning occur in each version. However, great care must be taken, as slight variations (like a '**,**' in place of a '**-**') can cause all sorts of syntax problems. All the direct coversions have been condensed in Figure 8. Figure 3 lists the functions provided in particular BASICs that are very machine-specific.

The widest disparities appear in the graphics and plotting commands. Many different terms and facilities are used, and to complicate matters further the same feature may be called by different names. Often the end result will have to be achieved by quite different means on different machines.

Line-drawing provides an example. In Applesoft BASIC the command takes the form **HPLOT X1,Y1 TO X2,Y2**. The BBC microcomputer on the other hand has first to use **MOVE** to position the cursor, and then **DRAW** to make the line. To help you avoid some of these pitfalls, we have produced a number of tables on graphics. Figure 5 defines the common features like draw a line, circle and point, and lists the basic differences between dialects.

Figure 6 lists the screen resolution – the number of points that can be plotted in both the X and Y axes – and the locations of the screen RAM start and finish points. The screen RAM typically stores a byte representation of what is displayed on the screen. If the character 'A' is displayed, the corresponding byte (65 decimal in ASCII) will be found in the screen RAM at a location directly related to its position on the screen. These locations can be **PEEK**ed and **POKE**ed like any other area of RAM. Many arcade games use assembler routines to achieve very fast action by working directly on the video RAM like this.

Figure 7 explains the functions and facilities unique to each machine. Many computers are able to define character sets to replace or supplement the standard character set. This technique can be used to produce shapes which can either be used individually or grouped together to make larger shapes.

In addition to sophisticated visuals, many of the popular systems support sound. Amplitude (loudness), pitch (frequency) and duration are often supplemented by another parameter, the envelope, to provide greater control of the sound against time. There are two envelopes, pitch and volume, both expressed against time.

Some machines offer only a bleep, others provide commands which will define a whole range of

enveloping. As there is little in the way of common features, we only list the unusual machine-dependent features in Figure 9.

Programs which make use of internal **POKE**s and **PEEK**s to achieve unusual effects are living dangerously, and good programmers avoid this approach wherever possible. Different hardware revisions from the same manufacturer may have different memory maps.

When you come across a program written for a machine other than your own, use the appropriate charts to find out what it is attempting to achieve, and which machine-dependent facilities are being used to achieve it. The majority of 'straight' BASIC programs, (not sound-, screen- or joystick-dependent) can be quickly converted by reference to Figure 8. If an unusual command appears in the program, Figure 3 should explain what it is. Be prepared to restructure a routine, replacing, say, a **DO UNTIL** with a loop controlled using an **IF THEN GOTO**.

Figure 4 will make sure you have the correct screen graphics colours. There are a number of common screen commands which produce the same end result even though the words used may differ, so we have followed the same two-table approach as with BASIC proper. If the general conversion chart, Figure 5, does not answer the question then look at the machine-dependent table in Figure 7. Figure 6 will help with scaling and plotting facilities as it lists machines modes and XY resolutions.

Again, completely different solutions to the same end may have to be programmed. On machines whose BASIC does not have a direct circle drawing facility you will have to resort to raw trigonometric functions (Sines, Cosines and the rest). Sound is even more variable in the features provided, from the Spectrum's sole sound command **BEEP**, where only time and frequency parameters are supplied, to the many-voiced BBC with its multi-parametered **ENVELOPE** command. You'll have realised by now that conversion can often only be an approximation.

However, we have listed the machine-dependent commands for reference in Figure 9.

Specific BASICs

This section looks in turn at a number of popular machines, concentrating primarily on the method of operation and differences between it and the common modes.

The **Apple** is a general purpose business and home computer with a good range of graphics facilities. The main variations in APPLESOFT BASIC are the use of **VTAB** and **HTAB** for vertical and horizontal screen cursor positioning and **PR(n)** and **IN(n)** (where *n* is a number) for I/O directing. The Apple has the normal line and colour plotting facilities as well as the **SHAPE**, **DRAW** and **LOAD** features.

Shapes can be drawn and stored. The screen has three modes, two with hi-res. Only very a very limited sound facility is available from BASIC, just a simple 'bleep'. The position of the paddles can be sensed directly using a BASIC variable and the fire button status is monitored using a **PEEK**.

The **Atari 400 and 800** are both quite powerful home machines. There are some unusual BASIC commands, of which the most interesting is the **STATUS** command, which can monitor the port status. It has the full range of plotting commands as well as a fill option, which like the **PAINT** of the IBM PC fills a shape entirely with a single colour. No less than nine modes are available, with resolutions of up to 320 x 192. A simple sound command with basic facilities is provided, together with four nine-position joystick commands, eight paddles and eight fire button commands.

The **BBC**, though primarily aimed at education, has many non- standard BASIC features. It provides an unusual **REPEAT/UNTIL** loop structure, and the ability to define complex procedures, as a sort of extension of BASIC's **DEF FN**. The normal range of plotting commands are available, together with a complex multi-parameter **PLOT** statement with many options. There is no polygon fill, only a triangle fill. Eight screen modes offer up to 640 x 256 pixels – rather more, in fact, that a domestic TV can handle. It is the machine with the most sophisticated (and complex) sound facilities, its multi-parameter envelope putting it at the very top of its class. Joystick connections are handled by way of the **ADVAL** command.

The **Commodore 64** is at the top of the Commodore range of home machines. The only unusual feature of the BASIC stems from Commodore use of the IEEE principles of primary and secondary devices. So using a special version of the **OPEN** command and **CMD**, I/O can be directed throughout the system to screens, printers and so on. Virtually all screen work on the machine is achieved by using **POKE**s or control characters.

Quite sophisticated graphics can be constructed using the concept of sprites (*see* Graphics), which replace the more usual BASIC plotting commands. The sprite is a graphic object which is user programmed and can be plotted on the screen, moved, have its colour changed and which can be scaled; all under the control of **POKE**s. As this facility is available on few other machines, direct conversions are virtually impossible. Three voices are provided for sound, with quite complex synthesiser-like attack and decay facilities accessed, again, using **POKE**s.

The **Commodore Vic** is the introductory machine of the Commodore range, and its BASIC has the same attributes as the 64 (*see* above). As with the 64, there are no direct BASIC plotting facilities, and in this case no sprites to help fill the gap, which forces the user to resort to **POKE**s to drive the graphics. The system provides a simple three tone, one noise channel sound facility, also accessed with **poke**s.

The **Dragon 32** is a full keyboard mid-priced home machine. The BASIC is compatible with MBASIC version 5, and has some added features. For example it can control the cassette motor and direct the sound through the TV speaker. This machine probably has the best range of graphic commands directly executable from BASIC including a polygon paint and a useful **CIRCLE** command capable of drawing arcs and ellipses. There is also a shape drawing facility much like that of the Apple, and the ability

to dump and reload screens to an array. It has five modes; up to 256 x 192 resolution. An interesting sound facility similar to GWBASIC is available, where the actual note names (like 'C') are entered for playing, along with parameters to control tempo and volume. A four position joystick handling command is also provided.

The **Oric** is a reasonably priced home machine with a similar keyboard to the Spectrum. This also has the `REPEAT/UNTIL` loop structure, and a *decimal* `PEEK` and `POKE`. The machine has a good range of commands including `CIRCLE`, with a resolution up to 240 x 200. An unusual feature are the pre-programmed sounds – `EXPLODE`, `PING`, `SHOOT` and `ZAP`. In addition `MUSIC`, `SOUND` and `PLAY` allow a number of tone and noise channels to be controlled, complete with enveloping.

The **Sinclair Spectrum**, big brother of the Sinclair ZX-81, is the machine which brought more computers into the home than any other. The chief differences between Sinclair BASIC and the mainstream dialects are the direct input via `BIN` of binary, its insistence on the (usually optional) `LET` in assignment statements, and syntax checking of input lines as they are entered.

This machine has a similar range to the Oric, and also includes the `BORDER` facility which defines the colour of the screen surround. In addition it can control the brightness and flash of characters, on a screen with a resolution of 256 x 176. There is only a very simple (and quiet) single voice sound generator, with time and pitch options.

The BASIC language on the 64 is directly descended from the PET. Although the PET language was a leader in the mid-70s, it has not changed since, and the 64 BASIC lacks many features which other micros take for granted, including minimal structured features and support for new hardware features such as the disk drives, graphics and sound chips.

Real or floating point variables are held to nine significant digits; integer variables can only hold whole numbers in the range −32768 to +32767. Unlike some BASICS, there is no advantage in using integer variables: they occupy the same amount of space in memory and are actually slower in operation than real variables, because the interpreter takes the integer value and converts it to floating points before doing any calculations with it. The only exception to this is that integer arrays are much smaller than floating point arrays with the same number of elements.

Strings can vary in length from 0 to 255 characters. Each time a new value is given to a string variable the old value is left in memory as 'garbage'. Eventually the memory becomes completely full and a routine in the interpreter known as the 'garbage collector' is called to clear it all out. It therefore pays to keep the size of string arrays down to a minimum because in a program with a large number of string arrays, garbage collection can take a long time (up to 45 minutes), during which period the machine is completely 'dead'.

The 64 BASIC supports a limited user-defined function facility. Only one argument is allowed, and both it and the result must be numeric. In addition the function definition can occupy only one line. There is also no special function (such as INSTR) to search for a string within another string. You have to do it yourself, the hard way.

The INPUT statement is very limited, not accepting commas or quotation marks amongst other things. The maximum length of an input string is only 80 characters, which means that you have to keep disk or cassette records shorter than this. On the output side there is no way to format numerical output to make it easy to print tables of data. You can get around this by using a routine like this:

```
1000 Z$ = RIGHT$ ("          " + STR$(INT
((100*V+0.5)/100)),6)
```

which rounds the value V to 2 decimal places and converts it into a string which is always six characters long.

The 64's video and sound generator chips are very powerful and can produce a wide range of audio and visual effects, including a three-voice sound generator and a graphics display of 320 x 160 dots in 16 colours. No special commands have been built into the BASIC to take advantage of these, so all sound and graphics programming is done by means of POKEs. The BASIC has also not been expanded to include the Disk BASIC commands on 4000 and 8000 series PETs, so disk handling, although no more difficult than on those machines, is a little less natural.

Nearly all the BASIC keywords can be abbreviated using shifted characters when you type them in:

```
Go   for  GOTO
GOs  for  GOSUB
REt  for  RETURN
```

This does not save space in the machine; all keywords are converted to single-byte tokens internally, and the full word appears when you list the program.

Languages: BCPL

BCPL – the initials stand for **Basic Combined Programming Language** – is a block-structured, ALGOL-like language chiefly noteworthy as being the forerunner of the language C (*qv*).

Where other programming languages have reals, integers, strings and arrays, BCPL has no data types at all. This flexibility distinguishes it from other programming languages, and makes it suitable for low-level system programming or non-numerical applications, such as writing compilers, editors, word-processors or games, where it is often an advantage to be able to convert data of one type to another without the usual constraints.

BCPL's rich syntax has several advantages:

It allows a number of ways of branching, looping and of defining subroutines.

As in C, the programmer can manipulate pointers without the compiler or run-time system losing track.

Arrays are simple but can be very powerful.

All subroutines can be called recursively.

The lack of data types makes BCPL compilers small and fast.

```
GLOBAL $(   global name;
              :
           $)
MANIFEST $(   one = 1;
                :
           $)

LET funct. 1 (arg 1, arg 2,  . . .arg n)
           = VALOF

    $(       LET int 1, int 2 = 1,2
             LET mult (a,b)= a*b
             :
    $)
    :

LET subroutine. 1 (arg 1, arg 2,
                   . . .arg n) BE
    $(
    $)
    :
LET START () BE
    $(
    :
    $)
```

Fig. 1. Structure of a BCPL program

```
LET array = VEC 9
LET row 1, row 2, row 3 = 0,3,6
  :
array!(row 1+2):=7
array!(row 3+1):=10

array–
```

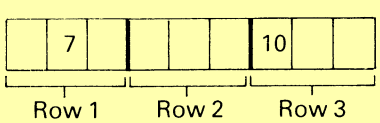


Fig. 2. Three by three matrix

Statement	BCPL Statement	BASIC equivalent
Assignment	A,B:=3,4	100 A=3 110 B=4
Compound Assignment	C–:=1	100 C=C–1
Conditionals	IF1<=N<=10 THEN	100 IF(N<1) OR (N>10) GOTO 130
	$(N:=N+4	110 N=N+4
	C*:=N	120 C=C*N
	$)	130 A=C+N
	A:=C+N	
	TEST A+B THEN	100 IF A+B<>0 GOTO 130
	A:=3	110 A=5
	OR	120 GOTO 140
	A:=5	130 A=3
	B:=3	140 B=3
Repetitive	WHILE N<> O DO	100 IF N=0 GOTO 140
	$(N–:=1	110 N=N–1
	A*:=A/2	120 A=A*A/2
	$)	130 GOTO 100
		140
	$(N:=A**2+3	100 N=A**2+3
	A–:=2	110 A=A–2
	$)REPEAT WHILE	120 IF A<>0 GOTO 100
	A<>0	
	FOR1=1 TO 14 BY 3	100 FOR1=1 TO 14 STEP 3
	$(AII:=0	110 A(I)=0
	BII:=0	120 B(I)=0
	$	130 NEXT 1
Case	SWITCHON N INTO	100 IF N<> 1 GOTO 120
	$(CASE 1:A:=4+B	110 A=4+B
	ENDCASE	115 GOTO 160
	CASE 2:A:=C*4	120 IF N<> 2 GOTO 150
	ENDCASE	130 A=C*4
	DEFAULT:A:=C+50	140 GOTO 160
	$)	150 A=C+50
		160

Fig. 3. BCPL commands with their BASIC equivalents

BCPL's basic data object is a **word**. A word can be treated as a number, a bit pattern, an address or a subroutine entry point.

The BCPL systems make no attempt to check whether a haywire program is overwriting itself or anything else that happens to get in its way. This characteristic of the language gives it both the flexibility and some of the problems of machine code. For example, you could have the following assignment statement which would be accepted by the compiler.

```
ARRAY:=INTEGER + 'C' * TRUE-SUBROUTINE+#XF4A2
```

ARRAY is an array name (subscripted); **INTEGER** is an integer variable; **'C'** is the character C (in ASCII); **TRUE** is the logical literal for TRUE; **SUBROUTINE** is a subroutine name; **#XF4A2** is the hexadecimal constant F4A2.

This freedom opens up powerful opportunities for an experienced programmer, but is definitely not for beginners.

The programs

A BCPL program can be described as a sequence of declarations. Most of these will be function or subroutine declarations, often preceded by the declaration of global names and constants, as in figure 1.

Execution of a program begins with the first statement of the START routine, and finishes with the last statement in that routine. The functions and subroutines defined in a program can be executed from the START routine or from each other, and library routines can also be called from anywhere within the program as long as they are defined global names.

The action of a function like `funct 1`, or a subroutine like `subroutine 1` is defined by the sequence of commands contained within the section brackets **$(... $)** following its declaration. Each such body of commands can itself contain declarations. In `funct 1` the variables `int 1` and `int 2` are defined with initial values of 1 and 2, and a second function, **MULT**, is defined, with *two* arguments whose result is the product of **a** and **b**. If the programmer tried to call the **MULT** function from `subroutine 1` it would fail – the **MULT** function can only be called from within `funct 1`, the function which defines it. To use **MULT** within `subroutine 1` it would need to be declared again there. This characteristic of a language is known as locality, and in this example **MULT**, `int 1` and `int 2` are local to `funct 1`.

As with other data items in BCPL, an array is held as a word. This word holds the address of the first word of an area of contiguous storage which forms the elements of the array, each element being one word. The array elements are retrieved by accessing offsets from the base address.

The one disadvantage of BCPL arrays is that they can only be one-dimensional, so *matrices* have to be explicitly created by the programmer. An example of a 3 x 3 matrix and updating it is given in figure 2. In this example, the first assignment puts a value of 7 into the second element of row one, and the second assignment puts a value of 10 into the first element of row three. In arrays like this the lack of data-types can be very useful. For example, our matrix could have each row as a record, with the first element being an integer, the second a label and the third a character string.

The commands in BCPL can be grouped under various headings; the major ones are given in figure 3 with their BASIC equivalents. As you can see, most of these can be translated without too much problem. Notice the use of logical expressions in the conditionals to determine whether a variable is within a given range. A **conditional** tests whether the condition is true or false, and in BCPL the result of a conditional is judged to be false if – and only if – it is zero, hence the example **TEST** statement in figure 3.

One other helpful command is the **SWITCHON** statement. In the example, if **N** has a value of 1, execution starts at the **CASE 1** label, ending at the first **ENDCASE** statement. Similarly, if **N** has a value of 2, then execution starts at the **CASE 2** label. If it has neither of these values then the **DEFAULT** option is performed.

BCPL's recursive design means that any of the subroutines or functions you define can call themselves from within themselves. The language is compiled, and the code it produces is quite compact and efficient. And the BCPL compiler itself is small enough for it to be used on most microcomputers.

Languages: C

As computer languages go, C is relatively new, having been put together by Dennis Ritchie at Bell Laboratories in 1972. Stories about the origin of its name differ: the most convincing traces C back to the language BCPL (*qv*). Ritchie based his own creation on a language called B, a development of BCPL. One step better than B is – naturally – C!

C, UNIX and terse programming

The first practical use of the language was to liberate Bell Laboratories' own UNIX operating system (*qv*) from the processor-dependent assembler tying it to the ageing PDP-11 minicomputer environment. Ritchie rewrote the whole thing in C and began to move it across to other processors with a speed that made the rest of the industry sit up and take notice. Since then UNIX and C have been close companions. UNIX without C is unthinkable, although C on its own is doing a lot of very useful work in alien environments like CP/M, MS-DOS and PC-DOS.

C shares many of the characteristics of UNIX. It's a kind of mezzanine language positioned somewhere between the ground floor of assembler and the top floor of high-level languages like BASIC. Like its name, C is terse – a language designed for busy programmers who do not like typing 'integer' when 'int' will do. Compare the way the different languages increment a variable. BASIC makes the assertion, baffling to beginners:

```
LET i = i + 1
```

A similar construction is available in PASCAL, though the dyed-in-the-wool PASCAL programmer will probably prefer the alternative:

```
i := succ(i);
```

to be read as 'let i take on the value of the successor to the current i'.

All this keystroking is far too tiring for the C-farer, who increments his variable with the momentous statement:

```
++i;
```

A similar compression is available to replace the prolix:

```
if i > x then y else z
```

which appears with minor variations in all three languages. C offers the alternative syntax:

```
i > x ? y : z;
```

C is economical in its overall construction too: no more than 30 reserved words define its commands and functions, in combination with a rich but relatively consistent 'grammar'. In this important respect it differs from BASIC. Although conceived as a disciplined kernel of rudimentary commands, in commercial life BASIC has proliferated until it sometimes seems there is a separate command for every conceivable thing you might want to do on a computer.

Crossing the hardware frontiers

The messy parts of any language are the input/output routines which control writing files to disk, reading from disk and communicating with the world outside the computer through the ports. In C these parts of the language are gathered into a library of standard functions, themselves written in C. This helps to make its central core independent of the hardware. Features like this are supposed to make life easier for the pioneers who write C compilers for new processors, although implementing some of the language's subtler features can produce bugs that are difficult to track down and eradicate. Hardware architecture which flatly contradicts the underlying assumptions made by C can provide the compiler writer with considerable headaches. But whatever the balance of theoretical simplicity and practical difficulty, love it or hate it (and passions run high among programmers), C is now among the first languages to be available on each new processor.

Naturally a language that that pops up everywhere provides valuable insurance for people who invest their lives and money in the production of new software, and C appeals to programmers struggling to keep up with the fast-moving developments of hardware technology. When a typical applications package might take a team of three more than a year to develop, C offers hope that their new software will still match the hardware by the time it comes to the marketplace.

The language has some well-documented shortcomings: floating point numbers always evaluate intermediate values in double precision, which often slows calculations unnecessarily. The order in which expressions are evaluated is not guaranteed, and varies from machine to machine, causing a program which runs perfectly well on one computer to crash mysteriously on another. But this difficulty is by no means confined to C – it would be fairer to regard it as a problem highlighted by the general portability of the language. You wouldn't even begin to transport a BASIC program across from, say, a PDP-11 to the IBM PC without an enormous revision of the source code.

C is a language for people who know what they are doing, and it gives them the flexibility to do it. It is far less severe than PASCAL. The compliant C compiler turns a blind eye to outrageous breaches of logic and commonsense – it's a yes-man assistant providing no help at all and it will happily run into a brick wall if that's what you tell it to do. BASIC attracts a number of software authors who are not entirely sure what they are doing, but are happy to procede through long hours of coding on a 'suck it and see' basis. Programmers like these will have a hard time writing in C.

The portable assembler

High level languages compromise the efficiency of the code they generate in order to make program writing easier. Assembler produces highly optimised code but is very time-consuming to write. The difference is comparable to that between using a mechanical loom or handweaving a carpet.

This is where C comes in. It embodies many of the structured features that make languages like PASCAL

useful for tackling real life problems, but by including a number of low-level constructs like arithmetic on pointers, it allows assembler-style programming without becoming machine-dependent. Uniquely in a high-level language it is able to reach deep down into the internal functioning of the processors themselves in a quest for machine efficiency. That's why C has come to be known as the 'portable assembly language'.

In C the sacrifice of efficiency for ease of programming and a guarantee of portability across a wide range of processors is not as great as with true high level languages like BASIC. But at best you can assume that a C-compiled application package would perform 30% better if it were re-coded in assembler.

COBOL is a high-level language developed for business users on mainframe computers. It stands for COmmon Business-Oriented Language.

A distinguishing feature of COBOL is that its syntax, designed to resemble 'natural' English, is very verbose – in fact quite tedious to write if you are used to other, more scientific, languages. In BASIC, for example, you assign **A** to **B** simply by writing **B=A**; the comparable COBOL instruction is **MOVE A to B**. A command like **MOVE** or **DISPLAY** is called a verb in COBOL. COBOL's powerful file-handling is much appreciated by writers of business programs.

Batch processing

One difference between mainframe and microcomputers is that a lot of mainframe processing is done in batch mode. Batch processing was developed in the days when computing resources were scarce, it is a way of allowing the human side of putting programs and data together to take place at a separate time from the actual computation. A job consisting of data – and maybe its program – is prepared at one site, then sent (originally on punched cards or paper tape, but mostly on magnetic tape these days) to a computer 'centre' to be run later – possibly hours or even days later.

For this reason there was little point in including interactive screen-handling statements in COBOL. But on microcomputers, batch processing is far less relevant, and a great deal of interaction is done using the screen. For this reason, micro-inspired COBOLs like CIS-COBOL (pronounced *Kiss* – it stands for *Compact Interactive Standard*) and MICROCOBOL, both based on the traditional ANSI (American National Standards Institute) version, have been extended to include two new screen-handling verbs, **ACCEPT** and **DISPLAY**. These are very similar to the **INPUT** and **PRINT** of BASIC.

COBOL on micros

ACCEPT will, as its name implies, read values typed at the keyboard. **DISPLAY** will display data of any kind on the screen. In CIS-COBOL, these commands are extended to **ACCEPT X FROM VDU** and **DISPLAY X UPON VDU**. A further extension, **AT**, will take screen co-ordinates as parameters to accept or display values anywhere on the screen using a sentence of the form: **ACCEPT value AT X, Y FROM VDU**.

The ANSI 74 standard on which CIS-COBOL is based was specified at two levels: the nucleus (level 1) and the full implementation (level 2). Level 2 COBOL is specified for full mainframe implementations, and includes many features not available in level 1. For example, level 2 specification includes a sort and merge facility, alternative indeces for indexed files, **COMPUTE** statements and conditional operands – The **COMPUTE** verb can save a lot of lengthy statements, it can precede any computation in the form **COMPUTE A=3*(B*C)+D+E**, and avoids tedious over-use of the verbs **MULTIPLY** and **ADD**.

Languages: LISP

LISP was developed at Massachusetts Institute of Technology (MIT), and shares the honour with ALGOL and FORTRAN of being one of the first high-level computer languages, dating back to the early 1960s. As an interpreted and easy-to-learn language using symbol processing as its central concept, the language makes a plausible alternative to BASIC.

Languages like PASCAL are made up of 'bricks' – the commands and functions built into the compiler and a set of unbreakable rules defines the way the bricks are allowed to interlock. LISP, on the other hand, has all the features of clay – pure natural substance enabling any form to be modelled. It exists as a sort of harmonious anarchy, adapting itself to accommodate the user.

LISP is most widely known as one of the key languages used in work on artificial intelligence (*qv*), an application for which it is especially suited. In addition, symbolic mathematics, such as algebraic manipulation, may also be readily represented using LISP. However the language is less suitable for numerical mathematics: it can do the sums all right – but only comparatively slowly. The current trend in micros is for arithmetic to be carried out by special hardware (the Intel 8087 processor chip is an example), and this may well bring LISP maths into line with other more numerate languages.

There is a fundamental logical problem at the heart of all list processing languages: the language has to treat the outside world as a continuous list which makes it hard to incorporate the concept of I/O handling – something quite easily implemented in languages such as BASIC.

Where timing is critical, as in games (*qv*) and real-time programming, LISP is not a suitable language. As well as being slow, it suffers from unpredictability. Because LISP is so extravagant with memory it easily runs out of RAM space but, instead of crashing (as in some versions of BASIC), LISP searches through memory for bytes that it doesn't need anymore in a process known as garbage collection. This involves looking at everything twice, and can take a couple of seconds, possibly causing hiccups in the running of the program at random intervals.

Word processing and text editing are not normally associated with LISP, but some very successful packages, such as EMACS (also from MIT), have been produced using LISP. Systems programming can been handled entirely in LISP and there are even dedicated LISP machines, using LISP architecture hardware, and LISP-written operating systems.

Other languages can be implemented in LISP. SMALLTALK, LOGO and PROLOG were all initially written in LISP, and in fact LISP itself can be defined in LISP – a useful way of adding specialised extensions. Redefining the LISP interpreter is done in much the same way as defining a new LISP function.

Being based on abstract mathematical theories, rather than the more pragmatic approach of BASIC makes LISP at the same time elusive and far reaching. Luckily a full understanding of all the theoretical underpinnings isn't necessary in order to use the language and once you have escaped from your prejudices, LISP becomes a perfectly natural way of expressing program ideas in an elegant, simple and concise way.

Essentially there are three aspects to LISP: functions, lists and abstraction.

Functions

The concept of a function should be familiar to most people who have used computers. Consider a function that removes unwanted dirt from clothes: dirty clothes and soap go into a washing machine, and damp clothes, free of dirt, emerge. As far as you are concerned, the washing machine and its aqueous thrashing around is unimportant – what matters are the inputs and outputs. A function is something which transforms an input into an output. This simplification enables us to live normal lives without encumbering ourselves with details of all the mechanisms that sustain us.

LISP code consists entirely of functions, each being made up of a set of calls to further functions. This imposes a sort of uniformity in the use of the language, and time is not spent learning to cope with the differences in behaviour of all these other function-like objects. Functions can specify any process or relationship. The simplest functions are built into the language at machine code level. `ORDINAL 'T'`, a function with one input and one output, calls a built-in function producing the ASCII value of the letter `T`. Functions can take more than one input, finding the average value of a set of numeric inputs, for example could have an open-ended number of inputs.

Lists

A LISP list is a collection of objects which are classified as being either atoms or lists. Atoms are the indivisible units on which LISP is based. A list of lists is a way of composing program ideas into a hierarchical structure. A book could be represented as a list of three items: part one, part two, and part three. Each part would itself be a list of chapters, each chapter a list of paragraphs, and so on all the way down to the atoms. In this example the atoms might be words (to a writer) or individual characters (to a compositor).

Atoms are encountered in other programming languages. In LISP they can be numbers, characters, strings or indentifiers such as `T`, `HELLO` or simply `X`.

Technically speaking, lists are dynamic, heap-based data structures. In practical terms this means that their structure and contents may be freely changed at any time. To revert to our example of the book, a chapter could be removed by a tiny change to the list, and it could then be inserted somewhere else – even at another level.

The unrestricted opportunity to restructure runs the risk of total anarchy, allowing you perhaps to put a whole chapter into the middle of a sentence but at the same time it can be very useful to be able to move large amounts of data around (say in a data-base) so easily.

It might appear that an array, as used in BASIC, can do much the same job, but the two are fundamentally different. An array holds a number of items that must be all of the same type. No mere array is permitted to hold an

integer in one position, a string in another, and another array somewhere else. And to insert a new item in the middle of an array means having to shift everything over to make a free space.

At this stage we have a good picture of LISP as consisting of just two things, functions and lists, with a few atoms at the heart of it all and it shouldn't be hard to concede that an atom is really only a special case of a list. What's more, the code that defines a function is also nothing more than a list. So data and functions are the same. Surely there is some distinction? Only when necessary.

A function which modifies a list can be used to modify a list that is a function. This naturally leads to the idea of recursion. A function can call a function, which may be itself, from within itself. Theoretically too, a list can contain a list, which may be itself, within itself. This happens a lot in LISP, giving the clear-headed programmer plenty of opportunity for concise expression.

Abstraction

Abstraction is the third aspect of LISP: the descriptions of functions and lists have already given many examples of this. The basic idea is to gather related fiddly details together to enable them to be thought of as a simple unit. This is a good programming technique, because it reduces the amount of information you nave to hold in your head to understand any part of a program.

Using a free form list structure is rather a good way of implementing abstraction since it imposes no limitations and allows the natural structure of the subject matter to dominate. This is the idea behind top-down programming. You define the outline of the program idea at a very high level what you want, and then descend level by level to fill in the details. This sort of programming is inherent in the idea of LISP itself, in sharp contrast to BASIC, which has a plethora of 'features' all bolted together.

```
(DEFUN
  MEMBER
    (PATTERN BASE COMB)
    (COND
      ((NULL BASE) NIL)
      ((ATOM BASE)(EQ PATTERN BASE))
      (T
        (COMB
          (MEMBER PATTERN (CAR BASE) COMB)
          (MEMBER PATTERN (CDR BASE) COMB)))))
```

Fig. 1. Lisp listing to define a new function

The layout of the LISP program that follows has been processed by an output function to make it more readable. If you type it in at a keyboard, only a single space is required between each identifier, and carriage-returns can be next to any gap.

Typing in the listing in figure 1, defines a new function called **MEMBER**, the list is made up of four elements:

An atom **DEFUN**.
An atom **MEMBER**.
A list starting **(PATTERN.....**
A list starting **(COND...**

The lists may themselves be broken down. The third element is made up of:

An atom PATTERN.
An atom BASE.
An atom COMB.

LISP does not label the elements in a list in this way. It only sees two things – the head of the list and the tail of the list. The head is the first item in the list, **DEFUN** in the outermost level of our example. The tail is the rest of the list that follows the head.

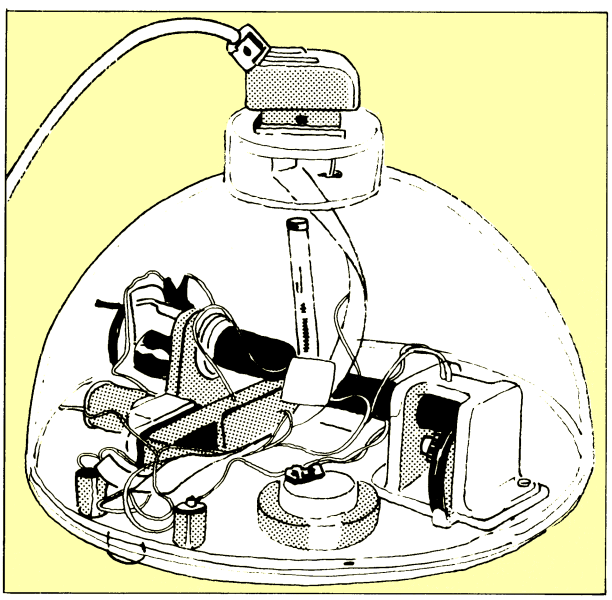
This division into the elements of head and tail is important in the functional interpretation of the list. The LISP interpreter takes the head of a list to be a function, and uses the tail to supply it with parameters. **DEFUN** is a function used to define new functions. It takes three arguments: the name of the new function **MEMBER**, a list of the parameters used by the function and a list describing what is done with these parameters.

Languages: LOGO

LOGO was developed with the young learner specifically in mind by a man who has been dubbed the 'guru' of childrens' computing, Seymour Papert. A quietly-spoken, Walter Matthau lookalike, Papert worked during the 1960s with Jean Piaget, the Swiss psychologist and educator, and in the late 1960s returned to the US to work in the artificial intelligence laboratory of the Massachusetts Institute of Technology, a hallowed ground in the field of educational research.

Papert has held the chair of Professor of Mathematics at the Massachusetts Institute of Technology since 1968. Fascinated by the subject of how children learn and, more specifically, how they learn when they are assisted and stimulated by computer intelligence, Papert started to experiment with the language which was to become LOGO in the early 1970s. His book, 'Mindstorms - Children, Computers and Powerful ideas', has been an international bestseller.

LOGO is an adaption of LISP (*qv*), But instead of the confusing rigours of nesting brackets, Papert devised the important front end of LOGO to allow children easy access to computer power. He also included the concept of the turtle, and now there are a dozen different types of screen turtles and several robot versions available for a variety of machines.



The Edinburgh Turtle robot

Turtle graphics

The turtle allows children to draw, either on the floor or on the computer screen. In commanding this small beast, children learn to express their intuitive grasp of the world in mathematical terms without any conscious involvement in the usually dry and irrelevant subject of 'pure maths'.

The Robot Turtle – For the youngest children (three and upwards), the turtle takes the form of a transparent plastic dome, which houses an electric motor and a pen, and which runs about the floor on wheels. This plastic, clearly mechanical turtle is attached to a computer by a long cable, through which it receives its commands. Children are introduced to this beast and told that the turtle will do their bidding if they talk to it in 'turtle talk'.

Children treat the turtle like a toy, and want to 'make it go'. The turtle is placed on a large sheet of paper in the middle of the floor, and the teacher or parent discusses which way the turtle is to move. 'Forward' is a likely suggestion and the teacher, with the child's help, types `FORWARD` into the computer. One other decision remains to be made by the child: 'How far forwards?' asks the teacher, 'How many steps?'

In procedures like this children relate the movement of the turtle to their own physical movement, and learn to express commands which cause physical effect, the pen inside the turtle providing the physical proof on the paper. This is the beginning of an understanding of geometry

Once the concept that the turtle will move under the control of the computer has been grasped, the teacher can introduce the next and most important concept: that the computer/turtle can be taught to remember how to do things. The turtle can draw a square on a piece of paper, and the computer told to remember the sequence. After that, the child or teacher has only to tell the turtle to `SQUARE`, or `SQ`, for the turtle to execute repeat the process.

The next step is to help the child discover that the procedure may be adapted, the size of the square altered, for instance, just by altering a variable. So the concepts of programming are learned.

An initial LOGO listing instructing the turtle to draw a square reads like this:

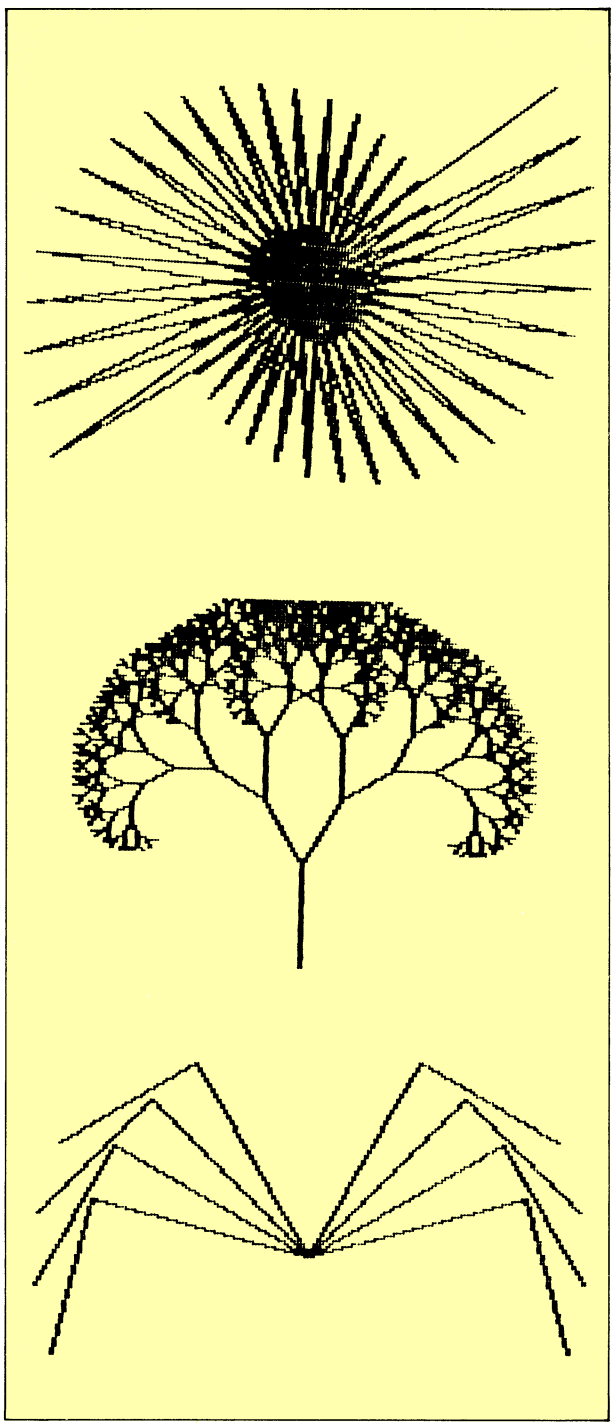
```
TO SQUARE
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
FORWARD 50
END
```

From there it is a short step to abbreviate both instructions and syntax, until an efficient program list for the same procedure would read as follows:

```
TO SQ REPEAT 4 (FD 50 RT 90) END
```

The Screen Turtle – Once a child reaches six or seven, the whole situation can take a small leap into the abstract. The concept can be introduced on the computer screen as a small cursor-like triangle of light, also called a turtle. If one point of the triangle is taken to be the front, the turtle can be turned on its own axis (rotated) or moved about the screen (translated) with a series of commands identical to those used by its floor-bound cousin.

Sprites – Another concept that the LOGO language brings to graphics on the screen is the 'sprite', an idea which makes it possible to animate images easily, and even to move them in three dimensions. The sprite is a pre-defined graphic image which can be moved around the screen in its entirety. Each sprite inhabits its own plane,



Screen dumps of some LOGO Turtle graphics

and those designated as logically 'nearer' the viewer have a higher priority than those further away. So if a sprite moves into an area occupied by a sprite of lower priority, it will mask it from view; if it moves to a space occupied by a sprite of higher priority, then the moving sprite will in turn be masked. This facility also opens up lots of possibilities which would be difficult without LOGO.

Sprites can be – and have been – implemented on computers outside the LOGO language (*see* Graphics).

Beyond the turtle

The limited memories of microcomputers have, until now, given casual onlookers the impression that LOGO is no more than turtles drawing pretty pictures. But the fuller implementations of the language now becoming available have powerful list processing abilities which make them formidable competitors to BASIC as all-round beginner's languages, and offer considerable advantages over well established languages such as COBOL.

The many micro versions of LOGO were initially developed over a ten year period, and the structure of the language has proved good enough to satisfy expert programmers as well as teachers who apply it in kindergartens. Implementations of the language have been produced by a number of computer manufacturers including Texas Instruments, Apple, Atari, Sinclair, RML, Radio Shack, and Acorn. Some versions are distinctly better than others, but all offer a delightful introduction to the concepts of programming.

A number of American schools have started to use LOGO, and the artificial intelligence laboratory at Edinburgh University has spent ten years investigating the language. So powerful is the potential of LOGO that Digital Research – inventors of the ubiquitous CP/M (*qv*) – have now declared it the language of the next decade, and are using it for writing many business orientated programs.

The philosophy of LOGO

The main thrust of Papert's argument is that children must be put in command of the new technology – although a glance into many elementary school classrooms reveals that the reverse is happening: computers are used in 'drilling' the pupils - asking them repetitive questions, and then offering a score. Teachers are just trying to replace themselves, without investigating the far more powerful and profound role that computers will surely play in education.

