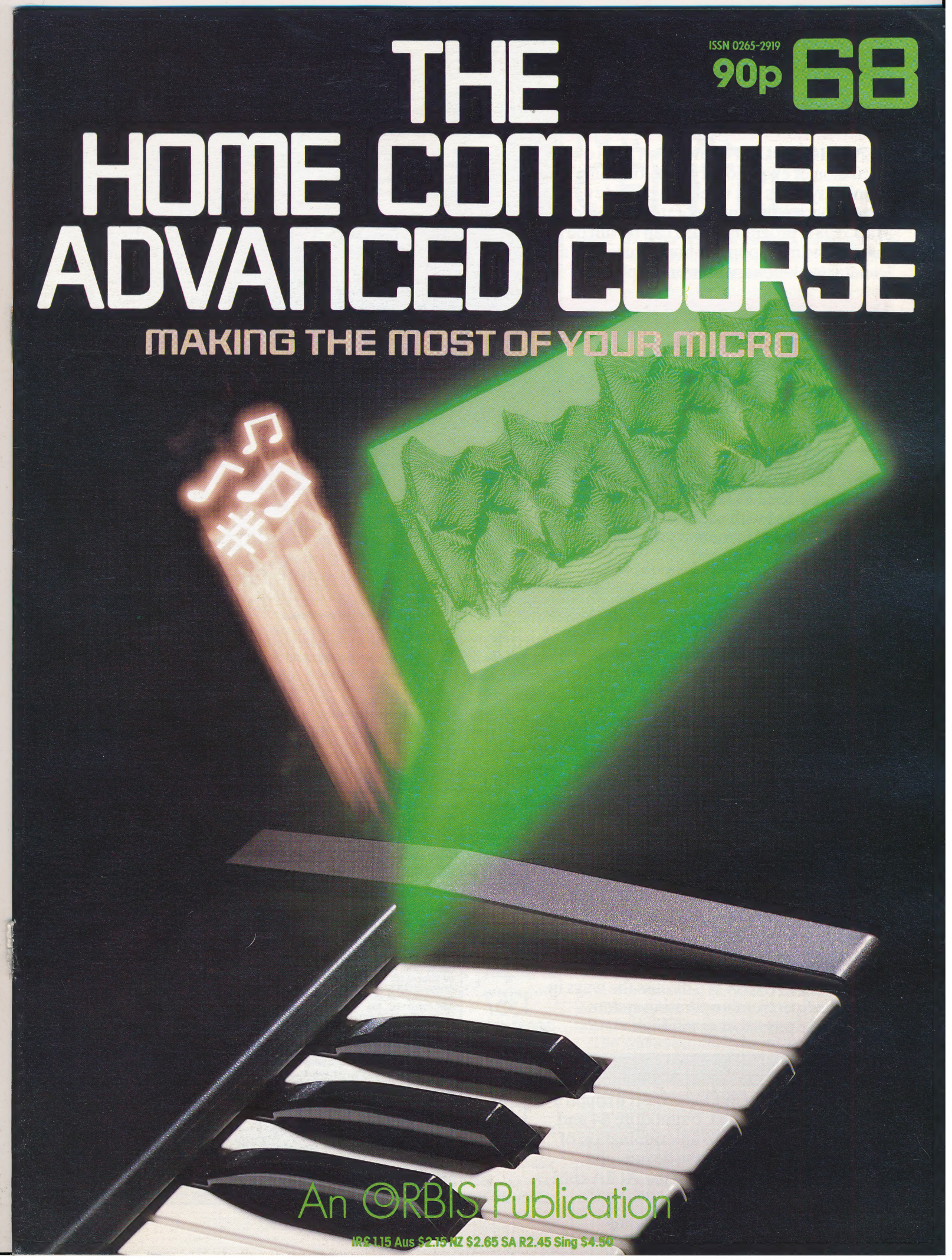
ISSN 0265-2919

90p 68

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

GENETIC CODE Following on from our discussion of evolutionary 'rule-breeding' learning systems last week, we provide a BASIC program that incorporates these ideas. The GENE program, which can be run on three of the most popular home micros, is a 'general evolutionary network explorer' 1348

HARDWARE

MUSIC CENTRE The Echo package from LVL, for the Commodore 64 and BBC Micro, provides a sturdy keyboard on which budding home maestros can compose their digital masterworks 1341

SOFTWARE

A PROMISING FUTURE We wind up our series on CP/M with a look at how the program is stored in memory, and speculate on the system's future 1346

COMPUTER SCIENCE

SELF EXAMINATION A mobile robot simulation written in PROLOG demonstrates the usefulness of the language's ability to treat its own programs as data 1352

JARGON

FROM PRINTER TO PROCEDURE A weekly glossary of computing terms 1360

PROGRAMMING PROJECTS

THE NEW WORLD III After 16 weeks exploring the intricacies of designing a simulation game, the final goal is achieved and your performance can be evaluated 1354

MACHINE CODE

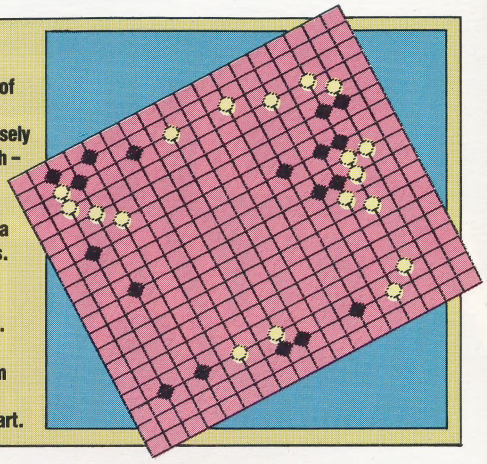
TAPE MEASURES We discuss the ways in which the Spectrum's operating system accesses the tape storage medium 1357

WORKSHOP

CIRCUIT SOUNDS Our project to build a MIDI interface for the Commodore 64 and BBC Micro moves on to an examination of the detailed circuitry for the device, and investigates the operation of the ACIA chip at its heart 1343

Next Week

- **We begin a short series of articles on another programming language closely associated with AI research – LISP.**
- **Having a computer that writes its own programs is a dream of many micro users. We look at 'program generators' – software designed to write software.**
- **Our new programming project to design a program for the ancient Japanese game of Go gets off to a start.**



QUIZ

1) What happens if a file is too large to fit into the Transient Program Area of CP/M?
2) What is contained in the first byte of a header block, and what does the programmer have to specify within it?
3) What happens within the Echo software when the value in the lower six bits of the data register matches that received from the keyboard?

Answers To Last Week's Quiz

1) Natural selection is a process used in AI whereby the computer decides which items in a list of possible solutions work best — thus creating a new list from them which will (hopefully) create a better solution.

2) Because of the problem of ground hum, a break is necessary in the electrical circuit. The opto-isolator is one such break used in a MIDI interface.

3) False. The processors in the Nimbus and the IBM PC are entirely compatible, the difference lies in the different configurations of the I/O system.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Deputy Editor** Steve Colwill; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistants** Caroline Clayton, Mike Clowes; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Graham Storrs, Martin Young, Joe Pritchard, Steve Malone, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. **ADDRESS FOR BINDERS AND BACK ISSUES** – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.

NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY: PAUL CHAVE

MUSIC CENTRE



PAUL CHAVE

Key Notes
The Echo keyboard from LVL, shown here, is the first 'real' musical keyboard to be produced for the BBC Micro or the Commodore 64 that directly accesses the sound chips of those computers and does not require an expensive interface

The Echo music package, intended for use with the Commodore 64 and BBC Micro, attempts to fill the gap in the home computer market created by the lack of products that directly access your micro's sound chip. The package has one of the better keyboards currently available, but the software leaves a little to be desired.

The Echo package from Leasalink Viewdata Ltd (LVL) consists of an authentic three-and-a-half octave keyboard plus the software to run it. Also provided is a manual and a 20-way interface cable to connect the keyboard and computer together, which fits into the machine's user port. If you own a Commodore 64 you'll be provided with two cables — the standard cable, which is for the BBC Micro, and a small adaptor cable that is provided to produce the correct interface for the Commodore machine.

The hardware of the Echo is simple in design; most of the work is performed by the software provided with the package, which obviously keeps the production costs down. The software itself, for both computers, is based around the same specification.

There are two distinct modes available within the software, the first of which is an organ mode. Depressing one of the keys on the computer keyboard will select one of the various 'instruments' listed. These range from Hawaiian guitar to cello and harpsichord, although the settings on each of the two machines are slightly different. A number of parameters will be listed at the top of the monitor, and depending upon which one you select these will allow you to alter the tones of the instrument you are playing.

The Commodore 64 version of the software has the facility to add tremolo and vibrato (but not both) or change to a major or minor key. These effects are produced by pressing one of the number keys.

The software on the BBC Micro version also incorporates the computer's keys, but in this case it is the function keys that allow you to set the tremolo and major or minor chords. The programmers have taken advantage of the extra function keys and the speed of the machine's processor, however, and have added other facilities. If you want to add extra emphasis or effect while playing a piece on the Echo keyboard, you can press one of the function keys, and the appropriate sound — woodblock, bass, cymbal and so on — will be played. Unfortunately, there is no facility to allow accompanying rhythms to run while the keyboard is being played.

PITCH CONTROLS

At the bottom of each screen are the pitch controls. On the BBC Micro version, the pitch can be raised or lowered by pressing the appropriate cursor keys. On the Commodore 64 version, however, this is performed with the < and > keys. Unfortunately, it says something about the quality of the software for the Commodore machine that both of these functions are labelled PITCH MINUS on screen.

What is also unfortunate is that the sounds produced by the 'instrument' keys bear little resemblance to the instruments they are intended to sound like. This is an all too common occurrence in the entire field of electronic music — not just in computer programs. However, the Echo's sound effects seem qualitatively worse than most. For example, the violin, viola and cello on the Commodore 64 version merely sound like an organ held at different pitches for each instrument.

The synthesiser mode is also available to both machines. The parameters provided by the synthesiser mode are dependent upon the facilities provided by the sound chip in each machine. Thus the Commodore 64 version allows you to adjust the envelope parameters of the waveshape and program the filter. As with the organ mode, parameters are altered by pressing the appropriate key on the computer keyboard until the required value is obtained.

ECHO SYNTHESISER
PRICE
£99.95 inc VAT

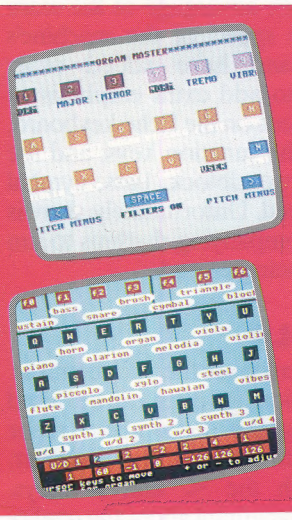
KEYBOARD
37 full-travel piano keys

INTERFACES
Single 20-way cable that plugs into the BBC Micro's user port. Cable adaptor for the Commodore 64

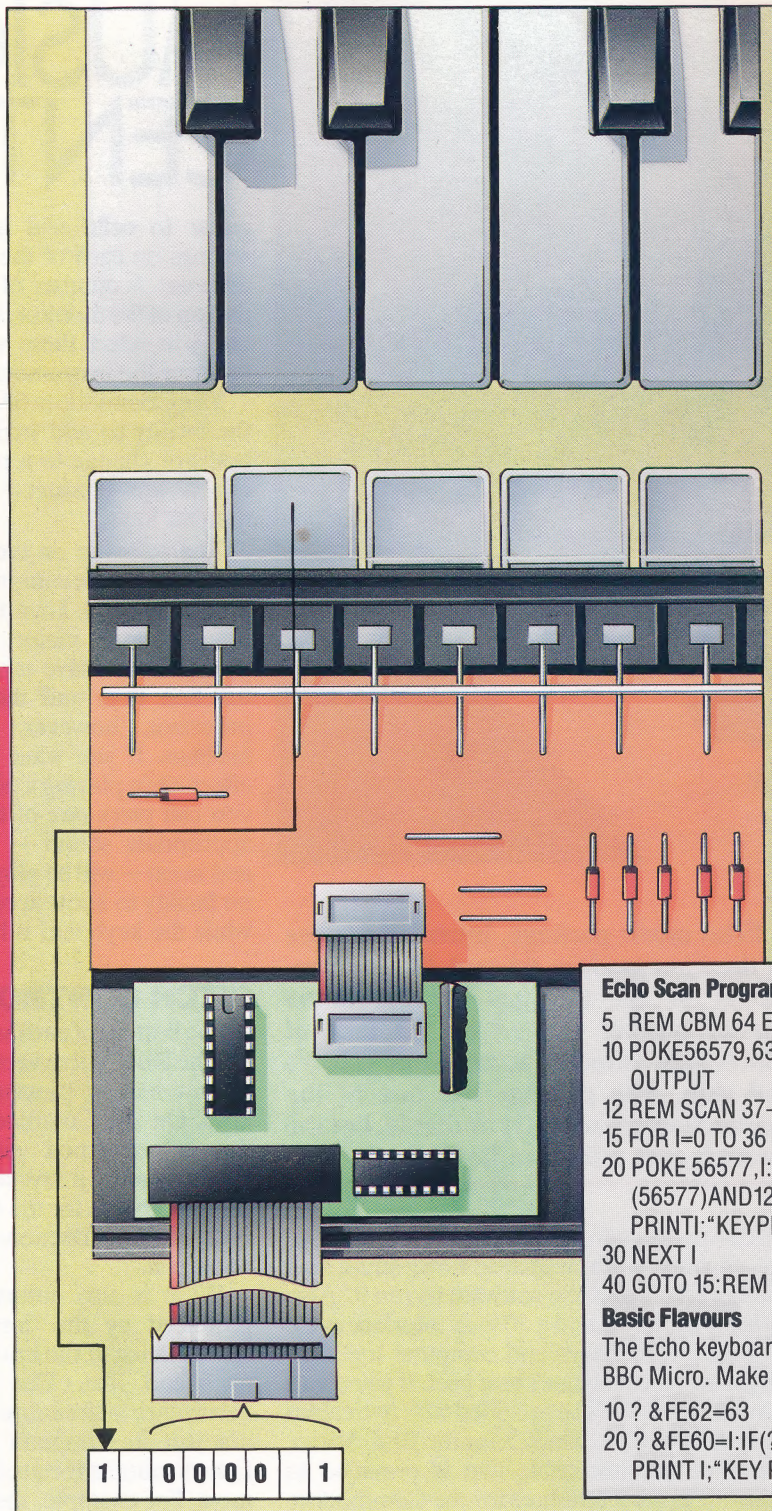
SOFTWARE
BBC Micro version is available on cassette or disk; Commodore 64 version is on cassette only

STRENGTHS
The Echo package allows the full facilities of the sound chip in both micros to be used from an authentic keyboard

WEAKNESSES
The poor quality of the software lets the system down badly, although an improved version is promised



Screen Music
Here are the screen displays for the Commodore 64 software (left) in organ mode, and the BBC Micro in synthesiser mode. Although superficially alike, the versions of the software are quite different. Although both use keypresses from the computer to alter the sounds, the BBC Micro version has a much greater variety of settings than the Commodore 64



Echo Scan

The Echo keyboard plugs into the user port of the BBC Micro or Commodore 64 and can be easily interrogated from software. The principle of operation is very simple. The lowest six bits of the user port data register are set to output so that data can be sent to the keyboard, and the most significant bit (bit 7) is set to input to receive data back from the keyboard.