There is an old Chinese proverb: 'I am told, I forget. I am shown, I remember. I do and I understand'. This is the spirit in which Papert feels that children should use the computer to create for themselves the programs that will answer the questions they want to find out about.

Papert and his team have developed LOGO as an 'international' language, a kind of technical Esperanto which can be learned as though the student were in 'Mathland' – an imaginary computer country which provides the programming equivalent of a student of French visiting France.

When computer technology is applied to help children teach themselves by 'heuristic' processes of discovery, says Papert, geometry, the mathematical expression of our physical world, can be learned many years earlier than would otherwise be possible.

Languages: PASCAL And MODULA-2

Programmers have said that coming to PASCAL after BASIC is rather like throwing open a window the morning after a heavy party and letting in the sunshine. But a cold draught can come in through that window just as easily. Prof. Dr Niklaus Wirth of the Institut fur Informatik, Zurich, did not write PASCAL to curry favour, and the language is a strict mistress.

He invented it specifically for teaching the principles of programming, and it certainly provides good discipline for any budding programmer. When PASCAL blocks you by not permitting certain 'obvious' programming steps there's often a very concrete reason why you should be going about the problem another way.

What PASCAL does particularly well is help you write source code that is transparent. Transparency in this sense means that there is no thick veil of obscurity between you and the computer: you can understand the clear structure and near-English text you have written, and so can the compiler.

Like Algol, PASCAL is a block-structured language built around procedures. Block-structure means that the program can be thought of as a series of Chinese boxes, the smallest contained inside the next one up and so on up to the largest which contains them all. Identifiers like variables can be assigned as 'local' to a particular block, which means they can only be accessed by that block and the blocks within it, and are 'invisible' to the program outside the block.

This structure corresponds well with, and encourages, 'top-down' programming – designing the outlines of a program and working down into the depth of detail step by step. The main program can then be very simple, something of the form:

```
begin
      Do_This;
      Do_That;
      Then_Do_The_Other
end;
```

Each of those three routines will then be defined elsewhere in the program, and not necessarily in elementary detail – they may themselves call other routines and so on.

Because of the starkness of strict PASCAL, most commercial versions incorporate a mixed bag of enhancements for string and file handling. The de facto standard for this super-set derives from UCSD PASCAL. One important contribution of UCSD to PASCAL has been the introduction of a new structured type called a 'dynamic string', a bit like a record:

```
type
string[max_len] =
      record
        length_byte : 0..255;
        string_part :  array[1..max_len] of char;
         end;
```

There is no dynamic string type in strict PASCAL. The term derives from the string types used in BASIC, called 'dynamic' because their lengths are not defined. Wirthians justify this short-coming by pointing out that an implicit string-type already exists in the 'packed array of char' – an array of single characters. The word 'packed' actually has little meaning on micros, but is handed down from the mainframe world where it was contrasted with a more quickly accessible 'unpacked' array which took up more space. If a packed array won't do you can create your own record type along the lines above. But once you've used the dynamic string extensions and the other helpful handles that UCSD provides, you'll find that life without them is very much the poorer.

MODULA-2, son of PASCAL

Perhaps if UCSD were the only contributor of extensions to PASCAL, the language would have grown naturally to a new standard. But the strengths of PASCAL have lead to its very wide use outside the academic environment in which and for which it was designed; its shortcomings have invited many non-standard extensions from all quarters. This process threatened to destroy the rationale of the language in much the same way as happened to BASIC when it left Dartmouth College.

It wasn't long before the uncontrolled spread of these various dialects alerted users in 1978 to the need to define a standard set of extensions. Two such attempts failed, mostly because the interested parties from universities and software houses had invested time and energy into implementing their own favorite extensions, and were reluctant to let go of them to make way for others 'not invented here'.

Wirth's own response will be recognised as typical of his thinking by programmers who have come to know it well through the use of PASCAL: 'If a language proves to be only marginally suitable for some application that was obviously not envisaged by its originator, we should muster the courage to build a new, truly adequate tool, instead of just grafting a fix onto an existing one.'.

The new tool was **MODULA-2**. Based on a sub-set of PASCAL (stripped of all its extensions), Wirth has improved the consistency of the syntax, added some support for multi-processing, and given it a few tools for probing into the lower levels of the machine's workings. But the development which gave the language its name was the provision for writing sections of a large program in modules which could be tested separately before being joined together to form the finished product. Modules behave like software 'black boxes' whose only inputs and outputs are those defined by identifiers explicity included in 'Import' and 'Export' lists. This avoids most of the awkwardness of large programs in PASCAL, where the 'Chinese Boxes' of the block structure can bury the program's logical shape under reams of code.

A necessary ingredient of any language which supports modular programming is a standard set of slots, or interfaces, for joining the modules together. They must be sufficiently easy to handle for one module to be written independently of another – perhaps even by a completely different person. Wirth was also experimenting with a new way of uniting software and hardware. The machine on which the language ran, LILITH, was designed with an architecture reflecting the structure of MODULA-2.

Maintenance: After The Sale

Ideally, from the customer's point of view, the dealer from whom you buy your micro should look after any problems arising after the sale. He should give uncomplaining assistance, answering all your questions, taking time to explain obscurities in the manual and helping you out with any disagreements arising with the manufacturer and so on.

In fact a dealer doing all that would be out of business in a matter of weeks, leaving all his customers high and dry. The profit a high street shop takes on the sale of the average home micro, making no allowance for the time spent with you during the sale explaining the machine, will just about pay for one hour of an engineer's time.

On top of all this you probably expect the dealer to cope with returning your micro to the manufacturers for repair under warranty, and to maintain the machine once it is out of the guarantee period, which means keeping at least one service engineer and a workshop. In fact the best dealers will do exactly this and more if necessary, but they need to work efficiently with a lot of help from you if they are to run this side of their business profitably.

When things go wrong

If things go wrong with your micro, don't immediately assume a hardware disaster. Something like two thirds of all the faults reported by new users are a result of not having read the documentation properly (which is often as much a fault of the documentation as the users), so it is always worth going back and re-reading the manual. What you need to know is probably in there somewhere. There is usually a check list at the back that begins by asking you if you are sure the machine is plugged in, that the fuses haven't blown and so forth. Go through it carefully, and the chances are that you will find that the fault is something you can correct yourself.

If you do have to go back to the dealer, remember that the extent to which you can depend on his help for after sales service is a matter of good relations rather than legalities. Supply him with a detailed description of the problem – type it or print it in capital letters and attach it to the machine with sticky tape. He may be able to fix it on the spot, and if he can't, a properly made out fault report will help speed the repair. Normally a dealer will exchange a micro if it breaks down during the couple of weeks after purchase, but otherwise it will have to be returned to the manufacturer and you will have to be patient. It is worth exploring the possibility of borrowing a micro from him in the meantime, but don't be too disappointed if he turns you down.

Third party maintainance

If your micro is making money for you, you may need to rely on more than a good casual relationship with your dealer. Traditionally, mainframe and minicomputer maintenance has been one of the more rewarding sides of a manufacturer's business because of the high capital cost and the relatively small number of installations. But microcomputers are sold in such large numbers that it is impossible for the manufacturers to cope unaided. So third party maintenance companies stepped in to fill the gap.

Microcomputers dealers often have certain maintenance companies to whom they will refer customers who want a contract, and this could result in a better deal than you might get from approaching the same company independently. But it is worth making general comparisons anyway, and if you decide to take the dealer's offer, to look closely at the service offered by the maintenance company he has recommended.

You want to be sure the company has qualified engineers. Ask for a customer list, so that you can see just how good they are. A reputable maintenance company will have a healthy list of clients, perhaps including accountants who rely a lot on their computers and demand a very good service.

Buying a fast service

One of the most important questions you should ask yourself is 'what is the longest tolerable response time you need if the computer breaks down?' If your work revolves around the computer a couple of hours might be the maximum, but if you only use it occasionally a twenty-four hour instant service will be a luxury you don't need. If you must have a quick response time, it is important to ensure that if the engineer can't repair the micro on site, you will be given an immediate replacement.

How much you pay will depend on several things. A standard annual contract offering a response time of a few hours will lie between 10% and 15% of the initial cost of the system. If you can let the fault ride for a day or two until an engineer happens to be in your area you should be paying a great deal less. This percentage only applies to microcomputers though – items like printers and disk drives are considered to be less reliable and charges may be as high as 18%. Some companies also consider certain makes of microcomputers to be more reliable than others – for 'reputable makes' maintenance can be cheaper. If you know before you buy that maintenance is going to be important it is worth phoning around the maintenance companies to sort out the wheat from the chaff.

Maintenance contracts

You may come across something called 'preventative maintenance' in the contracts. Although this it sounds rather impressive don't take it at face value. A very long time ago, BBC radio studios were run on a program of preventative maintenance that meant that recording sessions were regularly disrupted by visits from service staff to dismantle the potentiometers, spray them with cleaning fluid and reinstall them. Studies showed that the potentiometers lasted a good deal longer and gave a lot less trouble if they were left well alone and only attended to when they went wrong. The general lesson from all this is that once a technology matures to a point where

the manufacturers can deliver a consistently reliable product, preventative maintenance can be more trouble than it is worth. The modern microcomputer is a natural for the philosophy of 'let well alone'.

The insurance alternative

When you take out a maintenance contract you hope you'll never need it – and so does the maintenance company. They make their profits from having as few engineers as possible and selling you an intangible kind of 'assurance' so that you can carry on your business with confidence.

Why not go the full distance then, and sign up with a company which has no engineers at all, and has been in the 'confidence' racket for centuries. Outfits like this are called 'insurance companies', and they have recently discovered that they have a part to play in the micro market by offering a cheaper option to maintenance. Once again you are paying for something you might never need, but the cover usually works and costs far less than the maintenance rates of 10% to 15%. It is certainly an alternative worth considering, particularly for the home user, who will seldom require a full maintenance contract.

The same insurance company can also offer software cover. On a maintenance contract this is something that usually has to be negotiated separately, because it will be handled by different departments within the company. But insurers – who know as little about software as they do about hardware – are very happy to bundle the two together. For a premium of a few pounds a year you should be able to get cover for a small micro which offers reinstatement as new on all equipment up to five years old if it gets damaged or stolen. This price also includes full cover on equipment when the user is moving it around, as well as the cost of replacing software if it is destroyed either maliciously or accidentally. For the same price you will also get compensation for loss of business while data is being reinstated, as well as the cost of paying staff to re-key in data, and buying time on other machines.

Insurance companies, like maintenance companies, will charge more for covering anything with mechanical moving parts, such as printers and disk drives. They also charge more for full breakdown cover, when the computer merely refuses to work for some unknown reason, although this usually costs less than a normal maintenance contract.

Personal computer users might be able to negotiate even lower premiums, on the principle that the computer is used for less time during the day, and that non-business users don't need cover for financial loss to business, while the computer is repaired or the software replaced. But a home user might also find that cover for material damage to the computer can be higher – based, presumably, on the idea that personal computers sit not on desks, but on the living room floor, ready prey to careless feet. It is cheaper to have the average home micro included in the insurance for house contents.

Maintenance or Insurance?

If insurance is so much cheaper than a maintenance contract, where is the snag? The answer might be that maintenance companies generally honour their contracts without a murmur, whereas insurance companies have special departments to argue the toss and delay payments. Although a home user would probably be happy to wait a week or two before the compensation came through, a business user can't afford to shut up shop while the insurance company's loss adjuster goes through his leisurely paces.

Insurance companies insist that insurance is becoming more and more attractive to business users, who are not bothering to renew maintenance contracts at the end of the first year, but there is still a case for the business user staying with a maintenance contract for hardware, but taking out insurance for software and compensation for loss of business.

Memory Map: Key To The Micro

In computing as in cartography a 'map' is simplified pictorial overview of an environment. The verb 'to map' means (more or less) to translate. A computer's memory map is formed by translating the locations of the physical RAM chips as known to the processor into a logical table – an invaluable item to a computer, as we shall see.

But, confusingly in computer terminology, mapping can sometimes have a quite different meaning. To refer to an area of computer memory as 'memory mapped' does not mean that it has had a table drawn up. Memory mapping describes the process of taking a logical address and of translating (mapping) it into the physical address space. So producing a memory map is translating from physical to logical, while memory mapping is translating from logical to physical.

Physical to Logical Mapping

For systems or assembler programmers, the memory map is the most important piece of documentation supplied with a computer – the equivalent of the circuit diagram to the hardware engineer – giving the user a complete overview of the computer's memory arrangements. When you program in BASIC the interpreter will handle most of the planning, but if you use **PEEK**s and **POKE**s, knowing your way around the memory map becomes is essential.

An army general uses a map of the terrain to position his forces and to plan his campaign. Similarly, a programmer uses a map to place modules of the program in the most convenient position, avoiding areas pre-empted by the system, and making the best use of system subroutines and available memory. To stretch the military analogy a bit further, you could say say that certain areas of memory are minefields. **POKE** a byte into the wrong location, and the whole program blows up.

There are also areas such as I/O ports which are hives of activity and need to be well organised and co-ordinated, otherwise chaos and confusion will prevail. For ordinary high level programming this will be taken care of by the systems software, which designates buffer areas and defines the procedure for transferring data. But if you go in at a low level, a map becomes vitally important.

Many computers need two levels of map: a general map showing the positions of screen areas, cassette and disk buffers, reserved spaces for the operating system, the stack, and so on; with another far more detailed map of all the 'minefields'. A typical 'minefield' is the zero page on 6502 machines, deadly if you don't understand its layout, but potentially an area of prime real estate for programmers who do.

The zero page is the first block of 256 bytes of RAM, and is called this because the hex addresses of all the bytes in this block begin with the digits 00 (0000 to 00FF). Certain specific instructions for some processors (including the popular 6502 chip) allow this initial 00 to be ignored, so shortening the length of the instructions for the microprocessor and making them that little bit faster. Figure 1 shows the style of a typical memory map.

For a general map, a diagram showing labelled blocks of 256 bytes each is usually sufficient, with the hex address of each significant area beside it. But the usage of every individual byte needs to be known and documented on the more detailed map.

Making Your Map

Before you start programming in any depth on a new computer it is well worth going through the following procedure. First, using graph paper draw a table of the whole RAM memory and a second table (or tables) of any areas requiring detailed attention. Then go through the documentation of the computer, detailing – and perhaps even colouring in – all the areas used by the system; the buffer, screen, and operating system areas would all be prime candidates for inclusion.

The features of the memory map will depend on what facilities you are using – there is no point marking out the area used for a high resolution screen display if you are going to stay permanently in text mode. For the detailed maps, note the individual function of all the bytes used by the operating system – screen cursor positions, cursor character, screen modes, window sizes, bytes used for saving register values, keyboard scan counters, and so on.

A map is far more effective and easier to understand than simply presenting the same information in the form of a list. Once all this information has been collected, you will be free to assign areas on the general map to modules of your program, or storage areas for arrays and records. The detailed maps will allow you to assign certain bytes as global variables and pointers. It is very important to document the way in which you are using these bytes so that you don't over-write them by mistake. When the program has been finished, the memory map should be updated and preserved, since it can form the basis of the essential documentation.

If you have followed the suggestions in the entry on programming (*qv*) you will be developing your larger programs in modules, each module containing routines and data areas relating to one fairly specific function. One program module may control the drawing on the screen, another may deal with keyboard input while yet another handles the disk or cassette accessing procedures. It is very useful to have these separate modules documented and their RAM locations clearly marked on the map.

ROM memory can also be presented in the form of a memory map, but since nothing in ROM can be changed there is little benefit to be gained from working out how much space is used up. What are important are the subroutine entry points because some of these routines, such as keyboard scanning and output to the screen or printer, are well worth borrowing for your own programs. This sort of information is best presented in the form of a table of entry points and routine functions.

Assembler and machine code programmers will often use some of these very low-level routines to enable them to cut corners. For example, routines designed to log the position of the cursor after something has been printed on the screen can be very useful when programming a

complex display. Programmers will also have to use ROM routines when writing I/O routines such as communications packages.

The amount of system code stored in ROM depends on the type of micro. Home micros tend to be heavily ROM-based; that is, they hold nearly all the operating system code in ROM, along with the language interpreters and utilities. A ROM-based micro can be programmed without having to load any code from backing store. Systems with built-in disks usually have only enough code stored in ROM to load in the operating system from disk.

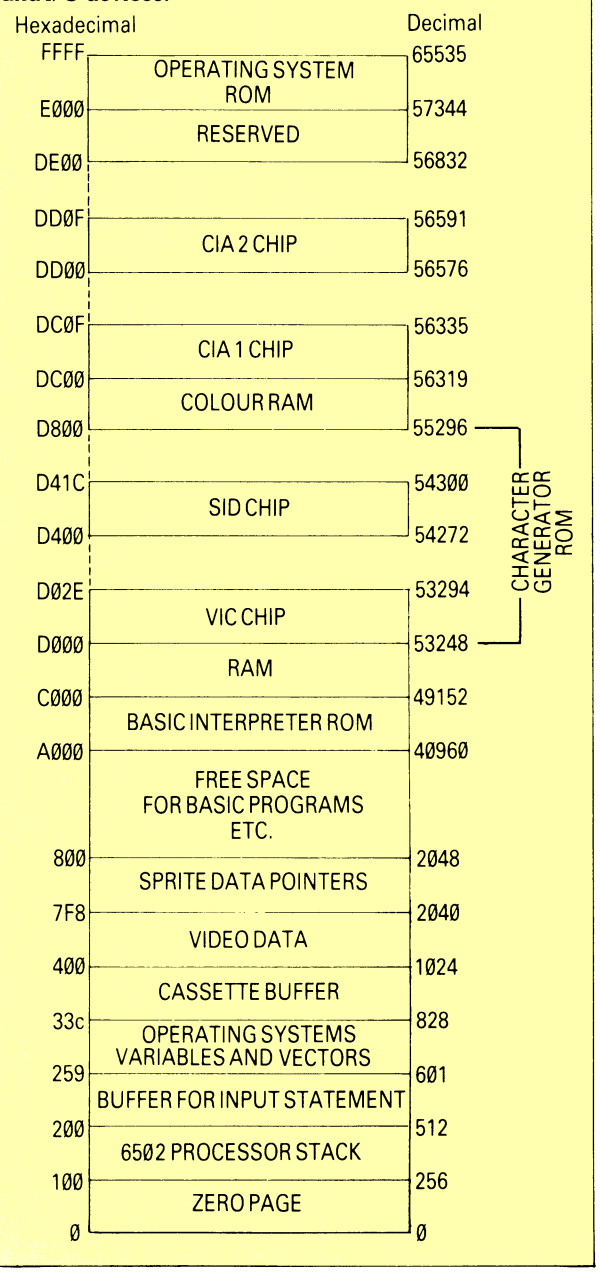
Even the best documentation supplied by manufacturers rarely presents the required information in the most useful form. So it is worth the effort of compiling these maps and tables, and adding to them whenever you come across a new routine or a new use for a variable. They are handy ways of recording useful information which should not be left in the hands of highly volatile grey matter.

Logical to Physical Mapping

One advantage of a memory-mapped area in the second sense is that it can be out of the directly addressable memory space of a micro, extending the amount of memory that a limited processor can handle, as well as providing a measure of security by making it harder to overwrite sensitive addresses by mistake. A mapped-out area of this kind will still have addresses, but a process has to be gone through to persuade the CPU to access it in preference to the main memory.

The screen display of most computers is sometimes described as memory-mapped. The addresses of the screen area in RAM is usually fixed by the hardware, and a separate processor or video display circuitry maps the contents of the display RAM onto the actual screen. This process should not be confused with the more general usage of the term memory map.

This shows the memory map of the 64 computer. The zero page and stack areas are essential reserved area for the processor and should be left well alone, apart from those locations already covered in other sections. Data entered in response to an INPUT statement is stored in the INPUT buffer. The operating system variables and vectors contain many locations which can be altered to achieve special effects. Data going to or from the cassette unit is stored in the cassette buffer, which is otherwise unused. The video data is the area which normally appears on the screen. Above the BASIC RAM which normally holds user programs are the interpreter and operating system ROMs and I/O devices.



The memory map of the Commodore 64 micro

Memory: Logical Space In Your Micro

Like the other integrated circuits in the system, memory chips are made of one the world's most abundant elements, silicon. Arrayed in neat rows inside your micro like a hibernating colony of silver-legged black beetles, these small components embody the logical universe in which your programs come to life.

Each chip contains a large number of tiny cells, or elements. In a variety of ways depending on the type of chip, each element is a 'boolean atom', a logical entity capable of being in one of two states. This isn't just an overblown way of saying that each cell is a 'switch' which is either 'on' or 'off', although the image has its uses and will crop up again in this book. The essential thing about a switch is that it controls something – a light bulb, the central heating, a car ignition system. In fact the word 'switch' is derived from the name of the small flexible rod used by a rider to direct a horse.

The logical elements inside the memory chips control nothing in this sense – they simply exists in one state or another by way of keeping a record of events, current or in the past. There is an illusion of control because the heart of the system, the processor, constantly refers to these elements as it goes about its business, rather like a contestant in a paper chase running from clue to clue.

There is a direct correlation between the state of these cells and the 'bits' which form the numbers (*qv*) that we like to think we are storing in the memory. But like the concept of the 'switch', the idea that the computer stores numbers is also a kind of illusion, created by the levels of software intervening between us and the microscopic physical reality of the chip's interiors. At the deepest level the memory knows nothing of our elaborate ideas; it simply stores patterns of cell-states, rather like a quad full of faithful recruits that individually will either salute or stand at ease on command.

There are two distinct kinds of memory chip inside most microcomputers, **ROM** and **RAM**, and it is important to understand the distinction.

RAM: The erasable blackboard

Random access memory, or **RAM** as it is almost always known, is memory that works like an erasable blackboard, allowing its contents to be changed, either by the user, or internally by the machine's operating system in the course of operations. In this way the processor is able to use the RAM space for the programs it runs, as well as for storing data for immediate use. It is this talent for allowing access to data directly without having to wind through a sequential process which, historically, gave random access memory its name. In fact (like 'random files') the name is a misnomer; the process of storing and reclaiming data from RAM has to be highly organised, and is far from random.

Any piece of information in RAM is changed simply by 'writing over' it, and the information over-written is lost for good. The new information is stored for as long as the chips are supplied with power, but if the computer's supply is cut off, whether by deliberately powering down or by accidentally tripping over the cable, everything in RAM is instantly lost for ever.

Some newer machines (typically battery driven) use a form of RAM called CMOS (Complementary Metal Oxide on Silicon) memory which retains its memory after power-down, but even for this a very low constant voltage is still supplied, even though the machine is officially off.

Because of this 'volatility', information in RAM needs to be transferred or 'saved' to some more permanent storage medium like cassette tape or magnetic disk before the computer is switched off (*ee* Backing store). Although the copy still in RAM disappears when the power goes, the saved version can be loaded back into RAM at the start of the next session.

ROM: Memory carved in stone

ROM is the alternative. As its name implies, 'read-only memory' can't be written to, but neither can its contents be destroyed by overwriting or by depriving the chip of power – the information stored is fixed by the manufacturer in the silicon as surely as lettering carved in marble. Typically this archival power is harnessed to store programs which help the user operate the system. Micros with BASIC 'built in' will keep the code of the interpreter in ROM; together with the rules for displaying the character set on the screen; maybe even a program to test the computer hardware whenever it is switched on might be integrated into the hardware in the same way.

Motley memories

These two functionally different kinds of memory are implemented in practice by a variety of technologies, offering the hardware designer many different opportunities when building a micro. The technical differences are immaterial to the average programmer, although system software writers may have to consider the speed of response of the different types of chip in certain routines expected to run in 'real time'. On the whole, the way the 'logical space' is created is considerably less important than its size and accessibility, so if your main interest is programming you may want to skip the hardware discussion that follows.

Static and **Dynamic RAM** are categories of random access memory as different from each other as they are collectively from ROM. Dynamic memory is about a quarter of the price of static, being engineered around the simple principle that each element behaves like a capacitor. At any given time a capacitor is either storing a charge, or not storing a charge, and this is enough to make the distinction between an 'on' bit and an 'off' bit. Unfortunately the charge in a capacitor tends to leak away in time – something like a millisecond in the case of tiny capacitors like these.

You might think that this unhelpful phenomenon would provide a reasonable excuse for abandoning the technology, but luckily electronic pioneers tend to be tenacious in their pursuit of a half-way good idea. They came up with what on the face of it sounds like the most outrageous bodge in the history of invention: they added

complex circuitry to examine each cell every thousandth of a second – and there are some five million of them in a typical 8-bit business micro – reading all the charges before they have decayed too far and then writing them back again! This meticulous, frenetic and completely invisible activity is modestly known as 'refreshing', and it is because of all this activity refreshing cells that they are described as 'dynamic'.

Static memory has a distinctly dull inner life by comparison. Each element is twice as large in each of its two dimensions as in the dynamic type, which means you will find a quarter as many on each chip, and they require more current. Instead of a capacitor-like cell, each element is made up of a set of minute transistors ganged together in such a way that at any given time they are either providing a voltage or not providing a voltage. The essential feature of this arrangement is that the cell can be flipped over from one state to the other by applying a small external voltage. This kind of set-up is referred to as a 'bi-stable flip-flop'. Its great advantage is that no external refreshing is needed, a flip-flop once flipped will stay flipped until it is flopped or until the power is removed.

Up until the end of the 1970s static RAMs were thought of as more reliable, but dynamic RAM technology has since developed to the point where this distinction is little more than a prejudice in the minds of a few nostalgic designers. Despite its higher price, static RAM still has its uses: rather than spend time and money on the additional design work necessary to support dynamic memory, manufacturers occasionally prefer the more expensive but simpler static form.

A more recent development is a range of chips called 'pseudo-static RAM'. These are simply dynamic RAM chips with the additional refresh circuitry built in. There make like a good deal easier for the circuit designer, because although they have all the virtues of dynamic RAM, to the outside world they look like static RAM.

Read-only memory comes in many more varieties. The two main ROM technologies are Fusible Link (FL), and Avalanche Induced Migration (AIM). Fusible Link ROMs contain small fuses which are blown by saturating them with current, so that they snap and break contact. AIM technology is similar: the difference is that a high energy electron flow (the avalanche), containing aluminimum atoms migrates through the silicon, causing the appropriate junctions in the circuitry to short-circuit and burn out. The number of fuses or junctions in a chip is equal to the number of memory bits, and so represents its capacity.

The best known **erasable** PROMS are the kind that you wipe under an intense ultra-violet light, but there are two other common erasable technologies, electronically *alterable* and electronically *erasable* (E**A**ROMs and E**E**PROMs). The electronic technologies alter the distribution of electric charge in the chip, and the presence or absence of a charge is used to represent a 1 or a 0. The charge is induced by a strong electric field, and by applying a reverse electric field, the chip can be discharged and the resident program erased.

The UV erasable chips are analogous to this. The gates (memory bits) are floated on an insulating sea of silicon dioxide, and by applying a high voltage, the insulating sea becomes conducting. Some electrons cross this sea to the gate, and alter its electric charge. But exposure to UV radiation also raises the conductivity of the silicon dioxide sea (photo-energetically), so that excess electrons leak back in the absence of a high voltage to prevent them. UV erasable chips are distinguished by a small crystal window through which the UV exposure takes place (and through which the surface of the integrated circuit itself can be seen – try looking with a magnifying glass if you get the chance). The erase process is not selective, and the whole chip is deprogrammed. Electronically erasable devices on the other hand can be selectively erased.

Both the fusible link and avalanche induced migration types are popular as ROM chips since they are fast, cheap, and resistant to radiation. Electronically erasable PROMS are proving to be most useful in control and instrumentation applications in cars, TV and radio controls, machine tools and so on.

Masked ROM is programmed during the manufacture of the chip, using an appropriate mask in a photolithographic process not unlike the printing of photographs from a negative. PROMs on the other hand are manufactured empty of information, and are programmed subsequently by the manufacturers. Users can even blow their own PROMs (see below).

The straightforward maskable ROMs are intrinsically cheaper, but their use implies considerable confidence in the program, and the system is not usually cost-effective for production runs of less than a few thousand. This is why many hardware add-ons, especially those from small manufacturers, have their system software stored in erasable memory – EPROM's.

Addressing memory

All microcomputer systems include both ROM and RAM for useful programs and working space respectively but the division between the two memory types in a system is not absolute. The programs in ROM, particularly the BASIC interpreter, usually need to reserve certain areas of the RAM space for their own use (for temporary storage of variables, for example). By definition, of course, it is impossible to store variables in ROM. For this reason, the actual amount of RAM the user has to play with is usually less than the total RAM space. These reserved areas should be documented in your manual, and if you value your sanity you will steer well clear of them (*see* Memory Maps).

Information is stored in memory in binary form, in blocks made up of nothin but ones and zeroes. Most common microcomputers use eight binary digits, or bits, as the standard unit of storage; and eight bits make one byte. You'll find a fuller discussion of this in the entry on Numbers.

Eight binary digits, each of which can have only two values, have 256 possible combinations; that is, one byte of memory can contain one of 256 bit patterns. This is fine as far as storing things in RAM or ROM goes. Two hundred and fifty-six byte patterns are more than enough

to allocate one pattern each to all the letters in the alphabet, both upper and lower case, all the numeric characters, all the mathematical and punctuation symbols on a typewriter keyboard, and still have plenty over for special control codes the computer might require (*see* ASCII).

But when it comes to getting information out of memory, the computer needs to be able to recognise where in RAM or ROM particular bytes are. So each byte location in memory is given an 'address', each address being a different pattern of binary digits, ascending from the start of the memory (at address zero) to the end.

Obviously, the 256 patterns given by a byte of digits is far too small to label each location in a useful amount of memory space. The corollary is that each address needs to be longer than a byte to give a respectable number of possible location address patterns. Most common microcomputers therefore use 16-bit addresses, allowing enough combinations to give 65,536 patterns – the value of 2^{16} (two to the power of 16). It isn't a particularly easy number to remember in decimal, so it's usual to think of it instead as 64 lots of 2^{10}. This unit is chosen because it works out as 1,024, which is sufficiently close to 1,000 to be called a kilobyte. So 65,536 becomes 64 kilobytes, or 64K for short.

The maximum memory on 8-bit computers is 64K, since only 16-bit addresses are provided. But 16-bit microprocessors like the Intel 8086 and 8088 microprocessors can handle 20-bit addresses, letting them select from any of 1,048,576 locations which is 1024^2 (1K squared), or simply a megabyte. The Motorola 68000 chip, has an even greater addressing range; its 24-bit addresses can access 16 megabytes of memory, or 2^{24}.

Remember that we are talking about the number of addressable bytes in a system. Don't be confused by the fact that in discussing individual chips rather than the total storage capacity of memory, the symbol 'K' takes on a humbler meaning. A 16K RAM chip does *not* hold 16 Kbytes. *The capacities of chips are quoted in terms of bits*, so to work out how many bytes a chip holds, divide its given capacity by eight.

Memory chip specifications

During the heyday of the 8-bit era the 'bread and butter' RAM chip in microcomputers was the 4116 dynamic RAM, with a capacity of 16 Kbits. The most frequently used static RAM chip was the 2114, holding only four Kbits. But now, in all except the most conservatively designed machines, these have been replaced by a new development in RAM, a chip with a variety of type numbers, one of them being 4164. Comparing this with the 4116, you might guess that it can hold 64 Kbits of information.

Apart from capacity, the most important specification is the 'access time', given in nanoseconds. This is the time it takes between the processors sending out the address of a location to RAM and the information from that location appearing on the bus. A nanosecond (10^{-9} seconds) is one thousandth of a microsecond, which is in turn a millionth of a second. A typical access time is 200ns, which means that the computer can access five million different memory locations in one second.

Common numbers for ROM chips are 2716, 2732, and 2532. These are all EPROMs, the erasable and reusable sort of ROM; the first has a capacity of 16 Kbits and the other two have capacities of 32 Kbits each. ROM chips are generally much slower than RAM chips, but their access time of 450 to 800 nanoseconds is not so critical, since the processor is only running programs stored in ROM and not switching information in and out of the chips.

The other chip parameter that designers need to consider is the power supply requirements. Early ROMs sometimes needed as many as three different power supply voltages. Now the big selling point is that a single five volt supply is all that is required, the same as that used by the microprocessor, RAM and other chips in the system.

Adding more memory

One of the rare points of agreements in microcomputing is that more memory is a good thing. Just as Napoleon is quoted as saying 'you can never have too many troops on the battlefield', a programmer cannot have too many bytes. With more bytes, more sophisticated software can be developed faster. And memory has the advantage of being one of the few upgrade options that can be plugged into most systems with a minimum of fuss.

Prices for memory add-ons can vary enormously, so shop around. Memory is rarely unreliable as long as it is appropriately packaged for your machine.

The most obvious reason for adding on extra memory is to hold bigger and therefore more sophisticated programs. This benefit is not always obvious; software developers have shown great resourcefulness in cramming code into limited RAM, though this impressive economy does have its disadvantages: there are simply some things that cannot be done as well with less memory.

On some systems, a lot of fairly sophisticated operating software is packaged up in the ROM. But one of the disadvantages of this (besides having to pay for all that memory when you buy the system) is the lack of adaptability. In order to load in another operating system, you either have to plug in a new set of ROM chips or bypass the ROM and thus render it redundant.

Operating software probably benefits most from more memory, but the sophistication of applications software is almost as dependent on the amount of memory available. It's hard to gauge the benefits because there's no direct relation between sophistication and memory capacity. Much depends on how well the program is written, and the quality of display and so on.

On the smaller machines ambitious software has to be written in assembler to make the most of the limited space, a process that involves long hours of coding and debugging. Expanding the memory allows very similar programs to be written in a high level language and compiled on the machine, which is far easier and quicker.

The way to go about adding extra memory is to start where all hardware considerations should begin – with the software. What is it that you want to run but can't because your machine is short of space? If the answer is nothing, then perhaps you should leave extra memory alone for the moment – there's no point in being the first one on your block with 48K. But perhaps there is a specific program you want to run that won't load without that deadly 'Out of Memory' error. Most of the cassette-based packages are written for the common configurations of the system: 1K, 16K, 48K or whatever, and will announce the size they need on the packaging.

Programmers will quickly find that anything under five Kbytes is very limiting, particularly in the restrictions imposed on arrays. But don't think that extra RAM can yield better graphics – only rarely is this the case. Extra memory may give you better resolution and more colours but only if the system is properly configured.

Usually the system sets aside an area of RAM for the display to hold the screen map, where sets of one or more bits represent each of the screen's pixels. Enlarging this video RAM allows the map to be more detailed, giving more pixels or more bits per pixel, but only if the system software is adapted to recognise the new arrangement. Manufacturers rarely provide software refinements like this, so the resolution and colour offered by a micro usually reflects the amount of RAM supplied with the minimum system.

Eight-bit disk-based machines will seldom benefit from a memory upgrade, as the usually come with the full 64K the processor is capable of addressing, and most 8-bit software is designed to run in a minimum of 48Kbytes. In the 16-bit league, the minimum is 128K, but many of the more advanced packages need twice this. The eccentricities of the individual hardware allowing, you can go on piling memory into the average 16-bit system up to the 1 megabyte limit.

RAM as cache and disk

Microcomputers are essentially very speedy devices: a typical pseudo-16-bit chip like the 8088 is capable of performing 100,000 instructions a second – as long as all it has to do is compute. What slows it down in practice are the real-world operations that make the machine useful. Operations like writing to and reading from disk, and producing a print-out.

For a long time mainframe designers have used a simple solution of buffering I/O (Input and Output) devices wherever possible. Buffers literally absorb the impact on the internal system of external operations by interposing a section of memory. Instead of writing directly to the external device and having to wait while it accomplishes all the stages of its task, the processor sends the data to a block of memory and returns immediately to get on with its main job. Thereafter the data is either transferred by independent I/O devices without the intervention of the main processor, or will wait in the the buffer until the processor has an idle moment in which to complete the transfer.

The same idea works in the opposite direction, when the processor is reading from the disk. If the disk can somehow have the wanted sectors already available in the buffer, the processor need not wait while physical motors churn round the magnetic media and heads click into position to peel off the data. A buffer used for this purpose is called a 'cache', from the old conjuring term meaning a hidden pocket. This cache idea is susceptible to some interesting developments. One is to make the buffer very large and eliminate completely the business of transferring data to the real hardware disk. You do this by giving the buffer (often called a RAMdisk in this context) a logical sector structure that makes it look to the system exactly like a disk. Reads and writes involving this 'pseudo-disk' take place hundreds of times quicker because no physical components have to be moved. The traditional disadvantages are:

RAM is something like one hundred times more expensive per byte stored than magnetic media.

Data written to RAM is volatile – when the machine is powered off the data is destroyed.

This picture is changing because RAM prices are falling, and the wide availability of battery-backed CMOS RAM means that volatility is now avoidable – albeit at a higher price. But even before these recent advantages the pseudo- or RAM-disk had already begun to show itself worthwhile in applications including real-time process control and communications, where speed is essential.

Extending memory with disks and banks

The principle of extending memory beyond the immediately available RAM is still relatively new to microcomputing, having first been developed for use with mainframes.

Virtual memory, sometimes called 'paging' or 'swapping' is one technique which reverses the idea of the RAM disk by making the backing store look like RAM to the system. This is achieved by having routines invisible to the user to map the physically disparate available RAM and a large section of disk space onto a logical continuum that presents itself to the running program as 'available memory'. In order to fulfill this promise, behind-the-scenes software busily transfers sections of core memory to and from disk in chunks called 'pages'. At least two word processing systems, PERFECT WRITER and FINAL WORD, use this technique to very good effect, allowing almost limitlessly large documents to be edited effortlessly. Virtual memory is also a feature of the language APL (*qv*) and the UNIX/ESC (*qv*) operating system.

Bank switching is another way of extending memory addressing, and is occasionally used to get round the limit of 64 Kbytes of memory space imposed by 8-bit processors. The processor still only addresses 64 Kbytes of memory at a time, but there are several sets, or 'banks' of 64K RAM, and these are shuffled in and out of the processor's address space, usually by sending

appropriate bank address values to a port set aside for the purpose.

Normally this switching of the banks is handled by the operating system, so that the user or any applications package running on the system is only aware of an enlarged amount of usable memory.

Blowing your own ROMs

If you write good software it deserves to be preserved, and one of the best methods is to burn it into a PROM. They are fairly robust, have very quick access times, and are difficult to pirate. Preserving software on chips is also useful for control applications which need very high reliability. Home users can also use programmable memory chips as an alternative to disks or cassettes.

A few computers come readily equipped with ROM reading circuitry, and others have their own special complications, like needing bank switching to bring the ROM into the processor's address space. But normally, as well as a computer, three additional pieces of hardware are necessary to start programming chips: a programmer, a ROM reader, and an EPROM eraser. You will also need a healthy supply of EPROMs. A programmer in this context is a piece of electronics consisting of a socket adaptor, a program card set, and some control electronics. There are manual programmers with their own keyboard, display and RAM, and these can be used independently of a computer although they aren't easy to program.

The best way is to develop the software initially on the computer, like any other program. Once it has been completely tested and debugged, determine how much machine code you have created so that you can choose an EPROM chip of the right size. Working out the size of assembler programs is quite easy, as you will already know the program and data areas, but in high level languages this problem is not so easily solved. An assembler routine has to be written, if not already supplied with the hardware, that will allow you to count the number of bytes your program occupies.

EPROM chips all carry code numbers like 2516 or 2532. The first two digits are the manufacturer's type identifier, and the second two are the size of the EPROM in K*bits*; so a 2516 chip contains 16K bits or 2K bytes. Usually, 25 refers to a chip that requires a 3 volt power supply to read from it, and a 27 chip a 5 volt power supply, but this convention is not strictly adhered to.

Now is the time to connect up the programmer to your computer. To load the program into the chip you will need to run the special system software supplied with the EPROM programmer, then put your program into RAM – obviously you will have to make sure you have enough RAM to hold both – and follow the instructions supplied. EPROM chips designed to run on small computers are sometimes prepared on larger machines using the same microprocessor because more powerful assembly and editing features are available.

Once the program has been loaded, or 'burnt in', the next stage is to test it. You won't be looking for bugs – they should have been ironed out in the development stage – but to makes sure the loading has been successful. This is where the ROM reader comes in, and you'll also need it to run the final product.

Wiping the slate clean – The EPROM eraser is your 'get-out clause' in case things don't go as planned. Erasure is achieved by exposing the EPROM chip to high intensity ultraviolet radiation, but don't expect it to be instantaneous. If no times are stated in the documentation, put a programmed EPROM into the eraser for a few minutes, then take it out and test it with the reader to see if it returns all its cells as FF_{16}. If it does it is blank, in which case you can multiply the time taken by three to give a reasonable safety margin. If it isn't blank, repeat the treatment.

It is the EPROM's ease of erasure that makes it ideal for prototyping, allowing you to prepare and test the software again and again until it is satisfactorily debugged. If you need a very large quantity – perhaps you are selling a game by mail order – then the next stage is to prepare a production run on ROM chips.

In very much the same way, a home computer operating system or BASIC interpreter starts life as a normal program, but when finished is stored in ROM and integrated into the internal components. Games cartridges are developed similarly, but are plugged into a suitable external interface.

Modem: The Micro Phones Home

Before you attempt to connect micros together over the telephone you need to be clear about what it is you intend to communicate. It is relatively simple to send text files, even between micros of different makes, because text is (nearly always) stored in your micro as ASCII code (*qv*).

But if your main purpose is to exchange BASIC programs, you are going to run up against problems unless the machines at each end are of the same make. In most machines a program in interpreted BASIC is stored not in ASCII but as a set of 'token' bytes, each token representing a separate instruction. The relationship between token bytes and the instructions they represent is unfortunately highly machine-dependent.

Some BASICs, particularly those on machines with floppy disk, have a **SAVE** command which gives you the option of storing the program in ASCII form. If your computer uses interpreted BASIC, but doesn't have a **SAVE** facility that works like this, you will have to open a file and write the program listing to it. This will then be created as a transmittable ASCII file.

The RS-232 connection

The next step is to check the input/output ports at the back of your micro. The more expensive micros usually come equipped with an RS-232 port, the socket you use to connect up your micro to the telephone network. Although a Centronics port is very useful for connecting high-speed printers and other devices, it isn't suitable for use with a telephone line, which can only cope with **serial** data.

For micros without a serial interface it is usually possible to buy an RS-232 add-on, although for the cheaper micros you may well find that the add-on costs nearly as much as you paid for the computer!

Unfortunately you can't just hook your RS-232 port straight into the telephone junction box. As well as being illegal, it won't work. You need a device which will turn the computer's output into a form that can be used on telephone lines. This is the function of the modem.

The Modem

The modem sits between the computer and the communications lines, **MO**dulating and **DEM**odulating signals to and from the computer, which is how it gets its name (**MO/DEM**). In simple terms it can be thought of as an interpreter, turning the computer's **digital** signals (*ON*s and *OFF*s) into **analogue** signals – more like the signals generated by your voice speaking into the handset's microphone. This is modulation, while demodulation is the reverse, turning analogue into digital.

Acoustic versus hard-wired

There are two types of modem: acoustically coupled and hard-wired. Micro users have usually opted for acoustic couplers because they're cheaper and easier to use. The acoustic type is usually no larger than a shoe box, and has two rubber caps which fit around the mouth- and ear-pieces of a telephone. Once a connection to the other telephone is made – by dialling in the normal way – a carrier signal acts as a kind of background tone to a series of blips corresponding to the digital values of the transmitted signal.

As all you have to do is place your telephone handset into the rubber cups; no extra wiring is needed. But this kind of modem tends to be slow and is inclined to garble messages. The design fits only handsets based on the British Telecom 700 series (the more usual type, not *Trimphones* or newer models).

Acoustic couplers are a cheap way into communications if reliability isn't crucial, as in the exchange of messages over a bulletin-board (*qv*) system, but when used to exchange program or data files, where errors would be ruinous, additional error-checking software has to be used.

The acoustic link is the source of the unreliability. The signal tends to be at the mercy of the internal hardware of the mouthpiece, external noise and vibration. In particular when acoustic couplers are used near a typist, the thud of the keys tends to upset the audio signal.

External sound and vibration can usually be dealt with by having the handset well insulated by the rubber cups, and placing the whole coupler on a sound-absorbent mat. The mouthpiece problem is worse in old phones. Carbon granules inside the handset's microphone can become compacted or very loose, causing crackles and disturbance on the line. The result can be the loss of several bits of data, and hence garbled transmission.

Hard-wired modems tend to be faster and more reliable. The snag is that you will have to get British Telecom to wire it in for you, it is still illegal for you to make your own connection to the public telephone system.

Originate and answer

The cheapest modems are sold as *originate-only*, which means that they can only be used to initiate calls. This makes it impossible for owners of originate-only modems to communicate with each other: any of them could start the call off, but none of them could receive it.

The more expensive devices have an 'answer' as well as an originate mode. This enables the coupler to receive a call as well, making it a far more practical device for hobbyists. But the owner of an originate-only modem could still dial into a bulletin network and receive an answer, because bulletin-board modems are left continuously in answer mode.

You would be well advised to buy a BT-approved modem, not only because you will be obeying the law but also because you can be confident that the modem will work. If the modem is approved, it will have a sticker on it with a green dot and the words:

> *Approved for use with telecommunications systems run by British Telecom in accordance with the conditions in the instructions for use.*

If it is not approved the law requires that it should carry a sticker with a red triangle and the words:

Prohibited from direct or indirect connection to any telecommunications system run by British Telecom. Action may be taken against anyone so connecting this apparatus.

Once the initial hurdle of making the connection has been overcome, a hard-wired modem is undoubtedly easier to use. Instead of placing the handset into a coupler once the recipient's number has been dialled, the handset is replaced and the phone is not touched again during the call.

Full- and half-duplex

The more expensive modems usually also offer the choice of full- or half-duplex transmission. Generally, if a modem isn't switchable between the two modes it will be wired permanently as full-duplex. Although not essential, half-duplex mode is quite useful, for example for printing.

Full-duplex mode allows two computers to transmit and receive data simultaneously. The transmitting computer expects the receiving modem to echo back each character sent. Should the message appear on the sender's screen as rubbish there's a strong inference that the line is corrupting transmission. Conversely a correct message on the sender's screen confirms the soundness of the line, because the message will have had to travel all the way down the line and then back.

Half-duplex mode allows data to be sent in only one direction along the communication line at any one time. A modem communicating in half-duplex is either transmitting or receiving, but not both. Unlike the full-duplex mode, characters are echoed 'locally' if they appear on the screen at all. In half-duplex mode a correct message appearing on the sender's screen is no assurance of the line's validity.

If at any time you use a modem and the same character appears twice on the screen (like this: SSOOMMEETTHHIINNGG FFUUNNYY) it's a good indication that you are mixing your duplexes.

At first glance there may seem no reason for using half-duplex, especially as most modems (including the cheapest) have full-duplex as standard. Bandwidth is the reason, but it is becoming steadily less important as modems get cheaper and better.

Bandwidth and speed

The limiting factor in communications along a telephone line is the bandwidth. This limit is well defined, by the international telecommunications standards body CCITT, and covers the frequency band used by the human voice, between 0.3 and 3.4 kilohertz. When a modem is operating in full-duplex (which needs *two* different carrier tones for send and receive) the bandwidth is split up into areas called envelopes. To keep the communications clean, faster transmissions require wider envelopes. This in turn uses up more of the bandwidth, which stretches the tolerence of the line. As the speeds increase (and the tolerences are stretched) you need more expensive modems to prevent errors creeping in.

Alternatively, you can go half-duplex. This uses less of the bandwidth, which means you can talk faster with fewer errors – although it is less easy to spot them when they do occur. As modems get cheaper and telephone lines and switches become more refined, faster speeds at full-duplex are becoming possible. But generally speed costs money.

Communication **speed** is measured by the number of modulations per second, where a modulation is the change in the signal of the transmission line (*see* baud rate). Micronet and Prestel, two public databases, work at 75/1200 and 1200/1200 baud (the first figure is the transmission rate, the second the receive rate). Most of the bulletin boards, such as REWTEL, work at 300 baud. Because of the differences, a 300-baud modem cannot be made to communicate with a 75/1200 system.

Staying in step

When two computers are communicating with each other, they must both be working either asynchronously or synchronously.

Most modems work on **asynchronous** transmission (also known as **Start/Stop**), where data is sent as separate characters or bytes, each as a batch of bits *framed* by extra 'start' and 'stop' bits. These serve to tell the receiving terminal when to start and stop reading that character. The framing bits also act as check marks for the receiving terminal's clock, getting it in line with that of the sending computer.

Synchronous transmission is a method of sending data at faster speeds than is possible with asynchronous transmission, but is not usually used with micros. Instead of sending data as single characters it is sent in large blocks. Synchronization between both sending and receiving terminals is maintained by a separate clock at each end, which means that data can be sent without gaps and therefore the need for start and stop bits.

The 64 is provided with an RS232 port, although its voltage levels are 0V and 5V rather than the standard −12V and +12V. This means that an external interface is required to connect the 64 to any true RS232 device – only another 64 or a VIC can be connected directly, and then only over very short distances. The RS232 port is, however, capable of supporting all the common baud rates from 50 to 2400, 1 or 2 stop bits; word lengths from 5 to 8 bits; mark, space, odd, even or no parity; half or full duplex and either 3-line or full X-line handshaking.

Having added a standard RS232 interface, or bought a modem specially designed for the 64, it is possible to use the port directly from BASIC, provided that high speed is not required. The limit will be at best 150 or 300 baud and will depend upon how the program is written. The normal file handling comands, OPEN, CLOSE, INPUT# and PRINT# are used – just as for a disk unit.

This is all you need to transmit text data between two machines, but to transfer programs you really need to use a program written in assembler.

Monitors: Micro Display Devices

Dedicated display devices for computers are expensive, and for this reason many home micros are sold without any sort of screen display, instead having a socket to feed a television set, either directly or through another unit called a modulator. This helps to keep down the manufacturing cost, but like that other cost-cutting compromise with existing domestic technology, the cassette interface, using a television set is cheap, workable and far from ideal. Once the initial excitement of owning and using a micro has worn off you may well feel that the time has come to think about buying a monitor. An alternative, if your micro has an RS232 socket (*qv*) and the supporting software, is to upgrade to a fully-fledged computer terminal.

The word 'terminal' in the data processing business technically means any piece of equipment that terminates an input/output line, and this includes printers and plotters as well as the more commonly associated visual display unit (usually just known as a VDU). The word 'monitor' is often used interchangeably with VDU, but is strictly speaking a write-only device, usually a cathode ray tube (CRT).

Using a TV

A lot of research has gone into the problems of using VDUs in offices; with very little into the use of a television set as a computer screen at home. Either way, the causes of eye-strain are likely to be much the same, aggravated by the fact that television sets were not designed to display stationary characters to be read over long periods of time, as happens when, for example, you are debugging a Basic program.

Specifically:

Television sets use a system of line interlacing that tends to create a juddering image particularly visible in an unmoving picture.

Character definition is restricted by the bandwidth of the interface.

Glare and specular reflection are caused by the untreated, unfiltered glass surface of the screen.

To cut down the eye-strain the first simple precaution is to make sure that the television set is properly tuned in to the computer. For reasons explained below, the character details will never be as clear as on a monitor or a dedicated VDU, but the characters are – by way of compensation – usually bigger, and with careful tuning they should become quite acceptable. T-junction coaxial cable connectors which enable you to plug in your TV aerial and micro cable together may produce a degradation of the screen image due to leakage of the signal into the aerial downlead. In this case you may think it's worth trading off the slight inconvenience of juggling the plugs around against the health of your eyes.

Once you have made the characters as clear as possible, check the lighting in the room. In a room where the ambient lighting is too dark, the screen shows up as a contrasting glare of light, quickly producing eyestrain. Conversely you don't want the room so bright that the characters appear faint against the washed-out background. It is often worth physically moving the setup so as to defeat any reflections on the screen from a window or bright light.

Many micro users make the mistake of sitting far too close to the screen. Make a habit of sitting at least five feet away, and if your connecting cable is too short, buy (or make) an extension.

TV compatibility

An American microcomputer with a TV interface bought in America will not work with a British television set because the line standards for television pictures are different in the two countries. The American television picture is made up of 525 lines and 60 fields per second (the NTSC system – which disgruntled television engineers swear stands for Never Twice the Same Colour), the British picture uses 625 lines with only 50 fields per second being displayed (the PAL system).

Conversion between the two standards is a messy and expensive business, and the simplest solution for owners of incompatible US micros is to get hold of an American TV set. There is obviously no need to be concerned about American computers bought in this country, as they will have been adapted to sell over here.

The PAL standard used in Britain is also the standard throughout Europe – with the uncomfortable exception of France. The contrary French use a quite different system called SECAM, so avoid any bargain micro you might be tempted to pick up during a holiday in that country. German television stations also transmit signals on VHF as well as UHF, and it is wise to check that a German computer is capable of matching British UHF standards.

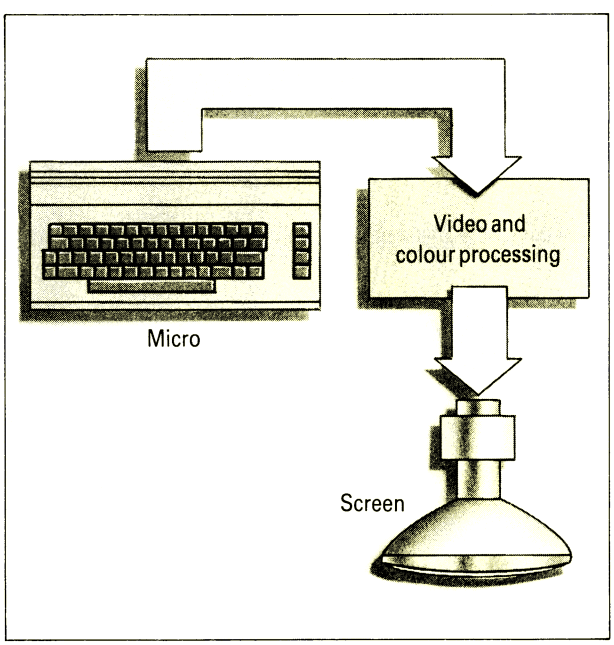


Fig. 1. The path taken by the composite video signal from micro to a monitor

Ancient British TV sets that hark back to the old 405-line standard used before 1964 will probably work with a modern home computer, but there will be a severe loss of quality with the characters being noticeably blurred.

The licensing laws

The licensing regulations in the UK allow you to run more than one television set under a single licence. So if the demands made by your micro on your domestic TV set tempt you to buy a second set, you won't be prey to the detector van as long as at least one of your sets is legitimised. A couple of exceptions are worth noting:

If you buy a colour TV set and your licence is for black and white only, you will need a colour TV licence.

If you keep your second television somewhere other than your home, two separate licences are required; a single licence only covers all the television sets in any one household. The definition of a 'household' is not always simple: a small flat attached to your house exclusively occupied by an aged relative has been deemed a separate household unless you can prove the relative regularly eats with you.

The law is more complex in the rather rare case of a household using a single television set exclusively with a micro. The test is not whether you watch transmissions, but whether the set is capable of receiving them. Effectively if you install a set in full working order with an aerial, you will have to buy a licence for at least for the first year, irrespective of your claims to being immune to the temptations of the broadcasting companies. If after that time you are still not watching programmes, you don't need to renew your licence.

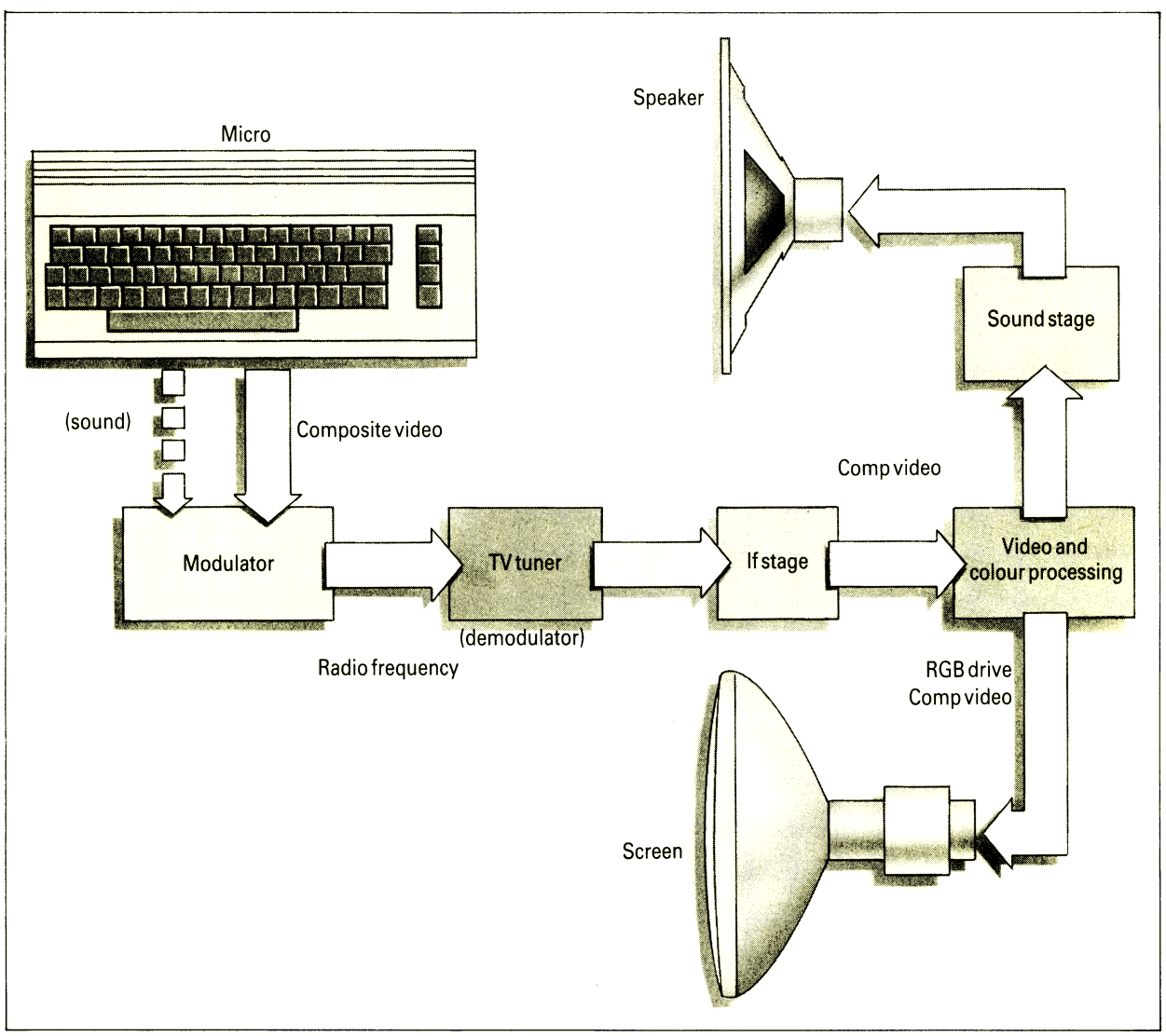


Fig. 2. The path taken by a modulated video signal from micro to a television

Thereafter you can still be taken to court for not having a licence, but as common law decrees, you are innocent until proven guilty, and it is up to the authorities to prove that you use your set to watch television programmmes.

You can avoid buying a licence altogether if you install a set which is not capable of receiving television programmes in the first place. In a fringe area it might be enough simply to keep the set away from anything resembling an aerial; where reception is good enough for a wire coathanger to do the trick it might be legally safer to have the tuning circuit modified by a properly qualified electrician.

When you buy or rent a television set, even a second-hand one, the dealer is obliged by law to inform the television licensing authority. If you don't already have a licence they will send you a reminder, but, subject to what we've said above, you can safely ignore it if you are genuinely only using the set as a monitor. Micro users need not worry about TV detector vans, because these detect whether you are receiving a signal, not whether you have the set turned on.

A monitor

Whether a dedicated monitor will give you a worthwhile increase in quality depends on the quality of the television set you have been using, and on the graphics software demands you make on your micro. If you are using a low-cost, black-and-white portable almost anything is likely to be an improvement. On the other hand, if you use very little graphics, and don't want high resolution, a good quality television set may be as much as you need.

Monitors advertised vary in size and quality. For general use you want a 12" or 14" unit; 9" units are really too small to be useful. Anything larger is also not particularly recommended unless you are able to move it a reasonable distance away. You will also need to work out whether the monitor has a good enough definition for your needs, which can be done from the specification of the unit.

How a monitor works

The cost of a good monochrome monitor will be in three figures, and colour may be several times that. As this may be more than the computer itself, it is worth understanding exactly what you are getting for your money. Figures 1 and 2 show the video signal path from the computer to the screen when using a monitor and when using a TV.

For the purposes of a monitor, the video signal produced by the computer has only to be amplified before being displayed on the screen. For the TV set however the computer output has first to be modulated to convert it into a UHF signal – the same sort of signal transmitted by the broadcasting companies. This 'mock transmission' is then fed into the aerial socket of the TV, demodulated to turn it back into the original video signal, and finally amplified and passed to the screen.

The processes of modulation and demodulation are never one hundred percent efficient, and the small losses which occur will inevitably affect the video signal. When the signal finally reaches the video amplifier circuit in the TV, it is a less than perfect copy, and this shows up on the screen as blurring, instability and colour aberration.

The picture on the screen is formed by a rapidly-moving dot, which scans the screen, varying in brightness as it goes. It is this variation in brightness which produces the visible image. To produce a steady display the coating on the screen (called a phosphor) continues to glow after the dot has passed.

The dot is driven by two oscillators: the **line**, or horizontal oscillator and the **frame**, or vertical oscillator, and it is important that these two stay in perfect step with the incoming video signal otherwise the picture will 'roll'. The frequencies of these oscillators can often be controlled by externally adjustable potentiometers called 'line hold' or 'frame hold', but the real task of keeping them in step is performed by a signal called the 'synchronisation' signal (sync for short).

Persistance refers to the time required for the phosphor to stop glowing after the dot has passed. Computer monitors need to have quite short persistence, otherwise moving objects will leave behind disturbing 'trails'. If you buy a proper computer monitor, you are unlikely to find this a problem, but some TVs or non-computer monitors may not be up to scratch.

The monitor may be connected to the computer by way of one of two main type of interface: **composite video** and **RGB**. In the first, the 'sync' signal is combined with the video information (hence the term 'composite'), while in an RGB interface, separate lines are provided for the Red, Green and Blue signals and the sync signal (hence 'RGB'). Composite video monitors normally use a 'phono' socket while RGB units usually have a DIN socket – some monitors are equipped to deal with both sorts of signals. The two types of connectors can easily be bought from hi-fi shops, if you are able to wield a soldering iron you can save money by making up your own cables (which are otherwise often quite unjustifiably expensive).

Monitors with a sound amplifier and speaker are available but you may have to shop around. In addition to the video signals, a separate audio signal has to be provided if the computer is to generate sounds through the monitor.

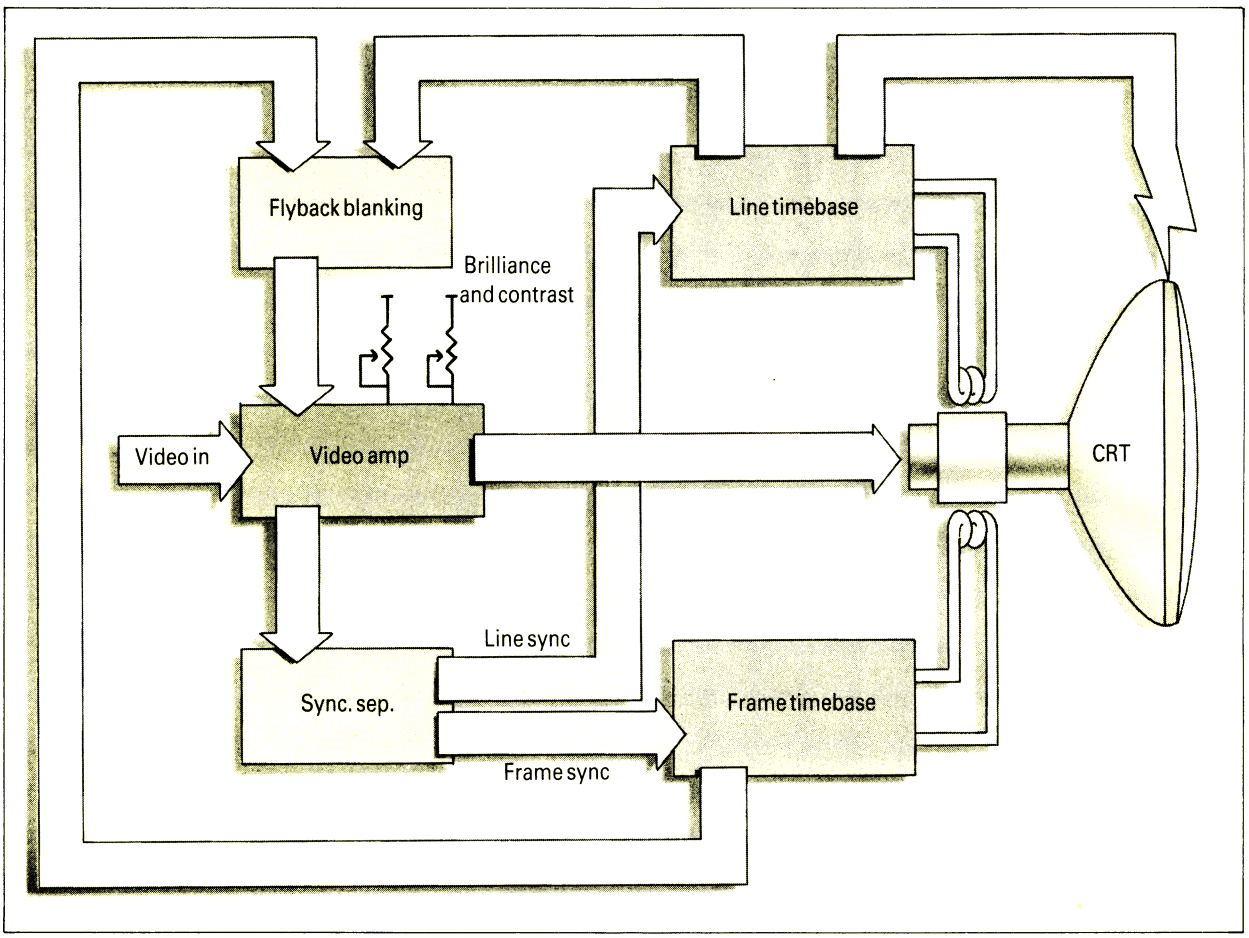
The advantages of a monitor

A monitor may well be worth its price if you are spending a lot of time with your computer. The advantages of using a monitor are not simply aesthetic:

Since the display is sharper and steadier, using the machine for long periods causes less eye strain.

Monitors are specially designed to be used for long periods at close range, they have lower radiation emission levels than a domestic TV, which is usually viewed from a distance of 2 or 3 meters.

The computer user can work happily out of the way,



Block diagram of a monitor showing the main components

leaving the family TV free for more conventional viewing. The monitor is physically smaller than a TV and can easily be put on a table or even on top of the computer itself.

Colour or monochrome

Obviously a keen games player is going to want a colour monitor to get the best visual effects. But if you're using the microcomputer as a word processor, maybe with an accounting package or spreadsheet program, a monochrome screen may be easier to work with.

For monochrome screens the most popular colour is green, although white (really a sort of bright but light blue) is also common. The choice is not simply a matter of taste: you want a colour which is easy on the eyes in order to use the machine in as relaxed a manner as possible. Tests in industry have shown that people usually do not realise when they are stressed, but they work less efficiently and tend to make more mistakes.

Green is generally considered to be easier on the eyes, but Continental ergonomists have successfully persuaded the market that the only screen colour worth having is a faintly greenish orange. Arguably far more important than colour are the factors of image size, resolution and stability.

The character display

Shapes on the screen are built up from dots arranged in a rectangle (the dot matrix principle again, *see* Printers); dots are illuminated as required to form the character. An eight by eight grid of dots is common among the cheaper displays, but this low resolution either produces characters too small to be seen easily, or characters that are noticeably 'dotty'. Better terminals will have a grid size of around 12 x 12, resulting in clear, large, easily distinguished characters. Look particularly at the descenders: 'g's, 'p's and the like should have a generous depth. Line spacing should be sufficient to avoid descenders clashing with ascenders.

Stability is crucial; the eye tires quickly trying to read a text that wobbles or judders. The swimming phenomenon found on many imported monitors due to the difference in mains cycle speed can also result from positioning the display near power supply sources such as those found in printers and computers. It's insidious, because the motion can produce tiredness and feelings of

nausea without necessarily being visibly obvious unless you know what you're looking for.

Screen definition

This can be specified in one of three ways:

Bandwidth – the video amplifier can only amplify signals up to a certain frequency, called the cut-off frequency or bandwidth. The higher the bandwith, the more rapidly the video signal can change, and the sharper the image on the screen. A low bandwith means that the video signal passed to the tube can only change relatively slowly, so that the picture looks blurred. For most common micros you should aim for a bandwith of 16-18 MHz. Anything much less than this is not going to give good definition.

Rise time – this is another way of expressing the bandwidth of the monitor, and again defines how fast the signal can change. A good rise time to aim for is 20 nanoseconds, that is twenty thousand millionths ($20*00310_{-9}$) of a second. This sounds fast, but the logic chips inside your computer may have rise times of less than five nanoseconds.

Resolution – yet another way of specifying the definition is by reference to the number of dots which can be displayed in the vertical and horizontal, directions. The monitor must be capable of displaying at least the number of dots which your computer can draw in high-resolution graphics mode.

The dedicated terminal

The majority of terminals (or VDUs) are of the 'glass typewriter' variety, with a screen positioned roughly where the paper used to appear out of the platen, and a keyboard (*qv*) with the familiar QWERTY arrangement supplemented by keys engraved with names like CTRL, DEL, ESC and ALT. Many, particularly at the lower end of the market, are still sold as a single piece of furniture with the keyboard and screen all in one unit. A fixed keyboard VDU has the advantage of being cheaper and easier to move, but tends to bring the screen far too near to the user. If you put the screen in the best position to avoid reflections or cricks in the neck, the odds are that the keyboard will be in the wrong place, and vice versa.

A badly designed keyboard only results in typing mistakes, but a poor display screen can cause serious eye strain. In addition to the short-comings discussed under TV sets and monitors above, some poorly designed VDUs display small characters with the 'm's so crushed together that you can't tell them from 'w's without close scrutiny. Like monitors, VDU's don't suffer from the interlacing judder of TV sets, but occasionally exhibit their own brand of instability: a slow seasick-making 'swimming' of the characters.

Readability of all types of screens depends on how well the screen surface is able to discriminate between the display and external light reflections. The plain, untreated screen surfaces found on cheaper monitors and VDU's, which show a dull, grey colour, cope very badly with this problem.

A simple solution is the black mesh, or **'fish-net' filter**, which works on the principle that external light coming off the screen surface has to make two trips through the filter, while screen information only makes one. The system is remarkably effective in increasing the contrast, appearing to turn the background colour from grey to dark grey or black.

Psychologically this is a great improvement, although laboratory studies confirm what you might expect – a slight degradation of the image. A more practical problem is that dust is attracted to the grid by the static electricity which the electron bombardment produces on the screen surface.

Better quality screens have a lightly etched anti-reflection surface which helps to cut down the glare. Even so, the screen's relation to the ordinary room lighting has to be carefully considered. The 'fishnet' contrast enhancement filter is excellent for reducing the general wash of light spilling across the screen at narrow angles to the surface, but does very little to get rid of reflections of light sources more directly in front of the screen. This is why, for example, it is usually impossible to operate a VDU with your back to the window.

Reflections of this kind are known as 'specular', because the image of the light source is reflected as in a mirror. To defeat them you need what is known technically as a **'circular polarising contrast enhancement'** filter. Unfortunately these are not cheap, and their surface is bloomed like an expensive camera lens and has to be treated with respect.

Circular polarising filters give a well contrasted, reasonably reflection-free presentation of your data in broad daylight. Specular reflections are still visible, but subdued to a deep purple colour which should be quite easy to ignore. The screen is a sandwich of two polarising filters, one a conventional sunglass type, the other a circular polariser twisting the light plane through 45 degrees. External light is therefore twisted twice, through a total of 90 degrees, only to find itself blocked by the sunglass layer on the way out. The practical result is that placing of the VDU becomes more or less independent of the general illumination of the workplace.

The 64 has three outputs for TVs or monitors. A phono socket carries the modulated UHF signal for a TV, and a 5-pin DIN socket carries the video and audio signals. Pin 4 is the composite video output for a colour monitor which has this type of input (not RGB monitors). Pin 1 is a luminance signal – either it or pin 4 can be used for monochrome monitors. Pin 3 is the audio output which can go to some monitors or to a separate amplifier.

MS-DOS: A Standard For 16-Bits

The entry on BASIC (*qv*) tells the story of the rise of the house of Microsoft, built up by Bill Gates and Paul Allen around MBASIC. As well as developing high level languages for micros, including COBOL and FORTRAN, Microsoft became very familiar with CP/M (*qv*) as a result of implementing it on a wide range of hardware on behalf of many manufacturers. Bill Gates claims that Microsoft has implemented more CP/M systems than anybody else, including the manufacturers, DRI. It was this subsidiary side of the company which suddenly became dominant when IBM came to them in the late seventies asking for an operating system for the new PC.

According to Gary Kildall, the founder of Digital Research, IBM approached Bill Gates because they knew so little about the world of microcomputing that they thought CP/M came from him. At that time a launch of CP/M-86 was imminent, so Gates sent IBM off to talk to Kildall. But (as Microsoft tell the story) Gary Kildall was out flying his plane on the day the gentlemen from IBM arrived. Disgruntled they returned to Bill Gates and said the contract was his if he could make the deadline.

The truth is probably that Gary Kildall was nervous of the legal difficulties to be cleared up before he could talk to IBM. IBM were insisting that a waiver be signed releasing them from any risk of litigation arising out of the discussions. If an idea they discussed were to turn up later in an IBM product, they didn't want an expensive legal squabble about who thought of it first. That was the intent of the waiver, but its effect was far wider, and Kildall's lawyers had warned him that technically speaking IBM might walk away from the meeting virtually owning CP/M.

Bill Gates had less to lose, and when he subsequently talked to IBM the result was an agreement to produce a 16-bit operating system for the IBM PC in an uncomfortably short time. But Bill Gates had one card up his sleeve – he knew of a company in his home town of Seattle which made 16-bit processor boards and which was so tired of waiting for DRI's CP/M-86 that they had written their own version, closely modelled on CP/M-80.

So it was that the original authors of 'the 16-bit CP/M' were not Digital Research, nor even Microsoft, but a company called Seattle Computer Products. Their operating system was always intended as a stop-gap measure, and as such it was exactly what Bill Gates needed while he geared his team up to meet his sparkling new contract with IBM.

MSDOS today

That release, version 1.25, was subsequently replaced by version 2, which developed in a direction owing much to UNIX (*qv*). The most immediately noticeable advance over CP/M is the hierarchical directory which treats directories as files capable of including other directories – *ad infinitum*. A less noticeable difference is the 'byte granularity' of the MSDOS filing system. Instead of blocking files onto the disk in allocation units of, typically, 2k (although it can be as much as 16k on some Winchester systems) – an approach which means that even a small file can end up occupying a sizeable chunk of the disk – MSDOS can be implemented to write no more than the exact number of bytes needed for the job.

The structure of MSDOS

The internal redesign which made this possible was the way the BIOS and I/O handling was changed to work through 'floating' device drivers which can be thought of as separate from the operating system. A device driver is a software routine acting as a go-between in the transactions between the operating system and hardware devices such as printers, consoles and disk drives. The wildly varying and sometimes awkward requirements of these peripherals are translated by device drivers into formal, orderly requests which the operating system can understand. Since most device drivers are bi-directional, the translation process also works the other way round.

As with the earlier version (and with CP/M), the operating system proper is isolated from the hardware by means of a chunk of hardware-dependent code, the BIOS (BASIC Input/Output System), some 15K of assembler which has to be supplied by whoever implements the system. Traditionally the device drivers are embedded in this code like the cogs and wheels in a clock. But BIOS 2.0 is quite different: the jump table which was the traditional CP/M-style entry point is now replaced by a linked list of devices that can be reconfigured at will, even (in simple cases) by the user. On booting up, MSDOS 2.0 reads a text file called `CONFIG.SYS`, which defines the drivers to be loaded into RAM. Initialisation code is loaded, executed and then overwritten, a useful technique for conserving memory by making sure that routines needed only at boot-time do not outstay their welcome.

The MSDOS 2.0 drivers 2 each define two entry points, called the strategy and interrupt routines, another technique borrowed from UNIX. The idea is that processes calling the drivers mustn't be allowed to hog the resources. So the first call from the process registers a request for the hardware (port, drive or whatever), and then immediately relinquishes control. The interrupt routine has the job of dealing fairly with the queued requests and flagging them as completed when they are finished.

This rather technical description of an important operating system concept is perhaps more simply visualised in terms of a queue of people at the delicatessen counter of the local supermarket. In order to deal with them fairly, the proprietors install a ticket system: instead of instantly demanding the assistant's attention, each customer arriving at the counter collects a numbered ticket, the equivalent of calling the strategy routine. By this method the customer is free to continue self-service shopping, and the delicatessen assistant carries on dealing with each of the customers in the queue in turn until an appropriate announcement brings them together.

In MSDOS 2.0 however the idea is there only as a shadow (perhaps that should be a foreshadow) of the

multi-tasking i/o-handling the technique implies. The interrupt and strategy routines are simply welded together, the one calling the other with no chance for other processes to break in.

In the UNIX tradition, two types of device are supported: character and block. Character devices such as consoles, modems and printers will pass their data into and out of the packet one byte at a time. Block devices, notably disk drives, move their data in chunks that are allowed to vary in size so that exact numbers of bytes can be accommodated.

For system implementors the ease with which drivers can be added is the best thing about MSDOS. For users, the most important thing about MSDOS is the wealth of available 16-bit software opened up by the runaway sales of the IBM PC.

In terms of user friendliness, the Microsoft product continues to score over its DRI rivals through its extended facilities for recalling and editing the command line, so that long but incorrectly entered commands don't have to be retyped in full.

Upgrades to MSDOS 2 include Microsoft Windows, a resident system process bolted-on to the operating system allowing you to run more than one program at a time. Each program can be allocated a part of the screen to itself, and data can be passed between programs. As it stands, Microsoft's Windows isn't a full multi-tasking system after the style of Concurrent CP/M, and there is no mechanism for juggling RAM or backing store (as in UNIX) so only a limited number of small programs can be accommodated.

MSX: Towards The Ideal Home

The homes of lucky consumers in the Western world are rapidly filling up with high-tech products such as hi-fi's, television sets, video cassette recorders and microcomputers. They all talk to us busily, and we talk to them – imagine what might be done if they could all talk to each other. . .

Compatibility has been an important issue in the information technology industry for some time now, and it is beginning to look as if this ideal might soon be extended to the world of consumer products other than computers. The heart of a household control system would be a low-cost, high performance and utterly compatible microprocessor, and that's where MSX fits in.

Machine and language

MSX is a BASIC-based operating system standard with a software and hardware specification designed to achieve compatibility among low-cost 8-bit microprocessors in the home. The MSX project was initiated in Japan by NEC and Matsushita, and the specification was drawn up by ASCII Microsoft. More than a dozen Japanese micro companies actively supported the system from its inception, launching MSX-based products into the European market in the hope of inducing European manufacturers to follow suit.

The MSX project defines the hardware and software which forms the base system for a low-end, 8-bit micro. Zilog's Z80A is the CPU of the machine, and it communicates with the outside world via a common set of I/O chips. The language used is an enhanced BASIC supplied by Microsoft on a 32k ROM, and has just about all the advanced features of GWBASIC (*see* BASIC), supporting multi-voice music and hi-res graphics. It also includes a host of new features designed to simplify programming support for external devices such as video disk control.

MSX holds out the promise of a software standard across a huge range of home micros – in competition with Personal CP/M (*qv*). Compatibility will extend from BASIC applications packages right down to assembly language programs, and there should be no need to rewrite code to accommodate the vast number of BASIC dialects offered on home micros. The corresponding drop in the cost of software production may well result in a cheaper and larger software base for home micro users.