To check for a keypress on the Echo keyboard, we need to do two things: first, POKE a number between 0 and 36 into the user port data register (the Echo keyboard has 37 keys numbered from zero at the left-hand end). This defines the key we want to test. We then have to test bit 7 of the data register (by ANDing the register value with 128). If this bit is one, then the key in question is currently depressed; the bit having a value of zero indicates the key is not.

In order to scan the whole keyboard, we therefore need a loop to test each of the 37 keys in succession. The program given here shows how this can be done from BASIC.

Echo Scan Program

```
5  REM CBM 64 ECHO SCAN
10 POKE56579,63:REM SET BITS 0-5 FOR
   OUTPUT
12 REM SCAN 37-KEY KEYBOARD
15 FOR I=0 TO 36
20 POKE 56577,I:IF(PEEK
   (56577)AND128)<>0 THEN
   PRINTI;"KEYPRESSED"
30 NEXT I
40 GOTO 15:REM REPEAT
```

Basic Flavours

The Echo keyboard can also be used with the BBC Micro. Make the following changes:

```
10 ? &FE62=63
20 ? &FE60=I:IF(?FE60 AND 128)<>0 THEN
   PRINT I;"KEY PRESSED"
```

KEVIN JONES

Similarly, the synthesiser mode on the BBC Micro makes use of the machine's ENVELOPE command. When this is called, you will be asked which of the four user-defined synthesisers (as opposed to those which are preset in the organ) you wish to alter. The screen then displays the 14 parameters required for the ENVELOPE command; each one is selected by the left and right cursor keys and their values can be altered by the up and down cursors. Once these have been set, you will be free to play the tone on the keyboard.

The software interfaces the computer to the keyboard by placing a value from 36 to zero in the bottom six bits of the user port data register. The method of scanning the keyboard used by the software package means that if a number of keys are held down simultaneously, then only the higher one will register. Unfortunately, this does not allow for rapid fingering changes on the Echo keyboard because the previous key must be fully released before the software will recognise another keypress.

CIRCUIT SOUNDS

In the previous instalment we presented the requirements set out in the MIDI specification. We can now begin to design an interface for the Commodore 64 and BBC Micro, enabling them to communicate with the increasing range of MIDI-compatible equipment now on the market.

The principal component of the interface is an Asynchronous Communications Interface Adaptor (ACIA). The ACIA performs the function of converting the computer's parallel data into the serial form required by MIDI, and vice versa. The device is usually contained within one integrated circuit. We will use the MC6850, which is found in many home computers as a control for asynchronous data transfer to and from cassettes and RS232-type interfaces.

Parts List

The following parts can be obtained from Maplin:

No.	Ref.	Item	Maplin Number
1	C4	1nF polycarbonate capacitor	YY24B
2	C2/C3	100nF polycarbonate cap	YY11M
1	R1	270ohm resistor	M270R
3	R2/R3/R4	220ohm resistor	M220R
2	R5/R6	680ohm resistor	M680R
1		Reel wire-wrap wire	BL82D
2	SK1/SK2	PCB-mounting 180° 5-pin DIN sockets	YX91Y

The following parts can be obtained from Technomatic, 17 Burnley Road, London NW10 1ED

1	IC1	MC68B50 ACIA chip
1	IC2	6N139 opto-isolator chip
1	IC3	74LS04 TTL hex inverter
1		24-pin DIL socket
1		8-pin DIL socket
1		14-pin DIL socket
1	X1	2.0MHz crystal
1	D1	1N914 diode

The DIP breadboard used to mount the components can be obtained from most branches of Tandy: part number 276-164

Circuit Connections

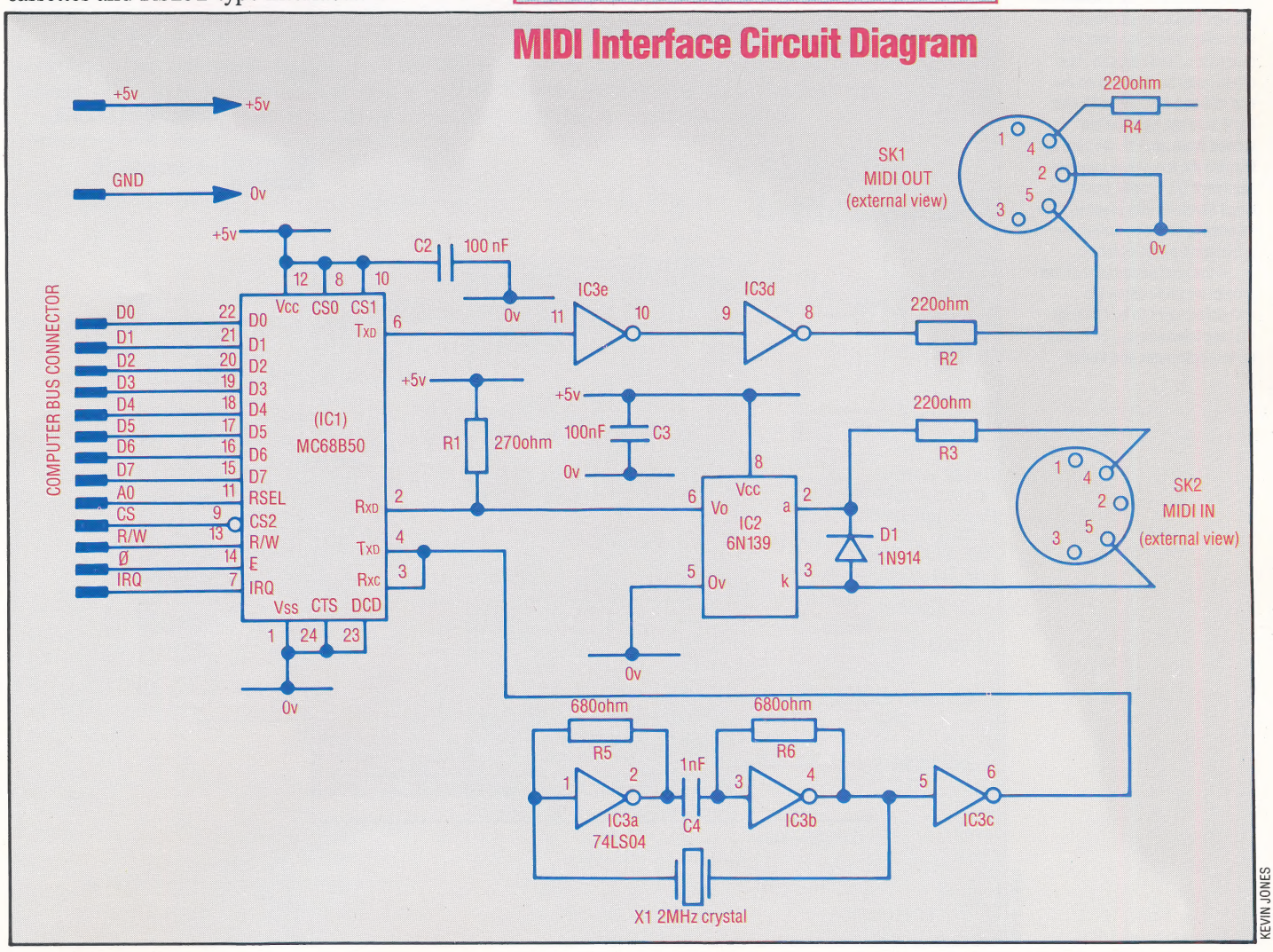
The opto-coupling is achieved by a 6N139 opto-isolator (IC2). The transmission clock is derived independently of the computer clock by a 2MHz crystal oscillator circuit formed around IC3a and IC3b.

IC3c, d and e each provide buffering functions for the transmitter and the clock generator.

The MC6850 is bus-compatible with the 6502 CPU used in many home computers. This allows us to make direct connections from 6502-based micro Read/Write, IRQ and Clock lines. (Note that we have labelled the clock input to the ACIA as Enable (E) to avoid confusion with the transmission clock inputs TXD and TXC).

The two chip select lines — CS0 and CS1 — are connected to the +5v feed and are thus tied permanently high, leaving CS2 as the only active chip select (or device enable) input. This line is forced low for a certain range of addresses, allowing read and write operations on the data bus to access the internal registers of the ACIA.

MIDI Interface Circuit Diagram



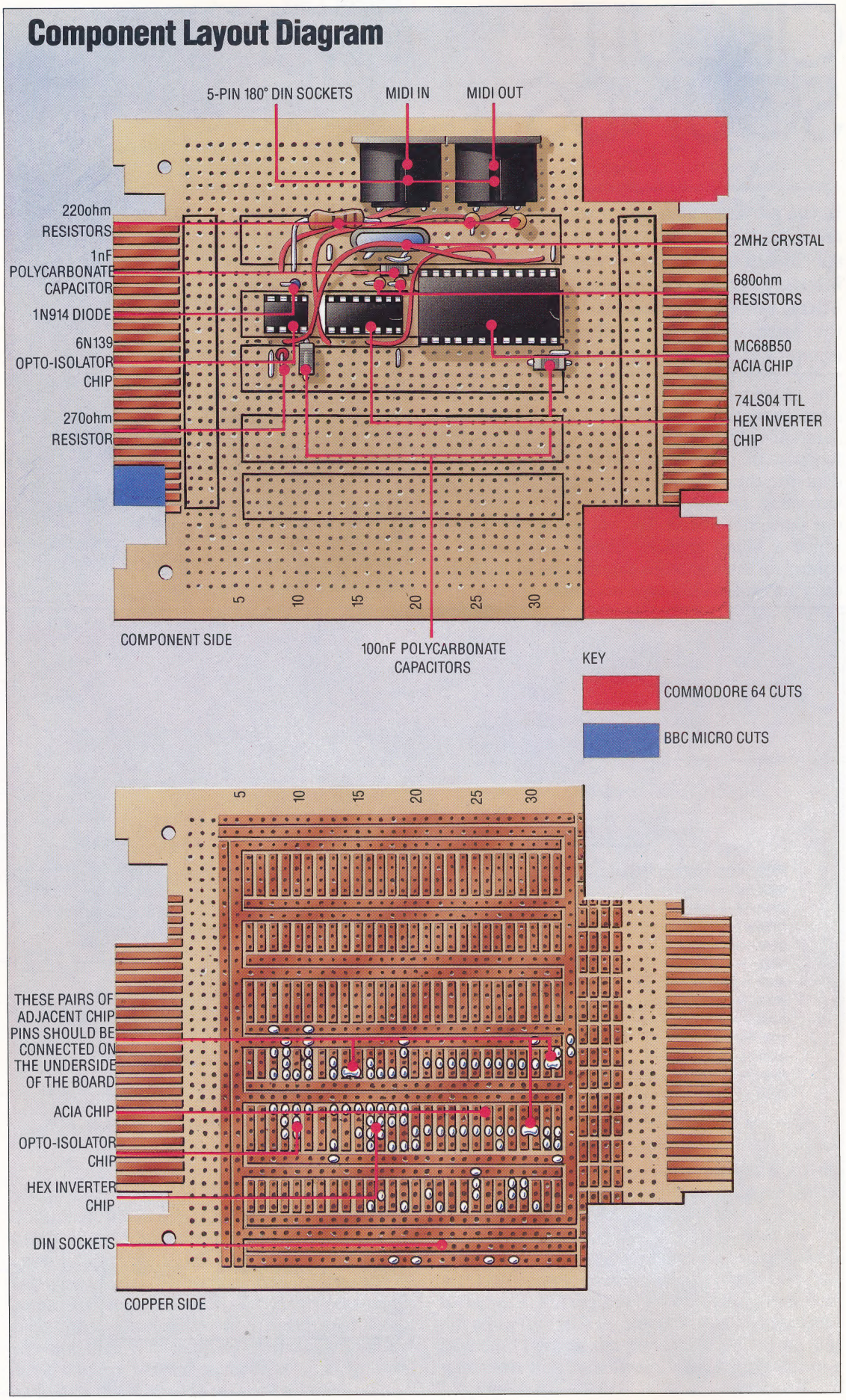
MIDI Construction

The major components for the MIDI interface should be mounted on the special DIP breadboard specified in the parts list. This board has double-sided edge connectors at both ends; the right-hand end being used for connection to the Commodore 64 and the left-hand end for connection to the BBC Micro. Before mounting the components it is necessary to make some cuts to the board. Note that it is only necessary to make cuts to one end of the board or the other depending on the computer you wish to use the interface with. Use a sharp knife or small hacksaw to remove the required portions, as shown. All the links are made on the top of the board using single-strand wire-wrap wire, with the exception of three links between adjacent IC pins. These links should be made by running solder between the appropriate pin pairs on the copper side of the board.

Start by mounting the passive components on the board: the resistors, capacitors, DIN sockets and DIL sockets, as shown. Make the necessary links with wire-wrap wire and mount the 2MHz crystal. The diode must be oriented so the end marked with the coloured band is to the right (when viewed from above). Be sure not to miss the small link beneath capacitor C3. Finally, push the chips carefully into place in their respective sockets, noting the orientation of the notches.

In the next instalment of the project we shall detail the remaining work to be done so that the interface can be linked up to a computer and tested

Component Layout Diagram



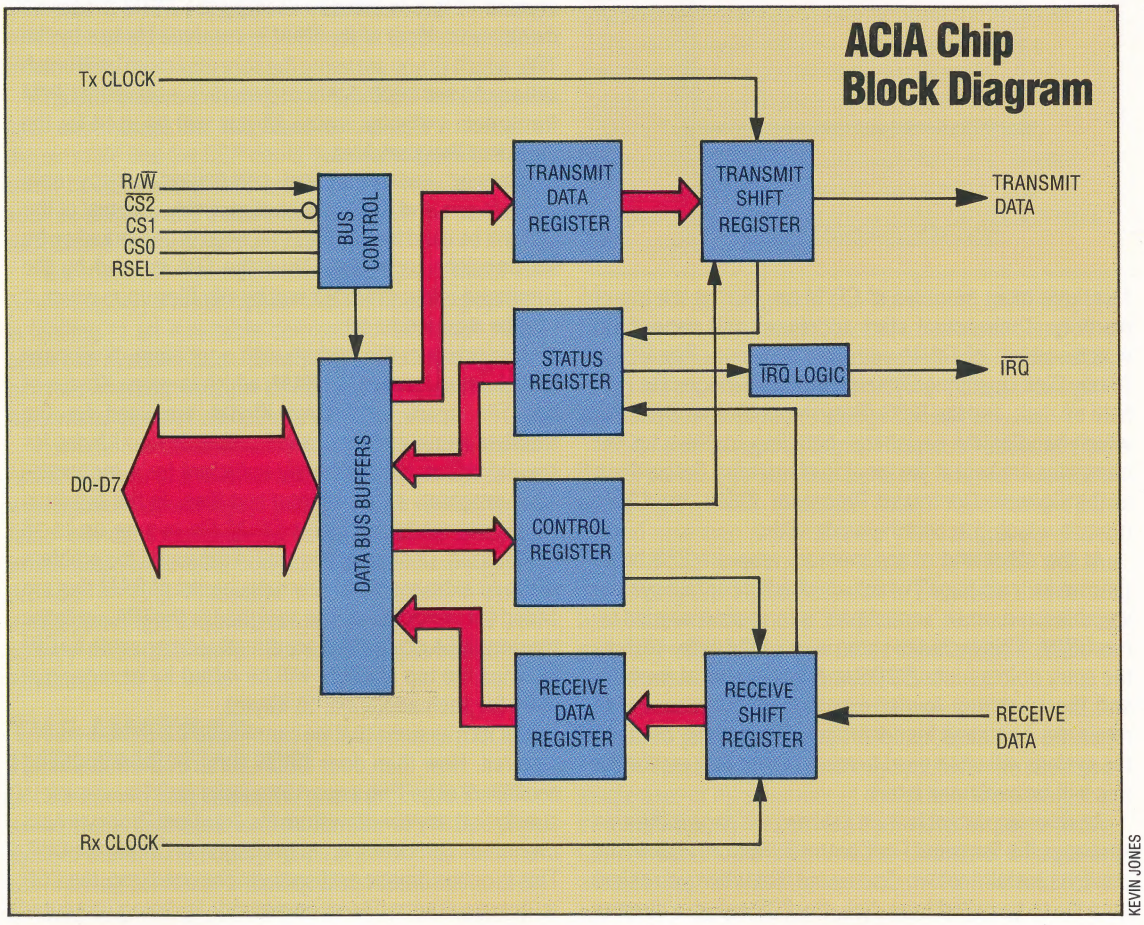
Other components we'll require at this stage include an interface clock, opto-isolator and input and output sockets, which form the basis of the interface. In the next instalment we will detail the computer port connections required to link the interface to the Commordore 64 or BBC Micro.

The internal structure of the device shown consists of a number of eight-bit registers. The Register SELect input line (RSEL) allows access to the registers from the data bus. Although it is not essential to understand the functions of the ACIA registers to run the software that will be provided, users who want to program the interface will need to know the following information:

- The *status register* consists of eight bits that describe the current state of the ACIA chip. They are available to the programmer by performing a read operation on the ACIA with the RSEL line at a low (logic zero) level.
- The *control register* contains eight bits which, as the name suggests, control the operation of the ACIA. The control register is accessed by writing to the ACIA with the RSEL line at a low level.
- The *transmit shift register* (TSR) performs the parallel-to-serial conversion required to transmit a data byte. The register is loaded with a byte from the *transmit data register* (TDR) whenever the TDR is full and transmission of the previous byte is complete. This operation sets bit 1 of the status register. The byte is then shifted out at a rate determined by the transmit clock. At this time, the start and stop bits are added to the eight data bits. This is an internal register and cannot be directly accessed from the data bus.
- The *transmit data register* (TDR) forms a buffer between the TSR and the system data bus. The byte to be transmitted is loaded into this register by writing to the ACIA with the register select line high. This operation clears bit 1 of the status register. If data is not to be lost, TDR should never be loaded when this status bit is clear. This is because the previous byte is still in the TDR awaiting transmission and will be overwritten.
- The *receive shift register* (RSR) performs the serial-to-parallel conversion required when receiving data from MIDI. When the register receives a complete byte, it will load the data into the RDR, setting status bit 0. If this bit was previously set, then status bit 5 will also be set, indicating that the previous byte had not been read by the CPU and that the data will therefore be lost.

If the received byte did not have the required number of start and stop bits (see page 1335), then status bit 4 will be set. This could occur if electrical 'noise' is present on the serial input line. Like the TSR, the RSR cannot be accessed directly.

- The *receive data register* (RDR) is loaded whenever a complete byte is received from the RSR. It therefore contains the most recent data byte. It is accessed by performing a read operation to the ACIA with the RSEL line held high. This operation clears status bit 0.

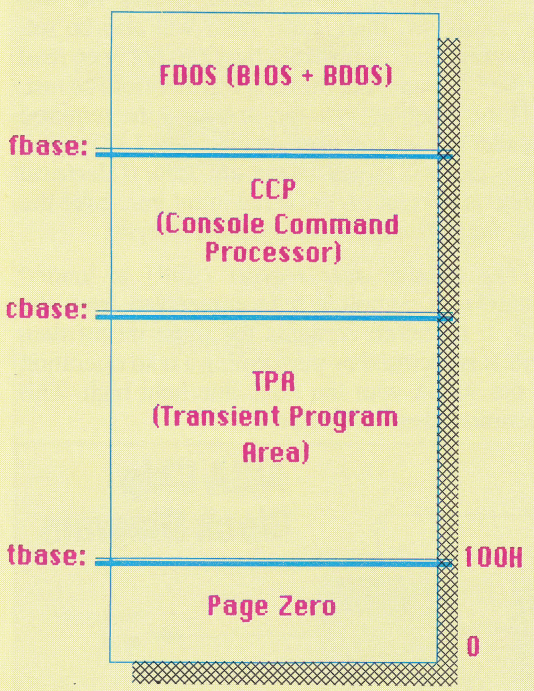


A PROMISING FUTURE

In this final instalment of our series on CP/M, we look at how the system is organised within the computer's memory to make the most efficient use of the available space. We review the three major components of CP/M, and conclude with a peek into the future of the operating system.

Tight Fit

This diagram shows where the various components of CP/M are stored in memory. At the bottom of memory, on zero page, are the system variables and entry points, as well as the bootstrap loader program. Above this is the Transient Program Area. The top boundary of the TPA can be altered to allow files that would not normally fit into the area to overwrite the Console Command Processor. At the top of memory are the BIOS and BDOS routines — which must not be overwritten since they are important to the execution of commands



The three main modules of CP/M are kept at the top of memory. Their actual start addresses will depend on which version of CP/M you may be using. The BDOS (Basic Disk Operating System), which manages the files held on disk, and the BIOS (Basic Input/Output System), which handles the peripheral–handling routines for the computer, are kept at the very top of memory. Below them is the Console Command Processor (CCP), the CP/M module which you communicate with.

At the other end of the memory map are 256 bytes (known as 'zero page') which are used to contain system variables and other 'scratchpad' information needed to run CP/M. This includes some very useful data, such as the entry points to the BDOS and BIOS areas of memory and the bootstrap loader program. It is necessary to retain the bootstrap loader program in memory because the system will often need to reload CP/M (the reason for this will be explained later).

Another important feature held on zero page is the area known as the Transient File Control Buffer, or TFCB. In the previous instalment we discovered that when you asked for a file to be called from disk, the CCP set up a dummy File Control Block in memory. When the BDOS located a file it compared the file with that held by the CCP and then passed further information from the directory track to the CCP. This information is stored in the TFCB.

Between zero page and the CCP sections is an area of memory known as the Transient Program Area, or TPA. This is best thought of as the workspace area of CP/M. As we have seen, when an instruction is submitted to the operating system, the CCP will first search the list of the resident commands held in the CCP area for a match. If it is not there, the CCP will assume that the instruction is a transient command and order the BDOS to locate it. Assuming that the command is resident on the system disk, the BDOS will load a copy of the machine code program into the beginning of the Transient Program Area, which is always at hex location 100 (the start of page one in the computer's memory map). Once loaded, the command is ready for execution and will be run automatically.

The specifications for using CP/M state that a minimum of 16 Kbytes of RAM are necessary. This is really quite a large amount of memory considering that CP/M is intended to remain in the background while other programs are being executed. Fortunately, only a small amount of this is actually used by the CP/M programs themselves. Most of the RAM is reserved for use by the TPA, which (if you are using the bare minimum) will finish at hex address 2900. Since page zero is reserved in CP/M for system variables, seven Kbytes will be used for the Transient Program Area.

However, most transient commands occupy only between two and three Kbytes of memory. The rest of the TPA will be used for any files that need to be worked on by the transient command. Thus the TPA is used by the command being executed and by any files upon which it may be required to act.

This presents the system with a problem. If we have the minimum amount of RAM, which is seven Kbytes of free TPA space, and three Kbytes are used for the transient command, we are left with only four Kbytes for any extra files. This is barely sufficient for a moderate-sized program, let alone long text files. In the previous instalment we found that a CP/M file can be up to 16 Kbytes long. Of course, one way to get round the problem is to add extra memory. Any extra banks of RAM that are added to the system will be assigned to the TPA, up to the maximum 64 Kbytes that are directly addressable by CP/M. Thus the top of CP/M will be raised by hexadecimal 4000 for each additional 16 Kbytes.

But let's assume that we cannot add the extra memory required. How then does CP/M perform the computing equivalent of getting a quart into a pint jar? The answer is simply that any overflow from the Transient Program Area overwrites (or 'overlays', in the jargon of CP/M) the CCP. This is not as drastic as it sounds. For one thing, the CCP is not required while a command is being executed.

CP/M, in common with all operating systems, will not accept any further commands while it is executing the previous one. Therefore, there is no need for the CCP to be present to interrupt them or to execute unnecessary resident commands.

Of course, once the command program has terminated, it will be necessary to reload the CCP programs in order to ready the system to accept the next command. This is performed as the final act of a transient command, and the program will then call the 'location boot' routine, which is found at location 0005 hexadecimal on zero page. This address is the entry point into the operating system that reloads the CCP module back into memory ready to receive the next instruction.

DESIGN CONSTRAINTS

We are now in a position to understand the nature of transient commands. When CP/M was invented the available hardware (RAM chips, in particular) was very expensive, and consequently few machines were equipped with more than 16 Kbytes as standard. Therefore, any software designed to run on these machines had to be tailored to suit the hardware limitations.

In order to make CP/M as comprehensive an operating system as possible within the memory space available some parts had to overlap. Since both the BIOS and the BDOS were required to be resident permanently they could not be used. Many of the commands made use of them for accessing files or performing some input and output operation (such as spooling a file to a printer or displaying it on a screen). Thus it was decided to have the CCP and the TPA 'occupy' the same area of memory, since one could not be used at the same time as the other in any event.

The CCP area could be overlaid by a transient program and once executed the transient program became dispensable and the CCP could be reloaded safely. It is a credit to Gary Kildall and his team that they produced such an efficient and durable system within the hardware restrictions imposed at the time.

Now that business machines have begun to adopt 16-bit processors, it would appear that the operating system is leading to an inevitable demise. But although 16-bit operating systems such as MS-DOS now dominate the business market, the eight-bit Z80-based micro seems to have many years of usefulness yet.

Home and small business micro systems often do not need the power of a 16-bit processor, but their disk drives will require an operating system. CP/M, for so long the de facto standard for eight-bit disk operating systems, would seem an ideal choice for manufacturers eager to provide a large software base without crippling development costs. Some computer manufacturers have already taken this route, most notably Memotech and Amstrad.

THE AMSTRAD IMPLEMENTATION

Of course, users dazzled by the promises of the 10,000 or so software packages written for the CP/M operating system are still faced with several problems. Amstrad users have a particular difficulty. CP/M was developed for 8in and 5¼in disks, and Amstrad has opted for the 3in Hitachi format. This means that it could be some time before many CP/M-based packages appear on the market for Amstrad users.

A further problem with the Amstrad computer is that supporting its screen leaves only 38 Kbytes of free memory for programs, which is far less than some of the more recent CP/M programs require. Therefore, unless some of these programs can be tailored to fit into the available memory space, Amstrad owners could find themselves having to use outdated software. However, by going 'down market', eight-bit CP/M looks as though it will continue to generate a large consumer base.

This is not to say that Digital Research has abandoned the 16-bit market. But the company has not found the going easy, since there are many more competitors around now than at the time Gary Kildall first developed CP/M. A later version of the system, CP/M-86, was the first attempt by Digital Research to break into the market based around the Intel 8088/6 chips. However, this was generally rejected by the market, which opted instead for IBM compatibility and the MS-DOS standard.

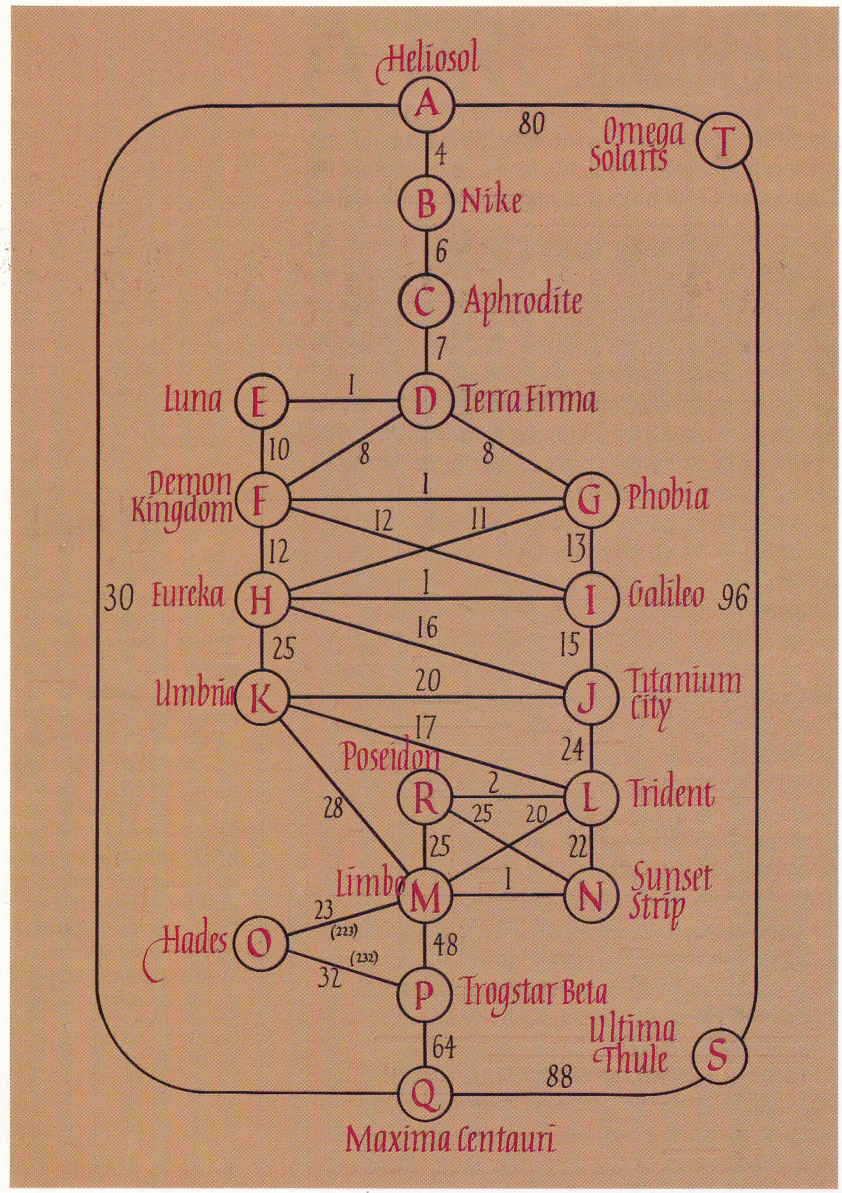
More recently, however, a multi-user version of the CP/M, known as Concurrent CP/M, has been launched by Digital Research. This operating sytem can be used by a single user or by a number of computers linked together in a network. It is perhaps too early to say whether Concurrent CP/M will achieve the success of its eight-bit ancestor, but CP/M looks as though it will be around, in one form or another, for many years to come.

Inside MP/M And CP/NET

In this series we have dealt almost exclusively with CP/M version 2.2, by far the most common version of the operating system available. This version (like all CP/M systems) is designed for single-user, single-machine applications. However, Digital Research has developed other systems (known as 'multitasking systems') based around CP/M for use with a number of machines and users.