MSX programs are distributed in two forms: ROM cartridge and cassette. The systems cassette interface and ROM cartridge formats are also specified for MSX, so that application and media problems are both banished at a stroke. ROM-based software from one manufacturer's machine can be plugged into another MSX offering from a different source, with the guarantee that it will run first time.

Games, Graphics and Music

As a games machine the system offers very impressive sound and graphics features, and a support for joysticks, touch-pads and games paddles. MSX-BASIC provides a large number of commands designed specifically for all of these devices and their attendant push buttons. The hi-resolution screen is a matrix of 256 by 192 pixels, with a total of 16 colours to play with. The graphics functions of GWBASIC are carried over into MSX-BASIC, with the *Graphics Macro* language available to draw any desired shape, while standard commands draw the run-of-the-mill ellipses (including circles of course), boxes and lines.

An interesting feature of the graphics is the way in which a picture may be manipulated once on screen. A fast way of streaming the picture data to and from the screen in a bit-wise fashion has been provided by the **GET** and **PUT** statements. **GET** will store the current picture or section of the screen as a binary pattern in a BASIC array. **PUT** will restore an array-stored picture to its original position on the screen. This method is very fast since a single statement is all that is needed to refresh a screen image. This leaves open the possibility of elementary computer animation.

A programmer can specify musical notes and their duration as numbers, or define them as character strings using the *Music Macro* language. All the features of the three-voice AY-3-8910 sound chip are exploited, while a music buffer speeds up execution time quite considerably.

CPU:	Z-80A Soft Compatible (Clock 3.579545 MHz)
Memory:	ROM 32Kbytes (Microsoft's M-BASIC) RAM more than 8Kbytes. Both ROM & RAM are extendable.
VDP:	TMS 9918A Compatible
PSG:	AY-3-8910 Compatible
PPI:	i6255 Compatible.
Display:	Text screen:256 x 192 pixels 16 Colours
CMT:	FSK System, 1200/2400BPS.
Sound:	8 octaves, Triple chord output.
Keyboard:	ASCII, Hiragana, katakana, Graphics, Jis Keyboard, Japanese alphabet arrangement standard.
ROM Cartridge:	Has an extended bus slot.
I/O Bus:	50 pin cartridge bus.
Printer:	8 bit parallel I/F.
Joystick:	one or two joystick adaptable.
Kanji function:	Optional.

The MSX standard

MSX and home electronics

As most of the Japanese manufacturers involved in the MSX project number microcomputing as only one of their many consumer electronics concerns, it is very likely that MSX will find its way into video recorders and other consumer devices. Should home controller systems become established, it is likely that MSX will become one of the dominant microsystems in this field.

A system such as video recorder will not necessarily have all the hardware elements present to be called an MSX computer, but future models should be able to accept computer signals from a standard machine. There is perhaps scope for an entire implementation of such a microsystem, especially given that one of the future proposals for MSX is that it will support voice control.

Too good to be true?

Critics of the system have suggested that manufacturers who adopt MSX will be obliged to produce a stream of carbon copy machines. The MSX lobby refute this: the design and layout of MSX products will be left up to the manufacturers and there is plenty of leeway for non-standard additions to the basic machine. The particular support chips for video processing, parallel I/O and the like remain undefined although a standard for their function has been defined.

The commercial practicality of the standard over the long term remains to be seen. Compatibility is a potential boon for new manufacturers (and to the consumers), but it is often traditionally resisted by established manufacturers reluctant to jettison their investment in existing systems. MSX may also be rejected by manufacturers interested in pushing forward into the new technology, as its standards are very conservative.

Music: Making Software Sing

Of all the creative arts, music is probably the most naturally suited to computer work. Music can be seen as revolving around mathematical structures, and within limits this can be expressed as a series of rules to be applied to melodies, harmonies, rhythm and timing. Sound can be transmitted in analogue or digital form as electrical pulses. It can be stored in the memory as binary ones and zeroes and can be reproduced very exactly.

Luckily you don't need to buy the expensive equipment used by the professional musicians involved in this new approach to music. Your existing micro is enough to start you off, and if you don't like the sound that comes out of it, or if you want to experiment with harmonies and rhythms, you can buy add-on boards, and with some micros you can even plug in additional piano-style keyboards. People who have never played an instrument can turn their micro into a programmed music box.

One of the complaints against computer music is that it produces a sameness of sound and takes away the individual's skill and interpretation. Computers can certainly make people lazy about what they produce, and for those with no musical imagination, computerisation will build walls rather than bridges. But the power of the micro undoubtedly gives real musicians an invaluable tool – and puts music into the hands all who need it.

The Rock musician Peter Gabriel, one of the foremost exponents of computer-produced music, has described how the musical computer removes the professional musician's exclusive control over the realisation of ideas and passes it on to the layman. People with a feel for music but without formal training can now organise music for themselves, possibly calling in specially-trained musicians later when the outline of their ideas have gelled. The possibilities of the Fairlight are exciting. If Peter Gabriel wants to work with a particular musician in a hurry, he has only to pick up the the phone, and use a modem (*qv*) to transfer the musical outline so that it can be added to and sent back in the same way.

Micro Music at Home

Most home micros are equipped with a sound generator, usually with several channels. A typical configuration would have three channels to produce fairly pure electronic tones as well as one for white noise. It isn't too difficult to drive these from BASIC, setting up the QWERTY rows as a musical keyboard – the very bottom row, Z X C V B N M can represent the notes C D E F G A , with the keys on the row above being used as 'black notes'. Provided you can find your way around this rather odd combination of typewriter and clavier, you can now pick out tunes. But this is only the beginning of what a computer can do in the world of music.

Suppose instead that you want to play somebody else's music. Take a score, perhaps with two voices if you want to try out a little harmony, and with the keyboard restored to its normal functions start tapping in the notes, possibly as **DATA**statements in a simple BASIC program. After a hundred or so such entries, with the user memory filling up nicely, you can run the program, sit back, and listen. Satisfying perhaps, but if you have to repeat the exercise too often you'll agree it's a lot of hard work.

Software synthesisers – For many micros you can buy a software emulated music synthesiser, where instead of the hardware controls and switches found on commercial synthesisers, a graphics representation of current settings appears on the screen, adjustable from the keyboard.

Professional synthesisers usually incorporate 'sequencers' – circuits which remember a keyboard sequence and can play it back to you as if it were tape recorded, enabling you to add a new layer of notes to build up a fully orchestrated score. A limited facility along the same lines is available on most micro emulations, sometimes with the luxury of being able to alter the speed and tone of a track – or even of editing individual notes stored in the sequencer. With help like this it is possible for even the most modest musician to turn out a very creditable performance.

Computers can be useful tools for teaching the fundamentals of music, and packaged programs can take most of the sweat out of producing ordered sounds from a machine. But, given the quality of sound produced by most microcomputers, for serious use as an electronic instrument a better bet would be to buy a cheap hand-held synthesiser, which will be at least half the price of an ordinary home micro. That will give you a small clavier-style keyboard, editing functions (without programming), user-defined sounds, a 100-note memory, and a choice of accompanying rhythms with a range of speeds.

The micro as Mahler – the computer really scores over the dedicated instrument when it comes to acting as a composer. An 8-bit machine with a few kilobytes of RAM may not, on the face of it, show a lot of promise as a Mahler, a Mozart or a McCartney. But if you are willing to lay aside a few preconceptions about what music ought to be, you can embark on some fascinating experiments.

Suppose, leaving other parameters aside, we just want a succession of notes, which is after all the basis of melody. A programmer could go through the procedure of defining every note, repeating the value of duration and loudness each time. But there is a shorter route, if we introduce the idea of a string of pitches as a dimensional array.

In the first example a few lines of programming has produced 10 notes. Clearly, the same lines could produce any number of notes (given the limits of user memory). But dimensional arrays in BASIC are unwieldy. Try, for instance, creating a string that adds variable duration, and therefore rhythm, to the program.

Randomness (*qv*) can also be quite tuneful (up to a point):

The second example shows how a few lines of program can instruct the computer to ramble on in its own tuneful way for as long as you can stand it. The 'music' is a random distribution of notes between the limits set for pitch. There is no order to the notes as they appear, and you would have to stretch your imagination a little to call it music.

You will probably find the music produced by the RND statement a little trite, but it does introduce a technique used by many professional composers nowadays. Even in its crudest form, it is an example of *stochastic composition*. Stochastic techniques are a statistician's device, used to define, or measure the limits for an event (in this case the playing of a musical note) to predict with some degree of certainty what the event will be.

In the preceding example the only certainty is that the next note will have a pitch value between the limits 60 and 200. But, by tightening the limits for each musical event, a programmer can control a computer's composing technique much ore closely.

In this way computers have been programmed to 'create' compositions '*in the style of...*'. That is, by analysing the scores of a composer, a programmer can supply the computer with rules for repropducing the statistical chances of a certain rhythm arising in the composer's work, or of one particular note following another. These composing programs produce music with a certain amount of novelty value in short stretches, but longer pieces can be tedious because this approach never takes sufficient account of the unpredictable (and appealing) side of a composer's work.

Unfortunately no home computer has enough memory even for a crude version of such a program. The closest we can get is to do something like replacing the RND(X) function in line 20 of the last example with the term RND(RND(X)). This very simplistic stochastic device adds bias to the randomness to weight the chances of a note being played toward the bottom end of the range. The value 60 will appear more often than 70, which in turn will appear more often than 80, and so on.

Try the program and you'll agree that the computer is hardly threatening any professional musician's livelihood, but it is beginning to develop a sense of order, and seems almost to be wandering about in search of a tune!

But for more interesting variations on the RND(X) function, a computer will need a grounding in the mathematics of music. The basic components of simple harmony involve frequencies bearing simple mathematical relationships to the 'tonic' note of the familiar seven-note (*do, re, mi* ...) scale. You can gain a better idea of how sounding 'right' to the Western musical ear actually has an elegant basis in maths by digging out a text on acoustics from the library. Helmholtz's primer, a century or so old, will give you all the basic formulae you will need.

While a computer can easily get to grips with the simple rules of harmony, there is no clear explanation in mathematics or psychology as to why a particular melody should sound attractive to the ear. A human might have the benefit of several centuries of melodic experiences, whereas a computer can act only within the relatively few guidelines punched into its keyboard by the programmer. In spite of all this and even though music is riddled with conventional and hidden rules, it seems at the moment to be a very interesting line of research for computer scientists developing artificial intelligence in machines.

Even our simple program for generating random notes can serve as a limited basis for some experiments in artificial intelligence. Try to organise some shape into computer 'melodies' with a little imaginative use of random functions. You may never produce a masterpiece – on the other hand, you might find that a machine can give you a more interesting insight into music than you'll ever get from practising scales.

One of the most persistent problems with getting micros to make music has been finding a way of matching the computer's no-nonsense, black or white, digital way of working with sounds, which have a highly analogue quality. However, various ways of programming sounds have been developed, pioneered by Robert Moog, the inventor of the music synthesiser.

ADSR

The most common technique for programming sound is called ADSR, standing for Attack, Decay, Sustain and Release, four variables that document the history of a sound from start to finish.

Attack charts the *rise in amplitude* (volume) of the sound from silence to its initial maximum volume. There are two parameters, the time taken for the attack, and its level when it reaches its peak.

Decay is the *fall in amplitude* of the note once the attack is over, it is measured in terms of a single parameter, the decay rate. Decay brings the sound down to the *sustain* level.

Sustain is the time the sound is *kept at the same volume* after the decay, and before the next stage, the *release*.

Release is the *time taken for the note to come to rest*. In normal musical sounds we can assume the 'release level' to be silence, but ADSR technique doesn't rely on this and notes can be created which will continue until the next note starts.

These parameters make up a sound's shape, or 'envelope'. However, ADSR is not capable of producing a huge variety of sound. It doesn't take into account changes in the pitch of a note when it is played, and it assumes that all the amplitude changes are linear.

ADSR is based on amplitude modulation (*see* Baud), but sound synthesis based on frequency modulation is becoming popular, principally because Japanese companies have made the cost of producing the appropriate chips comparatively cheap. Frequency modulation is a more complicated technique but has the advantage of producing sounds with a tonal richness which amplitude modulation is unable to provide.

Pop goes the micro

Micros mean money in popular music: three-quarters of the hits in the top ten charts use some form of computer music. Understandably, the Musicians Union is very concerned about the effects of computers on their members' jobs, particularly the new computerised drums used on many records in place of professional session drummers.

There were similar rumblings from professional musicians when the gramophone record was introduced, but the micro is having an even more fundamental effect on the way musicians perform and record their music. Micro-based instruments are readily available now and many can reproduce virtually any sound known to man, as well as many others undreamed of. Keyboards can sound like saxophones, violins, dogs or motorbikes, and digital circuitry can turn a human voice into a hyena's cry. The role of the micro even extends to controlling entire performances; timing fades, mixing the tracks and synchronising the lighting.

The music profession draws on computer hardware in four ways:

By adapting a general purpose micro as a stand alone instrument or as a controller for other instruments.

By upgrading modern electronic organs and synthesisers so that they have some processing power.

By producing a computer dedicated to music.

By creating 'hybrid' equipment which is partly a computer and partly a conventional musical instrument.

Hybrid instruments look slightly strange to the untutored eye. One of the most successful of these, costing several thousand pounds, is the Emulator, a piano style keyboard with a disk drive at one end.

The manufacturers, E-mu Systems in California, supply a number of different disks, each of which has two sounds. When you've booted up the disk, the four octave keyboard is split into two, giving you two octaves for one sound and two for the other. For instance, your left hand could play a trumpet on the keyboard while your right hands scrapes a violin.

The Emu isn't limited to pre-programmed disks – it can pick up any sound using a microphone and store it in memory. You could have one half of the keyboard playing a bird song while the other half produces the the sound of breaking glass. You can also feed in notes from a record or tape; recording just one note from a flute enables you to reproduce it at any pitch in the keyboard's range.

Sampling rate – The quality of sound (reproduced digitally) is dependent on the *rate* at which the computer samples the sound, and the *bandwidth* capacity of the computer. A note of the same pitch can sound very different on different instruments, because a real note is never a pure smooth wave, but a very complex jerky shape, shrouded in the 'overtones' which give the note its quality. That's why an oboe sounds reedy and a harpsichord zingy – distinctly different even when they're both playing middle C.

Because the wave varies in shape so much, for a computer to capture the quality properly it must analyse the sound many thousands of times a second. On the best music computers the note will be sampled about 50,000 times a second. The more frequently the sound is sampled, the better the quality of reproduction.

The dedicated music computer – The first and best known of these is the Fairlight, which first appeared in Australia in 1980 and it is now used by musicians all over the world. These dedicated music computers are much more sophisticated and powerful than a hybrid instrument like the Emulator, and consequently cost something like five times as much. But there's is a great deal more to them: the Fairlight has one or two musical keyboards, a central processing unit with two floppy disk drives, a computer keyboard and a video monitor with a lightpen.

As with the Emulator you can input sounds from a microphone or floppy disk and play them as keyboard notes, but you can also program sounds through the computer keyboard. The sound is represented on the monitor as waves, which can be altered and added to by the lightpen. You can even draw your own wave and then listen to it. This ability to change any sound into another by way of the screen and lightpen gives enormous scope for editing, mixing, inverting and merging sounds. The Fairlight also has its own computer language, the Music Composition Language (MCL), which allows you to set up notes, volumes, time values and special effects, and build up complex rhythms and harmonies.

The Fairlight is an extremely sophisticated musical tool, and it will be some time before facilities of this kind come within the price range of the average micro owner. As the cost of hardware falls the Fairlight will be improved with add-on boards and software, so the price is likely to remain fairly stable.

Silicon drums – But other instruments used by professionals are rapidly coming down in price. The best example of this is in the field of computer percussion: fairly reasonable electronic drums are down in price to two figures. These won't give you the capabilities of the more sophisticated equipment such as the Linn-Drum, but will let you mix several different sounds and beats.

Although the Linn-Drum costs almost as much as a small car it is very popular in Britain and is used on a number of hit records. Instead of the synthesised sounds of the cheaper computer drums, the Linn-Drum uses digital recordings of real percussion instruments. You can produce fifteen sounds, ranging from cowbells and hand-claps, to congas and base drums, and up to forty-nine different rhythm patterns can be stored.

Although it is programmable, the Linn-Drum bears no resemblance to a normal micro and intentionally looks more like a mixer to make it easier for use by drummers with no experience in computing. Drummers feed in their own style of rhythm patterns by hitting the keys.

The MIDI interface

In an uncharacteristic display of conformity, most of the computer music manufacturers have adopted a standard interface for connecting digital musical equipment. It's called MIDI, which stands for Musical Intrument Digital Interface.

MIDI is a fast serial communications link which can connect, say, a keyboard to a computer via standard 5-pin DIN plugs. Most professional electronic musical equipment and some micros have MIDI interfaces built in,

so that micros can be used to program sounds and store sequences of notes for editing.

Musical walls or bridges?

One of the complaints against computer music is that it produces a sameness of sound and takes away the individual's skill and interpretation. Computers can certainly make people lazy about what they produce and, for those with no musical imagination, computerisation will build walls rather than bridges. But the power of the micro undoubtedly gives real musicians an invaluable tool – and puts music into the hands of all who need it.

The rock musician Peter Gabriel, one of the foremost exponents of computer-produced music, has described how the music computer removes the professional musician's exclusive control over the realisation of ideas and passes it on to the layman. People with a feel for music but without the formal training can now organise music for themselves, possibly calling in specially-trained musicians later when the outline of their idea has gelled. The possibilities of the Fairlight are exciting. If Peter Gabriel wants to work with a particular musician in a hurry, he has only to pick up the phone, use a modem (*qv*) to transfer the musical outline so that it can be added to and sent back in the same way.

The 64's powerful sound generator chip SID – Sound Interface Device – almost forms a sound synthesiser on its own. Its facilities include three completely independent variable frequency oscillators, each capable of four waveforms (triangle, sawtooth, pulse and noise), three amplitude modulators, three envelope generators and a programmable filter offering high and low pass, bandpass and bandstop modes. The SID chip starts at location 54272. This address and the six addresses immediately above form the seven control registers of voice 1. Voices 2 and 3 similarly occupy the fourteen next highest addresses, and their operation is almost identical, so we'll just concern ourselves here with voice 1.

To play a note you need to set the registers. If you begin with the statement LET S = 54272 to point to the start of SID, the registers now become S, S+1, S+2 and so on. S and S+1 together control the frequency of the note, S+4 sets the waveform, S+5 and S+6 the envelope (attack, decay and release rates and sustain level). Lastly, S+24 sets the volume of the note. This location controls the volume for all three voices, and is divided into two nibbles (4-bit numbers). The lower one (0 to 15) is POKEd to get a volume (0 for low, 15 for high).

Here is a little routine that plays a completely random 'tune', using all these registers. You could adapt it to add the other two voices, the filter and modulation effects.

```
100 S=54272
110 FOR L=S TO S+24 : POKEL,0
: NEXT
120 DIM H(12),L(12)
130 FOR I=1 TO 12 : READ H(I),
L(I) : NEXT
1000 POKE S+5,256 * RND(0) : P
OKE S+6,256 * RND(0)
1010 POKE S+24,8 + 8 * RND (0)
1020 I = 13 * RND (0)
1040 POKE S+1,H(I) : POKE S,L(
I)
1050 POKE S+4,33
1060 FOR I = 1 TO 50*2^(5 * RN
D(0)) : NEXT
1070 POKE S+4,32 : FOR T=1 TO
50 : NEXT
1080 GOTO 1000
60000 DATA 4,48,4,112,4,180,4,
251
60010 DATA 5,71,5,152,5,237,6,
71
60020 DATA 6,167,7,12,7,119,7,
233
```

Numbers: The Language Of Quantity

The power of numbers lies in the way we can abstract them and manipulate them independently of the real world. But counting was not always like this. When man first learnt to count, numbers bore only a one-to-one relationship to the quantities they represented. Counting was comparing: fingers were held up against the sheep in a flock, and if they matched, then that was the count of the sheep. Even when traders began to rely on devices like the abacus, the relationship between numbers and the world they represented was still largely physical in its basis.

It wasn't until the 8th century A.D. that the idea of zero as we understand it today rescued the business of counting from its dependence on a clear physical relationship. The result of this profound invention, attributed to the Hindus and passed on to Europe by the Arab traders, was an enormous step up the ladder of abstraction, as numbers came to be manipulated positionally, that is to say that the position – the column in which a digit appeared – came to dictate the multiplier to be applied to arrive at its actual value.

Numbers were ultimately severed from the real world by the invention of algebra, freeing mathematicians to think of numeric manipulation as something altogether separate from the constraints of the real world – and even of numbers themselves. The results began to show fruit in the 17th century with the invention of mechanical devices that could make complex algebraic calculations, culminating a century later in Babbage's invention of his famous 'analytical engine', often regarded as the archetypal computer.

Numbers and computers

Computers are a sort of electronic abacus – although simpler in some respects than even this ancient device. The abacus is able to deal with an elementary representation of our decimal system, whereas the computer can understand only two 'states': whether a particular switch is 'on' or 'off'. These switches can be treated as digits or fingers, as in the decimal system we are familiar with, but they carry less information. The digit '9' in the decimal system carries enough information to differentiate it from the nine other digits 0 - 8, whereas the binary digit 'switch-on' (usually represented as '1') is only differentiated from one other digit, 'switch-off', or '0'.

Because of the meagre amount of data carried in the **BI**nary digi**T**, early computer scientists amused themselves by contracting the phrase to the portmanteau word 'BIT'. The contraction is so useful that it has stuck, and given rise to a vocabulary that carries the same idea further in words like 'BYTE' (a group of eight *BITS*) and 'nibble' or 'nybble' (a smaller byte of just four bits).

Bits can be quite useful as things in themselves. Kings and Queens have long used a flag which is either flying (1) or not flying (0) to tell passers by whether they are in for tea. Bits are often used by computer programmers as 'flags' in a sense not a million miles from this, although most of the information we need to store in a computer is more complex than just 'in' or 'out'. As a result of this there arose a tendency to group the individual bits together to make **binary numbers**.

Number bases

In the positional decimal system the number 352 actually means three hundreds plus five tens and two units. This can also be expressed as:

$$352 = 3 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$$

The maths in this example may seem like an obscure way of expressing a simple number, but in fact it is laying bare the bones of the positional system to expose something which we take for granted in daily life, but will now have to examine with some care if we are to understand how the binary system works. The thing to realise is that all positional numbering systems work in essentially the same way but using different 'bases'.

A positional numbering system can be constructed around any base (except one), with the most commonly used in computing being:

Binary (base **2**).
Octal (base **8**).
Decimal (base **10**).
Hexadecimal (base **16**).

All these different bases are designed to make life easier for the expert, but they can certainly add to the confusion of the beginner. The principle is not difficult to grasp once we overcome our inbuilt prejudice in favour of numbers to the base 10 (based on our ten fingers). Nothing in the nature of the world dictates that ten is the point at which we must stop inventing new symbols for the next number, and begin combining the ones we already have – which is really all there is to a number base. Numbers to the base 2 have only two symbols to play with, base 10 has 10 of them, base 16 has 16, and so on.

Once you understand this it is relatively easy to derive the value of a number by means of a simple calculation taking account of the *digits* used in forming the number, their *position* in the number, and the *base* in use. For example, the figure 153 (not necessarily to the base 10) can be analysed as follows: *take three lots of whatever the base is raised to the power of zero, add five lots of the same base raised to the power of 1, then add one lot of the base raised to the power of 2.*

Ten to the power of 2 is of course 100, ten to the power of 1 is 10. The only slightly tricky one is 10 to the 0 (or for that matter *any* finite number to the 0). If you multiply a number by itself zero times (which is what 'to the 0' means) the mathematicians have decided you always get 1. It works out very neatly, so let's not argue with them.

In rather more formal terms you can say that the *n*th digit of a number represents that digit times the base to the power of the digit's position minus 1. This algorithm is used in the example as the basis of a practical base-conversion program.

Binary and hex

Counting in binary is just like counting in decimal (base 10), except that instead of using units, tens, hundreds and so on, binary counts in units, twos, fours and so on. With one bit, we can represent two things – 0 and 1. With two bits, there are four different patterns, 00, 01 10 and 11. With three bits, there are eight different patterns. As a rule, with *n* bits, there are 2^n (that's $2\times2\times2...n$ times) different patterns.

Decimal	Hex	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

216 decimal = 128 + 64 + 16 + 8 + 1 = **11011001 binary**

128	64	32	16	8	4	2	1
1	1	0	1	1	0	0	1

11011001 binary = 1101 1001 = D9 hex
D g

3F hex = 0011/1111 = 00111111 binary =

(0 x 128) + (0 x 64) + (1 x 32) + (1 x 16) + (1 x 8) +

(1 x 4) + (1 x 2) + (1 x 1) = **63 decimal**

Fig. 1. Sample conversions from decimal to hex and binary

The binary world inhabited by computers is awkward for human beings. Because each bit carries so little data, even quite small numbers which might be represented in decimal with three digits (eg. 772) require a long string of bits in binary (1100000100). It's difficult to remember, to write out, to type in and even to display on a computer screen. One obvious alternative is to write numbers in our usual decimal system and to let the computer convert it to binary when it is ready to use it, but it turns out that converting from one to the other is a fiddly job. So when programmers and computers want to communicate at

```
1 REM NUMBER BASE CONVERSION
2 REM CONVERTING ANY BASE BETW
EEN
3 REM BINARY (2) AND HEX (16)
10 DIM A$(15)
20 FOR I=0 TO 15
30 READ A$(I)
40 NEXT I
100 PRINT "ENTER NUMBER TO CON
VERT:"
110 PRINT "QUIT BY TYPING 0"
120 PRINT
130 INPUT "NUMBER";NM$
140 IF NM$="0"  THEN END
150 PRINT
160 PRINT "ENTER BASE OF NUMBE
R-"
170 INPUT "SOURCE BASE";BS
180 PRINT
190 PRINT "ENTER BASE OF CONVE
RTED NUMBER:"
200 INPUT "CONVERTED BASE";BC
205 IF BC<2 OR BS > 16 GOTO 20
0
210 TT=0
220 FOR I=0 TO LEN(NM$)-1
230 RE=ASC(MID$(NM$,LEN(NM$)-I
,1))
240 IF RE<64 THEN RE=RE-48
250 IF RE>64 THEN RE=RE-55
260 IF RE>=BS  THEN GOTO 100
270 IF RE<0 THEN GOTO 100
280 TT=TT+((BS^I)*RE)
290 NEXT I
300 REM CONVERT DECIMAL-BASE2
310 PRINT
320 PRINT "CONVERTED NUMBER:";
330 FG=0
340 FOR I=20 TO 0 STEP -1
350 N1=(TT/(BC^I))+0.00001
360 IF INT(N1)<>0 THEN FG=1
370 IF FG=1 THEN PRINT A$(INT(
N1));
380 TT=TT-((INT(N1))*(BC^I))
390 NEXT I
400 PRINT
410 INPUT "READY";YY$
420 GOTO 100
430 DATA 0,1,2,3,4,5,6,7,8,9,A
,B,C,D,E,F
```

Fig. 2. Number base conversion program

the low level of pure numbers, a halfway house between the two number bases is very useful.

That is why the convenient **hexadecimal notation** has become so popular. 'Hex' uses base 16, so it counts in units, sixteens, two-hundred-and-fifty-sixes and so on. The convention is to supplement the ten decimal digits **0** to **9** by borrowing the letters **A** to **F** from the alphabet to represent the extra numbers 10 to 15 (thus hex **F** is the same as the decimal **15**). This makes hex look a little odd at first, but it is surprisingly easy to get used to.

Hex is compact to write and easy to convert to and from binary. Each nibble in a binary number corresponds directly to a particular hex digit since the four bits of a nibble conveniently cover the range between 0 (0000) and 15 (1111).

The only major trap lies in forgetting which base is being used at any given moment. Conventions vary confusingly about how to signify that a string of digits is in hex, and different systems use **0x**, **$**, **&**, **&H** preceding the number, or **H** trailing it. Another traditional notation uses *subscripts* to give 112_{10}, 160_{8}, 70_{16} (but doesn't usually bother to write 1110000_{2} since the ones and zeroes and the length pretty well give it away) – most computers have not yet got the hang of dealing with subscripts even though it is a very simple and elegant way of displaying the information. Figure 1 shows some corresponding decimal, hex and binary values and some sample conversions as a guide to how they all fit together.

Bases in BASIC

The BASIC program shown in Figure 2 will convert any number from any base to any other base. As set out in the example the largest base available is 16, but it could easily be extended.

The program reads in three values: the number to be converted (`NM$`), its base (`BS`) and the base to be used for the output number (`BC`). The number itself is input as a *string* so that the characters A to F will be accepted in bases greater than 10, it is converted by working out the value of each of the individual digits from their position in the source number which is then converted to the new base by repeated division.

Notes: When the number string is being converted to decimal, the characters **0** to **9** are represented by ASCII 48 to 57, so 48 is subtracted from the character to obtain the number. Similarly, 55 is subtracted from ASCII **A** to get decimal 10, and so on for the 'numbers' **B** to **F**.

If the digit from the string lies outside the range **0** to **BS - 1**, then it is illegal.

A small number (0.00001) is added to N1 when dividing by BC to avoid rounding errors.

The variable `FG` is used in leading zero suppression whereby a number such as 00012304 will be printed as 12304.

Rounding errors

The binary numbering system used inside computers sometimes throws up rounding errors when the binary numbers available are not large enough to keep track of the intermediate results in a calculation. It is worth being aware of this since it can happen quite often and easily in calculations using some of the commoner dialects of BASIC.

A mini-BASIC often can't work out a simple sum correctly. Ask it to multiply 0.6 by 100 and the answer you will get is 60.00001. The error also occurs with much larger numbers which yield correspondingly larger errors. For example, 981 divided by 100 and then multiplied by 100 will yield the result 981.0001 in many simple implementations of BASIC.

To understand this error, remember that the computer has to convert your decimal numbers into binary to make its calculations, and then back again into decimal to give you your answer. This conversion is not always straightforward. If you have ever tried to divide one by three, you will have run into a problem: the result is a recurring number which produces a series of threes going on towards infinity. The problem is just the same in binary.

Divide six by 10 in decimal and the answer is straightforwardly 0.6. But now try converting 10 and six into binary, and then repeat the same sum and you are likely to end up with a recurring binary number. There is no way of writing down or storing the infinite series of digits the recurring number requires, so the traditional method is to put down as many digits as you have space for and then round off the last digit you've got. If you are working to five decimal places, the way of representing say, 0.66666... is 0.66667. The last decimal place is rounded *up* because it is larger than five -- the decimal half way house.

A computer usually takes the precaution of storing a number to greater numerical precisions than will be displayed on the screen. Under BASIC on a typical 8-bit system the number will be stored in two parts:

The mantissa – the pattern of digits ignoring decimal places or trailing zeros, and

The exponent – the power of ten by which the mantissa must be multiplied to arrive at the value we are storing.

In a number such as 9×10^{5}, the 9 is the mantissa and 5 the exponent. Typically BASIC will use a 32-bit space to store an integer, in other words there will be room for 32 binary digits before rounding up or down is necessary. Some common BASICs, including those developed by Microsoft for the 8088 processor (used on many 16-bit machines including the Sirius and the IBM PC) have space for only about 21 significant (binary) digits. In decimal terms this is equivalent to seven digit precision, although there is usually provision for greater accuracy using a *double precision* mathematics option working with roughly twice as many bits.

In BASICs like this, because the results printed on the screen are to the same precision as the numbers stored in the memory, stray figures like the 0.00001 will turn up from time to time as a result of recurring binary numbers being rounded up or down.

Operating System: Housekeeping Software

An operating system is a program, a series of commands that tell a computer what to do. But unlike other programs it is very introverted. Rather than make changes to the real world (for example by calculating your monthly accounts and printing letters to the bank manager), its main concern is with organising the hardware it runs on, a process sometimes known as 'housekeeping'.

In the early days of computing huge banks of valves and matrices of delicate wires could only be coaxed into computation by a team of operators attending their every need. Computers started out by being incapable of doing very much for themselves until, with operating systems, they grew up. Suddenly they could find programs for themselves, display results on a screen, detect when the user was typing something in, administer the memory banks, direct data down a line connected to a printer and organise data held on disks. All thanks to the operating system as it administers the system.

An analogy might help. Suppose you owned a warehouse which was used to store supermarket supplies. The operating system is equivalent to the system you would set up to ensure that incoming goods are stored at the right place and in the right environment, that the warehouse space is used efficiently, and that the supermarket lorries have access to the goods they need via a suitable access point.

Using the operating system

In general the domain of an operating system covers thestorage and execution of your programs and data, handling the input and output devices, and managing your files on disk or cassette. When you switch on, the resident operating system wakes up and, on a BASIC machine, immediately connects the BASIC interpreter. It prepares to receive input from the keyboard and to send output to the screen. On systems where there is a choice between different forms of backing store it will also select the appropriate disk or tape filing system. Some operating systems even do a lot more than this.

When you hit, say, the **R** key, you naturally expect the letter **R** to appear on the screen. There is, however, no direct connection between the keyboard and the screen – the simple-seeming action is actually one of the many unseen jobs of the operating system. To understand what is taking place is the secret to understanding, and then to exploiting, your machine operating system.

Striking the key causes an electrical change that puts the ASCII code of the key in the **keyboard buffer**, the section of memory reserved as a temporary storage area for the keyboard's use. The operating system is periodically scanning this buffer to see if anything has been happening to it lately. Technically this is known as **polling** – an alternative approach called **interrupting** alerts the operating system directly a key is hit. Either way, it detects the presence of the input character, picks it up and takes it off to be dealt with.

To use another analogy, suppose you write a letter and post it in a convenient letter box. This is an input buffer for the operating system of the post office. It is not processed immediately, unless you happen to post it at just the right time, which is to say the post office uses a polling system in dealing with your mail. You expect your letter to wait in the pillarbox until the next poll or collection. The postman picks it up and takes it off in his van to the sorting office. Here its address is read so that it can be transmitted by the system to its destination. The postman puts the letter through the letterbox, and it falls onto the intended mat (a buffer for the addressee). It now waits in this buffer until someone passes by, picks it up and opens it. Only now can the letter be said to have been transmitted from sender to receiver, just as a letter is transmitted from the keyboard to the screen.

Harnessing the operating system

The point of this brief description of one aspect of the operating system is to demonstrate that a great deal of unseen activity is going on. There is a way of joining in this activity as a programmer, but to do this you need to inform the system what you really want done, rather than let it do what it has been pre-programmed to do by default.

There are, in essence, three different ways that you can tap the power of the operating system as a programmer:

By direct instructions as described in the manual.
By making CALLs to specific locations.
By PEEKing and POKEing to specific locations.

The first of these is tried and tested, coming straight from the manufacturer, and is efficient and easy to use. The second requires a detailed list of the relevant subroutine addresses. Such details are supplied by the manual or revealed in specialist publications. The third usually involves some investigative playing with the micro to find these special locations, and by avidly reading through informed articles in magazines. Be warned that these operations may only work on that particular version of the operating system.

Operating systems in perspective

One of the themes of this book is that true computer literacy means having a knowledge of microcomputing reaching beyond the necessary limitations and particular talents of your own machine. So to put your own operating system into perspective it is worth having at least an outline knowledge of the main microcomputer operating systems.

CP/M (*qv*) has tremendous importance, both historically, and because of the vast library of software written for it, much of it in the public domain. **MSDOS** (*qv*) has been made so familiar by the ubiquity of the IBM personal computer that it can't be overlooked. The **UCSD-p** system (*qv*) has some very firm adherents in academic circles and, with its concept of a 'total programming environment', it will almost certainly play its part in the design of the software systems of the future. **UNIX** (*qv*) shows signs of becoming an important standard on the new range of true 16-bit chips, and much

of its design has inspired MSDOS.

As operating systems develop they are acquiring a number of new functions. Facilities to speed up disks, for instance, can note which files you are regularly calling in from disk and keep them within easy reach. Concurrent CP/M (*qv*) includes a **RAM disk**, an area of the RAM which is treated like a disk drive and can be accessed almost instantly. Personal CP/M goes even further by providing an adaptation of a traditionally disk-based operating system that will run on a machine with no disks at all.

It may be some time before all these developments reach the home market, but because RAM is coming down in price faster than disks, operating system development is becoming more involved in organising files in memory rather than on disks.

There are many PEEKs and POKEs available and lots of these are detailed in Commodore's *Programmer's Reference Guide*. But remember that making use of these features will mean that the program will almost certainly not work if you change your 64 for a different machine. For some other useful PEEKs and POKEs see the sections on Keyboards and Data Processing.

It is often necessary to chain from one program to another keeping the variables intact. Using the LOAD command in a program does not clear the variables but, if the second program is larger than the first, it will overwrite the start of the variable table. You can get round this by setting location 46 to make the first program seem larger than it really is, so that BASIC will store all its variables higher in memory, out of the way. You'll first need to load each program in turn from the keyboard and make a note of the value of PEEK (46), which gives the number of the 256-byte blocks that the program occupies. Find the highest of these values and add 2 onto it. Let us suppose this gives us 36. If you now put this statement right at the start of the first program:

```
10 POKE 46,36: CLR
```

Adding 2 to the value POKEd into 46, gives you a little breathing space. The programs are bound to grow a little as you develop them, but you will not have to change line 10 until the largest program has grown by almost 512 bytes. You can make this safety margin larger if you want. There are two pitfalls to watch out for though. Because BASIC uses the program text as the string data, if any string variables are set up like this:

```
1520 H$="COMPANY HEADQUARTERS"
```

and then not altered in any way, when you load in another program the text will change and the data will be lost. The way round this is to change the statements to:

```
1520 H$="COMPANY" + "HEADQUARTERS"
```

The second form (and any other string assignment which involves calculation) stores the data in string space at the top of memory, where it is safe. The other thing to watch out for is once the program has been run and you have chained to a second program, to alter this second program you must first load a fresh copy with a LOAD command from the keyboard to reset the correct value in location 46.

The section on the 64 keyboard shows you how to disable the STOP key. You can also disable the STOP-RESTORE combination by a POKE 792,188, which disables the RS232 port too. You can return things to normal by a POKE 792,71.

There are many occasions when you might want to know if a key is being held down. You can do this by looking at location 197, which contains a different number for each key on the keyboard. Unfortunately the number is not the same as the ASCII code, but if you just want to test a few keys you can easily work out what their codes are. Similarly, location 653 will tell you if the SHIFT, CTRL and CBM keys are being pressed:

```
PEEK (653) AND 1   tests for SHIFT
PEEK (653) AND 2   tests for the CBM key
PEEK (653) AND 4   tests for CTRL
```

Printers: Hard Copy Made Easy

There are four broad price ranges of printer, the cheapest being the small heat- and electro-sensitive types. Next come the slow dot matrix printers and the adapted typewriters followed by the fast dot matrix printers and top quality dedicated daisywheels. At the top end, with prices measured in thousands of pounds, are the superfast printers, using laser and other ultra-modern technology.