A multitasking system is one in which the central processing unit shares its time (by means of 'time slicing') between a number of users or peripherals. The operating system that has been developed to allow a number of people to use the same CPU is known as MP/M (Multi-Processing Monitor Control Program). Using this system a single computer is connected to up to 16 different terminals, all of which are entirely independent of each other. A person at one terminal will not be aware of the other users. In order to cope with all these users, a number of features have been included in MP/M which are designed either to assist in the sharing of files or system devices such as printers.

Another development of the CP/M system which has been introduced by Digital Research is CP/NET. This is a network version of CP/M and allows a number of users on different machines to communicate with each other. Unlike MP/M, CP/NET does not require a master system with subordinate terminals, although there is a 'master node' to control the network. Futhermore, this master node must be running under MP/M and have disk drives connected



BERNARD JENNINGS

Star Gazing
The partygoers' interplanetary map shows the hyperspace routes between the planets, and their respective travelling times. It is possible to travel between planets that are not directly interconnected by hyperspace routes, but any journeys made that do not use the network routes take 1,000 time units each. The GENE program's task is to find quick routes through the network, so that each of the twenty planets is visited

GENETIC CODE

In this second part of our investigation of machine learning, we develop our discussion of the evolutionary model for a self-improving system, which we began on page 1321. We also provide a program, called GENE, that allows us to duplicate the evolutionary process and find better solutions to complex network problems.

When AI scientists look to nature for ideas on how to design self-improving systems, three things catch their eyes — the nervous system, the immune system and the evolutionary process itself. The nervous system, especially the human brain, is a wonderfully effective learning mechanism. But its operations are still mysterious — we have yet to understand it well enough to copy it. The body's immune response system can also be said to learn, in that it comes to distinguish self from other, so that it can attack and destroy foreign bodies. Over the course of a lifetime it learns to recognise millions upon millions of different proteins and without its amazing adaptability and reliability, we would quickly die. Yet it is essentially a rote memory system: its powers of generalisation are rudimentary.

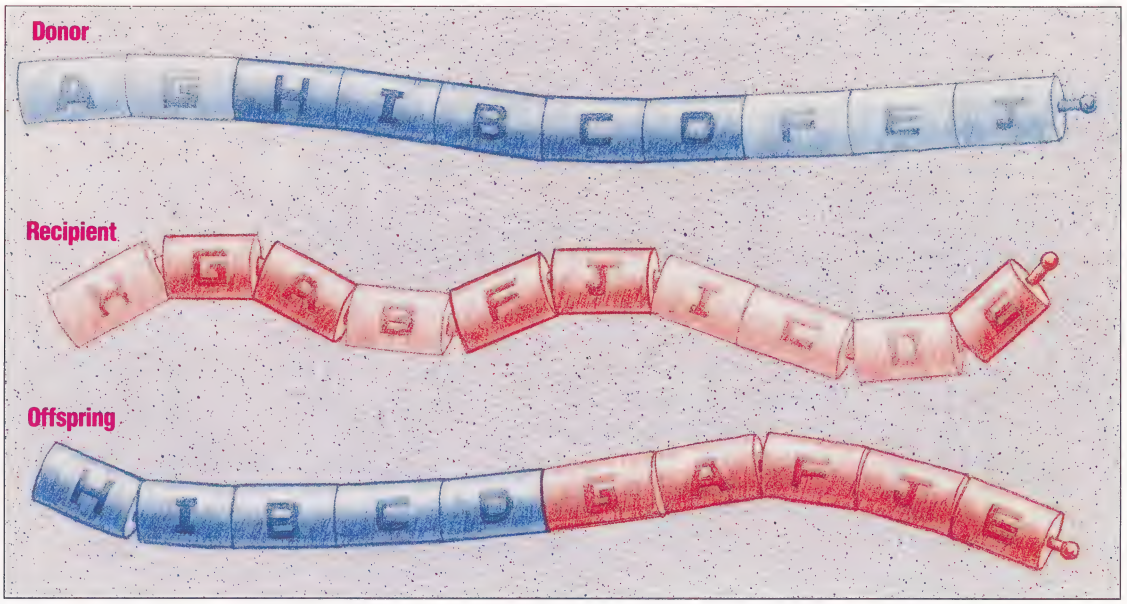
The evolutionary process is certainly effective as a means of devising ever more advanced organisms: it may be a bit slow for our purposes, but can be speeded up in computer simulation. Above all, it is relatively well understood and simple enough for us to copy with some hope of success. In the previous instalment we stated that a 'Darwinian' approach to machine learning had some theoretical advantages. Now we will see what happens when we put it into practice.

To illustrate machine learning using an evolutionary algorithm, we will look at a version of the notorious 'Travelling Salesman Problem' (TSP). This problem is usually posed in the context of the 48 state capitals of the continental United States (excluding Alaska and Hawaii). The salesman has a table of inter-city distances and is required to visit each city once only and return to his starting point. The object is to minimise the total distance travelled.

It sounds deceptively simple, but the number of possible routes or tours is (N-1)! (see page 549) where N is the number of cities. On a tour of 48 cities, there are 47 possible first stops that can be made, followed by 46 possible second stops followed by 45 third stops, and so on. In fact, there are more potential routes than atoms in the universe! Even with 20 cities to visit, the number of potential tours is over a hundred and twenty thousand million million!

There are several approaches to the TSP. It can be treated as a search problem, using methods similar to those we described in the second instalment in this series (see page 1241). It can also be handled by random hit-or-miss Monte Carlo methods. However, we are going to take the unorthodox approach of considering it as a learning problem. We want a method that will explore the enormous number of potential solutions in an economical fashion. We will not expect it to deliver the optimal solution, but we expect it to find a good one.

The GENE (General Evolutionary Network



KEVIN JONES

A Good Breed
GENE breeds new network routes by mating two existing successful routes. As these are held as 20-character BASIC strings, the process of producing offspring is essentially a simple string manipulation. In this example, the substring HIBCD is taken from one of the parents (the donor) and is spliced with the other parent (the recipient), ensuring that no letter is duplicated. This process ensures that, as in real genetic crosses, the characteristics of the parents are passed on to their offspring

Explorer) program that we give here is a learning system specifically adapted to explore networks and develop rules that evolve to produce increasingly economic routes through the network. Before describing how the program works in detail, we need to develop a network for it to explore. A good example can be gleaned from *The Gatecrasher's Guide To The Galaxy,* which lists, among other things, all the planets within an 80-light-year radius where you can find a decent party on a Saturday night. With the aid of this map we can transform the TSP into the PPCD (Planetary Party Crawl Dilemma), in which the objective is to put in an appearance at all 20 parties and return to your starting place in one night.

The map shows the hyperspace trunkroutes in our galactic locality, with the times taken to travel between each stop. Nodes that are not connected by hyperspace expressways are arbitrarily given a traversal cost of 1,000 time units.

EVOLUTIONARY LEARNING SCHEME

In a classic evolutionary learning scheme, there is a population of structures that are treated as 'pseudo-organisms'. Each of these structures (which we will refer to as 'rules') defines a potential solution to the problem at hand. They are also used to propagate new structures (the 'offspring') in ways that mimic some of the features of biological reproduction, such as parents passing on some of their characteristics to their children.

Depending on their performance at the task, rules are selected which are likely to survive the longest and have the greatest likelihood of breeding. In GENE, the organisms/rules have a very simple structure. They are held as strings in the BASIC program. For example, there is:

ABJHMNCDKTSFRQEGILOP

in which each letter denotes one planet in the planetary network and occurs only once in the string. Thus, each 20-character string is a

Alphabetic Engineering

First, two parent structures are chosen at random from those that have survived the culling process (implying that the parents must therefore be fitter than average for the task). One of these is called the 'donor' (R1%) and the other is the 'recipient' (R2%) (see lines 3030 to 3060).

A chunk of 'genetic material' (which is merely a substring) is picked at random from the donor and placed in string S$. Then subroutine 3300 is called, which splices the chunk from the donor with the recipient, by pulling in all the characters of the recipient string (except those already present from the donor) in the order in which they appear. The mating process is therefore asymmetrical. X mating with Y does not give the same result as Y mating with X, even if the random positions P1% and P2% are identical.

Let us look briefly at a scaled-down example. Given the two parents:

Donor: A G H I B C D F E J
Recipient: H G A B F J I C D E

we can select the substring

H I B C D

as the contribution from the donor, and splice in the remaining letters of the recpient to produce:

H I B C D G A F J E

as the result.

It should be stressed that this is not the only conceivable way of crossing a pair of rule-strings. You may be able to think of others. But it does ensure — as in real genetic crosses — that chunks of parental information are passed on to the next generation. It also ensures that each mating produces a 'valid' offspring. This means that the tidying routine at line 4000 of the GENE program has nothing to do; but it is included here for completeness of the program outline.

permutation of the planets to be visited, and defines a particular route around the network.

It is quite simple to evaluate each string by adding up the distance needed to visit the planets in the order specified. The smaller the combined distance travelling the tour, the better the route is.

We arrange for the strings that are worse than average to be deleted after each 'generation'. The question then arises of how to replace them. Obviously, if we pursue the biological analogy, we will be carrying out something similar to sexual reproduction to replace discarded strings. But sexual reproduction is not the only means of generating new strings. Around eight per cent of surviving strings are mutated.

The mutation subroutine, beginning at line 3500, merely performs a certain amount of random swapping. Mutation is not, however, the primary genetic operator but is a background operator that ensures the system does not get stuck at a local optimum. There may be a limit to the improvements that can be made in a successive generation by purely sexual reproductive methods.

If you experiment with the program yourself, you will find that the average standard of the population of rules does indeed improve with time, although — since even the best rules can 'die' eventually — this is not a continuous progression. The program actually cheats by preserving the best rule, and replaces it only with a better one.

You can 'tune' the system by playing with what amounts to the death-rate in line 2520. The random element in this line sets the survival rate of good rules from one generation to the next at 88 per cent, but this can be altered. You can also adjust the mutation rate in line 3520, as well as introduce alternative mutation operators, such as complete inversion of a rule-string.

Two questions remain:

1. How good is the method?
2. Why does it work?

The first can be answered by comparing a pure Monte Carlo approach. All genetic algorithms are in effect modified Monte Carlo procedures. In a pure Monte Carlo method, random solutions are generated and the best solution is retained, all within a specific time-limit. In this example, that would mean trying a random rule-string, evaluating it, keeping it if it was the best yet, and mutating it. The process could be repeated as long as you like.

If you write such a program, you will discover that it quickly finds a fairly good solution, and improves to a lesser degree as time passes. The best solution after 20,000 trials is likely to be only marginally better, if at all, than the one after 10,000. On the whole, genetic algorithms improve for longer durations, and by considering why this is so, we go much further towards answering the second question.

A pure Monte Carlo method is essentially blind search. A genetic algorithm, on the other hand, makes use of what it finds to direct further search. Specific patterns that contribute to good performance are preserved and propagated throughout the knowledge base (the population of rules) and are recombined in slightly different contexts. In effect, the search is preferentially guided towards regions of the (multi-dimensional) 'problem space' in which good results have been found. Unless the evaluation function is extraordinarily discontinuous, you could expect this to lead to a reasonable search strategy.

The GENE Program

The main data structures used to implement our evolutionary learning system are the following arrays.

NAME$(SIZE%)	Names of the network nodes
LINK%(SIZE%,SIZE%)	Inter-node distances
NX%(SIZE%)	Used in the mating routine
R$(NR%)	The rules themselves (strings)
RV(NR%)	The values of each rule

NX% is used when splicing in the recipient's contribution to a new rule that's being made (routine 3300), just so that each node/letter occurs exactly once in the offspring.

R$ holds the current population of rules as strings of length SIZE%. Each one defines a particular tour.

RV gives the 'value' of each rule corresponding to the distance of the tour it represents. If RV(R%)=0, rule R% will be 'dead' and need replacing for the next generation. (Zero-cost tours are obviously not possible.)

SIZE% can be altered if you want to try different map nets, in which case you will need to alter lines 8000 forwards. NR% can be adjusted to see the effect of more or fewer rules in the population

```
  10 REM ************
  11 REM **  GENE  **
  13 REM ************
 100 GOSUB 1000  : REM set up map matrix
 101 G%=0: SN%=0
 105 B=SIZE%*1000: B$=""
 110 INPUT "How many generations ", MG%
 111 GOSUB 1700  : REM initial rules.
 115 IF SN%=0 THEN INPUT "Start Node is No. ", SN%
 120 REM **** MAIN PROGRAM LOOP ****
 130 G%=G%+1
 140 PRINT "Generation ";G%
 150 GOSUB 2000  : REM evaluate rules
 160 GOSUB 2500  : REM kill bad rules
 170 GOSUB 3000  : REM mate good ones
 180 GOSUB 3500  : REM mutations
 190 GOSUB 4000  : REM tidy up
 200 IF G%<MG% THEN 120
 220 GOSUB 5000  : REM dump new rules
 250 END
 999 :
1000 REM -- ROUTINE TO SET UP MAP-NET:
1001 SIZE%=20: NR%=24
1010 DIM NAME$(20), LINK%(20,20)
1011 DIM NX%(SIZE%)
1012 DIM R$(NR%), RV(NR%)
1013 REM rules and rule-values.
1015 FOR I%=1 TO SIZE%
1020 FOR J%=1 TO SIZE%
1022 LINK%(I%,J%)=1000  : REM de
1023 IF I%=J% THEN LINK%(I%,J%)=
1025 NEXT : NEXT
1030 NC%=0 : L%=0
1032 FOR I%=1 TO NR%: RV(I%)=0: 
1033 RESTORE
1040 REM **** READ NAME AND NODE
1050 READ N$,ID%
1055 PRINT N$,ID%
1060 IF ID%<>0 THEN GOSUB 1500
1070 IF ID%<>0 THEN 1040
1080 PRINT NC%;" LOCATIONS READ
1088 PRINT L%;" non-default link
1090 RETURN
1100 :
1500 REM -- INDIVIDUAL NODE & CO
1510 NC%=NC%+1
1520 IF ID%<>NC% THEN PRINT "WAR
" out of order."
1530 NAME$(NC%)=N$
1550 REM **** SET UP INTERNODE D
1560 READ NI%,NT%
1570 LINK%(ID%,NI%)=NT%
1588 REM zero-links don't matter
1590 L%=L%+1
1600 IF NI%>0 THEN 1550
1610 RETURN
1620 :
1700 REM -- Initial Dummy Rules:
1710 S$=LEFT$("ABCDEFGHIJKLMNOPQ
1715 GOSUB 1780 : REM read file
1720 FOR R%= 1 TO NR%
1730 R$(R%)=S$
1740 PRINT S$;R%
1750 I%=INT(RND(1)*SIZE%+1): J%=
1760 GOSUB 6000:REM SWAP
1770 NEXT
1775 RETURN
1777 :
1780 INPUT "Old Rule File (RETUR
1790 IF RF$="" THEN RETURN
1800 F%=OPENUP(RF$)
1810 INPUT #F%, B$, B, SN%
1820 CLOSE #F%
1825 S$=B$
1830 REM only the top one.
1840 RETURN
1850 :
2000 REM -- Rule Evaluation Rout
2010 T=0
2020 FOR R%=1 TO NR%
2030 IF RV(R%)<=0 THEN GOSUB 220
2040 T=T+RV(R%)
2050 NEXT
2060 AV=T/NR% : REM average valu
2070 PRINT "Average score = ": A
2080 RETURN
2100 :
2200 REM -- Single Rule Evaluati
2210 S$=R$(R%)
2220 P1%=SN% : REM start node
2230 GT%=0
2240 FOR S%=1 TO LEN(S$)
2250 P2%=ASC(MID$(S$,S%,1))-64
2260 IF P2%=SN% THEN GOTO 2290
2270 GT%=GT%+LINK%(P1%,P2%)
2280 P1%=P2%
2290 NEXT
2300 RV(R%)=GT%+LINK%(P2%,SN%)
2310 RETURN
2320 :
2500 REM -- Routine to Kill Bad
2510 FOR R%=1 TO NR%
2515 IF RV(R%) < B THEN B=RV(R%)
2520 IF RV(R%) > AV OR INT(RND(1
R$(R%)="": RV(R%)=0
2530 NEXT
2540 RETURN
2550 REM -- NB smaller values ar
2560 :
3000 REM -- Mating Routine:
3010 FOR R%=1 TO NR%
3020 IF RV(R%)>0 THEN GOTO 3120
3030 R1%=INT(RND(1)*NR%+1):IF R
3050 R2%=INT(RND(1)*NR%+1):IF R
3070 REM 'parents' chosen.
3075 P2%=INT(RND(1)*(SIZE%-1)+1
3080 P1%=INT(RND(1)*SIZE%+1) : 
3090 S$=MID$(R$(R1%),P1%,P2%)
3100 GOSUB 3300 : REM splice in
3110 R$(R%)=S$
3120 NEXT
3140 RETURN
3150 :
3300 REM -- Gene Splicing Routi
3310 FOR S%=1 TO SIZE%
3320 NX%(S%)=0 : NEXT
3330 FOR S%=1 TO LEN(S$)
3340 SX%=ASC(MID$(S$,S%,1))-64
3350 NX%(SX%)=NX%(SX%)+1
3360 NEXT
3370 FOR S%=1 TO LEN(R$(R2%))
3380 SX%=ASC(MID$(R$(R2%),S%,1)
```



```
-values.

 : REM default
%(I%,J%)=0

V(I%)=0: NEXT

 AND NODE NO. ****

UB 1500
0
ONS READ IN."
ault links."

NODE & CONNECTIONS:

RINT "WARNING: Node ";ID%;

TERNODE DIST ARRAY ****

't matter.

my Rules:
IJKLMNOPQRSTUVWXYZ",SIZE%)
ead file if any

%+1): J%=INT(RND(1)*SIZE%+1)
P

le (RETURN if none)", RF$
URN

SN%

ne.

tion Routine:

GOSUB 2200

rage value
re = ": AV

 Evaluation:

t node

)
,1))-64
TO 2290
P2%)

2%,SN%)

 Kill Bad Rules:

 B=RV(R%): B$=R$(R%)
INT(RND(1)*100+1)>88 THEN

 values are better.

ine.

GOTO 3120
%+1):IF RV(R1%)<=0 THEN 3030
%+1):IF RV(R2%)<=0 THEN 3050
sen.
ZE%-1)+1)
2E%+1) : REM splice-points
%,P2%)
plice in rest

ng Routine!

)
%,1))-64
-1

$(R2%))
%),S%,1))-64

3390 IF NX%(SX%)>0 THEN GOTO 3420
3400 S$=S$+MID$(R$(R2%),S%,1)
3410 NX%(SX%)=NX%(SX%)+1
3420 NEXT
3440 RETURN
3450 :
3500 REM -- Mutation Routine:
3510 FOR R%=1 TO NR%
3520 IF RND(100) > 8 THEN GOTO 3580
3522 S$=R$(R%)
3525 FOR T%=1 TO 7
3530 R1%=INT(RND(1)*SIZE%+1)
3540 R2%=INT(RND(1)*SIZE%+1)
3560 I%=R1%:J%=R2%:GOSUB 6000:REM SWAP
3565 NEXT T%
3570 R$(R%)=S$
3575 RV(R%)=0
3580 NEXT
3590 REM only swapping at present.
3595 REM needs inversion too.
3600 RETURN
3620 :
4000 REM -- Tidying-up Routine:
4010 RETURN
4020 REM dummy at present.
4040 :
5000 REM -- O/P of results:
5010 PRINT'"Routes are:"
5020 BR=SIZE%*1000
5030 R%=0
5040 FOR I%=1 TO NR%
5050 IF RV(I%)=0 THEN 5100
5060 PRINT I%,RV(I%)
5070 PRINT R$(I%)
5080 IF RV(I%)<BR THEN BR=RV(I%): R%=I%
5100 NEXT
5105 A$=GET$
5110 PRINT
5120 PRINT "Best one is :"
5130 PRINT R$(R%),R%
5131 S$=R$(R%):GOSUB 6400
5133 PRINT"Distance  = ";RV(R%)
5134 A$=GET$
5135 PRINT'"Best ever :"
5136 PRINT B$
5140 S$=B$:GOSUB 6400
5143 PRINT"Distance = ";B
5144 A$=GET$
5145 PRINT
5148 GOSUB 5500 : REM file dump
5150 RETURN
5160 :
5500 REM -- Dump Routine:
5510 INPUT "New Rule File (RETURN if none)", RF$
5520 IF RF$="" THEN RETURN
5530 F%=OPENOUT(RF$)
5540 PRINT #F%, B$, B, SN%
5550 CLOSE#F%
5555 REM only the best.
5560 RETURN
5570 :
6000 REM **** SWAP TWO CHARS IN S$ ****
6040 IF I%>J% THEN T%=I%: I%=J%: J%=T%
6050 X$=MID$(S$,I%,1)
6060 Y$=MID$(S$,J%,1)
6070 S$=LEFT$(S$,I%-1)+Y$+MID$(S$,I%+1)
6080 S$=LEFT$(S$,J%-1)+X$+MID$(S$,J%+1)
6090 RETURN
6100 :
6400 REM **** TRIP ****
6430 PRINT " 0  ";NAME$(SN%)
6440 FOR I%=1 TO LEN(S$)
6450 N%=ASC(MID$(S$,I%,1))-64
6460 IF N%<>SN% THEN PRINT I%;" ";NAME$(N%)
6470 NEXT
6480 PRINT I%;" ";NAME$(SN%)
6490 RETURN
6500 :
8000 REM -- DATA FOR PLANETARY MAP:
8010 DATA HELIOSOL, 1
8020 DATA 2,4, 17,30, 20,80, 0,0
8030 DATA NIKE,2
8040 DATA 1,4, 3,6, 0,0
8050 DATA APHRODITE,3
8060 DATA 2,6, 5,7, 0,0
8070 DATA LUNA, 4
8080 DATA 5,1, 6,10, 0,0
8090 DATA TERRA FIRMA, 5
8100 DATA 3,7, 7,8, 6,8, 4,1, 0,0
8110 DATA DEMON KINGDOM, 6
8120 DATA 5,8, 7,1, 9,12, 8,12, 5,10, 0,0
8130 DATA PHOBIA, 7
8140 DATA 5,8, 9,13, 8,11, 6,1, 0,0
8150 DATA EUREKA, 8
8160 DATA 6,12, 7,11, 9,1, 10,16, 11,25, 0,0
8170 DATA GALILEO, 9
8180 DATA 10,15, 8,1, 6,12, 7,13, 0,0
8190 DATA TITANIUM CITY, 10
8200 DATA 9,15, 12,24, 11,20, 8,16, 0,0
8210 DATA UMBRIA, 11
8220 DATA 8,25, 10,20, 12,17, 13,28, 0,0
8230 DATA TRIDENT, 12
8240 DATA 10,24, 14,22, 13,20, 11,17, 18,2, 0,0
8250 DATA LIMBO, 13
8260 DATA 12,20, 14,1, 16,48, 15,23, 11,28, 0,0
8270 DATA SUNSET STRIP, 14
8280 DATA 12,22, 13,1, 18,25, 0,0
8290 DATA HADES, 15
8300 DATA 13,223, 16,232, 0,0
8310 DATA TROGSTAR BETA, 16
8320 DATA 13,48, 17,64, 15,32, 0,0
8330 DATA MAXIMA CENTAURI, 17
8340 DATA 16,64, 1,30, 19,88, 0,0
8350 DATA POSEIDON, 18
8360 DATA 12,2, 14,25, 13,25, 0,0
8370 DATA ULTIMA THULE, 19
8380 DATA 17,88, 20,96, 0,0
8390 DATA OMEGA SOLARIS, 20
8400 DATA 1,80, 19,95, 0,0
8410 DATA NOWHERE, 0
```

Basic Flavours

The GENE program is written for the BBC Micro.

Commodore 64:

Make the following changes:

```
1820 CLOSE 2
5105 GET A$:IF A$="" THEN 5105
5134 GET A$:IF A$="" THEN 5134
5144 GET A$:IF A$="" THEN 5144
5550 CLOSE 2
```

For disk systems add these lines:

```
1800 OPEN 2, 8, 2, RF$+",S,R":F%=2
5530 OPEN 2, 8, 2, RF$+",S,W":F%=2
```

or these for cassette systems:

```
1800 OPEN 2,1,0,RF$:F%=2
5520 OPEN 2,1,1,RF$:F%=2
```

Spectrum:

Remove % signs from all variables, replace NAME$() by N$(), LINK%(,) by L(,), NX%() by N(), RV() by R(), SIZE% by SI and RF$ by F$ throughout and enclose planet names on DATA lines within quotes. Make the following changes:

```
1010 DIM N$(20,15),L(20,20)
1012 DIM R$(NR,20),R(NR),D$(3,25)

1710 LET  S$="ABCDEFGHIJKLMNOPQRST
     UVWXYZ",(TO SI)

1800 LOAD F$ DATA D$()
1810 LET B$=D$(1)
1820 LET B=VAL(D$(2)):LET SN=VAL(D$(3))

2250 LET P2=CODE S$(S TO S)-64
3090 LET S$=R$(R1)(P1 TO P2-P1)
3340 LET SX=CODE S$(S TO S)-64
3400 LET S$=S$+R$(RS)(S TO S)

5105 IF INKEY$="" THEN GO TO 5105
5134 IF INKEY$="" THEN GO TO 5134
5144 IF INKEY$="" THEN GO TO 5144

5530 LET D$(1)=B$:LET D$(2)=STR$(B)
5540 LET D$(3)=STR$(SN)
5550 SAVE F$ DATA D$()

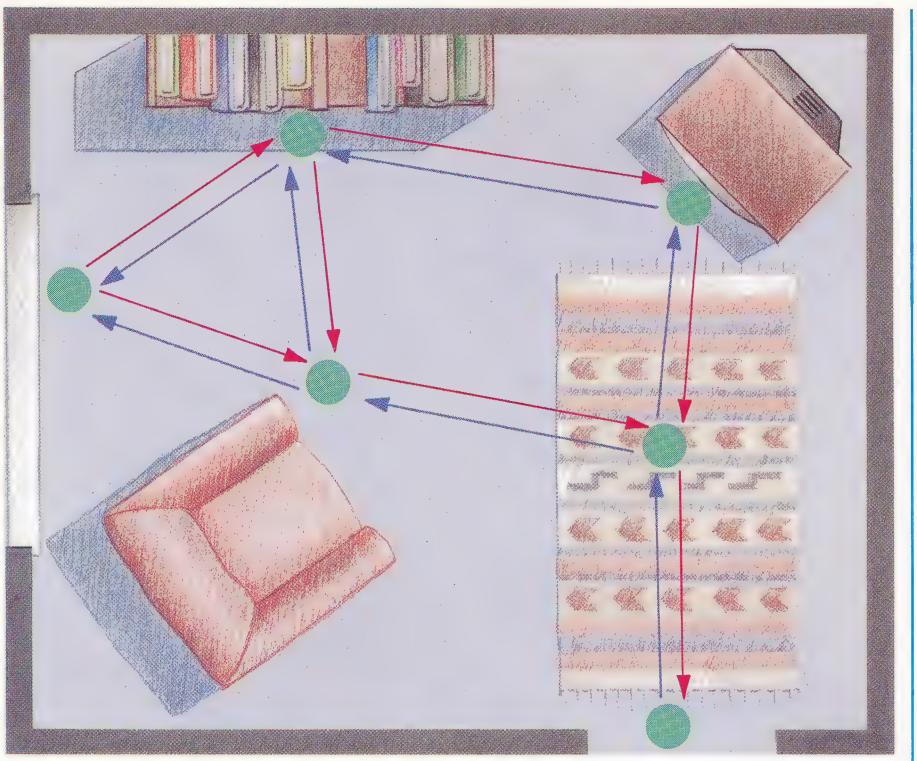
6050 LET X$=S$(I TO I)
6060 LET Y$=S$(J TO J)
6070 LET S$=S$(TO I-1)+Y$+S$(I+1 TO)
6080 LET S$=S$(TO J-1)+X$+S$(J+1 TO)

6450 N=CODE S$(I TO I)-64
```

SELF EXAMINATION

Room To Move
The robot's environment is defined as a set of locations that it 'knows about' – i.e. which have been entered as facts in its database. These locations are linked by a set of 'paths' along which the robot may move and which are stored in the format 'path(tv,bookcase)', and so on. How the robot can use this knowledge to construct a path from A to B is shown in our Mobile Robot Simulation Program

We conclude this series on PROLOG with an examination of the language's ability to use its own programs as data. This is one aspect of the language that makes it ideal for AI programming, and we illustrate its suitability with a program that simulates a robot's movement in a room.



PROLOG has been chosen by the Japanese as the 'core language' of their Fifth Generation computer project. There are two main reasons for this. First, PROLOG terms can have a very similar form to 'relations' in a relational database, and the Japanese see a sophisticated database machine as the basis of their system.

Secondly, PROLOG is an ideal language for writing artificial intelligence (AI) programs. This is mainly because of its ability to use its own programs as data and because its run-time interpreter is rather like the 'inference engines' (see page 1301) of many modern expert systems.

To see how using program statements as data can be useful, we have given a program that is a simple simulation of a mobile robot moving about a room in a house. This program will illustrate a few of the more advanced features of PROLOG and give us a taste of what PROLOG programs look like and how they behave.

The room is defined as a set of places and a set of paths joining them. We want our robot to obey commands, such as 'go from the television set to the chair', and find for itself the shortest route. First, consider the goal go(There), meaning: 'go from wherever you are to There by the shortest possible route'.

There are two clauses for go(There) in the program. The first is to trap the trivial case in which the robot is already at There. Note that for this we need a fact in the database (at(door)), which records the robot's current location. The second clause uses this fact to establish a starting position, and then calls another procedure: go(Place1,Place2).

Because go(Place1, Place2) can be used separately from go(Place), it first checks that Here and There are places it knows about and that Here is in fact the current position. If one of these checks fails, PROLOG abandons the present clause and goes on to the next. The second go(Here,There) clause is there to move the robot to Here if it is not already at that position. It does this with a call to go(Here) and, when that has succeeded, by a further call to go(There). The procedural reading of this clause is: 'to go from (Here) to (There) when not already at (Here), first go to (Here) and then go to (There). The third clause for go(Here, There) is included only to print an error message if neither of the places is known.

THE FINDALL PREDICATE

Assuming the preliminary checks for the first go(Here, There) clause succeed, the next sub-goal will be findall. This is a predicate that is occasionally built into PROLOG but can be added otherwise (like FORTH, PROLOG can be very easily extended by defining new predicates, since user-defined predicates and system predicates have the same status). Findall has three arguments. The first is a variable name, the second is a goal and the third is a list variable (RL).