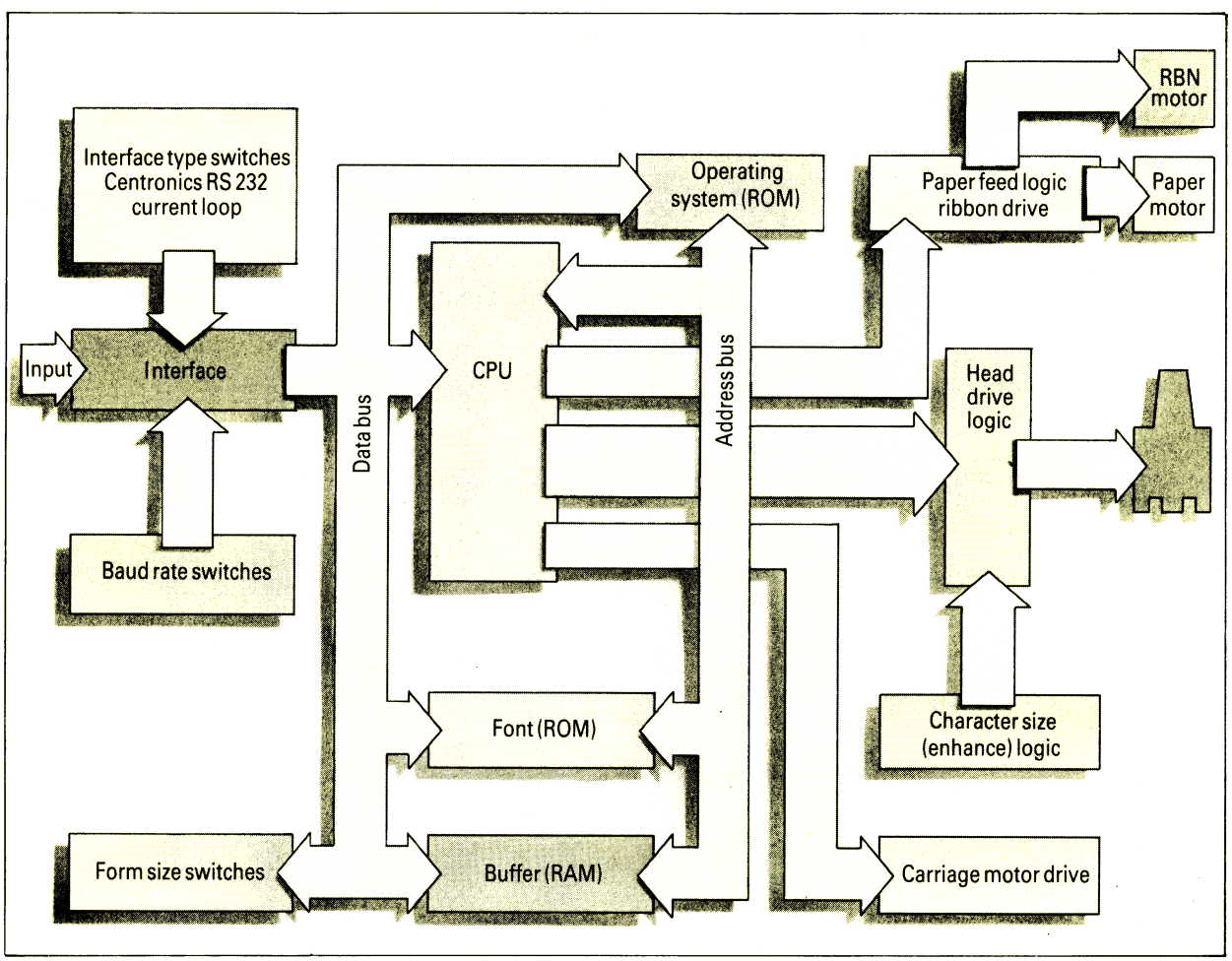
The quality of the print varies widely across the range, and choice depends on the function of the printed output. For draft work or for listing programs, virtually any output capable of being read without eye strain will do. For correspondence quality you will need good print, easily readable by eyes unfamiliar with its peculiarities. To produce artwork for publishing, only the very best quality of printer and typeface should be used.

High-quality printing can be either expensive and slow, or extremely expensive and fast, and it sometimes makes sense to sacrifice quality or speed in favour of cost. The range of print speeds, like cost, is again, very wide. The speed of a printer is usually quoted in one of two ways: **CPS** (characters per second) and **LPM** (lines per minute). The slowest printers, the converted golf-ball typewriters, manage about 10 CPS against the fastest, the laser and ink jet type, which print tens of thousands of CPS.

And then there's the matter of noise. Most printers have moving parts making rapid and repeated physical contact with the paper, combine this with with the rest of the mechanics needed to move the print head and the paper and you invariably end up with noise. Noise can be reduced by using an acoustic hood, but clever design is needed in creating baffles which will still allow adequate ventilation, and effective acoustic hoods are not cheap.

As to the business of connecting the printer to the micro. The simplest route is to get your dealer to connect the two and try them out. For a full discussion of the three common printer communication standards, RS232, Centronics, and IEEE, see the chapter on Communications.

Many printers are marketed under a variety of names. Manufacturers buy-in a printer shell and modify it, others just re-spray it a different colour, and yet more will just stick their own name over the manufacturer's label. The prices of these 'clones' can vary widely, so it is well worth shopping around. You may save up to a third by buying one in a different colour!



Block diagram showing the main components of a typical printer

Technically speaking a printer is a terminal, in the same way that a keyboard and screen combination is a terminal. Just like any terminal, a printer has memory and a certain amount of processing power, the amount varying fairly directly with the cost. The memory contains rules for printing all the available character sets, and when your micro sends a character code down the line, the printer looks up the code to see which character it should print.

However, the printer also has another set of codes which it uses for control purposes. These fall into two main categories. The first is the set of codes which do all the things you would expect from any printer: line feeds (advancing the paper by one line), carriage return (to return the print head to the beginning of the line), and form feeds (to advance the paper to the start of the next page or form).

The second category of controls handle the 'special effects'; changing the number of characters printed per inch (usually called the 'pitch'), line spacing, boldface, and underlining. Also included in this category are ^{super} and $_{sub}$ scripting – printing slightly above or below the previous character – as well as graphics for screen dumps, where the printer produces an exact image of what you've got on the screen.

The paper

The hard copy has to be printed onto paper, but even a seemingly simple substance such as this comes in many shapes, sizes and forms, some of it suitable for one type of printer but not others. Because of their method of operation, the heat- and electro-sensitive printers require special paper, and print on plain paper poorly if at all. Their simplicity keeps their purchase cost low, but special paper makes them more expensive in use.

Plain paper in single (usually A4) sheets is bought by the ream and either fed into the printer by hand as with a typewriter, or loaded into a sheetfeeder (expensive). Tractor feed or pin feed printers take paper in a continuous stream of perforated sheets with sprocket holes running vertically up the margins (these are sometimes also perforated so that they can be stripped

This is a demonstration of the Wordstar-driven Epson Mk III.

Standard typeface gives 80 characters to the line on an MX80.

By typing ^PA in Wordstar, you can get 132 characters per line, and they look better, too.

By typing ^PN, we can go back to standard type.

Expanded type is ^PE which poses problems. There are only about 40 characters per line here, and you must watch margins. Type ^PR to revert to standard.

Letter quality type is ^PQ emphasised. You cannot emphasise condensed type. Turn it off with ^PW.

On its own, ^PTfor superscript (^PT again to turn off) isn't too impressive. Do it with condensed print, however, and the machine will amaze you. How the designers had the nerve to aim for this detail is beyond me, let alone that they achieved it .

Subscript is identical, ^PYbut much lower down and again, looks more impressive when condensed.

Expanded and emphasised characters can be combined.

Is that enough ?

ends

Matrix printer hard copy, using a selection of special print facilities.

off after printing). So-called 'cut edge' or 'micro-perforated' sprocket-feed paper has such fine perforations that when stripped its edges are almost as smooth as separate sheets. In spite of all this, single sheets are definitely the best choice for word processing work, and if you can afford a good sheet feeder you will probably find it simpler and faster to work with.

Pin feed printers have small retractable nipples to the right and left of the platen which engage in the sprocket holes at the paper margins as the platen turns. To get the best from continuous stationery on a printer without this arrangement you can usually fit a tractor feed – a harness which mounts over the platen and drives the paper by means of a belt of nipples connected to the platen drive.

The point of pins or tractors is that the normal friction feed, perfectly adequate for typewriters or machines dealing with separate sheets, is not sufficient to prevent some slight but cumulative slipping as long lengths of paper wind through. It is possible to use continuous stationery with a bare platen, but the paper will always be vulnerable to sideways slipping and consequent jamming.

Some matrix printers come only with a sprocket feed and no platen, precluding the use of single sheets. Most daisy wheels can take a tractor feed option for continuous stationery such as custom printed invoices, payslips, and so on.

Plotting and graphics

In addition to their word printing ability, some printers have plotting and graphics facilities. Most of the cheaper matrix printers are capable of two kinds of graphic output: screen dump and true plotting. Screen dumping produces a dot-for-dot copy on paper of the screen image, but presupposes that the micro or printer manufacturer has written the necessary software to make the two compatible. Mixed in with this kind of graphics, many matrix printers have the ability to print user-designed characters.

Most daisywheel printers are also capable of drawing by entering into a 'graphics' mode that uses the full stop, shifting the paper and print head to produced continuous lines. It is a noisy and slow business, and very demanding on the full-stop character.

Reliability

Despite modern microprocessor technology, printers still have many electromechanical elements and are therefore inherently less reliable than the other components of a system. Economies in construction (and in support) will be false if they mean the system is often out of action. Rugged construction, simple mechanical operation, local service and an established reputation are all worth considering when choosing your printer.

Dot-matrix printers

Characters are created by dot matrix printers from a block of dots appearing or not appearing in a rectangle, in much the same way as characters are displayed on the computer terminal. In both cases the quality of the characters produced depends on the number of dots in the rectangle. Cheaper printers use a 5 x 7 matrix, and as you move up the scale of quality the number of dots is increased.

To offer more dots in the vertical plane the number of needles on the print-head has to be increased. The dots in the horizontal plane are created by the print head moving, and to produce higher quality results the density of these is also greater in the more expensive machines. The more dots present in the matrix, the better formed the characters can be. The first improvements tend to be proper descenders - q, y, p, g, j, and g have a more obvious 'tail', and this improves the legibility of the print. If the matrix provided is very dense it is even possible to print typefaces with serifs.

ASCII	CHRS	EFFECT
NUL	0	Used to end a control sequence, or as a dummy
BEL	7	Sound internal alarm (if fitted)
BS	8	Back space
HT	9	Move across to next (tab) print position
LE	10	Advance paper one line (line feed)
VT	11	Move (tab) vertically to next print position
FE	12	Advance paper to top of form (form feed)
CR	13	Return carriage (to home)
SO	14	Shift back to normal character set
SI	15	Shift to lower case/alter character set

ASCII printer control codes

The common matrix printers, such as the Epson, have a moving print head which contains a vertical row of needles, identical to the number of vertical dots in the character matrix. In operation, a magnet fires the appropriate needle against the ribbon to produce a dot on the paper. Head movement provides the horizontal component. Therefore, to print a character of a 7 x 9 matrix would involve 7 needles, some or all firing 9 times after a very small horizontal head movement, before leaving a small space, and then moving to the next character.

As the size of the dot matrix is increased this approach of needles being impacted on the paper by the sudden magnetic attraction created by pulsing a current into a coil, requires a large head and creates a great deal of heat. Newer heads are of the 'release', or 'stored energy' type, where the needles are pressed against the paper by springs, but held back by a permanent magnet. A pulse of much lower current is required momentarily to cancel the effect of the permanent magnet and allow the spring to fire the needle. Heads of this kind can be made smaller, and can carry far more needles. Twenty-four-needle heads are now beginning to make true correspondence-quality dot matrix printers a reality.

The 'near correspondence quality' printer is not cheap. But printers like this have at least three modes of operation to trade off speed against quality. At the fastest speeds, the print is as you would expect from a matrix, but when slowed down and with all needles in a

'compressed' mode, the dots are virtually invisible: close to the daisy wheel standard but still faster, and allowing font changes without having to stop the printer.

With this capability of changing fonts in mid-print, dot matrix printers have an enviable versatility. Virtually all of them can print characters of different sizes, widths and densities. With the facility for user-designed characters an electrical company, for example, could tailor the printer to produce transistor, capacitor, and resistor symbols. Given with this wide range of abilities, matrix printers are cheap, fast and not too noisy – altogether very suitable for a wide range of uses.

Thermal and electrosensitive technology belongs to the class of dot matrix printers, but instead of creating a mark by impacting needles on a ribbon, it relies on a special paper which is sensitive either to heat or to electrical discharge. The paper is instantly recognisable as it is waxy white or silver. Having to use special stationery puts the cost up and can be awkward – you can't use your own headed notepaper for example. The print heads are very simple, using heating elements or a spark discharge point, therefore they are virtually silent, but slow (since they need to wait while the paper reacts before they can move on to the next character).

The main advantage of this type of printer is its low cost and quiet operation. Some are capable of graphics and screen dumps.

Ink jet

These machines are yet another variant of the dot matrix principle where ink is squirted onto paper from a series of fine nozzles. The technology is quiet, uses very lightweight moving parts, and can be so fast that the only constraint on speed becomes the rate at which paper can be fed through the system.

A typical ink jet printhead comprises:

The ink cartridge
The ink flow system
The ink jet nozzles

These three units can be compacted into a small printhead with very low inertia. Unlike the needle printer, which forms character by impact, the ink jet printer projects tiny droplets of ink by applying voltage pulses to piezo-electric crystals mounted around each nozzle. Under the influence of electricity, piezo-electric substances deform slightly, and this property is harnessed in the ink-jet printer to apply short sharp pressure to the ink, sufficient to eject the droplets. Other systems are designed so that heating a tiny volume of ink produces steam which 'blows' the remaining ink in the drop onto the paper.

Special attention has to be paid by the designers of ink heads to the tendency of ink to spill and clog. Automatic jet cleansing which takes place while the head is idle is normally part of the system, and there is usually an arrangement to cap the jets (like putting the top on a pen) when the head is left unused for more than a few minutes.

Ink jet printers are extremely quiet in operation – often the noisiest thing about them is the movement of the paper! They have the disadvantage that specially absorbent paper is needed (ordinary paper produces an almost unreadable blurred image). Since the droplets arrive on the paper with no appreciable impact, carbon copies are out of the question.

Colour ink jet printers, using multiple ink cartridges are also available

Daisywheel

Daisywheel printers derive their name from the appearance of the print-head mechanism, a plastic or metal-coated disk punched out into the shape of flat spokes radiating from the centre like the petals of a daisy. At its tip, each petal bears the shape of one or more characters, and these create impressions on the paper by being hammered against an ink-bearing ribbon, which in turn makes contact with the paper. The wheel is rotated by a stepper or servo motor and stopped when the required character arrives in front of the hammer.

All have interchangeable wheels, with some makes offering a wider choice of fonts than others. Standard wheels from Diablo, Qume and Xerox are largely interchangeable, although the manufacturers try to keep you in line by suggesing that the 'ballistics' of other manufacturers wheels are somehow unsuitable. Don't try cross-matching proportional-spacing or metal wheels though; there are definite incompatibilities here.

A variation on the daisywheel has been produced by NEC with their Spinwriter. Here the petals of the printwheel have been turned up to form a sort of inverted thimble. Instead of the vertical orientation of the daisy wheel, the thimble rotates in a horizontal plane which, NEC claims, produces better results thorugh much reduced inertia. Cynical commentators have suggested that the thimble was a way of borrowing daisy technology from the Americans without infringing patents.

Because a rotating motor has to drive the daisywheel round, the printhead is usually heavier than that of a dot matrix printer. The increased inertia requires a larger motor to drive the printhead across the paper, and this in turn implies a sturdier power supply (and plenty of noise!). Daisywheel printers are still inherently slower that dot matrix printers because of the power supply requirements, and the fastest of the daisywheels (printing at a nominal 60 cps) requires heavy-weight – and therefore expensive – technology.

By lowering the speed requirements the printer designer can make savings throughout the whole chain formed by the daisy motor/printhead/power supply. The new slow speed daisywheels are able to use very low cost linear motors to send the printhead back and forth across the page. A linear motor works on the same principle as the magnetic suspension trains used experimentally for transport in Japan – the moving object slides above a rail on a frictionless cushion of electro-magnetic force which also provides the motive power.

If only a small amount of high quality printing is required, it is possible to obtain low cost daisywheel

typewriters with RS232 interfaces. Although they can only manage about 15 CPS, they represent good value for money as they are about one third the cost of a conventional daisy wheel printer. And you get a free typewriter!

Ribbons

Ribbons come in three main types, fabric, multi-strike film, and single-strike film. The film type is sometimes called a carbon ribbon – the terms are interchangeable. The fabric ribbon is an ink impregnated cloth (nylon in the better types) which travels back and forth, automatically reversing at the end of each cycle, until the ink runs out and the cartridge has to be thrown away. The print produced is of variable density with a slightly fuzzy edge caused by the cloth base, but it is economical as the ribbons last a long time.

Film ribbon is a thin plastic tape coated with carbon. The difference between single- and multi-strike ribbon lies in the distance the ribbon is moved after each character is printed. Single-strike ribbon travels a full character width after each print, whereas multi-strike ribbon edges forward only by part of a character width. Single-strike gives a clearer impression because each character is created from entirely fresh carbon, with no possibility of broken lines, but to do this the ribbon passes faster through the cartridge, and so is more expensive.

Printing with top quality ribbons on a daisywheel can be an expensive business. You can save money by changing ribbons according to the work being printed. Do your draft work with fabric ribbons and only use the carbons when you need top quality.

Though limited in speed, daisywheel printers are still the first choice where the highest quality printout is needed. However they are expensive and noisy, and for the ordinary home user it's probably fair to say that a dot-matrix printer would be just as good – and far cheaper.

Daisywheels also lack the dot matrix printer's talent for swapping fonts in mid-stream. As dot matrix technology develops and more needles appear on the print head, the daisywheel approach to printing begins to take on a distinctly old-fashioned look.

Golf-ball typewriters

A golf-ball typewriter can provide extremely good quality print at a much lower cost than other 'letter quality' printers. In particular you have the benefit of smart, legible print, the choice of dozens of different typestyles (by using different elements or 'golf-balls'), and the choice of quality carbon or utility fabric ribbon. Another advantage is the ability to use the machine as a normal typewriter.

There are some disadvantages as well – the machines are fairly bulky and noisy. Printing speed is 15 characters per second, and is restricted to alphanumerics – no proper graphics of course. You will almost certainly be using a secondhand machine, and it might need maintenance one day. But the machines often function for years without anything going wrong and spares are easily and cheaply obtained.

To work with a micro the golf-ball typewriter has to be converted – an ordinary office machine is not suitable. What you need is an IBM model 73 I/O (Input/Output) or KSR (Keyboard Send/Receive) machine, together with an electronic interface to convert the Centronics signals to IBM Selectric code. The model 73 is a special heavy-duty version of the more commonly found IBM 72 Selectric office typewriter, with added solenoids to enable it to be operated electrically by remote control. It has not been produced for some years now, having been superseded by quieter, faster (and more expensive) daisy-wheel printers.

A suitable machine ready converted and complete with interface can be bought for rather less than a low-cost dot matrix printer – but be warned, if you the buy items individually you will have to rewire the typewriter internally (difficult without a manual), and not all model 73 machines have 'correspondence' keyboards and golf-balls – some are intended for computer use (no lower case characters), and are totally incompatible. As they will not have been overhauled there is little cost saving anyway, and you could save a lot of heartache by purchasing a converted and guaranteed device.

Buffers mean speed

A printer buffer is a block of memory (external to the computer's own memory) in which characters are stored prior to printing. The computer can send data to the printer at speeds of at least 1,000 CPS, but an unbuffered printer can only accept new characters at its highest print rate, creating a bottleneck which keeps the computer waiting. Interposing a buffer is a bit like giving the printer a reservoir which the computer can fill quickly before resuming other work. The printer will then dip continuously into the reservoir at its own speed.

This picture presupposes that the storage capacity of the buffer is large enough for the computer get rid of all its data. If you send a 32K file to a 16K buffer, the buffer will fill with the first half of the file and send a 'busy' message back to the micro. The rest of the file will then have to dribble into the buffer at the same rate as the buffer empties data into the printer. The micro will only be held up for the time the printer takes to print half the file, so the buffer has made its contribution, but the advantage is substantially reduced.

There are two ways round this. You could estimate the maximum size of a typical print file (roughly speaking, one character is one byte, so the number of characters in a file is the number of bytes) and buy a buffer which is big enough to hold a file of that size. Or you could make a habit of splitting large print files into smaller files, each the size of the buffer, and each printed out one at a time.

Printer buffers are quite expensive – especially if the capacity required is greater than 16K. Obviously you need not buy one with this much memory – there are some available with as little as 2K – but if you are regularly

downloading larger files than this then the smaller capacity buffers may be more trouble than they are worth.

It is usual for buffers to have some other functions. Manufacturers like to use the phrase 'intelligent buffer', although all buffers must have some degree of intelligence to cope with the data flow correctly. For the buffer to function efficiently there must be some form of handshaking between the printer and buffer which tells the computer that its ready to accept more characters. The buffer need not be empty when it asks for more data, it all depends on the individual design.

Buffers dedicated to particular printers are available; these fit inside the chassis and come as an uncased board. They are normally fairly simple to install in a few minutes if you follow the step-by-step instructions provided with them.

One way of testing your buffer is to write a simple `FOR .. NEXT` loop, listing and printing the numbers from 1 to 100. Without a buffer you would have to wait for all the numbers to be printed before the computer comes back on line, but if the buffer is working properly the computer will be available for use while the print continues.

In practice writing data to the buffer becomes as fast as writing straight to the screen. Once you've tried it the chances are you will never want to be without it.

Most types of printer can be connected to the 64. If you have an RS232 interface you can drive any serial printer which uses hardware handshaking rather than an XON-XOFF protocol. Alternatively, if you fit a Commodore IEEE interface you can use any of the Commodore printers intended for the PET/CBM range. On the other hand you could fit a Centronics interface to use a standard parallel printer. If you have a printer which offers a choice of serial and parallel, it is best to use parallel as it makes graphics printing faster.

Processor: What Makes A Micro Tick

The central processing unit, or CPU, looks rather like an overgrown memory chip. But unlike memory, where there is a certain commonality between different systems, once we delve into the workings of the processor we find that there is a very wide variation between different products. Knowing how a Texas Instruments RAM chip works tells you probably as much as you need to know about the functioning of all the memory chips you are likely to come across; understanding the Zilog Z-80 is not an automatic entry to programming the Motorola 68000.

But there are enough general principles for us to be able to discuss processors without getting too bogged down in the divisive details of particular hardware. With the proper overview, understanding the eccentricities of an individual CPU shouldn't be too difficult. We can take as our basis the classic 6502 processor, because its elegant simplicity makes it an ideal starting point. Other ubiquitous chips like the 8086 and the Z-80 behave similarly enough as far as the essentials are concerned.

The job of the processor is to run programs – a simple enough statement that conceals a philosophical minefield too vast and risky to explore here. The origins of the processor lie in the history of numbers (*qv*) themselves and their association with that deceptively simple counting machine, the abacus. As numbers became more abstract, counting engines became more elaborate, until John Napier's invention in 1614 of logarithms released a whole new class of mathematical engines, like the slide rule, which calculated using physical analogue models.

In many ways these complex devices were a diversion from the principles we use in calculating today. The slide rule with its graduated scales, underpinned by Napier's remarkable and eccentric discovery (his original log tables were based, for no very good reason, on the number 7) is a highly sophisticated piece of equipment when compared to the idiotically simple activities going on inside the CPU. The processor does even less than an abacus, which can at least distinguish between a range of ten numbers. The processor knows of only two numbers, *one* and *zero*, and the only real maths it can do is to add.

Its secret of course lies in its speed. The binary method of counting used by the processor can embrace any number under the sun as long as the system handling it can cope quickly with huge strings of ones and zeros. And the simple mathematics of addition can be subverted quite easily to produce:

Multiplication by adding a number to itself the required number of times.

Subtraction by including the ability to handle negative numbers.

Comparison by seeing whether the difference between two numbers is zero.

Division by finding out how many times one number can be subtracted from another.

From there on you can see that the possibilities are limitless.

In fact by suggesting that addition is the elementary process on which everything is based we have simplified the whole business – and at the same time we have still somewhat overstated the case for the processor. Addition is itself derived from the even lower level activity of looking at pairs of single bits to detect whether they are the same or different – and that is the bottom line on what *all* the processor's internal circuity does.

From this elementary act another set of operations can be built up, complementing the arithmetic capabilities of the processor. In real life processors spent little of their time computing in the strictly arithmetic sense of the word, and many programs – like the word processors used to put this book together – draw mostly on the processor's other talent, the ability to perform *logical operations*.

Logical operations, sometimes known as Boolean algebra, are fundamental to computing, enabling the machine – or rather software running on the machine – to make comparisons, draw conclusions and take decisions. Unlike the arithmetic operators, the four logical operators accept as input only two kinds of values, which in the context of decision-making are usually thought of as being 'true', or 'false'.

AND accepts two values as input and outputs a value meaning 'true' if they are both true, or a value meaning 'false' if one or both of them is false.

OR accepts two values as input and outputs a value meaning 'true' if one or both of them is true. If both are false it outputs the value false.

XOR, like OR, will output 'true' if either of its two input values is 'true', and 'false' if they are both 'false'. However it also outputs false if *both* inputs are 'true'. XOR stands for 'eXclusive OR'.

NOT accepts only one value as input, and simply returns the negation of that value. The input 'true' returns 'false', and 'false' returns 'true'.

There is also a fifth logical operation, one which sounds rather different, but which, to logicians, belongs quite naturally with the rest. This is the implication, or **IMP** operation, which appears in BASIC in the guise of the `IF` statement. IMP accepts two Boolean values and returns the value 'false' only if the second input value is 'false'.

The processor in action

To find out what is happening when a micro runs a program, consider the simple instruction:

```
LDA 1000        A9          00  10
                Opcode      Data
```

In English this means 'get the content of memory location 1000_{16} (the tiny 16 tells you that this is a *Hex* number – *see* Numbers), and put it into the accumulator'. The accumulator is the processor's chief internal register, and is used for arithmetic operations. The three numbers following the instruction are the corresponding 6502 machine codes in hex for the instruction.

The first is the operation code, or opcode for short, which tells the processor what the instruction is. The other two numbers are the data to be used by or with the

instruction, in this case the address of the memory location used by the instruction. To execute the instruction, the processor first has to read the opcode – this is called an *instruction fetch*. It then has to decode the instruction to work out how many bytes of data follow. In this case there are two more bytes, but the number will vary from instruction to instruction. Running a program is just a continuous loop:

Read and decode an instruction.
Read any further data required.
Execute the instruction.
Go to step 1.

The illustration shows a simplified block diagram of a typical processor. We can see the three buses coming in from the outside world, and a number of boxes which represent the functional units within the processor. It is in fact a very simple processor, having only four registers for use by the programmer – the Accumulator, the Index Register, the Program Counter and the Stack Pointer.

The accumulator is the main register in which the results of the arithmetic and logical operations are stored, and is also used when shifting data between other register and external memory. The accumulator is the register most used by the programmer, and most of the processor's instruction set makes heavy demands on it too.

The index register is used to make it easy for a program to scan lists and tables – used a little like an array subscript in BASIC. It is usually not possible to do much arithmetic on the index register, apart from loading and storing it and incrementing and decrementing it (adding and subtracting one, respectively). Its main use is to specify addresses to be used by accumulator instructions. Figure 2 is a small program in 6502 Assembly language, with corresponding BASIC statements. The # is an assembler instruction saying 'take the following number as a value rather than a memory address'.

The program counter is an essential register with the job of keeping track of where the processor is in the program. After every byte of program is read, the program counter is incremented by one, and it is also directly altered by instructions such as `JMP` (like a BASIC `GOTO` statement), `JSR` (like a `GOSUB`), and `RTS` (like a `RETURN`), which in effect divert the flow of the program.

The stack pointer has many uses in assembly language programming. The stack is a data structure a bit like a pile of cards. You build up the pile by putting one card on top of another, and whenever you take a card off the pile, you always get the last card you put on. This property is known affectionately as **LIFO** storage – Last In, First Out.

The processor uses the stack frequently when it executes a `JSR` instruction. Every time the processor branches off to a subroutine it has to store the value of the program counter somewhere, because when it comes to an `RTS` at the end of the subroutine it will need to know where to return. As you can have subroutines calling subroutines calling subroutines, a stack is the best way to store these addresses so they will not get mixed up.

The **ALU** or Arithmetic-Logic Unit is the real 'brain' of the processor, where all the arithmetic and logical operations are carried out. It is the job of all the other units to feed the ALU with data, to give it instructions, and to take away its results.

There are three **holding latches**: the instruction latch, the temporary address latch, and the next address latch. A latch is a circuit used to hold a value for as long as required, rather like a memory location. The instruction latch receives the opcode from the data bus, and holds it until the instruction decoder has finished with it, even though the data on the data bus is changing during the execution of the instruction as data moves in and out of the processor.

An address on an 8-bit processor requires 16 bits to define, whereas the data bus is only 8 bits wide. If we have an instruction such as `LDA 1000`, which says 'Load the accumulator from memory address 1000_{16}', the address to be used is held in two consecutive memory locations after the opcode. These are read by the processor separately as the instruction is decoded, and the full 16 bit address is built up from the two halves in the temporary address latch.

The next address latch actually controls the address bus. If an address is put in here it will automatically appear on the address bus. In our `LDA 1000` instruction, this address would come from the program counter for the opcode fetch, and then from the temporary address latch ready to read address 1000_{16}.

The program is actually executed by the **instruction decoder** and the **control unit**. The instruction decoder is in fact a small computer, which runs not a program but a microprogram held in the fragment of very fast ROM which determines the instruction set of the processor. The necessaray speed of this ROM makes it an expensive item on the designer's budget – too many different instructions will make the chip too expensive.

The instruction decoder has the job of looking at the instruction, working out what has to be done, and breaking the job down into smaller tasks which it sends to the control unit to execute. It is the control unit which actually does the work, sending the necessary commands to move or manipulate the data.

Programming: The 'Black Art'

Programming is an almost religious experience – lots of suffering, occasional uplifting rewards and frequent groans of despair – all painfully good for the soul. Here are some elementary principles:

Always write in the highest-level language that produces acceptable results. There is no point writing a program in assembler if BASIC can do the job just as well.
Structure your solution. Suitably organised, commented and indented, a piece of software is a work of art. The structure of your program should reflect the data structures and operations of the task. This makes for a clean program that is easier to debug, test and modify than a 'spag-bol' program.
Don't rush into churning out code, but carefully design the program on paper first. Every hour spent on designing the logic will save five hours of frenetic debugging later on.
Document your programs. The variable **P** (or any other sort of P for that matter) will not mean a lot to you or to anybody else after a year or so when you want to improve, modify, debug or translate the program containing it. Document the functions of all routines – in plain English – as well as the use of all variables.
Use as many system and previously written routines as possible. Save pain, do not try to re-invent the wheel. Stand on other programmers' shoulders (not their toes).
Aim to produce as general and comprehensive a program as possible. The more specific your solution the more difficult it will be to alter your code if the specifications change. Simple solutions are best and often the most efficient.
Do not rely on your testing alone to assure yourself of the accuracy of your program. Testing proves only the *presence* of bugs, not their absence. Try to prove to yourself by a 'logical walkthrough' of the program that the sections of code do what they ought to do in all cases.
Backup your program until you are blue in the face with backupping.
Take regular hardcopy listings of your program. They make debugging much easier, save on the eyestrain and also serve as last-resort backups.
Do not assume that the user of the program will have a PhD in computer science. Make your programs informative and user-friendly. The ideal is a program that is completely self-contained, users never having to look at a boring old manual, all information being at their fingertips.
Oh yes, and try to enjoy it as well!

Structure

Structured programming is a method developed to make sure that code displays the logic of the program in a way that is easy to understand and maintain. Some languages have features that force structured programming on the programmer.

For example, in PASCAL all information about variables (apart from their values) is given at the beginning of the program. A list, called a declaration, gives the name (**X**, **Y**, **Z**, **NAME** etc.) of each variable, together with its type (whether it will be used to hold an integer number or not, or a string of characters). This encourages programmers to think about the variables they need before launching into writing the code.

It's a discipline lacking in BASIC, although other structured facilities have been introduced to versions of the language. One branching structure common to all but the most minimal BASICs is the addition of an **ELSE** clause to the **IF** statement, giving the syntax:

IF condition **THEN** action 1 **ELSE** action 2.

The condition might be to see if the contents of variable **A** are equal to 10, in which case the first action is performed; if **A** doesn't equal 10, then the program will execute action 2.

Other structured facilities available in some versions of BASIC include **REPEAT ... UNTIL** and **WHILE ... WEND**. Both are used to set up loops, and to complement the standard BASIC facility **FOR ... NEXT**. One shortcoming of **FOR ... NEXT** is that it entails specifying the number of times the loop is executed, as in this example:

```
70   FOR X = 1 TO 1000
75        intra-loop action executed here
80   NEXT X
```

Here the action defined in line 75 will execute 1000 times. However, with **REPEAT ... UNTIL**, the loop will execute for as long as it takes to reach a certain condition. Loops of this kind can be emulated in simpler BASICs without the **REPEAT** facility, using **IF ... THEN**:

```
10   LET X = X + 1
15        intra-loop action executed here
20   IF X < 1000 THEN GOTO 10
```

As well as clarifying the structure, **REPEAT** also brings the advantage of speed. **GOTO** statements require the whole program to be scanned until the target line is reached, but the **REPEAT** statement automatically marks that line as a target line. The loop can find it instantly because its address has already been noted.

WHILE ... WEND differs from other loop statements: it doesn't assume that a loop needs to be performed first. With **REPEAT** at least one pass will be made before the program jumps onto the next step. But if the condition defined by **WHILE** isn't met, the program will jump to **WEND** without executing the loop at all. For example:

```
10   WHILE X < 10
20   LET X = X + 1
30   WEND
```

If **X** is equal to or greater than 10 the program will ignore line 200 and carry on with subsequent lines.

One further point about the use of **REPEAT ... UNTIL**, as compared with **FOR ... NEXT**; sometimes you might use an **IF ... THEN** statement after the **FOR** in order to jump on when a condition is met, as in:

```
10   FOR X = 1 TO 1000
20   LET Y = 10 * X
30   IF Z = Y THEN GOTO 50
40   NEXT X
```

This is a classic example of what structured programming tries to avoid. What horrors await the unfortunate program counter when it gets to line 5000? Does it ever return to the loop, and if so, what has happened to the poor abused variable `X`? You might have to go through all the rest of the code with a magnifying glass looking for the answers.

The bugbear here is the `GOTO` statement. The deviser of PASCAL, Niklaus Wirth, was so aware of the potential destruction coiled up inside that little four-letter word that he left it out of his original specification of the language. `GOTO` is back in now, because frankly PASCAL is too rigid without it; Wirth discovered that its absence made error-trapping difficult, if not impossible. But the compiler insists that you declare your target labels beforehand.

The BASIC subroutine call, `GOSUB`, is an improvement; at least it 'holds onto the string' with some promise that execution will return to the point of departure. However, there is no guarantee of this, and the routine jumped to may spray off into a shower of `GOTO`s or fall though into some other code without ever finding the necessary `RETURN` statement. The worst thing about it is that there is no secure way of passing data to the subroutine. Certainly you can set variables before calling the routine, but this doesn't completely fit the requirements of sound structured programming. And it is dangerous, as we shall see.

Suppose you have a subroutine to draw an elephant. You obviously want to be able to draw more than one size of elephant, so it would be nice to have a generalised routine that accepts a different size each time it is called. And let's say you want these elephants in different colours. In a structured language the routine would look something like this:

```
procedure elephant(size, colour);
    const legs = 4;
    var size, colour,i;

      DrawHead(size);
      DrawBody(size);
      DrawTail(size);
      for i = 1 to legs do
        DrawLeg(size);
      FillAll(colour)
    return;
```

In this imaginary language (actually a mixture of PASCAL and C) we can call our routine with the line `execute elephant(pink,enormous)` or maybe `execute elephant(green,miniature)` or some such. The adjectives in brackets, values **passed** to the routine, are called **parameters**. The essence of a good parameter is that it is a sort of whispered instruction to the routine which isn't overheard by the rest of the code, like picking up the telephone and dialling `e-l-e-p-h-a-n-t` on a private line and saying: 'Do your thing again, and I'll have green and big this time.'

The secrecy between the calling line and the routine is fundamental to structured programming. In less fanciful language, the point about the parameters (`size,colour`) is that they are unknown to the rest of the program. Because of this, no casual action of the program outside the elephant routine can ever change the parameters by mistake. You can even have a parameter of the same name in some completely different routine without risk of conflict.

Conversely if something is going wrong with our elephant production, we don't have to scrabble through miles of code to find the error – if we are calling the routine with the correct parameters, then the error must lie inside the elephant procedure.

BASIC won't let you `GOSUB elephant`; it requires a line number, which doesn't help intelligibility. But the worst thing about standard BASIC is that it has no proper parameter-passing scheme. Values sent to the subroutine have to use **global** variables, so called because they are known to the whole program – and they are liable to be altered from anywhere. If your BASIC elephant production is going crazy it could be that the variable has accidentally got the same name as a variable in another routine (perhaps one which that positions the creatures on the screen, or something of that sort). You will have to check the whole program to find it.

The structures that we have loosely called 'routines' are generally discussed under two headings: **procedures** and **functions**. The difference between them is far from clear-cut: generally speaking a procedure performs an action (like drawing an elephant), whereas a function simply returns a value. But functions can be designed so that they perform procedure-like actions as a side-effect, and procedures can incorporate the actions of a function by returning with altered parameters.

Structured programming is more a way of thinking than a set of facilities. So even if your favourite language doesn't have all the bells and whistles we have been discussing, you can still form good programming habits and be aware of the opportunities for improvement. The rewards aren't immediate, but it's an approach you'll certainly appreciate when you come to revise your program or to borrow sections of it for use elsewhere.

Recursion

Recursion is a powerful tool and a useful technique to add to your armoury of programming skills. But it also has the reputation of being too simple to understand. Students have been known to wrestle long hours with complex misconceptions of what recursion is, before they eventually see the light. The time to get to grips with recursive programming is when you make your move from BASIC to assembler, PASCAL, LISP or whatever, or when you start to use some of the more sophisticated data structures. The concept isn't difficult to understand, as long as you 'think simple'.

Recursion in a programming language is the ability to define a routine (or procedure, or subprogram or

whatever) in terms of itself. A subroutine called **X** can call subroutine **Y** in virtually all languages. Recursive languages allow subroutine **X** to contain calls to **X** from within **X**.

A problem which is recursive is one that can be defined in terms of a slightly smaller version of itself. The classic example is the factorial function. The factorial function, for a positive integer *n*, is calculated by:

```
n * n-1 * n-2 * n-3 * ... * 2 *1
```

Factorial 5, for example, is **5 x 4 x 3 x 2 x 1 = 120**.

For any positive integer *n*, factorial (*n*) can be defined as *n* multiplied by factorial (*n*–1). Notice how this definition folds back on itself, evoking the function 'factorial' to explain itself. This sounds hopelessly circular, but it needn't be. Recursion can turn out to have a suitable termination condition preventing it from looping forever, and routines that *bottom out* (as this one in fact does) often provide a very simple way of solving otherwise long-winded problems.

The end condition for our factorial function is reached when *n* = 1. Factorial 1 is 1; factorials less than 1 are undefined. So this function can be written (in pseudo-PASCAL) as:

```
FUNCTION FACTORIAL (N: INTEGER): INTEGER;

BEGIN.
IF N=1 THEN FACTORIAL := 1
            ELSE FACTORIAL := N*FACTORIAL(N-1);
END;
```

The function would be called from another part of the program. For example:

```
X:=23;
Y:=FACTORIAL (X);
WRITE(Y);
```

The print statement in BASIC is equivalent to the `WRITE` statement in PASCAL. The `N` in the factorial function is called a **dummy parameter**, as it is replaced by the value used as a parameter whenever the routine is called from the main program.

Keen-eyed readers will note that the factorial function can quite easily be written as a `FOR` loop. In sequential languages like PASCAL, `FOR` loops are generally more efficient than recursive calls and should usually be be used in preference for speed of execution and for conservation of memory – although recursive procedures often show the logic of the solution far better. Recursion comes into its own in problems where no iterative (loop) solution is easy or obvious.

Recursion as a feature of a language requires local variables and **dynamic storage allocation** for subroutine calls, which is why it is not generally available in BASIC. BASIC and FORTRAN use **static** space allocation, which means the amount of storage space a program can use is fixed before the program is run. Dynamic space allocation allows a program to use more memory space if it is needed. Each recursive call of a routine requires more memory space, so a call of factorial (10) would need more space than factorial (5).

The logical clarity of recursion usually makes the business of coding extremely easy; the hard work comes in spotting the algorithm in the first place. Recursive algorithms are often obscured by their very simplicity.

Some data structures are well suited to recursive programming. A particularly good example is the **tree**. Each tree is either made up of a number of sub-trees, or is empty. Each sub-tree is either made up of sub-sub-trees, or is empty and so on. Notice how even the definition of a tree can be recursive.