It works by PROLOG trying to prove the goal it has been given — plan(Here, There, [], Route) in this case — as many times as it can. The variable in the first argument must match one of the variables in the goal and, each time that goal succeeds, the value that that variable has been instantiated to is added to the list in the third argument.

For example, the goal plan(Place1, Place2,[],Route) would find a route between Place1 and Place2 and put it in the variable Route. So in this case findall gathers up all the routes that can be found between Here and There and puts them into a list.

We should note at this point that we have not yet needed to define the procedure for findall. In fact, it is normal practice when programming in PROLOG to develop programs in just this way. Because of its declarative style, we write our procedures in a 'top down' manner, defining the high-level goals first and filling in the details later.

The predicate shortest(RL, Short Route) is another predicate we have not yet defined here. Its job is to take a list



of lists (List1) and to create a new list (List2) containing only the shortest list it finds. As all the lists in List1 are routes, the shortest list will be the shortest route.

We have reached a stage at which our robot has planned the shortest route to its destination; all that remains is for it to move there. Another 'dummy' procedure, walk(Route), enables this. We have defined walk so that it simply writes out the routes, but the route is actually a list of PROLOG goals. These could have been given to PROLOG to execute as a program instead of being printed to the screen. The predicate face(Place) could be defined to perform a sensor scan to locate Place and then turn the robot, while move(Place1,Place2) could be defined to allow the robot to move between the two points.

This program for moving about is written by the main program itself inside the plan procedure. This is simply done. Essentially, the algorithm it uses is: to find a route from A to B, first find a route from A to C, then find a route from C to B. The plan procedure is recursive. The first clause is there to stop the recursion when the destination is reached (that is, there is always a route from A to A).

The second clause in fact does all the work. The declarative reading of this is that a plan exists to go from A to B by route R if:

1. there is a path from A to C, *and*
2. C is not on the already-visited list, V (C will then be added to this list to prevent you from going round in circles in the future), *and*
3. there is a plan to get from C to B by route R1.

When all these conditions are true, the final route, R, is the list containing a program to get from A to C added to the route-so-far (held in R1). Recursive procedures such as this are notoriously difficult to follow (try working through it with a pencil and paper), but they give PROLOG tremendous conciseness.

To complete the program, we now need to update the robot's current position. This is done with the two built-in predicates — retract and assert. The first clause that matches X is removed from the database by retract(X), and asserta(X) adds the clause X as the first of that type in the database (assertz(X) adds X as the last). These predicates are powerful tools for AI programming.

Mobile Robot Simulation

Standard PROLOG Version:

```
at(door).          /* the robot's current position */

go(There):-        at(There),write('I am already there!'),nl,nl,nl.
go(There):-        at(Here),go(Here,There).

go(Here,There):-   place(Here),place(There),at(Here),findall(Route,plan(Here,There,[],Route),RL),
                   shortest(RL, ShortRoute),walk(ShortRoute),retract(at(Here)),asserta(at(There)),saywhere.
go(Here,There):-   not(at(Here)),write('I am not at the'),write(Here),write('so I will go there first.'),nl,nl,
                   go(Here),go(There).
go(Here,There)     write('I can only visit places I know about'),nl,nl.

plan(A,A,_,R).
plan(A,B,V,R):-    path(A,C),not(member(C,V)),append([C],V,V1),plan(C,B,V1,R1),
                   append([face(C),move(A,C)],R1,R).

walk(Route):-      write(Route),nl,nl.    /* to be fully defined later */

saywhere:-         at(Place),write('I am at the'),write(Place),nl,nl,nl.

/* a list of paths the robot knows about */     /* and a list of the places it knows */

path(bookcase,chair).                          place(door).
path(chair,bookcase).                          place(rug).
path(bookcase,tv).                             place(tv).
path(tv,bookcase).                             place(bookcase).
path(tv,rug).                                  place(window).
path(rug,tv).                                  place(chair).
path(rug,chair).
path(chair,rug).
path(door,rug).
path(rug,door).
path(window,chair).
path(chair,window).
path(window,bookcase).
path(bookcase,window).
```

Micro-PROLOG Version:

```
(at door)        /* the robot's current position */

((go X)          (at X) (P I am already there!) PP,PP,PP)
((go X)          (at Y) (go Y X))

((go X Y)        (place X) (place Y)(at X)
                 (findall Z (plan XY()Z)x)
                 (shortest x y)
                 (walk y)
                 (DELCL ((atX))) (ADDCL ((at Y)))
                 (saywhere))
((go X Y)        (NOT at X)
                 (P I am not at the) (PX)
                 (P so I will go there first.) PP PP
                 (go X) (go Y))
((go X Y)        (P I can only visit places I know about) PP PP)
(plan X X x1 Z)
((plan X Y x Z)  (path X X1) ( NOT member X1 x) (append (X1) x x1)
                 (plan X1 Y x1 Z1)
                 (append ((face X1) (move X X1)) Z1 Z))

((walk Z)        (P Z) PP PP)
((saywhere)      (at X) (P I am at the X) PP PP PP)

/* a list of paths the robot knows about */     /* and a list of the places it knows */

(path bookcase chair)                          (place door)
(path chair bookcase)                          (place rug)
(path bookcase tv)                             (place tv)
(path tv bookcase)                             (place bookcase)
(path tv rug)                                  (place window)
(path rug tv)                                  (place chair)
(path rug chair)
(path chair rug)
(path door rug)
(path rug door)
(path window chair)
(path chair window)
(path window bookcase)
(path bookcase window)
```

Programmed Manoeuvres

Information about the robot's environment is entered into PROLOG's database by the user. PROLOG uses the procedure go(place1,place2) to construct a route from A to B — which it does by first finding a route from A to C and then finding a route from C to B.

Note the use of the underline symbol as an argument within the clause plan (A,A,_,R). The symbol can be used in PROLOG as an 'anonymous variable' or 'wildcard' for which no value will be instantiated when the clause is executed. In this particular case, the anonymous variable is used as part of a clause that exists to show that there is always a route from A to A

THE NEW WORLD III

We conclude our New World simulation game project by giving the third and final part of the complete listing, together with the simple alterations that Amstrad owners can make to allow the program to run on their machines.

The program, although written on the Commodore 64, can be made to run (with a few minor alterations) on Amstrad computers. Every occurrence of PRINT CHR$(147) should be replaced by CLS. All lines that wait for a keypress, i.e:

<line no> GET I$:IF I$="" THEN <line no>

should be replaced by:

<line no> I$="" :WHILE I$="" :I$=INKEY$:WEND

Finally, the following line should be inserted so that the 40-column screen display mode is selected:

5 MODE 1

```
6530 REM LIFEBOAT
6535 IFM(1)=1THENRETURN
6536 PRINTCHR$(147)
6540 M(1)=1
6550 S$="A LIFEBOAT IS SIGHTED*":GOSUB9100
6552 S$="DRIFTING IN THE DISTANCE*":GOSUB9100
6554 PRINT:GOSUB9200
6556 S$="THROUGH THE TELESCOPE YOU SEE*":GOSUB9100
6558 S$="THAT IT CONTAINS:*":GOSUB9100
6560 PRINT:GOSUB9200
6562 S$="4 PEOPLE*":GOSUB9100
6563 GOSUB9200
6563 GOSUB9200
6564 S$="AND A LARGE CHEST!*":GOSUB9100
6566 PRINT:GOSUB9200
6568 S$="IF YOU CHANGE COURSE*":GOSUB9100
6570 S$="TO PICK THEM UP*":GOSUB9100
6572 S$="IT WILL TAKE YOU TWO EXTRA DAYS*":GOSUB91
00
6574 PRINT:GOSUB9200
6576 S$="DO YOU WANT TO PICK THEM UP (Y/N)*":GOSUB
9100
6578 INPUTI$:I$=LEFT$(I$,1)
6580 IFI$<>"Y"ANDI$<>"N"THEN6578
6585 IFI$="Y"THEN6600
6588 PRINT:GOSUB9200
6590 S$="THE LIFEBOAT DISAPPEARS....*":GOSUB9100
6592 PRINT:GOSUB9200
6594 S$=K$:GOSUB9100
6596 GETI$:IFI$=""THEN6596
6599 RETURN
6600 PRINT:GOSUB9200
6610 EW=EW+2/7
6625 IFCN<>16 THEN6630
6627 S$="YOU CAN'T PICK THEM UP*":GOSUB9100
6628 S$="YOU HAVE NO ROOM ON THE SHIP*":GOSUB9100
6629 GOTO6592
6630 X=16-CN:IFX>3THEN6635
6632 S$="YOU ONLY HAVE ROOM FOR*":GOSUB9100
6633 PRINTX;"MORE PEOPLE ON THE SHIP"
6634 PRINT:GOSUB9200
6635 S$="YOU PICK UP:*":GOSUB9100
6638 X=0
6640 FORT=1TO16
6645 IFTS(T,1)<>0THEN6679
6650 X=X+1
6655 IFX>4THENT=16:GOTO6679
6660 CN=CN+1
6665 TS(T,1)=INT(RND(1)*5)+1
6668 TS(T,2)=INT(RND(1)*50)+50
6670 PRINT"1 ";C$(TS(T,1))
6679 NEXT
6680 PRINT:GOSUB9200
6682 S$="THE CHEST CONTAINS:*":GOSUB9100
6685 FORT=1TO4
6690 X=INT(RND(1)*10)+10
6692 PRINTX;U$(T);"S OF ";P$(T)
6693 IFPA(T)=-999THENPA(T)=0
6694 PA(T)=PA(T)+X
6695 NEXT
6699 GOTO6592
```

A plague can strike the crew . . .

```
6700 REM PLAGUE STRIKES
6705 IFM(2)=1THENRETURN
6706 PRINTCHR$(147)
6710 M(2)=1
6712 S$="A PLAGUE STRIKES*":GOSUB9100
6714 PRINT:GOSUB9200
6716 X=1
6718 FORT=1TO16
6720 IFTS(T,1)=2ANDTS(T,2)<>0ANDTS(T,2)<>-999THENX
=0:T=16
6722 NEXT
6724 Y=1
6726 IFOA(1)<>0ANDOA(1)<>-999THENY=0
6730 Z=((X+Y)*10)+5
```

. . . and their ability to resist the disease will depend on whether you have hired a doctor and bought medicine before leaving port

```
6732 I$="YOU HAVE ":IFX=0ANDY=0THEN6740
6734 IFX=1THENS$="WITH NO DOCTOR*":GOSUB9100:I$="A
ND "
6736 IFY=1THEN S$=I$+"NO MEDICINE*":GOSUB9100
6740 S$="MANY OF THE CREW ARE AFFECTED*":GOSUB9100
6745 PRINT:GOSUB9200
6750 X=0
6755 FORT=1TO16
6756 IFRND(1)<.3THEN6775
6760 IFTS(T,2)=0ORTS(T,2)=-999THEN6775
6765 TS(T,2)=TS(T,2)-Z
6770 IFTS(T,2)<1THENTS(T,2)=-999:X=X+1
6775 NEXT
6776 IFY=1THEN6780
6777 S$="1/2 YOUR MEDICINE IS USED*":GOSUB9100
6778 OA(1)=INT((OA(1)/2)+.5)
6780 PRINT:GOSUB9200
6785 IFX=0THEN6797
6790 PRINT"AND";X;
6792 S$="CREW MEMBERS DIE*"
6794 IFX=1THENS$="CREW MEMBER DIES*"
6795 GOSUB9100
6796 PRINT:GOSUB9200
6797 S$=K$:GOSUB9100
6798 GETI$:IFI$=""THEN6798
6799 RETURN
```

Pirates may attack the ship . . .

```
6800 REM PIRATES
6805 IFM(3)=1THENRETURN
6810 X=0
6812 FORT=1TO16
6814 IFTS(T,2)=0ORTS(T,2)=-999THENX=X+1
6815 NEXT
6816 IFX=16 THEN RETURN
6818 M(3)=1
6820 PRINTCHR$(147)
```

. . . are there any guns on board to fight them off and minimise losses?

```
6822 S$="  PIRATES ATTACK THE SHIP!*":GOSUB9100
6824 PRINT:GOSUB9200
6825 K=2
6826 IFOA(2)=0OROA(2)=-999THENK=4
6828 S$="IN SPITE OF YOUR GUNS*"
6830 IFK=4THENS$="YOU HAVE NO GUNS*"
6832 GOSUB9100
6835 X=0
6836 FORT=1TO16
6838 IFTS(T,2)=0ORTS(T,2)=-999THEN6845
```



```
6840 X=X+1:TS(T,2)=-999
6842 IFX=KTHENT=16
6845 NEXT
6850 PRINTX;
6855 S$="OF THE CREW IS KILLED*"
6856 IFX>1THENS$="OF THE CREW ARE KILLED*"
6860 GOSUB9100
6865 PRINT:GOSUB9200
6890 S$=K$:GOSUB9100
6895 GETI$:IFI$=""THEN6895
6899 RETURN
6900 REM RUDDER
6905 IFM(4)=1THENRETURN
6910 PRINTCHR$(147)
6915 M(4)=1
6920 S$="TROUBLE WITH THE RUDDER!*":GOSUB9100
6925 PRINT:GOSUB9200
6928 X=4
6930 FORT=1TO16
6935 IFTS(T,1)=3ANDTS(T,2)<>0ANDTS(T,2)<>-999THENX
=1:T=16
6938 NEXT
6940 S$="ALTHOUGH YOU HAVE A MECHANIC*"
6945 IFX=4THENS$="YOU HAVE NO MECHANIC AND*"
6950 GOSUB9100
6955 S$="YOUR JOURNEY WILL TAKE*":GOSUB9100
6960 PRINTX;"WEEKS LONGER"
6965 EW=EW+X
6967 PRINT:GOSUB9200
6969 S$=K$:GOSUB9100
6970 GETI$:IFI$=""THEN6970
6975 RETURN
7000 REM STORM
7005 IFM(5)=1THENRETURN
7010 PRINTCHR$(147)
7015 M(5)=1
7020 S$="YOU ARE BLOWN OFF COURSE*":GOSUB9100
7022 S$="IN A STORM!*":GOSUB9100
7025 PRINT:GOSUB9200
7028 X=2
7030 FORT=1TO16
7035 IFTS(T,1)=4ANDTS(T,2)<>0ANDTS(T,2)<>-999THENX
=1:T=16
7038 NEXT
7040 S$="ALTHOUGH YOU HAVE A NAVIGATOR*"
7045 IFX=2THENS$="YOU HAVE NO NAVIGATOR AND*"
7049 GOTO6950
```

If you are running low on provisions you may decide to put in at this island and replenish your stocks, but in doing so you will lengthen the journey

```
7050 REM ISLAND
7055 IFM(6)=1THENRETURN
7060 PRINTCHR$(147)
7065 M(6)=1
7070 S$="YOUR CHARTS SHOW AN ISLAND*":GOSUB9100
7071 S$="WHERE YOU MAY BE ABLE TO*":GOSUB9100
7072 S$="RE-STOCK YOUR PROVISIONS*":GOSUB9100
7073 S$="BUT IF YOU GO THERE*":GOSUB9100
7074 S$="IT WILL ADD TIME TO YOUR JOURNEY*":GOSUB9
100
7075 PRINT:GOSUB9200
7080 S$="DO YOU WANT TO GO THERE*":GOSUB9100
7082 INPUTI$:I$=LEFT$(I$,1)
7084 IFI$<>"Y"ANDI$<>"N"THEN7082
7086 PRINT:GOSUB9200
7090 IFI$="N"THEN7145
7100 S$="YOU REACH THE ISLAND*":GOSUB9100
7105 S$="AND OBTAIN:*":GOSUB9100
7106 IFB$="N"THEN7110
7107 PRINT:GOSUB9200
7108 PRINT"NOTHING!":GOSUB9200
7109 S$="(REMEMBER THE ALBATROSS!)":GOSUB9100:GOTO
7130
7110 FORT=1TO4
7112 IFRND(1)<.25THEN7129
7115 X=INT(RND(1)*10)+5
7120 PRINTX;U$(T);"S OF";P$(T)
7122 IFPA(T)=-999THENPA(T)=0
7125 PA(T)=PA(T)+X
7129 NEXT
7130 S$="BUT THE JOURNEY WILL NOW TAKE*":GOSUB9100
7135 X=INT(RND(1)*2)+1
7139 PRINTX;:S$="WEEKS LONGER*":GOSUB9100
7140 EW=EW+X
7145 PRINT:GOSUB9200
7150 S$=K$:GOSUB9100
7155 GETI$:IFI$=""THEN7155
7159 RETURN
```

Seven factors can combine to precipitate a mutiny among the crew. This routine generates a mutiny factor, MF, by checking the conditions prevailing at the end of each week of the voyage and adding the appropriate weightings to MF

```
7200 REM MUTINY
7210 MF=0
7215 IFH$="Y"THENMF=MF+30
7220 NC=0
7225 FORT=1TO16
7228 IFTS(T,1)=5ANDTS(T,2)<>0ANDTS(T,2)<>-999THENN
C=1:T=16
7230 NEXT
7235 IFNC=0THENMF=MF+30
7240 IFA$="Y"THENMF=MF-20
7245 IFB$="Y"THENMF=MF+30
7250 IFCN>12THENMF=MF+30
7255 IFWT>MOTHENMF=MF+30
7260 IFWK>8THENMF=MF+((WK-8)*10)
7275 MF=MF+INT(RND(1)*30)
```

If the mutiny factor rises above 75 (but is less than 100) then the player is warned of unrest among the crew

```
7280 IFMF<75 THENRETURN
7282 PRINTCHR$(147)
7284 IFMF>100THEN7300
7285 S$="CONDITIONS ON THE SHIP*":GOSUB9100
7286 S$="ARE GETTING WORSE*":GOSUB9100
7287 S$="AND SOME OF THE CREW*":GOSUB9100
7288 S$="ARE TALKING OF MUTINY!*":GOSUB9100
7290 PRINT:GOSUB9200
7292 S$=K$:GOSUB9100
7294 GETI$:IFI$=""THEN7294
7299 RETURN
```

If the mutiny factor rises above 100 then the crew rebel, listing their grievances before setting the captain adrift in an open boat

```
7300 PRINTCHR$(147)
7305 PRINT:GOSUB9200
7310 S$="THE CREW HAVE MUTINIED*":GOSUB9100
7312 S$="BECAUSE:*":GOSUB9100
7313 X=0
7314 IFH$<>"Y"THEN7320
7315 GOSUB9200:X=X+1:PRINTX;
7316 S$="THEY HAVE BEEN ON 1/2 RATIONS*":GOSUB9100
7318 S$="    FOR SOME OF THE VOYAGE*":GOSUB9100
7320 IFNC<>0THEN7325
7321 GOSUB9200:X=X+1:PRINTX;
7322 S$="THERE IS NO COOK*":GOSUB9100
7324 S$="    AND THE FOOD IS AWFUL*":GOSUB9100
7325 IFB$<>"Y"THEN7330
7326 GOSUB9200:X=X+1:PRINTX;
7327 S$="THE ALBATROSS WAS KILLED!*":GOSUB9100
7330 IFCN<13THEN7335
7331 GOSUB9200:X=X+1:PRINTX;
7332 S$="THE SHIP IS OVERCROWDED*":GOSUB9100
7335 IFMO>=WT THEN7340
7337 GOSUB9200:X=X+1:PRINTX;
7337 S$="THERE ISN'T ENOUGH GOLD*":GOSUB9100
7338 S$="    FOR THEIR WAGES*":GOSUB9100
7340 IFWK<=8 THEN7350
7341 GOSUB9200:X=X+1:PRINTX;
7342 S$="THEY HAVE BEEN AT SEA*":GOSUB9100
7343 S$="    FOR MORE THAN 8 WEEKS*":GOSUB9100
7350 PRINT:GOSUB9200
7360 S$="THE CREW SEIZE THE SHIP*":GOSUB9100
7362 GOSUB9200
7363 S$="AND SAIL AWAY*":GOSUB9100
7370 GOSUB9200
7372 S$="LEAVING YOU ADRIFT IN *":GOSUB9100
7373 S$="AN OPEN BOAT*":GOSUB9100
7374 S$="LET'S HOPE YOU ARE PICKED UP":GOSUB9100
7375 PRINT:GOSUB9200
7380 PRINT"        GAME OVER"
7382 END
```

These short subroutines are used extensively within the program to produce effects, such as slowing down output to the screen

```
9100 REM SLOW PRINT OF S$
9110 FOR S3=1TO32
9115 IFMID$(S$,S3,1)="*"THENS3=32:GOTO9140
9120 PRINTMID$(S$,S3,1);
9130 FORS4=1TO20:NEXT
9140 NEXT:PRINT
9199 RETURN
9200 REM DELAY LOOP
9210 FORS5=1TO1000:NEXT
9299 RETURN
9300 REM REDUCE CREW STRENGTHS BY WF
9310 FORS1=1TO16
9315 IFTS(S1,2)=0THEN9340
9320 TS(S1,2)=TS(S1,2)-WF
9330 IFTS(S1,2)<1THENTS(S1,2)=-999
9340 NEXT
9399 RETURN
```



The ship arrives at its destination. . .

```
10000 REM ARRIVAL AT NEWWORLD
10001 PRINTCHR$(147):GOSUB9200
10005 S$="YOU ARRIVE AT THE NEW WORLD*":GOSUB9100
10006 PRINT:GOSUB9200
10007 S$="AS YOU APPROACH THE SHORE*":GOSUB9100
10009 S$="NATIVES PADDLE OUT TO MEET YOU*":GOSUB9100
10010 PRINT:GOSUB9200
10015 IFOA(2)=0THEN10050
10017 S$="THEY LOOK FIERCE AND ARE ARMED!!*":GOSUB
9100
10018 PRINT:GOSUB9200
10020 S$="DO YOU OPEN FIRE? (Y/N)*":GOSUB9100
10022 INPUTI$:I$=LEFT$(I$,1)
10024 IFI$<>"N"ANDI$<>"Y"THEN10022
10026 IFI$="N"THEN10050
10028 PRINT:GOSUB9200
10030 S$="MANY NATIVES ARE KILLED*":GOSUB9100
10032 S$="BUT DURING THE NIGHT*":GOSUB9100
10034 S$="OTHERS RETURN*":GOSUB9100
10036 S$="AND BURN YOUR SHIP!!!!*":GOSUB9100
10038 PRINT:PRINT:GOSUB9200
10040 S$="      GAME OVER*":GOSUB9100
10042 END
10044 GOTO10042
10050 S$="THEY TAKE YOU TO MEET THEIR*":GOSUB9100
10052 S$="CHIEF. HE HAS MET YOUR RACE*":GOSUB9100
10054 S$="BEFORE AND IS FRIENDLY.*":GOSUB9100
10056 GOSUB9200
10058 S$="THE CREW ARE FED AND RESTED*":GOSUB9100
10060 PRINT:GOSUB9200
10062 S$="THE NEXT DAY TRADING BEGINS*":GOSUB9100
10064 PRINT:GOSUB9200
10066 S$=K$:GOSUB9100
10068 GETI$:IFI$=""THEN10068
10069 RETURN
```

. . .and trading begins

```
10070 PRINTCHR$(147):GOSUB9200:REM TRADING
10072 IFOA(2)=0THEN10080
10074 S$="THE CHIEF DOESN'T WANT YOUR*":GOSUB9100
10076 S$="GUNS - THEY WILL CAUSE TROUBLE*":GOSUB9100
10078 PRINT:GOSUB9200
10080 IFOA(3)<>0OROA(4)<>0OROA(5)<>0OROA(6)<>0THEN
10085 S$="YOU HAVE NO GOODS LEFT*":GOSUB9100
10090 S$="FOR TRADING*":GOSUB9100
10095 GOTO10038
10100 S$="IN EXCHANGE FOR ANY KNIVES*":GOSUB9100
10102 S$="SALT CLOTH OR JEWELS YOU HAVE*":GOSUB910
10104 S$="HE OFFERS YOU PEARLS CARVINGS*":GOSUB910
10106 S$="AND SPICES*":GOSUB9100
10108 PRINT:GOSUB9200
10110 S$="WHEN YOU LEFT PORT THESE WERE*":GOSUB910
10112 S$="WORTH:*":GOSUB9100
10114 S$="PEARLS - 2 GLD PCS EACH*":GOSUB9100
10116 S$="CARVINGS - 2 GLD PCS EACH*":GOSUB9100
10118 S$="SPICES - 1 GLD PC PER GRAM*":GOSUB9100
10120 PRINT:GOSUB9200
10122 S$="BUT THEIR VALUE MAY HAVE*":GOSUB9100
10124 S$="CHANGED WHEN YOU GET HOME*":GOSUB9100
10125 PRINT:GOSUB9200:S$=K$:GOSUB9100
10126 GETI$:IFI$=""THEN10126
10130 FORT=3TO6
10135 IFOA(T)=0THEN10200
10140 PRINTCHR$(147):GOSUB9200
10145 PRINT"YOU HAVE";OA(T);
10150 IFT=3THENS$="BAGS OF SALT*"
10151 IFT=4THENS$="BALES OF CLOTH*"
10152 IFT=5THENS$="KNIVES*"
10153 IFT=6THENS$="JEWELS*"
10155 GOSUB9100
10156 PRINT:GOSUB9200
10160 S$="FOR THESE THE CHIEF OFFERS YOU*":GOSUB9100
10165 PRINT"EITHER";OA(T)*EQ(T-2,1);"PEARLS"
10166 PRINT"    OR";OA(T)*EQ(T-2,2);"CARVINGS"
10167 PRINT"    OR";OA(T)*EQ(T-2,3);"GRAMS OF SPICE"
10168 PRINT:GOSUB9200
10170 S$="DO YOU WANT PEARLS,CARVINGS*":GOSUB9100
10172 S$="OR SPICES?*":GOSUB9100
10174 S$="(ENTER 1,2 OR 3)*":GOSUB9100
10175 INPUTI$
10176 I=VAL(I$):IFI<1ORI>3THEN10174
10180 AO(I)=AO(I)+OA(T)*EQ(T-2,I))
10190 PRINT:PRINT"THE ";T$(I);" ARE PUT ON THE SHIP"
10192 S$=K$:GOSUB9100
10194 GETI$:IFI$=""THEN10194
10200 NEXT
10210 PRINT:PRINT:GOSUB9200
10215 S$="    END OF TRADING*":GOSUB9100
10216 PRINT:GOSUB9200
10218 S$="YOU HAVE OBTAINED:*":GOSUB9100
10220 PRINTAO(1);"PEARLS"
10222 PRINTAO(2);"CARVINGS"
10224 PRINTAO(3);"GRAMS OF SPICE"
10226 PRINT:GOSUB9200
10228 S$=K$:GOSUB9100
10229 GETI$:IFI$=""THEN10229
10230 RETURN
```



Do you get involved in a local coup attempt? The rewards are high — but so are the risks if the coup proves unsuccessful

```
10300 REM REVOLUTION
10305 IFOA(2)=0THENRETURN
10310 PRINTCHR$(147):GOSUB9200
10315 S$="DURING THE NIGHT A RIVAL OF THE*":GOSUB9100
10316 S$="CHIEF VISITS THE SHIP IN SECRET*":GOSUB9100
10317 PRINT:GOSUB9200
10318 S$="HE WANTS TO BUY YOUR GUNS FOR*":GOSUB9100
10320 S$="A REVOLUTION*":GOSUB9100
10322 PRINT:GOSUB9200
10324 S$="HE OFFERS YOU 30 PEARLS PER GUN*":GOSUB9100
10326 S$="DO YOU SELL HIM THE GUNS?(Y/N)*":GOSUB9100
10328 INPUTI$:I$=LEFT$(I$,1)
10330 IFI$<>"N"ANDI$<>"Y"THEN10328
10332 IFI$="Y"THEN10400
10334 PRINT:GOSUB9200
10336 S$="THE CHIEF FINDS OUT AND IS*":GOSUB9100
10338 S$="PLEASED WITH YOU*":GOSUB9100
10340 S$="HE GIVES YOU FREE PROVISIONS*":GOSUB9100
10342 S$="FOR THE JOURNEY BACK*":GOSUB9100
10344 GOSUB9200
10345 IF RND(1)<.75 THEN 10350
10346 S$="AND 50 PEARLS!!*":GOSUB9100
10348 AO(1)=AO(1)+50
10350 PRINT:GOSUB9200
10352 S$=K$:GOSUB9100
10354 GETI$:IFI$=""THEN10354
10359 RETURN
10400 PRINTCHR$(147):GOSUB9200
10405 IFRND(1)<.75THEN10450
10410 S$="THE REVOLUTION IS SUCCESSFUL*":GOSUB9100
10412 PRINT:GOSUB9200
10415 S$="THE NEW CHIEF REWARDS YOU WITH*":GOSUB900
10420 S$="FREE PROVISIONS FOR THE RETURN*":GOSUB9100
10425 S$="JOURNEY.*":GOSUB9100
10429 AO(1)=AO(1)+(OA(2)*30):REM ADD PEARLS
10430 OA(2)=0
10431 GOTO10350
10450 S$="THE REVOLUTION FAILS!!*":GOSUB9100
10452 PRINT:GOSUB9200
10455 S$="THE OLD CHIEF IS ANGRY WITH YOU*":GOSUB9100
10457 S$="HE BURNS YOUR SHIP AND STEALS*":GOSUB910
10458 S$="EVERYTHING!!*":GOSUB9100
10459 PRINT:GOSUB9200
10460 S$="    GAME OVER!!*":GOSUB9100
10462 END
10464 GOTO10462
10500 REM END OF VOYAGE
```

The ship finally returns to port and an evaluation is made of the player's performance as a trader

```
10501 PRINTCHR$(147):GOSUB9200
10505 S$="WITH A STRONG CREW AND GOOD*":GOSUB9100
10507 S$="WINDS THE JOURNEY BACK GOES*":GOSUB9100
10508 S$="WELL AND TAKES ONLY 8 WEEKS*":GOSUB9100
10512 WW=0
10514 FORT=1TO5
10516 WW=WW+(8*CC(T)*WG(T))
10518 NEXT
10519 PRINT:GOSUB9200
10520 S$="WAGE BILL FOR RETURN VOYAGE = *":GOSUB9100
10522 PRINTWW;"GOLD PIECES"
10524 PRINT:GOSUB9200
10526 S$="WHEN YOU GET BACK*":GOSUB9100
10528 S$="YOU SELL YOUR TRADING GOODS*":GOSUB9100
10530 S$="THE GOODS ARE NOW WORTH:*":GOSUB9100
10532 PRINT"PEARLS - ";V2(1);"GOLD PIECES"
10534 PRINT"CARVINGS - ";V2(2);"GOLD PIECES"
10536 PRINT"SPICES - ";V2(3);"GOLD PIECES"
10538 PRINT:S$="YOU GET A TOTAL OF*":GOSUB9100
10540 X=(AO(1)*V2(1))+(AO(2)*V2(2))+(AO(3)*V2(3))
10542 PRINTX;"GOLD PIECES"
10545 PRINT:S$=K$:GOSUB 9100:PRINT
10547 GET I$:IF I$="" THEN 10547
10550 S$="YOU NOW HAVE:*":GOSUB9100
10552 PRINTMO+X;"GOLD PIECES"
10555 PRINT:GOSUB9200
10556 S$="THE WAGE BILL FOR THE VOYAGE IS*":GOSUB9100
10557 PRINTWT+WW;"GOLD PIECES"
10559 PRINT:GOSUB9200
10560 S$="YOU END THE GAME WITH:*":GOSUB9100
10562 Z=MO+X-WT-WW
10565 PRINTZ;"GOLD PIECES"
10566 PRINT:GOSUB9200
10567 PRINT:S$="YOUR RATING IS:*":GOSUB9100:PRINT
10568 IF Z>3200 THEN S$="ARCH CAPITALIST*":GOSUB 9
100:END
10569 IF Z>2500 THEN S$="MASTER TRADER*"GOSUB9100:
END
10570 IF Z>2000 THEN S$="MERCHANT CLASS III*":GOSU
B9100:END
10571 IF Z>1000 THEN S$="A PROPER JONAH*":GOSUB 91
00:END
10572 S$="MORE OF A DUCK THAN A DRAKE*"
10573 GOSUB9100:END
```

TAPE MEASURES

We investigate the ROM routines used by the Spectrum's operating system to access the tape filing system, which is the only method of storage provided on an unexpanded machine. We also provide programs to demonstrate how the save and load routines are used from machine code.