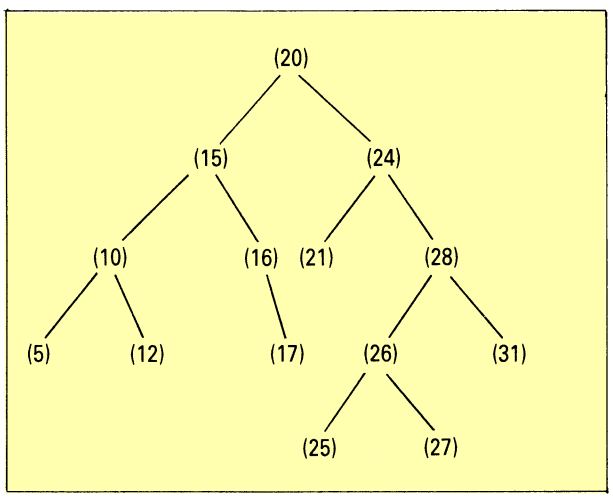


Fig. 1. A binary tree (see recursion)

As an example of how recursive techniques can be used, take the tree in figure 1 and consider the problem of how to print the values of the nodes in order. Remember that any node can be reached only from its parent node and not from any other. This is because that is the only node which contains a pointer to it, and the records can be in any order in our representation of a tree. The recursive solution is:

```
PROCEDURE BINARY TREE PRINT (N: NODE);

BEGIN
IF NOT EMPTY (LEFT BRANCH FROM NODE) THEN
BINARY TREE PRINT (NODE FROM LEFT BRANCH);
WRITE (NODE N VALUE);
IF NOT EMPTY (RIGHT BRANCH FROM NODE) THEN
BINARY TREE PRINT (NODE FROM RIGHT BRANCH);
END
```

Given the initial call on the first node (the **root** of the tree), this procedure will print out all the values in order. Note that the tree can be of any size (except totally non-existent), as the procedure is independent of any reference to size.

Recursion can be simulated in BASIC by using a stack, but the language is not really designed for it. Surprisingly perhaps, recursion is easy to implement in assembler. Most microprocessors use stacks to store return addresses and registers. A suitable subroutine calling method has to be devised to save all the current registers and local variables, so that when the recursively called

routine returns up a level, the environment can be restored to its original, pre-call, conditions.

Portable programs

You might think that that manufacturers deliberately try to foil programmers who want to write programs to run on a number of different machines. Code portability, as it is called, results from foresight and discipline on the part of the writer. But even so, many good computer programs which would be better described as transportable rather than portable, require an effort which increases with the degree of incompatibility between machines.

Hardware is the biggest problem because different machines have different ways of handling and processing data. The so-called 'universal' language, BASIC (*qv*), only helps to make matters worse because manufacturers all tend to have their own idea of what BASIC should be. Most versions of BASIC use a common set of commands, like **PRINT**, **LOAD**, **RUN**, **GOTO** and **GOSUB**, but will often also rely on commands unique to the machine being used. Users may well not realise this until they try to run a BASIC program on a machine it wasn't written for. (For a comprehensive toolkit to tackle this particular problem, *see* 'Crossing the dialect boundaries' under Languages: BASIC).

When moving software from one machine to another, by disk or down a serial line, you have to decide the form in which the software should be transferred. There are usually two options: in source form (eg. a BASIC listing) or in object form (eg. an assembled or compiled program). To transfer object code it is not enough that each machine has the same processor. You must also be able to cater for all the operating-system and hardware calls made by the program, and this usually means ensuring that the machines are identical, or at least running the same operating system.

Source code is a surer way of moving programs between dissimilar machines, but it does imply that you will need a similar type of compiler on the target machine, as a different compiler may not understand all the commands used in the source code. On top of this, you will have to ensure there's enough memory to cope with the program and the data it uses.

On the bigger systems, and particularly under the UNIX operating system, compatible source code is the only form of portability. This is fine for programmers and software vendors, but leaves users vulnerable when upgrading to new machines. That is when portability or the lack of it strikes home – often expensively. If you rely on packages supplied by software houses you may have to purchase a complete new set of software.

Sticking to software you have written yourself doesn't always help either, particularly with home micros, and in the worst case leaves you facing the prospect of having to rewrite all your programs.

To avoid the immensely time-consuming and tedious task of having to revamp your software library and also to help you write programs that can be shared or sold among a wider circle of micros, and there are a number of designing and writing principles to bear in mind.

Separate and minimise machine-dependent parts of the program. Make sure the interface with the hardware is clearly defined. An operating system like CP/M, with its separate and clearly defined BIOS machine interface, will do this for you automatically, except that cursor addressing and the manipulation of the screen is, in general, nearly always hardware-dependent. This is why it is essential to structure application programs so that the hardware-dependent routines are kept in a separate part of the code, or at least clearly signposted with comments.

On machines that use BASIC, you can easily identify the machine-dependent areas: they nearly always concern commands that manipulate particular areas of memory, specifically those areas of memory that deal with the screen, sound and tape or disk. So you should isolate all graphics and sound routines as well as the commands which input from or output to a file, whether on disk or tape. You will also have to avoid **PEEK**s and **POKE**s, as these will only work on a micro with its memory organised in exactly the same way (*see* Memory maps).

Use **international standards** that exist for most high-level languages. Unfortunately the International Standards Organisation, ISO, has not yet established such a standard for BASIC, but the most widely used version for disk-based machines is MBASIC, or its later development GWBASIC.

The low-level assembler ASM.COM provided with the operating system CP/M-80 is a very wide-ranging standard, and its source code will work on all CP/M machines – provided all the calls you employ go through the operating system. Unfortunately the same principle doesn't apply to MASM.EXE, the MSDOS assembler, which is not always supplied with the operating system.

The ASCII code is used by most micros, so at source-code and text level there is usually at least this rudimentary level of portability. But some machines may still need a degree of translation: in particular there is little standardisation of the character codes used to perform control functions (ASCII codes 0 to 31) and none at all when it comes to values greater than 127.

Beware of language extensions. Many languages are supplied with a few extra commands, and if you use them in your software, they won't be understood by a compiler which doesn't know about them.

Write the program in **small simple modules**, if possible and if the language allows. It's good structured programming anyway, and will also help when moving the software over to machines with smaller memory capacity.

If you're programming for the smaller end of the market it might be better to think of designing your software in a carefully structured ultra-high-level language that doesn't run on any of the machines, but is purely used to define the logic of the program. There are languages specifically for this purpose, but **plain English will do**, provided you're disciplined about it. Once you've got that right, writing different versions for the machines you want to cover is simply an exercise in translation.

Program generators

Designing programs is an art. But particularly in the business world where requirements are often so similar, designing programs is often the art of copying what has been done before. Programming itself – the churning out of the code – is largely a clerical task, the sort of job that information technology is supposed to be relieving us of. Can't all this be done by computers?

Program generators are programs that write programs. In use they often closely resemble database (*qv*) packages, the distinction being that program writers produce high-level code, such as BASIC, which can be run independently of its generating parent.

Ironically the high-level languages of the 1960s, like FORTRAN and then COBOL, were introduced as the ultimate man/machine interface – English-like languages comprehensible by computers, in which human beings could easily express problems. But coding turned out to be not that simple. It required professional programmers and program designers, and reams and reams of source code to achieve anything useful.

The creation of short-cuts through the painful and costly business of programming became an industry in its own right. Various report-generating languages were spawned to take care of the formatting and selection of output.

When in the late 1970s micros made computer power available to a much larger number of people, programming the machines was no easier. In some ways the common use of the 'simple' language BASIC made everything more complicated. Lacking sophisticated methods of handling data files, BASIC requires extra code to achieve results that more file-oriented languages like COBOL find easy. Naturally the introduction of software packages that promised to do most of the programmer's work sparked off enormous interest.

At the beginning of the 1980s the sensational announcement of a 'universal program generator' called THE LAST ONE turned out to be something of a disappointment to many who were only too ready to believe in miracles. Like the languages they write in, program generators should be capable of catering for all eventualities. But the more a program-generating system can do, the more difficult it is to handle.

A high-powered system that did everything you needed might be so ungainly that you would happily go back with a sigh of relief to writing in the raw language. An-easy-to-use system that took little time to master might well produce only simple programs that you could have knocked up without much difficulty by hand.

In practice program generators have two different markets, because the needs of the programmer/analyst and the non-programmer are different. The non-programmer has a grasp of what needs to be done, but not how to do it, while the programmer has a thorough grounding in program file structure and file generating.

Simpler functions, such as those sometimes referred to as the electronic card index, are well within the scope of an untrained user. The design can be easily visualised, and the terms file, record and field can be interpreted and assimilated painlessly. But when the task expands to multiple data files linked and indexed, the average micro user will become uncomfortably aware of the need for a thorough grasp of file-handling concepts and techniques.

So systems aimed at the end-user should cope with most of the simple file-handling without much intervention, and storage and retrieval should be almost automatic. On the other hand a programmer requires to be 'let into the system' to design complex file structures, such as trees, if necessary. Often systems written on micros are more complicated than their mainframe equivalents because of the need to optimise speed and reduce space.

The code generated by and for the non-programmer is unlikely to be modified, whereas the code produced for the programmer may well be 'tweaked' to produce results not possible with the generator alone. This puts a different emphasis on the code generated: if it is aimed at the programmer it must be well documented with remarks to make updating easier, and will probably have user-defined names.

Typically, program generators of all sorts will use **menus** in the design stage to guide you through the facilities offered. You'll find this very useful for getting to grips with unfamiliar software, although there may come a time when you wish you could speed up the process by dispensing with the menus. Usually you can't. Tutorial examples that take you through all the more important facilities make a useful introduction. This approach has the additional advantage of allowing you to produce a working system, an achievement that bolsters self-confidence and encourages progress.

Most of the currently available packages generate BASIC code, or more rarely COBOL. They all go through the phase of producing the system parameters, defining the files, records, fields and other aspects of the system in a fairly standard computer-type fashion.

THE LAST ONE allows a simple verbal **flowchart** to be developed as an English-like series of commands. Even when these have been defined, it is possible to alter them, using commands to insert, amend and delete. A copy of the flowchart can be printed out during development and also included as `REM` statements at the top of a program it produces.

Files and flowcharts can even be modified after they have been used in live running. The system then generates a new flowchart and suite of programs and alters the file structures accordingly. This is a powerful facility, but it should be looked upon as a last resort and should not be used as a substitute for insufficient thought at the design stage.

The initial flowchart contains about half of the information required (the *structure*, but no *content* as yet). Because the rest, such as branch addresses to other parts of the program (for `GOTO`, `GOSUB` and so on), are defined during the often lengthy process of producing the code. THE LAST ONE demands your constant involvement, stopping every now and again to ask you questions about how you want some particular part handled. This works well for the new user who will want to build up the general layout as a whole, before having to enter the fine

detail. But if you are an expert you will have (definitely should have) designed the system in its entirety beforehand and you may well prefer a package that takes all this information in at the beginning and then lets you go off for a cup of coffee.

THE LAST ONE can design practically anything from complex indexed-file handling of business systems to spreadsheets and calculators. At its simplest you can define a program to multiply two numbers together and display the result, although the generated program will be about 30 lines long since it will include input and error checking on a scale unlikely to be used by any programmer.

Packages like THE LAST ONE tend to employ straightforward sequential searches to find particular records, anything more complex often has to be defined by the user. This is fine for an experienced programmer but not for the first-timer who usually has little understanding of complex filing and searching structures.

Packages aimed squarely at the systems house and designed to produce a business system complete with menus, reports and input screens tend to be far more structured. **ISAM** (index sequential access method) and **AID** (amend, insert and delete) routines are probably automatically provided. Of course these extensive facilities will take longer to master.

As a general rule all these (we hope) labour-saving program generators still work best for programmers who have put a good deal of effort into preparing the system design, even going as far as to draft out report and screen layouts. The user who has not gone to this length will probably generate an inefficient system or waste much time in front of the terminal wondering what goes where. The more professional code-generators are not very kind to users who don't know their own mind.

Verbs, nouns and objects

Several program-generators are designed to create a particular kind of program: for example, a variety of games (*qv*) generators are available for home micros. These usually break up the standard features of a game (like background, foreground, moving characters) and allow the user to program them using a variety of drawing, painting and animating routines.

One of the most spectacular games generators is Bill Budge's *Pinball construction set*, which draws on graphics-based design principles that later came to popularity as the 'front-end' for many 16-bit business systems. The kit consists of an empty pinball table drawn on one side of the screen, with a selection of components on the other. By using the joystick to move a cursor shaped like a pointing hand, you can pick up components, place them on the table and immediately play a game to test the effect.

It provides a stunning example of the power of 'object-oriented' languages like SMALLTALK. This language-cum-operating system, developed on a mainframe by Xerox, enables the user to build up programs out of 'objects'.

An **object**, used in a rather technical sense by the Xerox programmers, is a combination of program and data – or if you like, a fusing of a 'thing' with its 'function'. Bill Budge, for example, includes a paintbrush among the components with which you create your game. In the sense that you are able to move the paintbrush around with the joystick it is a 'thing', but because it also embodies the concept of colouring the ground it passes over it is also a 'function'. In programming terms, an 'object' can be manipulated as if it were data, but at the same time, like a procedure, it can define the particular manipulation that is to take place.

Object-oriented languages are often contrasted with **procedure**-oriented languages like PASCAL and BASIC which depend on distinguishing data and procedures – the 'nouns' and 'verbs' of programming. Programmmers who have been brought up to think along these lines find object-oriented languages very hard to grasp and fail to see that the philosophy actually reflects the processes of real life far more accurately. When you read a newspaper (or this book), are you taking in data, or are you being programmed?

A programmer's menagerie

When an error turns up in a lengthy piece of software you will often have considerable difficulty in finding exactly which part of the program is responsible. At the University of Alaska, Edward Gauss watched the way his northern neighbours catch wolves and suggested the **wolf-fence algorithm** for debugging.

The ground rules are:

> Somewhere in Alaska there is a wolf to find.
> You may build a wolf-proof fence which divides Alaska as required.
> The wolf draws attention to its presence by howling loudly.
> The wolf does not move.

The algorithm is then:

> Decide on the territorial scope of your search. Initially this will be all of Alaska.
> Construct a fence along any convenient natural boundary, dividing the territory into two smaller regions.
> Listen for the howls; these tell you in which of the two regions you have trapped the wolf.
> Repeat the first three steps until the wolf is trapped in a tight little cage.

Any convenient print instruction can serve as a wolf fence. It should display its location so that the output can be identified uniquely. In BASIC this might look like:

```
1234   PRINT "WOLF FENCE AT LINE 1234"
```

You add the wolf fence to a program, run it and examine the output. The howl of the wolf – the report of the error – will occur either before the fence is reached or after. You erect additional fences until you clearly see the exact location of the fault. And of course there is nothing to stop you from speeding up the process by raising several fences each time round.

If you're familiar with some computer-science techniques you will recognise this idea as a variation on the **binary search** method. This is fine as far as it goes, but wolves that obligingly howl loudly are really pretty easy to catch. There are other bugs in programming that are much more wily and subtle – *foxes*, for example.

Foxing the foxes. Foxes don't announce their presence by howling. They keep quiet until the final results are printed, when the programmer discovers that some of the chickens have been eaten. To trap a fox you have to lay bait in the form of extra printouts of the program's intermediate results. The fox is located when it first takes the bait, causing incorrect intermediate values to be printed.

Coyotes and other animals. There may also be 'coyotes', which howl until a fence is built nearby and then become silent. This may occur when an array subscript goes slightly out of bounds, overwriting adjacent variables. Building a fence might move these variables out of harm's way, so that the coyote stops howling.

The **Cheshire cat** is even harder to corner. This animal appears and disappears and can imitate another beast when it is not invisible. A Cheshire cat is often caused by uninitialised variables: these contain random values, which cause unpredictable failures when the program is run.

Adding instrumentation

Programs that take up more than a few pages of source code are often better written step-by-step in separate modules, each consisting of a number of logically grouped routines and functions. During the development stage it is a common procedure to test each routine before programming the next. This methodical approach will help to make sure the bugs don't run rife, but even so, one part appearing to work correctly under an individual test does not mean it will work 100% correctly when run with the rest of the program.

In order to find out what is going wrong it may be necessary to build instrumentation into your program. Just as a car has a panel of dials to keep the driver informed about the state of the machine, so it is possible to add diagnostic lines to a program to persuade it to report on the changes in its important variables.

In its simplest form instrumentation can be the equivalent of warning lights, requiring only a single line of code. For example if a routine, called `SCAN` expects as input the variable `X` with a value in the range 1 to 10, a single line inserted before the main body of the routine could be:

```
IF  X<1 OR X>10  THEN PRINT "X out of range in SCAN "; X
```

A step up in the sophistication of the monitoring instruments may be helpful when the program manipulates tables or records (or other large data objects). One possibility is simply to list the appropriate table on the screen as soon as it has been filled. However this would throw any other output on the screen away, could take a tedious length of time, and could obscure other faults which may appear.

A better way is to design the screen layout so that one or two lines are specially set aside for displaying diagnostics. A simple one-letter prompt can be written on one of the lines and a few responses programmed. For example, pressing the Return key to the prompt could tell the program to carry on. Or typing a number could print the contents of that record in the free lines and then redisplay the prompt, allowing any number of records to be reviewed before the program proceeds. Rather than being just a single line of code, this obviously requires a small routine.

For any routine the two most important questions that instrumentation can answer are:

What is the **input**?
What is the **output**?

But other data can also be of use to programmers. When testing a new program the priorities are first that it works correctly, secondly that it works efficiently. A rev counter and speedometer on a car are not used to 'debug' it, but to aid in the car's efficient use.

Efficiency of programs can be divided into a number of areas: RAM space required, disk usage, speed and others. But the most important is very often speed – how long does the user have to wait while processing takes place? The speed of a program depends a lot on how good the algorithms are, as well as the machine-dependent factors such as processor speed, disk access time and so on.

A simple method of monitoring a program is to have a set of counts, one for each of the program's routines and arrange that every time a routine is called its count is incremented. You could even display the count values beside the name of its routine in some suitable space on the screen, just by turning a flag on. For example, a couple of lines could be added to the `SCAN` routine:

```
SCANCOUNT = SCANCOUNT + 1
IF BUGFLAG THEN PRINT "SCAN: "; SCANCOUNT
```

This sort of information can help to debug and improve your program. If any particular routine is called often, then improving the program should concentrate either on improving the algorithm of this routine or reducing the number of calls to it. By knowing the routines most often called, you needn't waste time improving routines that are called infrequently.

Debugging programs can be an enormous task. It requires careful checking, strict attention to detail and deductive reasoning. Sometimes hunting for a menagerie of beasts and bugs can involve more effort than writing the program in the first place, particularly if the program is large.

If you have to resort too often to bait, wolf fences and debugging instrumentation, it may well be that you should get out of Alaska and start again, putting your debugging effort into a more thoughtful planning of your program.

The 64 BASIC is somewhat lacking in structured features, but that makes it more of a challenge to write a well laid out program. It is simple enough to rewrite an IF-THEN-ELSE statement using only an IF-THEN form:

```
100 IF B=5 THEN C$="NO" ELSE C$="YES"
```

becomes

```
100 IF B=5 THEN C$="NO" : GOTO 120
110 C$ = "YES"
120 . . .
```

or better:

```
100 C$="YES" : IF B=5 THEN C$="NO"
```

REPEAT . . . UNTIL and WHILE . . .WEND statements have to be replaced with FOR-loops, but don't be tempted to jump over the NEXT statement before the loop has terminated:

```
100 FOR I=0 TO 100
110 IF A(I)=0 THEN J=I : I=100
120 NEXT
```

This could replace a REPEAT . . .UNTIL A(I)=0 statement, and a WHILE . . .WEND statement (where the body of the loop will not be executed if the condition is false at the start) can easily be produced by adding a test before the loop to see if it needs to be performed:

```
100 IF N=0 THEN 140
110 FOR I=0 TO N
120 IF A(I)=0 THEN J=I : I=N
130 NEXT
140 . . .
```

The 64 has many facilities such as the VIC and SID chips which are accessed by means of PEEKs and POKEs. It is important to realise that sooner or later you are going to upgrade to a better machine and that the new machine will have different addresses, and so your programs should be as easy to change as possible. We can do this by replacing the address in a POKE statement with a variable, so that a program controlling the SID chip can specify a variable SID:

```
100 SID = 54272
```

and then reference all the SID registers by POKE SID+5,37 or POKE SID+23,71. You can of course do the same sort of thing with the VIC chip and other important locations in the machine. Apart from making it easier to change the program if you upgrade your machine, having a meaningful variable name like this means that when you pick up the program in six months time – a POKE SID+5,37 is obviously to do with the sound chip, but could you say immediately what POKE 54277,37 does?

64 BASIC has no way of passing parameters to subroutines, and no way of declaring local variables within a subroutine, so it is doubly important to keep a written specification of each routine saying what parameters it uses, and which variables it changes. Similarly you should keep a list of every variable saying exactly what it is used for. A bug in this innocent-looking line:

```
100 FOR I=0 TO 30 : GOSUB 34000 : NEXT
```

where subroutine 34000 changes the value of I can be very difficult to find.

Random Number: Element Of Uncertainty

There is no such thing as a solitary random number – the concept only has a meaning when it is part of a series of numbers. 'Random set' or 'sequence' are the more normal technical names.

What is random?

A sequence of numbers is said to be random where each number is determined purely by chance. There is no relationship between the numbers apart from the fact that the sequence will have some kind of what is called 'distribution'. This term is a measure of the probability that each number will fall into any given range. In a random sequence of digits between and including 0 and 9, each will be one of the 10 possible choices, and there will be a given probability associated with each digit. If a sequence is generated by say, throwing a ten-sided die (instead of the more normal six-sided one), we could expect to get each digit with the same probability of 0.1.

This example illustrates two things: that the sequence generated is random; and that the sequence is characterised by its distributions, the relative chances of getting each of the possible numbers in the sequence. We could, in the extreme case, think of a ten-sided die that was very heavily weighted on one face so that it always showed a '6'. Under those circumstances the sequence generated would be 66666666 ... which is not very useful for anything other than the most transparent form of fixed gambling!

There are circumstances in the use of computer-generated sequences of random numbers where several different kinds of distributions are useful, but in practice these are usually derived from the so-called uniform distribution where each number in the sequence occurs with an equal probability (just as with the unbiased ten-sided die).

Using random numbers

Most obviously in microcomputing random numbers are essential to games, supplying the element of chance that gives a game spice. They can be used to produce surprising sounds as part of a music program, or to animate a graphic display. But the random element has a role in other applications as well, not least in simulation exercises, where you may want to test out what happens in a randomly selected set of circumstances.

One of the fundamental aspects of computer aided design is the ability to use the computer to simulate the operational use of the finished article. These simulation techniques rely heavily on random number sequences to supply a large amount of test data which can be used to measure results. Randomly generated test data is also sometimes the only practical way to test a general purpose algorithm, such as a new sorting subroutine.

Computer decision making, believe it or not, frequently depends on using random numbers. In games theory, for example, where a computer may just not have the information necessary to make a fully determined move (such as in the opening moves of a chess game) it may fall back on a predetermined sequence chosen at random from a library of move sequences, paralleling the way in which a human will approach the same problem.

Make your own

There's an obvious problem in using a deterministic device like a computer, which follows a series of sequential rules, when it comes to generating a series of random numbers. By definition each number cannot be dependent upon its predecessors.

A computer essentially gives the same result every time it performs the same sequence of actions (if yours doesn't, you have a great random number generator but it's not much use for anything else), and so any sequence of numbers generated by using ordinary computing rules cannot be truly random. The best that we can do is to produce a series of pseudo-random numbers. If we take suitable precautions, a sequence like this can be a very good approximation to a truly random series.

Pseudo-random numbers

To make clear what pseudo-random numbers are we'll take a simple technique known as the Linear Congruential Method (LCM), and use it to derive random integers between zero and some suitable maximum. The use of integers simplifies our investigations, but is no limitation in practice. If a random number in the range 0 to 1 is required (it often is) it is simply obtained by dividing the random integer by 10.

Let us take `m-1` as the highest element of our sequence of generated pseudo-random numbers so that there are *m* in all (`0` to `m-1` inclusive). Our objective is to invent an algorithm which will generate each of the `m` integers with equal probability. The algorithm will be a function, and will therefore require input and output, needing a new number each time to 'seed' the generated pseudo-random number. The obvious way to do this is to use the previous pseudo-random number as the starting point. Mathematically what we say is:

$$x_n = f(x_{n-1})$$

We have already broken one of the basic definitions of random numbers, namely that each member of the sequence should be independent of its predecessors. But this is an inevitable feature of pseudo-random sequences, although if we choose `f` cleverly, it will be a pretty good approximation to the perfect random sequence.

An eventual looping of the algorithm is inevitable because there are, as we have said, only `m` possible numbers which can be generated, and each one is fully determined by the one before. It is therefore impossible to generate more than `m` integers without repeating one of them, and as soon as that one is repeated the sequence begins to loop, because that particular number always generates the same one following it and so on.

Of course, there is nothing to say that the loop will always have the maximum length `m`. It could well be shorter, and it is part of the problem in choosing a suitable

generating function f. We can illustrate this with a trivial example: assume for a moment that `m=10`, that is, we are generating the integers 0 to 9 inclusive.
LCM simply multiplies the previous number by a constant, adds or subtracts a constant, and then performs modulus arithmetic on the result with a third constant:

$$x_n=(2*x_{n-1}+2)\text{mod } 10$$

'Mod 10' means the remainder when the contents of the brackets are divided by 10 (for example 14 mod 10 is 4 and 1979 mod 10 is 9). Start by choosing **1** as the first 'random' number; the generated sequence is then:

$$x_1=1$$
$$x_2=(2*x_1+2)\text{mod } 10$$
$$=4 \text{ mod } 10$$
$$=4$$
$$x_3=(2*x_2+2)\text{mod } 10$$
$$=10 \text{ mod } 10$$
$$=0$$

Similarly we derive $x_4=2$, $x_5=6$, $x_6=4$, $x_7=0$ and so on. The full generated sequence of numbers is thus:

```
1 4 0 2 6 4 0 2 6 4 0 ...
```

and we see that it begins to loop immediately after generating the first number. The number of generated numbers inside the loop is known as the 'period' of the sequence. In the example the loop consists of the sequence 4 – 0 – 2 – 6, and the generated numbers are therefore said to have a period of **4**. A low period implies a low degree of randomness, as we can see from this example.

But a high period does not necessarily imply a high degree of randomness. If we want our computer to simulate the throwing of dice and therefore to generate the numbers 1 to 6 on a random basis (or 0 to 5 and add 1 to the result), the maximum length of period we can expect is six. But by choosing our constants carelessly, making the function:

$$x_n=(x_{n-1}+1) \text{ mod } 6$$

Starting with 0 gives us the sequence:

```
0 1 2 3 4 5 0 1 2 3 4 5 0 1 ...
```

which clearly has a period of 6 but equally clearly is not in the slightest bit random.

So although a large period is really essential if we don't want the generated numbers to loop too soon, we still have to supply the function with the right constants. You can experiment with this formula yourself to find values that give the most convincing pseudo-random sequence.

Most BASICs include a `RND` function to return a decimal number in the range zero to one, and many of them use LCM, often with an additional BASIC command to change the seed, the command `RANDOMIZE`.

Genuinely random

True random number generation is possible on a microcomputer, but an external variant is essential to introduce the element of genuine unpredictability. One way this might be done would be to measure the heat of one of the circuit boards to an extremely high degree of accuracy. You might then use the 20th decimal place (or whatever) of the temperature as a basis for the next random number in the sequence. Weather variations and the many operating conditions that affect temperature suggest this as a very effective system, although it does depend on some rather expensive and not readily available thermometric hardware.

The most readily detectable source of unsynchronised input is our own communication through the keyboard. Measured in micro-seconds the variation in time between key-strokes is high, even for a skilled typist. From this raw input a simple, continuous and effective stream of true random numbers can generated.

On many micros that use polled keyboard input, whenever a key is touched a flag is set somewhere in the circuitry between the keyboard and the processor, and it remains set until the processor picks up the character code. A flag is simply one bit of a memory location or register capable of being in one of two states (set: 1 or reset: 0). When the processor retrieves the character it resets the flag. The process of testing the flag to see if a character is waiting, retrieving the character, and resetting the flag happens extremely quickly (10 to 20 micro-seconds). The processor could therefore cope with a stream of input of well over 25,000 characters per second.

A good typist can type around 10 characters per second – most of us are a great deal slower – so for the rest of the time the processor is sitting idle, waiting for the human to do something. This idle period therefore is a good time to generate random numbers. The processor is going monotonously through the loop:

```
Test the keyboard flag
Is it set? – No
Go back to test it again
Test the keyboard flag Is it set? – No
. . . . . and so on.
```

But this can be modified to:

```
Test the keyboard flag
Is it set? – No
Increment a random counter
Go back to test it again
```

When a key is finally hit (say after an interval of 0.2 seconds, or 10,000 loops), the character is retrieved and the random count stored elsewhere for later use.

Obviously a variation in time between key strokes from 0.2 to 0.21 seconds will be sufficient to generate unpredictable random counts. In practice, the time variation is usually far wider, from a sharp 0.1 seconds to daydreaming minutes. For every key stroke a random count is produced. In fact, it is safe to take two randoms from each key stroke, the upper and lower byte from each count.

Once generated, these random counts can either be used immediately, ignored, or stored in a table to allow randoms to be collected before they are required. The

demand for randoms may be greater than the current supply of key strokes, in which case a table of the last 128 randoms generated should ensure a ready supply. However, that does not mean the possibility of the table running out should be ignored. As a safety device, a check should be made to see if the table is nearly empty, and if it is, pseudo-randoms are generated when required.

To seed these pseudo-randoms is is enough if one genuine random is left in the table. As soon as more key strokes are made, the table can start filling up again with genuine randoms. One important advantage of this system is that the user is never explicitly asked to type keys for the sole purpose of generating randoms – it is all done invisibly.

Some computer manufacturers have recognised the need for randoms, and have a keyboard loop counter programmed into the computer's ROM. For the benefit of those users without this facility, the keyboard input routine is included in the listing below. The code is given in a pseudo-PASCAL format sometimes called PDL (Program Descriptive Language), which is supposed to be helpful in explaining algorithms, without being dependent on any one language's syntax. The text in italic is comment.

In order that the loop is executed sufficiently often, the `READ CHARACTER` routine should be written in Assembler.

The following memory space is declared:

```
RANDOM TABLE:an area of 128 bytes
NEXT RANDOM: a pointer to the next empty
             location in the random
             table
COUNTER 1:   a single byte
COUNTER 2:   a single byte
A.           Read character called when a
             character is required by the
             program
```

Then the program itself:

```
BEGIN        Stop any left-over character
             producing a random count
             equal to 0
CLEAR KEYBOARD FLAG;
             Increment random counts until
             keyboard flag is set.  Note
             that when a count reaches 255
             and 1 is added to it it
             becomes 0
REPEAT
COUNTER.1=COUNTER.1+1;
IF COUNTER.1 = 0 THEN COUNTER.2 = COUNTER.2+1
UNTIL KEYBOARD FLAG SET;
RETRIEVE CHARACTER CODE;
CLEAR KEYBOARD FLAG;
IF NEXT.RANDOM < 126 THEN table not full
BEGIN
    RANDOM.TABLE   (NEXT.RANDOM) =
    COUNTER.1;
    NEXT.RANDOM = NEXT.RANDOM + 1;
    RANDOM.TABLE    (NEXT RANDOM)   =
    COUNTER.2;
    NEXT.RANDOM=NEXT.RANDOM + 1;
END;
END OF READ CHARACTER;
B.RANDOM     Called whenever a random is
             required.  Range of required
             random is 1 to MAX-R
RANDOM=RANDOM.TABLE(1);
WHILE RANDOM > MAX-R
DO RANDOM - RANDOM - MAX-R;
             Shift the table down one space, so
             the next random is in the bottom
             of the table.  if the table is
             empty, a pseudo random is
             generated instead
IF NEXT>RANDOM > 2 THEN   table not empty
BEGIN
    FOR LOOP = 1 TO NEXT.RANDOM - 2 DO
    BEGIN
        RANDOM.TABLE(LOOP) - RANDOM.TABLE
       (LOOP + 1)
    END;
    NEXT.RANDOM - NEXT.RANDOM - 1;
END
ELSE         If table is empty, just mush
             last number around a bit
    RANDOM.TABLE(1) = RANDOM.TABLE (1)*5 + 25;
END OF RANDOM;
```

The program assumes that a key will always be hit before a random is required. If it is possible that this will not be the case, a seed value should be placed in RT(1), and NEXTR set to **2**.

The first text given with the program shows the random number generated with each key hit. The second text shows how the random table is filled, then emptied, then how pseudo randoms are used.

The 64 uses the linear congruential method described in the text, and gives quite a good pseudo-random sequence. Many people find it difficult to start a random sequence, so that their games always begin with the same events. But 64 users can tap into the time variable and start the program with a RND (–TI), thus beginning a different random sequence each time. Another common requirement is to generate a random number in a given range X.

I = INT (11 * RND (0))

returns a number in the range 0 to 10 (remember that the RND () function can never actually reach 1).

Software Design: Coding For Human Beings

Even with the new generation of graphics displays and with mouse-driven software promised by the advertisers and magazines, too much software is still written with the computer in mind – and the human operator a very secondary consideration.

Before the invention of the microprocessor in the early seventies computing was an elaborate ritual performed among highly paid professional computer users and rooms full of costly humming hardware. No wonder a sort of priesthood developed around it, and with it an esoteric aesthetic for addressing the machines, a cryptic shorthand totally incomprehensible to the layman.

Now we have that ugly phrase 'user friendly', and a mish-mash of micro software trying to live up to the description. Because there is no coherent design philosophy across the spectrum of micro software, the result can, depending on your temperament, be seen either as a Tower of Babel or a rich tapestry of opportunities for improvement.

Up-front v. Deep-down

Many of the new 'windowing' and mouse-driven packages illustrate rather well the 'up-front' school of thought that puts a great deal of design effort into screen displays and data entry routines. Packages designed along these lines can be a joy to use, as long as the effort proves genuinely helpful to the long-term user, rather than – as it often is – simply a marketing device to make the software look super-friendly during the demonstration.

There are still a lot of computer systems in use that are not graphics oriented, are driven mostly by keyboard commands, and whose output is chiefly lines of text. Packages designed for systems like these can still take advantage of good ergonomic principles, although many software producers don't take the trouble. Programmers often seem to lack experience in using software in real life, and don't realise how time consuming clerical activities can be speeded up for the user. Why should you have to type in the names of existing files, for instance, when with a little thought the routine could be rejigged to pluck file names from the directory simply by moving a cursor?

Deep-down design takes all this into account, it is more concerned with creating a living tool that will grow with the user's needs. The emphasis is less on spectacular graphics effects and more on economy of effort for the user. Economy of effort for the computer also has to be taken into account, because good designers are aware of the inevitable trade-off between 'up-front' software cosmetics and the speed and flexibility that internal design economy can bring. Graphics makes huge demands on the processor and memory. It is no good having a super-friendly system that draws helpful pictures all over the screen if it takes all day to do it.

But sensible screen-handling is only one aspect of good software design. Here are some other considerations that apply in general to a whole variety of different software packages:

How easy is the software to install? Does the designer show reasonable foresight about the problems and opportunities presented by your hardware?

How well does the software do what you want it to do? Do you have to adapt your work methods, or can you mold the program?

Your relationship to the software is going to change as you get to know it better, and as your own needs change. If there are copious help messages, will you be able to override them when familiarity makes them unnecessary? If you upgrade to a hard disk how easy will it be to reorganise the program for its new environment?

How well will the package fit in with the other software you're running? Are you going to have to learn several different sets of cursor control commands, or can you modify all your programs to work in a similar way? What data format does the program read and produce? Is there any hope of compatibility between, say, the new spreadsheet package you've acquired and the word-processor you're already familiar with?

Warning: Lone genius at work

The worst software design often appears in packages where the writer is a lone operator. Intelligent, deeply knowledgeable about one particular hardware installation, but with only a hazy idea of what other people are using, and what they are using it for, a software writer of this kind can be dangerous to your sanity. Writing software (even at the games (*qv*) cassette level) is a highly professional business requiring teamwork, method and thorough testing.

Programs like this may even be uninstallable on your equipment in the first place. It is not unknown for widely advertised packages to overlook the fact that many video terminals position the cursor by sending the X-coordinate first and then the Y-coordinate (the scientific convention) rather than the business-oriented approach of sending the Row followed by the Column. If you're using a terminal with the wrong sort of cursor addressing the program won't run.

There are sometimes design idiosyncrasies in the input routines. If the programmer never uses lower case, routines used to exclude undesirable control codes may make input impossible unless the shift key is latched. One notorious package waits until you have finished filling in the line then, if you've used lower case, wipes out your entry and sits there dumbly with the cursor repositioned at the beginning of the line. No error messages, no explanations, no assurance that something hasn't gone horribly wrong with the program.

Tucked away in the manual is an explanation for this sulky behaviour – or rather the bald statement that you *must* enter in upper case (why?). But constant visits to the manual are often a sign of poor software design. Many well-thought-out menu-driven programs can be used without any documentation, managing instead to

boil down the wisdom of the manual and serve it up within the programs themselves – needless to say, a logical and consistent structure in a program will make it considerably easier to use.

A menu is particularly useful for finding your way around a facility you call upon rarely, and it saves having to go back to the manual each time to remind yourself what it is all about. At their worst, menu-driven packages can make you feel a little like being cross-examined by the FBI. As a memory-jogger for the occasional user menus are very appropriate.

Help that hinders

There's a lot of sense in assuming the user needs as much help as possible, but the help mustn't be intrusive when it's not needed. Using software that holds your hand while you put data in – then offers you sheaves of options when the time comes to get data out – can be rather like driving a car with a fully qualified driving instructor beside you. It's fine while you're learning, but eventually when you pass your test and are navigating like a veteran you find the instructor is still glued to the passenger seat, and his conversation becomes monotonous.

Some packages let you turn off the conversation, WORDSTAR manages this rather well, offering a choice of help levels varying between 3 (beginners) and 0 (no help). A similar approach, coming to be used in word processors and spreadsheets, relies on timing instead of overt help level setting: if you hesitate between commands it assumes you want help.

This approach helps to blur the distinction between menu-driven and direct control-driven program steering. For example, the command to copy a region of text to the holding buffer might be a sequence of three characters: Control-X, Control-R, Control-C. The beginner who pauses after the first character will encounter the main menu.

```
          Main Dispersal Menu
          ~~~~~~~~~~~~~~~~~~~

(B)uffers              (R)egional Files
(C)apitalization       (S)et Help
(M)iscellaneous        (W)indows Layout
           e(X)it from menu
```

which points the way to the next menu by way of Control-R (or **R** or **r**). A second pause after this entry produces the 'Regional Files menu' from which we might, for example, select **G** – 'go to marker':

```
             Regional files menu
             ~~~~~~~~~~~~~~~~~~~

(A)ppend to KILLS      (D)elete to marker
(O)utdent              (U)ndelete
(C)opy to marker       (G)o to marker
(S)et marker           (I)ndent
           e(X)it from menu
```

But the sequence **R** – **C** executed as an uninterrupted string produces the same result without displaying either of the menus, so that once the commands become instinctive the software adapts unobtrusively. A programmable keyboard can even be customised by loading the character sequences into the function keys.

Verbose or succinct

Most programs expect input from the user at some form of command line prompt or from a file, and from time to time will put up messages on the screen. There are two distinct schools of thought about how this should be done. The verbose school maintains that data should be exchanged between the user and the program in the form of clear text, as clarity is more important than succinctness.

This has the advantage that the commands themselves more or less explain what they're up to, but doesn't exactly save your disk space or your typing fingers. In contrast the alternative 'succinct' command style promotes the idea of controlling a program with as little input as possible, and restricting error messages and screen reports to the minimum. This is very much the inspiration behind the operating system UNIX (*qv*).

Well designed verbose software anticipates the fact that the user may tire of writing out instructions in full, and allows one- two- or three-letter abbreviations of commands. An approach which takes the best of both worlds approach isn't hard to implement, one wonders why more software originators don't use it.

Sorting: Heaps And Bubbles

Sorting has puzzled mathematicians since the ancient Greeks. Put at its simplest, the problem is how to organise a list into some sort of sequence, which may be, though is not necessarily, alphabetic or numeric.

Sorting is useful for three reasons:

Information in long lists is easier to understand if it is in a logical order.
Two files containing similar types of data are easier to merge if they are sorted into the same order. They can be processed in tandem, with one updating the other.
A sorted file is easier to search. Instead of having to examine each item individually, a program like the binary chop can find individual items quickly and easily.

In computing terms the list and the items in it will probably be represented as a disk file containing records. In a telephone directory the equivalent of a record would be the combination of name, address and telephone number that makes up each entry. Each element of this combination is called a field, and in order to sort the entries, one of these fields (or sometimes more than one) has to be selected as the *key*. In a normal telephone directory the names are arranged in alphabetical orders, so the *name* field can be said to be the key.

Generally speaking the key is the piece of data that will be used to put the record in its correct position. It might be a number or a string of letters. Most BASICs have a facility for telling that, say, a word beginning with M is alphabetically later than a word beginning with G. You can normally use the greater and less than operators (< and >) with alphabetic characters – so that `B>A` is legal in most versions of BASIC. The value of the ASCII code for an upper case letter is 32 less than its lower case equivalent, so you can't sort a mixture of upper and lower case letters without taking this into account.

Over the years, many different, and often cunning, algorithms have been devised to permit rapid and efficient sorting of data. The motives of the designers varied from the commercial to the purely academic, but the modern requirements are normally for a sort algorithm which can be executed on a computer in the shortest possible time, using the fewest computer resources.

Unfortunately, no single sort algorithm performs best under all circumstances. Some which run very rapidly on randomly ordered input are incredibly slow if the incoming data is already ordered or nearly so!

Sorting algorithms

At this point we need to distinguish between external and internal sorts, because some of the characteristics are different.

External sorting orders the records in a file on some suitable chosen key, writing the result to an output file, and often using intermediate work files during the process.

Internal sorting orders records held in a program. All external sorts contain an internal sort, and work by reading in a large number of records, sorting them internally, and writing them out before processing the next group.

The **bubble sort** is the method of sorting most frequently quoted in the text books, and it makes a useful introduction to the subject. The bubble sort uses two rules:

Take the first pair of records and put them in correct order. Move one position down the list and repeat the process. Continue until the end of the file is reached. This whole process is called a 'pass'.
At the end of the file assess if swaps were made during the previous pass. If they were, return to the top of the file and make a complete new pass. If they weren't, stop. The job is done.

Let us assume we have a table of *n* records `R(1)` to `R(n)` in our program. We're sorting into ascending key sequence, although of course sorting into descending sequence presents no problem. Part of each record is the **key field**, and for convenience we will assume that the key is a contiguous set of fields. Let us call the key area in record `R(i)` by a convenient abbreviation, say `K(i)`.

To illustrate the algorithms we need a simple example set of keys to be sorted. Figure 2 shows a set of 10 randomly chosen numbers in the range of 0-99.

The bubble sort is simple: we run through the table looking at adjacent pairs of records, at each pass achieving a certain improvement in the ordering. In our example, the first pass proceeds as follows:

Compare 31 and 47; correct order so no change.
Compare 47 and 26; 47 is higher so swap corresponding records. List now reads, 31, 26, 47, 07 ....
Compare 47 and 07; 47 is higher so swap corresponding records.
Continue the comparison through the table.

The result of the full pass is shown as the second line in Figure 3. If we repeat the operation with second and subsequent passes we obtain the results shown in the rest of that figure. During the fifth pass no instance is found where it is necessary to swap the records and the table is therefore confirmed as being completely sorted.

Closer investigation shows that it is not necessary to scan the whole table in every pass. The first pass guarantees that at least the highest number reaches its final position in the table. Each subsequent pass ensures that at least one more item comes to its ultimate resting place. It is therefore possible in any one pass to terminate the scan at the last record which is known to be in its final position. The formal algorithm to express this appears in Figure 4. We can see now why the sort is called a 'bubble' sort: in each pass another high value 'floats' further up the table.

At first glance it might seem impressive that the bubble sort algorithm can sort our 10 numbers (by which, of course, we mean the 10 records of which these numbers are the keys) in only 4 passes of the data. If the data is

already sorted you can see that the algorithm will only perform a single pass of the data.

Unfortunately, appearances are deceptive. If the incoming data is already in *descending* key sequence it is necessary for the algorithm to perform $n-1$ passes of the data to achieve ascending key sequence as it only moves a single data item to its final position in each pass of the data. This is shown for our example numbers in Figure 5.

With randomly ordered input data, the bubble sort needs to make more passes as the number of data items grows. In other words the number of passes depends on n. Furthermore, the time of execution of a single pass obviously depends upon the number of records in the table – if we double the size of the table a single pass will take twice as long.

Combining these two factors tells us that the execution time, with randomly ordered input data, varies as n squared. If we double the number of records we quadruple the execution times – there are few sorting algorithms which are less efficient than this!

Sorting heaps. There is another internal algorithm that has shown itself to be a reasonable performer under normal circumstances, and which will far out-perform the bubble sort. It is fast in execution and uses very little memory either for instructions or for working space (*see* Figure 1).

This algorithm is called a heap sort, it appears in full in Figure 1. It is very concise and consists of surprisingly few steps. However it is not likely to be clear to you why it should actually perform a sort at all (unless you have met it before)!

The algorithm operates in three stages. The first is to organise the data input into a form which is internally convenient, the second is to pick out the data from that internal order in the final order we wish to achieve. We also have to consider how we get the incoming data into the special order in the first place.

A heap – so dubbed by the algorithm's designer, J.W.J. Williams – is the name used to refer to our table of records when their order is such that:

```
K(1) > K(2) and K(1) > K(3)
K(2) > K(4) and K(2) > K(5)
K(3) > K(6) and K(3) > K(7)
... and so forth
```

or in general:

```
K(i) > K(2i) and K(i) > K(2i+1)
```

This implies that the first record is the one with the highest key and that there is a clearly structured order of records following it. The clearest representation is that of a **tree structure** (Figure 6). Just as record 1 has a higher key than records 2 or 3, so record 2 has a higher key than records 4 or 5, and they than record 8/9 and 10/11 respectively, and similarly record 3 has a key higher than those of the records beneath it in the tree.

It might seem strange that the use of such an intermediate order makes the sort process simpler, until we consider how we can extract data in the finally sorted order from heap. Simply take the first record in the heap and swap it with the record at the end of the heap. The

1. Set up integers START, END, i, j.
   Set up a working storage area W which will hold one record, and with the key field area inside it defined as k.
2. If N ⩽ 1 no sort is needed.
3. Set the heap limits:
   START = N ÷ 2 + 1 (where ÷ implies rounding down)
   END N
4. If START > 1, we are building the Heap:
   Set START = START – 1
   W = (IR (START)
   If START = 1 we are in stage 2, creating the output sorted data from the Heap:
   SET W = R (END) pick up last record
   R (END) = R(1) swap with first
   END = END – 1 Heap now 1 entry smaller
   If END = 1
   Set R(1) = W
   End of algorithm; data is now sorted.

Steps 5–10 now insert record START into the correct position in the Heap.

5. Set j = START
6. Set j = j*
   = 2 j
7. If j < END go to step 8
   If j = END go to step 9
   If j > END go to step 11
8. If K (j) < K (j + 1) Set j = j + 1
9. If k ⩾ K (j) go to step 11
10. Set R (j) = R (j)
    Go to step 6
11. Set R (i) = W
    Go to step 4

Notes to accompany Heapsort algorithm

Variables

N = the number of records in the table
R = the contents of one of the records
K = the key to one of the records
START = the position of the first record in the pass
END = the position of the last record in the pass
i = the position of a record
j = the position of the record with the key of greater value below record in the heap
W = a working space (in Basic, a variable) used to store the record being worked on
K = the key to the record in area W

Notes

In the formula K(x) or R(x), where x holds the value of i, j, START or END, K refers to the key to the record at position x, and R to the record itself at position x.

Fig. 1. Heapsort algorithm

31	47	26	07	95	10	16	78	85	32

Fig. 2. Test data; ten random numbers

Start	31	47	26	07	95	10	16	78	85	32
After Pass 1	31	26	07	47	10	16	78	85	32	95
After Pass 2	26	07	31	10	16	47	78	32	85	95
After Pass 3	07	26	10	16	31	47	32	78	85	95
After Pass 4	07	10	16	26	31	32	47	78	85	95
After Pass 5	07	10	16	26	31	32	47	78	85	95

Fig. 3. Full pass of Bubblesort

1. Set integer TOP = N (number of records in table)
2. If TOP ≤ 1 sort is complete
3. Set integer SWAPPED =0; will be set to position of last record moved nearer start of table
4. Set integer i = 1
5. If K (i) ≤ K (i + 1) go to step 8
6. Swap K (i) and K (i + 1)
7. Set SWAPPED = i
8. Set i = i + 1
9. If i < TOP go to step 5
10. If SWAPPED = 0 sort is finished
11. Set TOP = SWAPPED
12. Go to step 2

Variables

TOP = the position of the last unsorted record before a pass of the table
SWAPPED = the last record to be moved nearer the beginning of the table in one pass
K = the key to a record
N = the number of records in a table
i,j = the position of a record in the table

See notes in figure 5

Fig. 4. Bubblesort algorithm

larger key record moved will then be in its final position and we merely reorganise the rest of the heap (which will now be slightly varied from a heap by virtue of our having moved a new record to the front) so that it is a heap again, and then repeat the process until the heap has shrunk to a single element, at which point the table has been sorted. The logic which we use to achieve the reorganisation into a heap is identical to the logic we use to create the heap in the first place.

We achieve the heap initially by a simple iterative process. The second half of the records in the table (which corresponds to the bottom row of records, number 8-15 in Figure 6) must form a heap in the trivial sense that we are not yet considering records in the upper half of the table (numbers 1-7 of Figure 6). Then include one record from higher up the table (this would be number 7 in Figure 6), and swap records 7, 14 and 15

Start	95	85	78	47	32	31	26	16	10	07
After Pass 1	85	78	47	32	31	26	16	10	07	95
After Pass 2	78	47	32	31	26	16	10	07	85	95
After Pass 3	47	32	31	26	16	10	07	78	85	95
After Pass 4	32	31	26	16	10	07	47	78	85	95
After Pass 5	31	26	16	10	07	32	47	78	85	95
After Pass 6	26	16	10	07	31	32	47	78	85	95
After Pass 7	16	10	07	26	31	32	47	78	85	95
After Pass 8	10	07	16	26	31	32	47	78	85	95
After Pass 9	07	10	16	26	31	32	47	78	85	95

Fig. 5. Worst case of Bubblesort in action

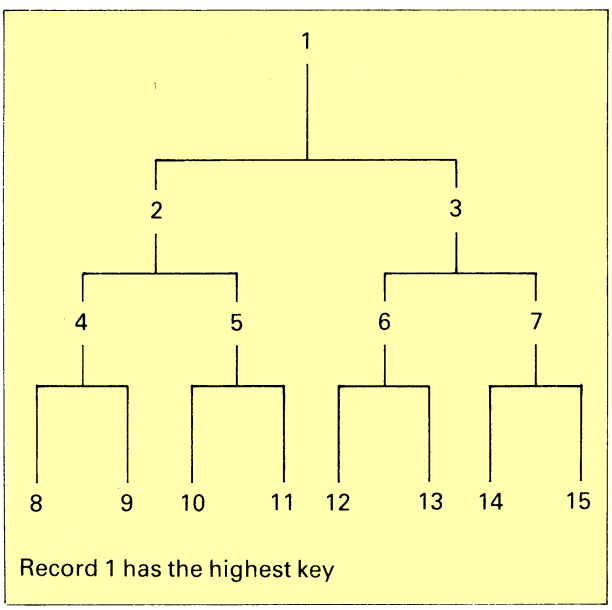


Fig. 6. Tree structure representing order of records prepared for Heapsort

around as necessary to achieve `K(7)>K(14)` and `K(7)>K(15)` so that these 3 records form a little heap of their own. Then include the next record up (record 6) and repeat the exercise with records 6, 12 and 13. Record by record we work back to the start of the table observing the heap organisation of the records so far included until record 1 has been incorporated and the whole table is thus held as a heap.

Let us see how it works in practice with our example of 10 numbers. The processing is shown in Figure 7 with the values of `START` and `END` given so that the formation conversion of the heap into finally sorted data is apparent.

Compare this sort algorithm with the bubble sort. If the comments are left out from the algorithm, the steps of the heap sort are insignificantly more complicated than those of the bubble sort and the memory space used by both algorithms is almost exactly the same. The key point is that neither algorithm requires a separate output record area (which some algorithms do require, thus doubling the amount of table space needed in a program).

We saw how the execution time of the bubble sort

Pass	Start	End	1	2	3	4	5	6	7	8	9	10
0	**6**	**10**	31	47	26	07	95	10	16	78	85	32
1	**5**	**10**	31	47	26	07	95	10	16	78	85	32
2	**4**	**10**	31	47	26	85	95	10	16	78	07	32
3	**3**	**10**	31	47	26	85	95	10	16	78	07	32
4	**2**	**10**	31	95	26	85	47	10	16	78	07	32
5	**1**	**10**	95	85	26	78	47	10	16	31	07	32
6	**1**	**9**	85	78	26	32	47	10	16	31	07	95
7	**1**	**8**	78	47	26	32	07	10	16	31	85	95
8	**1**	**7**	47	32	26	31	07	10	16	78	85	95
9	**1**	**6**	32	31	26	16	07	10	47	78	85	95
10	**1**	**5**	31	16	26	10	07	32	47	78	85	95
11	**1**	**4**	26	16	07	10	31	32	47	78	85	95
12	**1**	**3**	16	10	07	26	31	32	47	78	85	95
13	**1**	**2**	10	07	16	26	31	32	47	78	85	95
14	**1**	**1**	07	10	16	26	31	32	47	78	85	95

Fig. 7. Heapsort in action

varied as *n* squared. In the heap sort the dependence on *n* is far greater. Each item in the table is considered separately, which gives a dependence on *n*; and each time an item is considered the heap has to be reorganised. This re-organisation, however, only affects one leg of the tree structure and therefore causes a dependence not on *n* but instead on the number of nodes in the tree, which varies as $\log_n$. The total dependence of the heap sort is therefore $n \log_n$. This means for example that if we increase the number of records sorted from 10 to 100, the bubble sort execution time will normally increase by a factor of 100, whereas the execution time of the heap sort will only increase by a factor of 20.

It is extremely simple to code an efficient and concise sort routine. Why not try it yourself?

```
100 N=100
110 DIM K(N)
200 FOR I=1 TO N : K(I) = INT(
1000*RND(0)) : NEXT
210 TI$="000000"
1000 IP = N
1010 IF IP <=1 THEN 2000
1020 SWAPPED=1
1030 I = 1
1040 :    IF K(I)<=K(I+1) THEN
1070
1050 :       J = K(I): K(I) = K(
I+1): K(I+1) = J
1060 :       SWAPPED=1
1070 :    I=I+1
1080 IF I<IP THEN 1040
1090 IF SWAPPED=0 THEN 2000
1100 IP = SWAPPED
1110 GOTO 1010
2000 PRINT"SORT COMPLETED IN "
;TI/60:"SECONDS."
2010 FOR I=1 TO N : PRINTK(I)
: NEXT
```

Bubblesort program

```
10 N=100
20 DIM K(N)
100 FOR I=1 TO N : K(I) = INT(
RND(0)*1000) : NEXT
110 FOR I=1 TO N : PRINTK(I) :
 NEXT
200 TI$="000000"
1000 IF N<=1 THEN2000
1010 FIRST = INT (N/2)+1
1020 LAST = N
1030 IF FIRST=1 THEN1070
1040 FIRST = FIRST - 1
1050 W=K(FIRST)
1060 GOTO1130
1070 IF FIRST<>1 THEN1130
1080 W = K(LAST)
1090 K(LAST) = K(1)
1100 K(1) = W
1110 LAST = LAST - 1
1120 IF LAST=1 THEN2000
1130 J = FIRST
1140 I = J
1150 J = 2*J
1160 IF J<LAST THEN1190
1170 IF J=LAST THEN1200
1180 IF J>LAST THEN1230
1190 IF K(J)<K(J+1) THEN J=J+1
1200 IF W>=K(J) THEN1230
1210 K(I) = K(J)
1220 GOTO1140
1230 K(I) = W
1240 GOTO1030
2000 PRINT"SORT COMPLETE IN ";
TI/60;" SECONDS."
2010 FOR I=1 TO N : PRINT K(I)
 : NEXT
```

Heapsort program

Speech: Computers That Talk And Listen

Getting computers to understand human speech and to reply verbally to questions and instructions can help people to do jobs where their hands are not always free, such as airline pilots, baggage handlers, and machine operators. It also has enormous potential for the disabled, and people with impaired sight unable to read a screen. But the implications go a good deal further than that.

Speech recognition

It is considerably harder to get a computer to recognise speech than to synthesise it. For one thing, no two people say the same word in the same way. Apart from the enormous difference in the pitches of people's voices, and the differences in pronunciation, there is also a large variation in the sound of a word according to its context and the emphasis given to it by the speaker. Think how many different sounds we give to the word 'the'.

And how do we define a 'word' to the computer? In speech the sounds tumble together and blend into a continuum that is not obligingly broken up, as is text, by that useful stop and start character the blank space. Human beings can understand words spoken extremely rapidly; even in a normal conversation when we miss several words we can fill in the blanks just by unconscious reference to the context. This is just as well, because laboratory tests show that thirty percent of the sounds which stand for words in ordinary speech are completely unintelligible when delivered in isolation. That last sentence, for example, if spoken aloud in conversation might come out as something like: 'Thizz justice welby cuzzler borrow-tree tesho...'. A computer program which worked on simple pattern recognition would come nowhere near understanding the sense, but would be hunting through its dictionary for the meaning of 'borrow-tree' and wondering what it had to do with Justice Welby Cuzzler.

Once you include the 'blank spaces' the problem instantly becomes so much simpler that even modest micros can cope. Speech recognition products responding to 'isolated utterences' have been around for many years now, and with the appropriate software and hardware it is already possible to teach your micro to recognise a limited vocabularly, and possibly a few short sentences. Two types of system are emerging for micros:

> A standard package which will recognise a few words from a wide range of strange voices, though the speakers will have to talk clearly and deliberately. The vocabularly might typically consist of the numbers one to ten, and a few often-used commands such as 'Print', 'Return', 'Enter' and so on.
>
> A package that allows you to train the computer yourself, with your own choice of words. The computer would only recognise the voice or voices established during the installation phase.

Training the computer when using the second approach is a process of typing in the word you want the computer to recognise, and then repeating it several times into a microphone wired into the computer's speech recognition board. In some systems you may find that you have to retrain your micro each time you use it, because, strange as it may seem, people's voices change significantly, even over short periods. Once you have built up one vocabulary you can add to it by using a key word to access another set of words. So, for example, you would have a limited business vocabulary, but when you use the word 'account' it would bring you into a subset vocabulary of financial terms.

The way the computer recognises the words varies from system to system. Some of the early systems used a form of quasi-speech recognition, where a computer would recognise a very small set of words purely by the difference in length. So for instance, it would notice the difference between 'Yes', 'No' and 'Maybe', but not between 'Banana' and 'Gorilla'.

More recent speech recognition software looks at each word as a whole unit, or breaks words down into smaller units which the phoneticians call phonemes. Either way the different waveforms of each sound will be digitised during the training session for storage as a pattern of bits in the computer's memory and tabulated against the written word. Later, when trying to recognise a spoken word, the software will search the memory for the pattern that resembles it most closely. If there is no similar word in the memory, the system will probably come back with an error message or ask for a repetition.

For obvious, though disquieting, reasons the highest funding for speech recognition has come from government and military sources. It is now common knowledge that the US National Security Agency has developed techniques able to detect key words in transmitted speech (including telephone conversations), and there are rumours that some UK Government establishment such as the ne at Cheltenham may be attempting a similar project. It is one thing to listen in to phone lines and wireless communications, but sifting through the welter of material which accrues (particularly if you start putting your own nationals on the suspect list) either takes a huge workforce – with corresponding problems of security – or immense computing power, before the 'information' can become 'data'.

Rumour has it that Texas Instruments, who have invested huge sums in the application of computers to speaking and listening, have a plant where security is so tight that they use a speech recognition and voice synthesis system to check the identity of all who enter the top-secret area. According to the story, you find yourself in an enclosed room and an attractive female voice invites your response to a number of personal questions. While this is going on your weight is checked by a panel in the floor, and your voice pattern is collated against the computer records.

Whether the story is true or not is immaterial, but it does show how the science of digital speech recognition can bolster the ambiguous growth industry of security. However you might disguise your appearance, forge a signature, and wear gloves to conceal your finger prints, it is impossible to copy someone else's voice exactly.

Medical establishments also show great interest in

speech recognition, but of course come nowhere near commanding the same resources. There is a smattering of small companies working on voice control systems for the disabled, and a few projects at universities and polytechnics. The benefits which will result in cheap and reliable systems for the disabled, will eventually also be available to other micro owners.

Answering back

The synthesis of speech is so much easier than recognition that it is getting difficult to keep track of all the speech synthesis products coming out for micros – add-ons for a wide range of machines varying enormously in price and quality. Speech synthesis is taking off for the home computer user due to the mass production of special chips designed for the purpose.

People have been interested in speech synthesis for well over a century, starting off with early models of the human vocal tract operated by hand. Some systems still try to simulate the human vocal organs using electronic circuitry. Language researchers have calculated the number of different phonemes making up a language, and a chip can be programmed with a library of these sounds, together with the means to piece them together to make words.

But the result is an unhuman sounding voice. A more satisfactory form of machine generated speech is made by 'burning' a digital recording of a real human voice into a chip. But, even when using this method, the sound generated by the computer is a little strange, if only because of the quality of the computer's speaker. The human voice covers a wide range of frequencies, and the complexity of these voice signals can only be reproduced by very sophisticated equipment. On home computers this method brings about a distinct improvement in quality, but only at the price of limiting the words to a set vocabulary.

Most of the speech synthesis products are manufactured in the United States, and those which aren't nearly always use an American speech chip, usually from Texas Instruments. One notable exception is the BBC speech chip, which encapsulates the establishment tones of news reader Kenneth Kendall. Chip manufacturers try to aim for a mid-Atlantic male voice for home and business applications, but in military applications a woman's voice is often used, the idea being that lonely young fighter pilots and ship's navigators might pay more attention to this than to KK.

Putting it all together

The combined use of speech synthesis and speech recognition can make for a friendly computer. Texas Instruments lead the way in 1984 with a personal computer system built around the idea of voice input, with response in the form of speech. A system like this aims to allow you to tell the computer in plain English speech that you want, for example, to make a phone call – whereupon it will look up the recepient's number then dial and keep redialling until it gets a response. The system can also take phone calls in its owner's absence, behaving like an answering machine, but being able to make intelligent decisions based on the caller's input.

Systems like these can apply speech recognition to word processing, although the time is still some way off when the spoken word can be automatically translated into text. But you could, for example, tell the machine you wanted to write a letter to your bank manager. The key word 'write' would trigger the word processor, the word 'letter' would activate your pre-set formatting for correspondence and set up the day and the date correctly, and 'bank manager' would initiate a search through the database for the full name and address, which would then be positioned correctly at the head of the letter.

Speech recognition could even be used during the composition of the letter to pull in 'boiler-plate' paragraphs – chunks of pre-written text that you anticipate will recur in your correspondence and which are filed in a library against a keyword ready for instant recall.

Spreadsheets: How Figures Tell The Story

Electronic spreadsheets are available on many home computers now, but the people who market the software still tend to assume that the user has a diploma in accountancy and only needs to know how to handle the controls and te peculiarities of a particular package. But, unless you grasp the essence of what the software is up to, the refinements – how to format the numbers, how to add new columns or to sort the rows into alphabetical order – aren't going to be of much practical use.

A spreadsheet is a table of numerical values, laid out in a way which helps to convey the relationship between the figures. A simple example might show three sums of money (Gross Sales, Gross costs and Gross Profits) – the last (lower) figure being the *difference between the first two*. This table uses a single, vertical, dimension to clarify the relationship between the values:

1300.00
950.00
2250.00

A typical spreadsheet might contain twelve such columns arranged horizontally, one for each month of the year, introducing a second dimension (left to right) to depict the time element. Clarity is improved by labelling the rows and columns:

	Jan	Feb	Mar	Etc.
Gross Sales	1300.00	1430.00	1573.00	1730.30
Gross Costs	950.00	1045.00	1149.50	1264.45
Gross Profits	2250.00	2475.00	2722.50	2994.75

Entered and derived data

The table above has two tales to tell. First the top to bottom story of the conversion of sales into profits, and second the left to right story of the growth of sales, costs and profits as the year wears on. Notice particularly that there are two kinds of figures involved: **Entered data** (Gross Sales and Gross Costs) which are values originating in the real world and are typed in by the user; the **Derived data** (Gross Profits) consists of figures *calculated* by the spreadsheet using pre-defined formulae (in this case, `GROSS PROFITS=GROSS SALES - GROSS COSTS`).

The difference between these two kinds of figures is not always clear from the table, and neither are the formulae used for calculating the derived values. Often the user may not need to make the distinction or to know the precise formula, but for anyone constructing or changing the table it is crucial to understand these inner workings.

As it stands the table is a valuable way of presenting information whether or not it is put together with a computer. For the simple figures above, and for more complex ones, nothing more sophisticated than a typewriter is needed to produce a much larger spreadsheet where, for example, Gross Costs might be subdivided into additional rows of Direct Costs, Overheads and so on.

But the more ambitious a table, the more time it takes to prepare. Three clerical skills are used: laying out the table neatly so that, for example, the decimal points align properly, entering the data correctly and performing the calculations. A fourth, implied, skill is patience. The spreadsheet may be a budget, guesswork needing quick revision in the light of new information. The alteration of a few pence in a single item of entered data could have a knock-on effect which means that the entire table has to be re-calculated, re-formatted and re-displayed. A glance at the data in one layout might be enough to show that it would be far more effectively displayed by using a different layout – who's going to risk losing a good typist by asking for it to be re-typed?

These clerical skills are well within the capabilities of a computer, and mainframes have long been used for printing out tables. What VISICALC introduced in the late seventies first to the Apple, then to the Pet, was the idea of displaying the table on the screen and of allowing you to enter or alter the data by moving the cursor to the appropriate 'cell' and writing it in. All data affected by the alteration was then instantly re-calculated and re-displayed.

Entering the formulae

The formulae used to produce derived data are also set up by placing the cursor in a cell and typing. In drafting out the Gross Profits table (*above*) with a paper and pencil, you might very well use the margin or a separate sheet on which to scribble the formulae and to calculate the derived figures before transferring them to the appropriate cells. In the electronic spreadsheet you enter the formulae directly into the relevant cells where they lurk invisibly, their location defining where exactly the results are to appear. For checking purposes the formulae themselves will usually be displayed in some marginal location whenever you steer the cursor into a cell containing derived data.

Something quite new happens to spreadsheets when you paste them up on an instantly definable computer screen. Instead of being regarded as an end-result in themselves, fit only to be sent off to the board room, the table becomes a kind of two-dimensional calculator which can display a whole range of values derived from slightly varying inputs. This allows you to look at results on a 'what if we change this . . .' basis – hence the common name of 'What-if' software which has attached itself to this kind of package.

The Gross Sales chart (*above*) for instance, generated here using SUPERCALC, is a budget forecast based on only two items of 'real world' data, the present January sales figures and their associated costs. The 'model' used assumes that both of these are going to rise at 10% per month and the relevant formulae are embedded in *each* cell of the top two rows from February on wards. The precise expression of the formulae will vary depending on the make of software you are using, but it will say, in essence, 'go and look at the figure in the cell to the immediate left of this one, multiply it by 1.1 and display the result here'.

UCSD: The Programmer's Toolkit

The UCSD p-system grew up in a world of academics, and is rather more than just an operating system. It was devised and developed at the University of California at San Diego as a programmers' development kit. In any system the tools needed to make a program are themselves programs: editors, compilers, linkers, debuggers, interpreters and filing utilities. On most micros they are usually scattered around the place, and have to be regularly fetched and carried. The approach adopted by the UCSD p-system is different: from the user's point of view the programming programs are all part of an 'environment', and most of the time a menu of options stretches across the top of the screen, offering access to all the tools needed to write or run a program.

The first thing you see when you start up UCSD, is the main command menu. Its options include:

```
E(dit, R(un, F(ile, C(ompile, E(xecute, A(ssemble
```

The brackets are a UCSD flourish to remind you to choose the option you want by pressing the appropriate letter on the keyboard).

Each command brings you to a new sub-menu of options. If you want to edit something, you press 'E', and are then offered:

```
A(djust, C(opy, D(elete,  F(ind,  I(nsert
J(ump,  R(eplace,  Q(uit,  X(change, Z(ap
```

Wherever you wander in the system you are surrounded (hence the word 'environment') by the tools you will need, and selecting them entails nothing more than pressing a single key.

UCSD offers particularly sophisticated tools when you're programming in a high level language like PASCAL, and its special strengths are editing and filing. UCSD usually comes with two editors: a screen editor and a product called YALOE (standing for Yet Another Line Oriented Editor).

Editing and files

YALOE is provided for use on terminals which don't support cursor addressing, so it would hardly ever be needed with modern micros. The *standard* UCSD editor is the screen editor which allows you to move the cursor to anywhere on the screen, it is quite powerful enough to be used for word processing (*qv*). You can find sections of text, replace them with something else, define blocks of text, set margins and create columns.

When you press the E for Edit key the UCSD p-system (*see* below) clears the screen except for a row of editing commands across the top line. To write a program you simply press I (for Insert) and get going. All the standard keyboard facilities such as four-way cursor movement, backspace delete, space bar cursor movement, cursor tab, and line delete are freely allowed.

After the text has been created and saved to disk the same keyboard functions apply, but you have to be a little more careful how you use them. If you want to insert a new word in a line, you simply put the cursor at the spot where you want to begin writing, hit the Insert key and the screen editor moves all the text in front of the cursor to the right of the screen. You then enter the new word and go through a simple sequence of menu options to save the new program version.

To administer disk files you call up the filer, which offers these options:

```
G(et,    S(ave,   W(hat,    N(ew,   L(dir
R(emove, C(hange, T(ransfer, D(ate, Q(uit
```

Files are taken from or put into volumes, represented physically as disks or I/O ports. Each volume has its own name.

The filer can be used to list the contents of disks, to transfer files between volumes, to datestamp files and to put and get disk files. Instead of having to write a sometimes lengthy command at a prompt, as with an operating system like CP/M, calling up the appropriate program under UCSD is simply a matter of pressing the right key and entering the appropriate parameters. When you're working on a program, UCSD stores it as a 'workfile', so it is always to hand when you're editing or filing it.

The p-Code System

At the heart of the UCSD system is *p-code*, a sort of half-way house between source code and object code which promises portability between a whole variety of processors without the need for the often agonising process of recompiling the program. The UCSD concept is to use software to create what is sometimes known as a 'virtual machine' inside the real hardware. It sets up, if you like, a ghost machine inside your machine. And the ghost created by the UCSD system is an emulation of a processor running under its own machine code, called p-code.

To UCSD software, all machines running the UCSD p-system look the same: they are all p-code processors. In reality they are a selection of quite different hardware, some with sixteen-bit processors, some with eight. But in each machine, between the real hardware and the p-code sits a p-code converter, ready to translate p-code into the machine's native code. In this sense the system is an interpreter, but the pre-compiler has done much of the work already, minimising much of the overheads of time and code size associated with interpreter systems. All the programs you write with UCSD are compiled down to p-code, and stored in that form. That makes them portable across the entire range of hardware that the UCSD system supports.

P-code does impose restrictions. Originally it was designed as an intermediate code for PASCAL compilers, and is not very suitable for languages with a quite different philosophy. COBOL statements, in particular, do not compile comfortably down to p-code instructions. UCSD does offer BASIC and FORTRAN compilers, but the resulting applications tend to be noticeably slower than those produced using compilers which don't have to cope with an intermediary.

UNIX: A Universal Operating System

UNIX is the operating system developed by Bell Laboratories, the company which invented the transistor in 1945. UNIX is big by personal computing standards, although in the world of massive minis and mainframes where it grew up, it was regarded as the first of the human-sized operating sytems. As a general-purpose, multi-user, interactive operating system it offered a collection of features not before found in the large operating systems, and at the same time it foreshadowed the more intimate relationship between user and machine that followed the introduction of the micro.

The history

The earliest version of the operating system derived from an ambitious and sprawling project called Multics, a timesharing development in which Bell Laboratories had a hand. When in 1969 Bell Labs withdrew their support the news came as a blow to their computer scientist Ken Thompson. While the rest of the project was turning sour, his own Space Travel game was coming along nicely on one of the Multics timesharing terminals.

The only alternative hardware Thompson had to hand at Bell Labs was a neglected DEC PDP-7 computer. Its operating system offered little support for program development, so he got stuck into PDP-7 assembler to construct a rudimentary filing system and a few utilities. The improvisation was a far cry from the monumental Multics. There being nothing Multi- about it, Thompson christened his baby UNIX.

Another Bell Labs computer scientist, Dennis Ritchie, began to take an interest in the project which became official when they were able to offer some simple wordprocessing facilities to other departments within the company. In February of 1971, with the help of some official funding, the operation was resited on a PDP-11 in a radical revision inspired by Ritchie's experience of the previous year. PDP-11's sprang up in other departments, and users chose UNIX in favour of the native operating system supplied by DEC.

In 1973 came the next major revision, a crucial one as far as the future of UNIX was concerned. The new version was written in C, a language developed by Ritchie from BCPL (Basic Combined Programming Language) and designed to replace B, a bare bones re-arrangement of BCPL knocked up by Thompson. UNIX and its utilities now comprised over 300,000 lines ofCsource code. The virtual memory techniques used by the system required only 53K of the system to be resident in core memory – which was just as well, as there was only 122K of RAM in the PDP-11's.

During the 1970s UNIX hardly showed its head outside the ivory towers of academe and the sanctuary of Bell Labs. The company was blocked from making a commercial go of UNIX by US anti-trust legislation, which was already keeping a wary eye on their monopoly of the telephone system. Even as recently as 1982 there were no more that 20,000 UNIX-based computer systems in the US. The number rose by five times in 1983, as had been predicted by observers keeping an eye on the colleges where, with benign commercial foresight, Bell Labs had been sowing seeds for the future with cheap educational licences.

What UNIX does

You can think of UNIX as being in three layers. At the centre is the **kernel**, the code which directly accesses the hardware. This is equivalent to CP/M's BIOS, the section that has to be adapted for different hardware environments. The kernel remains in core all the time while UNIX is running, controlling the file handling, multi-tasking and I/O.

Wrapped around the kernel is the **shell**, a glorified command line interpreter which is also able to execute batch files. Although two shells are supplied as standard with most versions of UNIX, it was the intention of the designers that the users or the system implementors should be able to write their own quite easily. Three shells come with MicroSoft's XENIX, a derivative of Bell Labs' UNIX VERSION 7, with several additional extensions from the University of California at Berkeley.

You don't have to be a programmer to change the way the shells work. Non-programmers can write what are called 'shell scripts' in plain text to run under under either the Bourne shell or the Berkley C-shell – a shell whose syntax is a simplification of the theClanguage syntax.

Outside the shell lie the **utilities**. A huge variety of these are supplied as standard with most versions. C itself is included, together with a rich selection of aids for software developers. Although these are all regarded as part of the system, some UNIX implementations allow the purchase of subsets of the utilities (although this – to some extent – sacrifices the designers' fundamental idea that any UNIX system should be self-supporting.

File handling

The most important role of any operating system is to provide a method of handling files. The UNIX user sees files as being of three kinds: ordinary disk files, directories, and special files. Ordinary files include text files and binary, executable programs. Directories behave exactly like ordinary files except that they can't be written on by the users' programs; it is the system itself that controls the directory contents.

The system maintains several directories for its own use, the main one of which is called the root directory, dubbed / with typical UNIX brevity. All the other directories are files in the root directory, and any file in the system can be found by tracing a path through the branching chain of directories by giving a sequence of directory names separated by slashes (/) and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. A feature called linking allows a (non-directory) file to turn up in more than one directory, and even under different names.

With special files, UNIX allows filenames to be connected to I/O device drivers, enabling them to be read

and written to (where appropriate) just like an ordinary disk file. Special files connect the directory to communication links, disks, peripherals – and even to raw memory. When I/O devices are handled like this, file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can as easily respond to a device name. This arrangement also allows special files to be given the same access protection as regular files.

The root directory is always kept on the same device, usually a built-in winchester disk. But by a process of 'mounting', additional devices (usually floppies) can become temporary 'branches' of the file system tree. The mount system request is given with two arguments: the name of an existing ordinary file kept in the root directory specially for the purpose, followed by the name of a special file which is in fact a device with an independent file system containing its own tree'd directory. This approach has the advantage of leaving a UNIX machine open-ended for future hardware expansion.

UNIX protects its files by tagging each one with a set of ten 'mode' bits, the values of which are controlled by the user who creates the file. One bit is set if the file is a directory, and three triads of three bits each indicate read, write and execute permission for, respectively, the local user, the users group, and for all users in general. Listing the directory in long form (`ls -l`) will show the values held in this bit field.

For the purposes of restoring forgotten passwords, backing up, and other system maintenance where the protection system would be a nuisance, UNIX provides for a 'super-user' who may over-ride the usual file access constraints.

UNIX allows I/O redirection and piping. The standard assumption of programs running under UNIX is that input will come from the user's keyboard and that output will go to the console screen. Redirection allows this assumption to be altered at the last moment – from the command line – immediately prior to execution. Left and right arrows are used to indicate the device/file to be used for input and output respectively.

Pipes are an extension of this idea. By naming a pair of executable files in the command line (using a vertical bar character to separate them) the output from the first program can be directed into the second program. The vertical bar character means 'take the output from the program on the left and make it the input to the program on the right.

One use for this might be to look through a directory listing to find all the entries that are themselves directories. The command `ls -l` will produce a 'long' listing, where each filename is accompanied by a readable presentation of the protection bits. The first bit is displayed as a d if the file is a directory, and lines like this can be selected by using the pattern recognition program 'grep' with the parameter ^d, which means 'look for a d as the first character on the left hand edge'.

The command line to do this will look like this:

```
ls -l > grep '↑d'
```

Because UNIX is a true multi-tasking system, the two programs will actually run simultaneously. The pseudo-pipes of MSDOS 2 (and the CP/M enhancement MicroShell) are emulated by the creation of a temporary file to hold the output of the first file until the second begins its run.

UNIX makes no distinctions at the system level between 'random' and 'sequential' file operations, and no logical record size is imposed, unlike the 128 byte records of CP/M. As you might expect on a multi-user system there are internal interlocks to prevent users from deleting each other's open files or creating files with the same name in the same directory; but the level of file and record locking provided by, for example, MP/M-II, was rejected by the UNIX designers as being 'unnecessary and insufficient'. Modern commercial versions of UNIX however invariably include some additional form of record locking facilities.

The octopus

For an operating system of its size and capability, UNIX is very portable, thanks to its being written in a high level language with a layered structure which means only the kernel need be revised for new hardware. The multi-user, multi-tasking talents of UNIX have been well-tested for more than ten years in the scientific and academic environment.

Although customised shells will go a long way to meet the non-scientific, non-academic market, the more you cover UNIX the slower it gets. Even ordinary single-user versions still usually contain the multi-user kernel, which spends CPU resources in unnecessary housekeeping such as looking out for other, non-existent, users. UNIX has been called 'the octopus in your tank', and the current generation of 16-bit hardware needs all the help it can get to make UNIX on micros totally convincing.

Despite (or possibly because of) the built-in facilities in UNIX for generating and maintaining documentation the manuals are over-blown, inconsistent in tone, and in need of drastic editing. The documentation for XENIX is little better, though it does offer some improvement in the indexing and general organisation. It is the manuals more than anything which make certain that UNIX will not fall easily into the hands of the raw beginner.

The main features

- Multi-tasking, multi-user operating system.
- Hierarchical 'tree' directory system with demountable volumes.
- Compatible file, device, and inter-process I/O.
- The ability to run several processes simultaneously.
- Individual shells for different users within a multi-user system.
- A huge range of utilities and operating system functions, including a dozen languages.

User Groups: Getting Together

User groups are forming all over the country almost every week, and with such an obvious diversity of interests between clubs, selecting the one to suit your own needs can be difficult. On the other hand, the variety of computer clubs in existence does mean that there's a club to suit virtually everyone's tastes.

Supra-national clubs include such umbrella organisations as the Association of Computer Clubs (ACC) and the British Computer Society. Both can provide general information as well as putting you in touch with more specialised groups. The ACC (formerly known as the Amateur Computer Club) maintains a very useful up-to-the-moment database of about 400 computer clubs.

Micro-dependent clubs form the largest number of computer clubs. Which of these you choose will obviously depend almost entirely on the micro you own. You will probably get the most out of a local micro-dependent club, since there is a tendency for the national groups to hold only infrequent meetings. Some hold no meetings at all, but do perform a useful central organising function acting as publicity agents for local branches, organising stands at exhibitions and producing promotional material. The national clubs will sometimes have software libraries available to members. These aren't able to meet the enormous interest in games (*qv*) due to copyright protection, but many members do write their own games software, so they are worth investigating.

Software-allied clubs are for those who don't want to be restricted to a particular make of micro. Typically the software connection will be an operating system, as with those united by an interest in CP/M or Unix. Clubs such as these are strong on conferences, special interest groups, software libraries, and newsletters. It is also fairly easy to find a club catering for popular programs like WORDSTAR or spreadsheets. If your inclination is more towards games there are computer chess clubs and groups like the Hell's Temple Fan Club.

Fixed locality clubs tend to be very popular outside the big cities, because of the difficulty of grouping enough members around a single make of computer. Their size varies a lot, and the larger clubs often contain machine-oriented sub-groups. There is a lot to be said for a club like this where members will quickly acquire a wide knowledge of a range of micros, and get a much better picture of the micro world in general.

Some fixed locality clubs are set up by pupils and teachers within specific schools, or in-house like the ITN computing club or the Metropolitan Police's club at New Scotland Yard. Usually they aren't as isolated from the outside world as the naturally restricted membership makes them sound – clubs like these are rather good at establishing links with other micro clubs.

Aim-oriented clubs are groups like Computers for Peace, MENSA's home computing special interest group and the Primary Health Care group. Computers for Peace is a group which aims to use information technology and personal computers for peaceful purposes and which collects information on the peace movement worldwide.

The Primary Health Care Group is for computer users working in the medical field and is open to anyone in the profession – not just GP's and consultants. Activities include testing software in the field and reporting findings to other members. The group also hopes to set up its own bulletin board so that medical information can be easily and speedily exchanged.

Finding the right club

Finding the right club shouldn't need excessive travelling. Libraries, dealers, maunfacturers and the 'Club Spot 800' pages on Micronet are all good places to try looking. If you are still stuck, the ACC is particularly good at pinpointing fixed locality and micro-dependent clubs. The British Computing Society will be a good starting point for the more obscure specialised computing groups.

Club charges vary enormously. Some levy no subscription, but do charge a few pence for entry at meetings. Others have a regular subscription charge that will cost less than a games cassette a year. There are also some professionally oriented clubs where it is assumed that the subscription will be paid by the firm, the Inland Revenue, or both.

Most clubs sell their subscriptions on the strength of their newsletters or monthly bulletins, which contain hints and tips, programs, circuit diagrams, reader's classifieds, product reviews, competitions, and adverts. This may be valuable information not easily obtained elsewhere, or it may consist of incoherent ramblings, so before joining have a good look through an issue of the club magazine. It's a good idea too to go along to a club meeting before signing up in order to gauge what is on offer, and evaluate the intangibles.

Some benefits to look out for include:

A technical advice service able to answer specific computing problems over the phone by the club's own 'experts'.

Hardware and software demonstrations laid on by local dealers (clubs generally have little difficulty in enticing them over with the prospect of mass sales).

Discounts to clubs and/or club members (from those same dealers). Even though the resulting 'bargain' may still seem expensive when compared to mail order prices, this kind of offer is well worth thinking about. If things go wrong you can always go back and see the local dealer – but there's no point in shouting at the postman.

Talks and lectures from visiting speakers. This isn't always so easy for clubs to arrange, but some have tie-ins with local Polytechnics and can offer good in-depth lectures on subjects such as machine code and communications.

Some clubs are heavily tied to a specific manufacturer, importer or dealer, and are therefore less likely to be impartial. You are not likely to learn of bugs or faults through clubs like these, however their technical information, news and support may be more accurate as useful than most.

A good many clubs have membership of fewer than 50 with 20 or so people turning up to regular meetings. Joining such a club is really like undertaking any other social activity, with the added benefit of making direct contact with experienced micro users. You'll soon get to know the best time to arrive at meetings: clubs tend to encourage their younger members to turn up and play games first, and then move on to serious programming as the meeting gets on.

The Association of London Computing Clubs (ALCC) can arrange a transfer to another affiliated club for members who move house or who find their own particular interests better catered for elsewhere. In fact, few people appear to take advantage of this, preferring to stay loyal to their local club. But there is nothing to stop you attending more than one club if you are keen.

Start your own club

If you find that you don't like any of the micro clubs you've tried, or that there are none within easy reach of where you live, you might consider starting one yourself. There are almost certainly a lot of people in your district who would be interested in joining such a club, for the same reasons as yourself.

Finding the Venue – The first thing you will need is somewhere to hold meetings. Anything will do, from a room in a pub or community hall, to a spare room in your own house. Within your – almost certainly limited – budget it should be possible to find a venue which is centrally placed and easily accessible. Make sure, though, that the room is well heated – computers and computer clubs tend to draw most attention during the winter months! You'll need plenty of accessible power points too.

Avoid weekends for the first meeting, since too many people are likely have other things planned. A start somewhere between 7.30 and 8.30 allows for a full evening, giving everyone time to get home from work and to eat. Plan ahead and give yourself at least three weeks to do some advertising.

Telling the World – When you've passed the word to friends and acquaintances who you think might be interested, spend some time devising a simple, eye-catching poster to announce when and where the first meeting will be held. Making the first evening free of charge is guaranteed to double your initial membership.

Blitz your area with photocopies of the poster, starting at your local computer shops. It will be good for their business, so they should be only too happy to give you free window space. Don't be too disappointed if the group gathering in the appointed place at the appointed hour isn't exactly a multitude. With a nucleus of keen supporters, however small, the club will soon start to snowball.

Getting up to speed – On the first night put your heads together and decide what sort of things to include in your club meetings. Typically, the accent will be on programming, so you should make sure there are plenty of opportunities for programming workshops and talks on particular aspects of the Black Art. A course in BASIC is bound to be popular. Computer manuals are often daunting and confusing, and the chance to swap notes in friendly surroundings will soon swell the membership.

An equipment rota is essential to the smooth running of the club. Don't forget that TVs or monitors are just as important as the computers themselves. Try to pair computer owners without transport with car owners who have no computers.

In the early weeks a caretaker committee, elected informally, can take on the business of the club until the membership stabilizes. Few committee members are needed - just a chairman, secretary, treasurer and publicity officer. You will have to charge a membership fee, renewable annually, as well as a subscription for each meeting attended. A policy of allowing free admission for one initial meeting will encourage potential new members. The library of computer books, magazines and software need to be nothing more elaborate than a box into which members can put any computer-related material they want to share or dump. Renting library items out for a small weekly charge (if you can get away with it) makes a useful source of revenue.

The publicity drive – When the membership settles down the meetings will acquire a regular format. This is the time for a big publicity drive to tell everyone you can reach what the club is and what it does. Local newspapers are always looking for news, so if you go about it the right way you can get some very useful free publicity. Try to give them an angle rather than just a bundle of raw facts. Has your youngest member, aged 8, sold a program to the local computer shop? Tell the world.

Consider the national computer press. Several of the magazines publish lists of all the computer clubs in the country, and you only have to drop them a line to be included. Most areas have a local radio station, and it is well worth expending a little effort to get some of their airtime. As with newspapers, the difficulty is getting the right person on the telephone, so send a letter with a daytime phone number to both the programme controller and the news editor. If there's a story in the news you give them you should get a call.

Exhibitions in your area give you the chance to meet potential new members at first hand. Contact the organisers and try to get yourselves a stall in the main hall. If you emphasise that you are non-profit-making you may well be able to get one free. Be ready with copies of a simple information sheet, and be prepared to take subscriptions from new members and names and addresses of potentials.

Long term organisation – As the club develops, the first requirement will be for extra space. The biggest problem

with changing premises is ensuring that all your members move with you, particularly those who make only an occasional appearance. If possible give these a phone call as advance notice. Don't forget that the onus is on you to keep up contact.

Maintaining an interesting programme is especially difficult if the club meets every week. Before long you may find that more and more time is spent playing computer games and less and less is spent on the more constructive aspects of the club. Once this happens, some members will inevitably become bored and begin to drift away.

Keeping up the equipment rota will require tact and organisation. Nobody wants to see their computer receive a weekly hammering from the games freaks, if this happens then the number of machines brought to each meeting will dwindle rapidly. The moment you see this happening (and it will) it is time to take a long hard look at the club's format. Consider changing from a weekly meeting to a fortnightly or even monthly meeting. Plan well ahead so that there is plenty going on at each meeting. Organise discussions and workshops. Invite speakers on particular subjects and arrange demonstrations of new items of equipment. You'll find that members are more inclined to bring their computers when they know exactly what they are going to be used for.

Winchesters: Disks The Hard Way

Hard disks, often called Winchesters, improve on floppy disks (*qv*) as much as floppies improve on cassettes: they give greater reliability, speed of access and capacity.

The manufacture of Winchester disks is in many ways the most impressive of all the computer technologies, if only because disks have moving parts – some no larger than a few microns – which have to be engineered to an awesome degree of accuracy. The degree of precision required for the disk head is comparable to a fully laden Boeing 747 (itself much taller than most buildings) flying a couple of feet off the ground at 600 mph – for hours on end!

Winchester disks were originally built around 14" (and later 8") platters, the same diameter as the first floppies. As floppies have shrunk to 5¼" and then to something less than 4", Winchesters have followed suit. Eight-inch Winchesters are now a rarity; the 5¼" format, inevitably known as mini-winnies, are now the standard on business micros. The British company Rodime led the way to an even smaller standard when it announced a 3½" Winchester disk system early in 1983, with a formatted capacity of five or ten megabytes, depending on the number of platters the unit contained.

The smallest-capacity commercial 5¼" Winchester disks in common use today offer 5 megabytes (5,000K) of storage. A more usual capacity is 10 megabytes; this is 40 times the size of the standard IBM floppy, representing about a year's typing. Mini-winnies with the same physical format but with a capacity of 20 megabytes are fairly common, and current manufacturing techniques have already managed to cram 40 megabytes into the same-sized box. This represents four years' worth of typing for one typist, or something in the region of 85 ordinary-length novels.

The Winchester at work

Figure 1 shows a cross-section of a typical hard-disk system. Each disk, sometimes called a platter, has two recording surfaces, and two read/write heads are dedicated to each disk. The heads travel in only one dimension – radially either into or away from the central vertical shaft, all moving together. The vertical shaft itself spins at a very high speed, typically 3600 revolutions per minute.

As with floppy disks, data is stored in tracks and sectors on a ferric-oxide material coating the upper and lower surfaces of the platters. A **track** is one complete concentric circle of the platter, and will typically be divided into 16 or 32 sectors. The typical maximum number of tracks per inch of radius (**tpi**) is 600.

As the concentric circles get closer to the centre the tracks get shorter, which means more bits per inch of arc of the circle. But the rate at which bits are read or written is kept constant by laying the inner tracks fractionally closer together, helped by the fact that the linear speed of the disk relative to the head is slower as the head moves towards the centre.

Not all the surface area of a disk is used to store information. 3¼" drives typically use only the tracks that

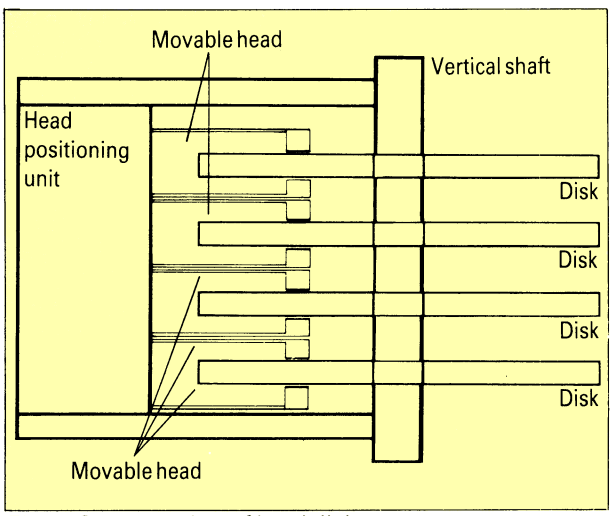


Fig. 1. Cross section of hard disk system.

are within about ¾" of the edge of the disk. At 600 tpi the maximum number of tracks used is 306.

Sometimes hard-disk specifications will talk of a **cylinder** rather than a track. A cylinder is the set of tracks on all recording surfaces that are equidistant from the centre – the complete set of tracks addressed by the head at any one time. On a four-platter drive, for example, each cylinder will consist of eight tracks.

Winchester aerodynamics

In order to be able to read and write from one tiny area of the disk without affecting surrounding bits, the magnetic head must be travelling very close to the disk surface. The distance between the head and the disk is of the order of a third of a micron (a micron is a thousandth of a millimetre). On the outer tracks the disk is passing the head at about 37 mph. So a micro-hiccup will be enough to send the head gouging into the disk at considerable speed.

The head itself is actually flying above the platter. As the disk is spinning so fast, the air close to the platter surface moves with it through a kind of friction effect. The head is kept up by this air pressure, and the result is very similar in aerodynamic principle to a hovercraft. Tiny springs are used to balance the upward force and maintain a constant height. The heads that are using the lower surface of a platter are held down by air pressure and held up by the springs.

Because of the centrifugal force of the spinning disk of air above the platter, air is pushed out from the centre towards the edges. This results in what is called a pressure gradient – the air pressure is much less at the centre than it is at the edges. To compensate for this, air is pumped up the central shaft and out onto the surfaces.

As a side-effect the outward airflow helps keep the platter surface free of particles. The drives are not air-tight but are dust-tight, and will keep out particles measuring more than half a micron. A mote even this small would

almost certainly cause a disk head to crash. Winchester disks have to be assembled in special areas air-conditioned to a standard known as 'class 100 clean'.

Head movement and access time

The distance between tracks, although tiny, is somewhat larger than the distance between the head and the surface. Positioning is usually carried out by a double-precision **stepper motor** in which the number of revolutions of the motor is exactly proportional to the number of pulses of current supplied to it (and computers are *very* good at handling pulses). The number of revolutions of the motor will also be proportional to the distance the head will move, so a specific number of pulses can be applied to will move the head accurately from one track to another.

The specification of the speed at which data can be read or written is usually divided into two timings: access time and transfer rate. **Access time** is the time it takes the head to find the right sector; **transfer rate** is the rate at which bits can be read or written.

Access time can be divided into three further specifications: the seek time, settling time, and the rotational delay.

> The **seek time** is the time it takes to position the head, and is typically around 80 milliseconds for a hard disk.
> When a head arrives at a new track it will oscillate slightly for a millisecond or so, and this is called **settling time**.
> The **rotational delay** is the time it takes the sector to spin round under the head, and is also sometimes known as **latency**.

Another technical specification quoted for hard disks is whether the head-moving circuitry is open- or closed-loop. **Open-loop** means there is no feedback; the circuitry has no way of telling where the head is so the positioning has to be very accurate. On a **closed-loop** system the head can pick up information from the disk and can use this to confirm that it has found the right track. This can be done in a number of ways, one of the best involves putting a code in the sector header. The head will read this code when flying over it, and the head-positioning circuitry can use this either to confirm that the correct track has been found, or, if not, to correct the movement of the head.

How the Winchester got its name

There are two stories to explain why the drives are called Winchesters. The first is that it is simply named after IBM's plant at Winchester, New Hampshire, USA, which developed the technology.

The second – a lot less likely but more fun – is that they were named after the famous rifle. The connection is slightly dubious: one of the original Winchester rifles was referred to as the 3030, and the first disk drives developed by IBM had two platters holding 30 megabytes each, so that 3030 became the product number.

Wordprocessors: Prose Control

Word processing – the business of creating, editing, manipulating and printing theoretically unlimited quantities of text – is a natural application for microcomputers, and a development of the text editors (*qv*) used for writing computer programs in source code. But with the wrong combination of hardware and software it can be a dreary, slow and even risky business:

An over-complicated commercial package running on modest hardware can bring your creativity grinding to a halt.

Sophisticated hardware can be wasted on software which doesn't know how to make the best of its environment.

Poorly designed software which crashes when you inadvertently change a disk can cost you hours of work in recreating the last edit – evaporated by rebooting the system.

Your micro is a resourceful machine, and – with the right software – will quite probably take you as far as you need to go in the direction of word processing. But if you get a taste for writing text electronically you may find that hardware which is perfectly adequate for ordinary computing becomes painfully unergonomic when you have to spend extended periods in front of the wrong kind of terminal reading and writing text. Many VDU's and most television sets are simply not up to presenting large tracts of text clearly enough for accurate editing. You may need to buy a special keyboard, and perhaps an expensive new printer. Where, then, do you begin?

The entries under 'Keyboard' and 'Monitors' (*qv*) should be studied if you're contemplating doing any serious word processing.

Memory

Systems with too little memory, or running ambitious packages, will tend to be sluggish, due to their use of overlays. Overlaying is the technique used to cope with the problems of running a large program in a relatively small area of memory by keeping the less frequently used routines on disk until they are actually needed. Even as mundane an operation as scrolling vertically through a file can require a change of overlay in some systems.

There will also be problems if you try to move around at random in a large document. Small internal memories can make it necessary for the software to write temporary files of already edited text on to the working disk, reducing the capacity available for storing document files.

Some manufacturers of small machines have provided particularly successful programs for their own equipment. But on a typical 8-bit CP/M system, 48K is likely to be the minimum practical size. If you are looking at new equipment with a 64K option, or going in for a 16-bit machine with anything up to a megabyte of memory, the extra money it will cost might well be buying you a quicker, more flexible system.

A number of packages described as word processors are available for cassette-based home micros, but don't expect them to offer anything like the facilities that cost the users of disk systems many hundreds of pounds. Generally speaking, anything that has word-wrap ability tends to be called a word processor, and many of these products aren't much more than line editors (*qv*) masquerading as screen editors. They usually have only a single line in the centre of the screen which you can edit, and you have to access areas of the text by scrolling up and down through the document until the part you want is on the screen edit line.

You can expect cassette word processors to have functions like string search – or perhaps find and replace – block delete and move, word-wrap and justification. They occasionally offer headings and footings, forced page breaks and print effects such as bold, superscript and subscript and underscore.

Backing store

Home micros using cassette tapes rather than floppy disks to store data are necessarily limited as word processors. Cassette tape recorders weren't designed for data storage and they are less reliable than floppy disks (*qv*). Tapes can be infuriatingly slow – disks with their random access can quickly reach for any item of data, but a cassette system has to search through a tape from the beginning until it finds what it wants. And as the tape has to be 'blocked' to store a number of files, storage efficiency – particularly for several short documents – can be alarmingly low.

Single drive systems are flexible, and using two floppy disk drives is even better. On a dual floppy system, disks can be 'backed up' (copied on to another disk for safe storage) in a single operation, and designated files can be copied on to another disk without any intermediate operations.

Mail merging (*see below*) is possible but slow on a single disk system, but dual floppy systems make a far faster and easier job of it. Editing large documents is also easier if the updated version is written away on to a clean disk in the second drive, while the original remains in the first.

Do be careful, however, when assessing the disk capacities quoted by manufacturers. Many of them will quote 'unformatted' capacities, but these don't take into account how disk tracks are divided into sectors, or how many tracks are used to store directory information rather than programs or text. Quite often the actual text storage capacity of a disk is less than 80% of its quoted unformatted 'capacity'.

Some systems need the complete word processing software program on every disk used, but even if yours doesn't it is important to find how much room these programs require on the system disk. This is unless you are lucky enough to own a system where, after the software has been loaded, the system disk can be removed before text is entered, edited or printed.

If you are not so fortunate, you can get an idea of the gross text capacity of your system by subtracting the volume of disk required to hold the program from the formatted disk capacity. However there are other factors

which can influence the maximum document size that your system can conveniently handle.

Page or document?

In that is **page oriented** word processing software the maximum unit of text is a 'page', arbitrarily defined by the software designer, and usually considerably larger even than a densely typed A4 page. One typical package uses a maximum page size of 80 lines, each of up to 80 characters. If you use software like this, and if your work is likely to include reports, manuals, proposals or other multi-page documents, it is worth making sure that you can at least link files together so that a whole document can be printed on sequentially numbered pages in one operation. It's obviously better if each link point doesn't force the printer to start a new page.

Alternatively, you can choose a **document oriented** package, of which WORDSTAR (*see below*) is the classic example. A document can be of any length (so long as your disk storage is sufficient) and can be edited, stored and printed as a single unit. The printer you use controls the line width; systems with a horizontal scroll facility are advisable for lines more than 80 characters wide.

Page-orientated systems usually use memory (both RAM and disk) more efficiently, and are often best if your system has limited storage. Document-orientated systems make creating, heavy editing, and printing of large text files simpler – memory permitting.

Disks and backups

The size of the largest document that can be handled effectively is related to how the system stores the original and updated versions during editing. Some systems simply overwrite the original as you edit. This is straightforward if the operator understands it properly, but can be dangerous if the file to be edited isn't regularly backed up onto another disk. A number of dedicated word processing systems based on the classic Wang word processing system operate in this way.

A safer approach is for the system to create automatic backups during editing. Then if any particular edit is a disaster, serious errors can be put right by rejecting all the editing changes and recalling the original version. WORDSTAR offers this facility. More sophisticated packages give you a choice – you don't necessarily want to make a whole new version if you recall a document to amend a comma, and holding two versions of a document will inevitably place an extra burden on storage capacity. Single sided, single density 5¼" floppy disks will provide about 80K of formatted text storage. A document of about 30K characters (say 5,000 words) which has already been edited once will already be using 60K of the disk. If a second update is required, difficulties may occur during the editing because of the system's need to create temporary work files and you will certainly run into trouble when you try to file the newest version. In a relentlessly backup-creating system like WORDSTAR, the biggest document that can be handled effectively is far smaller than the gross text capacities so often quoted – the recommended maximum file size under WORDSTAR is 40% of the total disk capacity.

Most of the limitations and difficulties associated with backing store evaporate if a Winchester disk (*qv*) is used, but the technology is not cheap.

Try it out

While the hardware is important, a detailed investigation of exactly what the software will do is vital. All the packages you examine should allow you to move the cursor to every character displayed (it is not always possible to access blank areas of the screen) and then delete or insert characters at this position.

In several packages, typing will normally overwrite the character at the current cursor position, in which case inserting is done by setting a special mode. Check whether this operation removes all or most of the subsequent text (that is, characters below the editing point) from the screen. Generally, the less disturbance to the display, the better. It is also worth seeing how easy it is to correct text which you have just entered while in insert mode.

Mode-less packages which default to inserting characters in front of the cursor are often simpler and more flexible to use when handling ordinary text but are much more difficult to use for tabular work, with the table alignment shifting around alarmingly as each correction is made. One reason for WORDSTAR's popularity is its mode-less default in routine text editing with a simple way of changing to overwrite mode.

It is important to understand that overwrite on its own can only be used to correct errors where the original and the corrected text have the same number of characters.

Many micro packages can mark existing blocks of text and then move or copy them elsewhere in the same document. How far this process can be extended depends on system sophistication; ideally, you should be able to copy individual blocks of text between any two documents with ease, whether they are held on the same or separate disks.

During printing you should be able to invoke automatic page numbering, to print only part of a document, and to use additional paper handling devices such as tractor feeds (*qv*) and single sheet feeders (*qv*). It is also very useful if you can print separate files appended together into one long text allowing, for example, a long report to be built from individual text modules copied into the final document in sequence.

Search and replace

Most systems now offer character string search and replace, a speedy way of updating certain types of document in an organised way and, when no direct page access is available, the only simple way to locate a passage in a long document.

Packages vary a lot in the permissible length of both the 'target' and replacement strings. Try to discover if a string

can still be located even if it does not wholly lie within a single line of text. Some systems treat the carriage return and line feed at the end of each line as if it were part of the text, which means that a search for:

```
carriage return and line feed
```

might fail to find the phrase in the paragraph above.

It is also still worth checking that there are no problems if the target string is a subset of the replacement string. Create a paragraph containing the word 'England' three or more times and then try to replace it each time with 'England and Wales'. Check that this has been done correctly in one single pass – word processors have been known to go into an infinite recursive loop during this test, with disastrous results!

Special features

Decimal and flush right tabs, various forms of indentation (including overhangs or reverse indentation), marginal notes, sub- and super-scripts and print enhancements such as centering, underlining and bold printing (with an offset double strike) should be checked if you expect to use them. It is also very useful to be able to record format changes (margins, tabs, line spacing, page lengths) within a document so they are automatically included when it is recalled for editing.

Mail-merging

Mail-merging – the technique used for generating all that junk mail personalised with your name and address – is probably the oldest word processing application. It allows the automatic repetitive printing of a standard document containing yet-to-be-defined place markers that work rather like variables in a BASIC program. The standard document is printed over and over again, with new values being supplied for the variables in each copy. At its simplest the system can be used to send an individual copy of a standard letter to a number of recipients; obvious applications include credit control, personnel management and invitations to tender or supply.

An extension of this idea, using whole files rather than specified blocks of text as variables is used in the very powerful document assembly programs of some professional word processors. WORDSTAR's optional extra module, MAILMERGE offers the same facilities on a micro, given enough disk storage.

Mail-merging has been praised as a shining example of the sort of productivity gains available through using micros, although the praise usually comes from the senders of mail rather than from the recipients – and never from the postmen!

When costing a word processing system it is only too easy to underestimate the time needed to learn how to use it. Manuals have improved greatly of late, by many otherwise excellent packages still come with documentation that is quite unsuitable for newcomers not already familiar with the basic operation of the hardware.

What to watch for

Make sure you understand all the operations required to fix permanently the edited version of a document on to a disk. Often the final editing is done well before the end of the document, so that when you give the command to commit the file back to disk the cursor is somewhere around the middle of the text. Some systems dangerously only save material lying between the beginning of the text and the current cursor position. If the save function has to be preceded by scrolling through to the end of the document, sooner or later a hurried operator will forget this, and the updated version will be stored without its concluding pages. It's surprising how long this can go undetected.

One odd quirk of some micro packages, not found in the dedicated word processing field, is the use of quite different formats for input and editing on the one hand, and for printing on the other. The edit mode fills the screen with characters, breaking words between lines and destroying the 'shapes' of paragraphs, making it hard to find the parts you want to edit or work out page breaks. In the print mode, text will be displayed on the screen very much as it will appear on paper, but cannot be edited in this form.

Software designers do this to simplify problems in other areas, and to get around the limitations of the hardware, so don't reject a package out of hand simply because it works in this way. Rather, compare all the facilities likely to be of value to you and judge on this wider basis.

The proper identification of pages can be important in work involving long documents. Sadly, many packages do not show page boundaries on the screen at all. Some of those that do, offer 'dynamic' page divisions – extensive editing near the beginning of a document immediately alters all subsequent page breaks. This can make it hard to find an editing point near the end of the document.

WYSIWYG versus EMACS

WYSIWYG stands for 'What You See Is What You Get', a school of thought exemplified – though with its many embedded control characters far from perfectly – by WORDSTAR. In WYSIWYG word processors the display on the screen shows more or less how you can expect to see the text when it is printed out. But there is a diametrically different approach known as EMACS.

One of the driving ideas behind the word processing system called EMACS, originally developed at the MIT artificial intelligence laboratory, is that what appears on the screen is important only in respect of its content. Formatting is a function of printing, and should not bother the head of the programmer or creative writer using the package, whose primary concern is to get around the text quickly, to access and edit several texts at the same time, and generally to treat the screen as if it were an electronic desk top on which papers can be shuffled. Later, when the time comes to produce hard copy, format codes embedded in the text will make calls on a library of

ready-made formatting instructions, guiding the layout of the printed word with no further human intervention. A format instruction like '@begin(quotation)' makes no change to the screen layout, but is read by the pre-print formatting routine which will understand it as a command to switch in the appropriate left and right margins and to observe other format conditions such as line spacing, all of which add up to what's called the 'environment'. This particular environment would be switched off again by the instruction '@end(quotation)'.

EMACS allows the screen to be split up into windows. A window is simply a section of the screen in which text is handled independently from the other windows. It also permits a number of different files to be held open simultaneously in a number of buffers (invisible and dynamically changeable segments of the virtual memory). Buffers are nothing more than invisible windows that you can bring into view at the touch of a button. The EMACS philosophy has many advantages for the experienced user, but is hard to sell to newcomers, who find the 'glass typewriter' aspects of WYSIWYG software more instantly reassuring. But in going the WORDSTAR route they sacrifice:

The facility to alter the entire file format quickly and simply by changing the style instruction at the head of the text.

The ability to move around the text (and move the text around) in logical increments of character, word, sentence, paragraph, rather than by using the physical increments of row, column and screen.

The opportunity to work on more than one file at the same time.

EMACS offers all this and more. While WYSIWYG is adequate – even preferable – for short documents needing close attention to layout, the carefully thought out approach of EMACS is a boon when manipulating large volumes of text.

WORDSTAR

WORDSTAR has become the *de facto* standard, not because it is the best, or even because of any brilliant marketing effort on the part of MicroPro, the authors. WORDSTAR got where it is today by being available across a wide range of CP/M systems, where it proliferated reputedly as much by pirating as by the sale of legitimate copies. Considered purely as an editing system, WORDSTAR is a fairly slow, largely because of its reliance on overlays.

For example, if you are just beginning to use the system, a menu of possible commands will always appear at the top of the screen. This will not appear as you begin to learn the editing facilities well enough to key in commands before the menu is displayed.

WORDSTAR has an insert on mode, which means that you can automatically insert new text at any point by simply placing the cursor in the correct place and typing in whatever you want.

If you run into problems there is a help facility to give some guidance, but this too can be a little confusing until you've used the system for a while and/or read the WORDSTAR manual thoroughly.

WORDSTAR allows two editing styles: **Document editing** is for text being prepared for ordinary printing, a category including letters, reports and so forth. A document edited under WORDSTAR contains many non-printing control characters (optionally displayable on the screen) which give information to the printer telling it to perform functions such as underlining and starting a new page. Its carriage returns (created by word-wrapping) are 'soft', using different values from the ASCII standard. WORDSTAR also has a habit of marking the characters that terminate words by setting their high bits on. These embellishments are accepted and understood by the printer driver inside WORDSTAR, but can cause chaos if fed directly to the printer, to a screen, or to an assembler or a high-level language compiler.

For this reason WORDSTAR can also be operated in **non-document mode** which creates pure ASCII text without any of these refinements. In this mode WORDSTAR behaves as a rather well-endowed text editor.

MicroSoft's WORD

WORD is one of the newer word processing packages, taking advantage of greater addressing space offered by 16-bit machines. It is a thoroughly designed package – a word processor with a smooth and presentable user interface that won't frighten away beginners, but concealing a surprising array of functions which can be unfolded one by one in the user's own time. Superficially it's a WYSIWYG word processing system, although internally it incorporates many of the features of EMACS. WORD includes libraries of format styles, multiple buffers, the lot – but bundled cleverly into the lower reaches of the software where the complexity won't interfere with the beginner's sense of control over the package.

When you first bring up the WORD a bordered window appears, occupying most of the screen but leaving four lines at the bottom for the menu. The window is empty except for a cursor positioned in the top left hand corner. The cursor appears to have a hollowed out diamond in the centre, but in fact the diamond is the end of text marker, a separate symbol temporarily overlaid by the cursor.

You type in the text just as you would on a typewriter. Automatic word-wrap takes care of line endings, and you only need hit return when you get to the end of the paragraph. Elementary correction works as you might expect, with the backspace key wiping out text as the cursor moves to the left. When the screen fills a very fast scroll clears it and puts the last few lines at the top.

Simple editing is almost as easy, using the cursor keys to go back to mistakes. Like WORDSTAR, the WORD sensibly forbids the cursor to stray outside the area of text, but in a welcome departure from WORDSTAR's cursor-handling, the WORD cursor travels intelligently up and down, tucking in towards the left margin to follow short lines, but remembering its original column when line length permits.

By default the editor is in insert mode, so that text to be replaced must first be deleted using the standard delete key. If inserted characters increase the length of any line beyond the margins readjustment of the paragraph takes place automatically. You don't have to tidy edited paragraphs manually, as in WORDSTAR.

Moving, deleting and copying blocks of text is simpler than with WORDSTAR. Instead of visible block markers, the cursor is 'stretched' to highlight the area of text you're selecting. And unlike WORDSTAR, deleted text isn't lost – it gets sent to a special buffer area called 'scrap', from which it can be recalled simply by hitting the insert key. This elegant idea imported from EMACS means that there is no essential difference between deleting and moving text. Progress in learning to use the software is fast because you can try out new functions without any fear of doing permanent damage to your text, thanks to the **UNDO** command which reverses the last action you took. Instead of the '@begin' and '@end' statements used to format EMACS, environment codes are stored at the end of every paragraph and every division. A division is a section of the text established by the user, typically chapters of a book, or preface to an article. The environments are defined by the user in a separate part of the software called the 'gallery', accessed by way of the main menu. The gallery can be used to establish a number of environments, which are then given one or two character codes and stored for future use.

The environments and their definition codes can be stored in a style library, grouped according to context. People who spend their time writing articles, reports, memos and letters tend to develop a individual way of laying out each job. WORD allows you to have style files called, for example, **ARTICLE**, **REPORT**, **MEMO**, **LETTER**, and to attach the appropriate one to the file you are working on. This means that someone regularly generating memos needs to think about the layout only once in a lifetime, not every time one is typed.

Like EMACS, The WORD allows the screen to be split up into windows, and you can have up to eight of them, attaching different text files to different windows. Grasshopper minds can edit several separate documents at the same time; more orderly authors can move text between documents, or use one window to scribble notes on while developing something more fully on another.

The WORD is useful enough on basic hardware, but can also be run as a LIZA-like environment using a mouse and hi-res graphics screen. Thanks to the clarity of WYSIWYG, printing out text holds no terrors, although the WORD is unable to display page breaks dynamically. If you want to know where they're going to appear you have to run a repaginating routine.

Committing it to print

You may feel that the typical seven-needle matrix printers – perfectly adequate for day-to-day computing – may not do justice to your prose. If you are willing to pay the price the more sophisticated, 'intelligent' matrix printers now available will certainly give better results. Professional word processor users will probably choose a fully-formed character printer (*qv*), such as a Diablo or Qume daisywheel, NEC thimble or Ricoh double daisy. Printers of this kind can cost many times more than your micro, so it is obviously tempting to seek economies.

One obvious economy is to accept equipment with a less sophisticated specification and slower printing. You may find this acceptable if:

You don't need to print very much.

You're extremely patient.

You are running a word processor capable of 'background' printing.

Background printing (sometimes known as spooling or despooling) is the ability to operate the printer at the same time as writing or editing the next batch of text. Many word processors for micros can't manage this; WORDSTAR is the exception, but the function has to be correctly installed for the hardware. Without background printing a slow printer can completely take over your system – a waste of a good micro. A typical low-cost printer of this kind takes nearly an hour and twenty minutes to print a twenty page (5,000 word) report.

Also, make sure that your word processing software can be properly adapted to work with your chosen printer and micro. Some quite well-known products can be remarkably sensitive to minor variations in protocol implementation, with the result that the printer stubbornly prints a different character from the one you ask for. If you need to print particular character like the English pound sign or continental accents, ask for a demonstration of the combined hardware and software before you buy.

Installing the printer

One problem with using a word processor that's heavily laden with print facilities is getting the printer to plays its part. Printers usually have a fairly standard set of control codes (based on the ASCII codes) for line feeds, carriage returns and so on. But the codes that the micro must send to produce print enhancements like bold and underscore vary widely from printer to printer.

In order to get the effects you want you need an installation program which will make the word processor send the correct codes to the printer. WORDSTAR is supplied with a program of this kind (called `install`), providing a choice of printer drivers which can be tailored to your own requirements. Writing a printer driver from scratch is a complicated business, and will almost certainly require some knowledge of assembly language. It will need to capture output from the word processor, search for the control codes, and then replace them with the codes that will understood by the printer.

Index

Also available from Century Communications:

BOBBY CATCHES A BUG
Ray Hammond and Michael Cole

BOBBY MEETS A PIRATE
Ray Hammond and Michael Cole

COMPUTERS AND YOUR CHILD
Ray Hammond

DICTIONARY OF COMPUTING AND NEW INFORMATION TECHNOLOGY
A.J. Meadows, M. Gordon and A. Singleton

The ELECTRONIC MAIL HANDBOOK
Stephen Connell and Susan Curran

MICROPHOBIA
Martin Honeysett

THE REALLY EASY GUIDE TO HOME COMPUTING ZX SPECTRUM
Sue Beasley and Ruth Clarke

THE WAY THE NEW TECHNOLOGY WORKS
Ken Marsh

WHAT TO BUY FOR BUSINESS: A HANDBOOK OF NEW OFFICE TECHNOLOGY
John Derrick and Philip Oppenheim

THE COMMODORE HANDBOOK
Peter Lupton and Frazer Robinson

THE ATARI HANDBOOK
Peter Lupton and Frazer Robinson

ASSEMBLY PROGRAMMING MADE EASY FOR THE BBC MICRO
Ian Murray

LOGO PROGRAMMING
Anne Moller

In association with 'Personal Computer World'

BEST OF PCW: SOFTWARE FOR THE BBC MICRO
BEST OF PCW: SOFTWARE FOR THE DRAGON
BEST OF PCW: SOFTWARE FOR THE SPECTRUM

THE CENTURY COMPUTER PROGRAMMING COURSE
Peter Morse and Ian Adamson

COMPUTER GAMESMANSHIP
David Levy

THE DATABASE PRIMER
Rose Deakin

35 EDUCATIONAL PROGRAMS FOR THE BBC MICRO
Ian Murray

EDUCATIONAL PROGRAMS FOR THE DRAGON 32
Ian Murray

EDUCATIONAL PROGRAMS FOR THE SPECTRUM
Ian Murray

INFORMATION TECHNOLOGY YEARBOOK
edited by Philip Hills

THE INTIMATE MACHINE
Neil Frude

THE MICROCOMPUTER HANDBOOK: A BUYER'S GUIDE
edited by Dick Olney

MICROCOMPUTING FOR BUSINESS: A USER'S GUIDE
edited by Dick Olney

MICROWARS ON THE COMMODORE 64
Humphrey Walwyn

THE ORIC HANDBOOK
Peter Lupton and Frazer Robinson

WRITING IN THE COMPUTER AGE
Andrew Fluegelman and Jeremy Hewes

This book will answer all your questions about your Commodore 64, and the world of computers. It is both a machine specific handbook concentrating on one machine alone and a general computer book of real practical use if you have a Commodore 64.

The Micro Enquirer explains all the important aspects of using your computer, with useful listings to act as a practical guide. Topics covered include:

Printers
Communications
Floppy disks
Programming languages
Memory
Software design
Bulletin boards
Cassettes
Games
Wordprocessors
Home control
Modems
Programming techniques

This book does not just concentrate on the peculiarities of your machine, but also puts the technology in perspective by looking at its development, history and influence. It is a reference book too, with a full index offering rapid access to the information you need; lists of codes, instruction sets, interface pin functions and explanations of technical terms.

The Micro Enquirer is the book for anyone with a Commodore 64 and an enquiring mind.

The material in this book is largely adapted from articles published in the magazine *Computer Answers* which have been up-dated and expanded to cover the Commodore 64.

CN 9424

This was brought to you

from the archives of

http://retro-commodore.eu