A standard Sinclair Spectrum is fitted with only a tape interface to allow the saving and loading of programs and data. The addition of Interface 1, however, gives us access to Microdrives, the serial interface and local area networks. These can be seen as being alternative filing systems in much the same way that the BBC Micro has alternatives available to it. However, on the BBC Micro we select the filing system of interest by a * command; on the Spectrum, we use a different command syntax to differentiate between commands meant for the tape filing system and any of the other filing systems available to the computer. For example, the command:

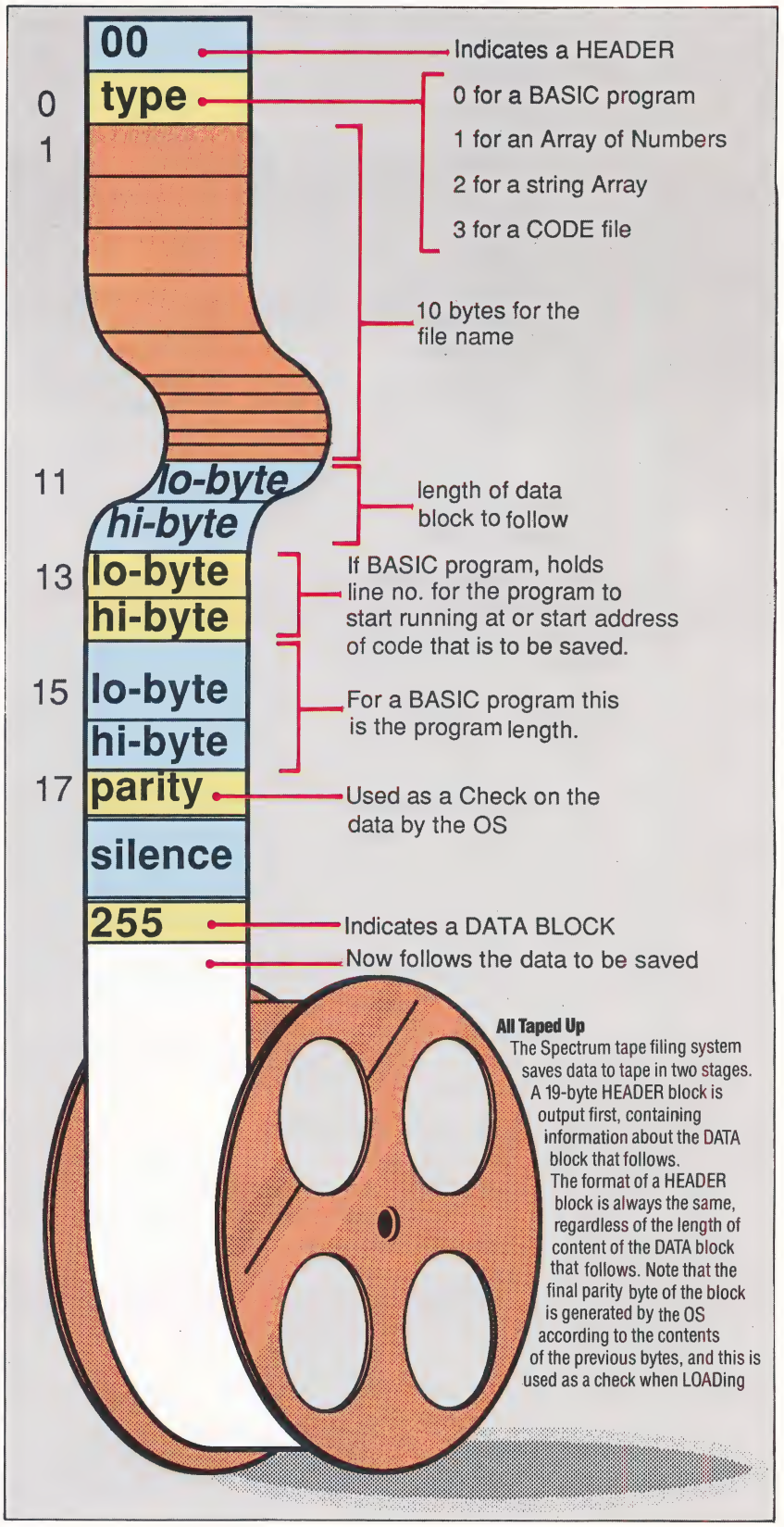
```
SAVE "fred1" LINE 1
```

will save a BASIC program to tape so that it will start running from line 1 when it is loaded back into the Spectrum. To save a program in a similar way on the Microdrive filing system requires the incredibly cumbersome command:

```
SAVE *"m";1;"fred1" LINE 1
```

The "m" part of the instruction specifies a Microdrive is being used; the 1 following this indicates the number of the drive required; and the filename — "fred1" in this case — follows this. We'll look at how the Microdrive filing system works in a later part of the course. Here we'll examine the use of the tape filing system from our machine code programs.

Irrespective of what the data being saved on the tape is, it is always arranged in the way shown in the diagram. The first 19 bytes sent to the tape are regarded as the *header*, and these bytes contain data relating to the following block of data on the tape. The first and last bytes of the header are provided by the operating system routines that write the data bytes out to the tape. The other bytes need to be specified before the ROM routine which writes the data to the tape is called. The 'type' byte indicates to the OS, on loading, the nature of the data in the block. Following this is a 10-byte file name — the reason why file names are limited to 10 characters on the Spectrum. Even when the file name is less than 10 characters long, the remaining spaces in the file name are filled out with the ASCII code for a space (32). The data in



All Taped Up
The Spectrum tape filing system saves data to tape in two stages. A 19-byte HEADER block is output first, containing information about the DATA block that follows. The format of a HEADER block is always the same, regardless of the length of content of the DATA block that follows. Note that the final parity byte of the block is generated by the OS according to the contents of the previous bytes, and this is used as a check when LOADing

the header from byte 11 onwards provides the OS with other details, such as information regarding where to load the data into memory. The final byte of the header is used to check for any errors when the tape is read in — if something is detected, the appropriate error message is generated.

The main data block itself starts with a single byte holding the value 255; this is again provided by the ROM 'write-to-tape' routine. Then comes the data, followed by another check byte, also generated by the 'write-to-tape' routine. In both the header and data blocks, the actual value written as the last byte of the block depends upon the bytes sent immediately before it.

Let's now look at how we can use the cassette interface routines from our own machine code programs. The ROM routines that are concerned with tape operations on the Spectrum are to be found in the addresses between &04C2 and &09F3 of the Spectrum ROM. It is thus quite a sizeable piece of software. This is not surprising, because all the timing operations and tone generation needed to put information on tape are software-based in the Spectrum.

There are no system variables concerned specifically with tape operations, but the following table contains a few that are used 'incidentally' by the routines.

System Variable	Description
TADDR (at locations 23668 and 23669)	Used by the ROM routines to decide which tape operations to perform when interpreting a BASIC command
BORDCR (at 23624)	Used to restore the Border Colour to its original colour after a tape operation
XPTR (at 23647 and 23648)	Used to temporarily store the IX register

The system variables VARS, ELINE and PROG are also used when loading information from tape into the Spectrum.

SAVING TO TAPE

First of all, let's consider the operation of saving data to tape — the routine which does this is found at ROM address &04C2. In a typical use, this routine is called *twice*; once to save the header information and a second time to store the actual data that we want to save to tape. The entry requirements for this routine are given in the table at the bottom of the preceding column.

Register	Header	Data Block
A	0	255
DE	17	Length of data
IX	Address of the start of the header or the start of data	

As can be seen from this table, the IX register is used to point to the data that is to be saved. To clarify matters, let's look at a simple example of how to save a block of data to tape. The following program will save 100 bytes of data (beginning at ROM address 0000) to tape.

```
        ;routine to save 100 bytes, starting address
         0000, to tape
        ;
        ;send header
3E00            ld   a,0        ;indicate a HEADER BLOCK
DDE5            push ix         ;save ix on stack
DD21D328        ld   ix,header  ;add. of header block
111100          ld   de,17      ;no of header bytes
CDC204          call #04c2      ;write header to tape
        ;send data
DD210000        ld   ix,0000    ;add of 1st byte to save
116400          ld   de,100     ;no of bytes to be saved
3EFF            ld   a,255      ;indicate a DATA BLOCK
CDC204          call #04c2      ;write data to tape
DDE1            pop  ix         ;restore IX value
C9              ret             ;back to BASIC
03      header: defb 3          ;data type 3=CODE block
54455354        defm "TESTPROG  ;f/name+spaces=10 chars
64              defb 100        ;lo-byte of data-length
00              defb 0          ;hi-byte of data-length
00              defb 00         ;lo-byte of start add.
00              defb 00         ;hi-byte of start add.
00              defb 00         ;used for BASIC only
00              defb 00         ;for start line no.
```

Calling this routine will save the relevant area of memory. However, no prompt will be issued (as would be the case with tape handling routines from BASIC). This isn't really a problem as the data on tape can be reloaded to see if the program has done its job correctly, using this command:

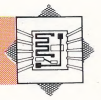
```
LOAD "TESTPROG" CODE 40000
```

Now compare the bytes loaded in with the bytes in locations 0 to 99 in ROM. Remember that although the HEADER data block starts with the type byte, the first byte actually sent to tape is provided by the OS (0 for a header and 225 for a data block).

A useful modification of this program can enable you to save a BASIC program from within a machine code listing. The HEADER block at the end of our previous listing should be altered to read:

```
00      header: defb 0          ;0=BASIC program
73737373        defm "sssss     ;fname filled to 10 chars
                ssssss"
00              defb nn         ;lo-byte) length of
00              defb nn         ;hi-byte) prog+variables
00              defb nn         ;lo-byte) start line
00              defb nn         ;hi-byte) number
00              defb nn         ;lo-byte) length of
00              defb nn         ;hi-byte) program only
```

The length of the program and variables can be worked out by subtracting the value held in PROG from the value held in ELINE, and the length of the program alone can be obtained by subtracting the value held in PROG from that in VARS. If you do not want the program to start running once it is loaded, then simply set the 'line number at which the program is to start running' entry in the listing to 32768. Similarly, we can simulate the SAVE SCREEN$ command from machine code by specifying the start address of the code to be saved as &4000 and the length as &1B00.



This routine could easily be extended by using techniques that we've learned in earlier parts of the course; for example, a prompt could be put on the screen, and a simple routine could be added to wait for a key to be pressed.

LOADING FROM TAPE

The ROM routine that loads data from tape is also called twice: once for the HEADER and again for the actual data block that is to be loaded. The routine is located at address &0556 in the ROM and its entry requirements are given in the following table (along with those for verify code):

Register	Load	Verify
C Flag	1	0
A	0 for header, 255 for data block	
IX	Points to place in memory to which bytes are to be loaded	
DE	Number of bytes to be loaded; D must be in the range 0 to 254	

We will also require some workspace for this routine; about 34 bytes, or enough to accommodate two HEADER blocks. The reason for this is quite straightforward. In order to load a named file to an address within RAM that we specify, we must first set up a second HEADER block containing details of the file that we want to load. We then compare the HEADER data of the files on tape with the HEADER we've set up in memory in order to locate the file we require.

On calling the routine (at &0556), the C flag should be set as indicated in the table. If you are verifying a piece of code, this should be located at the address pointed to by the IX register. On leaving the ROM 'load-from-tape' routine, the C flag will indicate the result of the operation. If it is set to one, then the load was performed smoothly. But if, for example, you were trying to load a HEADER block, and the first thing encountered on tape was a block of data, then the C flag will be set to zero. (Any tape loading errors will be handled by the Spectrum OS.) Let's now look at a simple example that will load a HEADER from tape into RAM.

```
        ;load a header from tape into RAM
        ;
37              scf             ;set carry 1=LOAD
3E00            ld   a,0        ;0 indicates a header
DDE5            push ix         ;save ix on stack
DD2148EE        ld   ix,61000   ;load header to 61000
111100          ld   de,17      ;no of bytes in header
CD5605          call #0556      ;do it
DDE1            pop  ix         ;restore ix
C9              ret
```

Once the HEADER is loaded then we can examine it. We can compare the file name with the one we require, ensure that its the correct type of file and check the length of the data block if we need to. The following program checks the name only, and if it is correct loads the data block following the HEADER into memory at a particular address.

```
        ;locate and load TESTPROG
        ;
DDE5            push ix         ;save IX on stack
37      loop:   scf             ;set carry for LOAD
3E00            ld   a,0        ;0=BASIC file
111100          ld   de,17      ;no of bytes in header
DD21CA2B        ld   ix,head2   ;load header into head2
CD5605          call #0556
D27B2B          jp   nc,loop    ;if no header then again
060A            ld   b,10       ;no of bytes in fname
11BA2B          ld   de,head+1  ;point to desired f/name
21CB2B          ld   hl,head2+1 ;point HL to found fname
1A      name:   ld   a,(de)     ;get header char in a
BE              cp   (hl)       ;comp with found header
2007            jr   nz,no      ;different so exit loop
13              inc  de         ;get next char
23              inc  hl
05              dec  b          ;decrement counter
20F7            jr   nz,name    ;same so check next char
1802            jr   ok         ;all chars checked + ok
18DB    no:     jr   loop       ;check failed, try again
        ;found right header so prepare to load data
DD21B92B ok:    ld   ix,head    ;get head in ix
DD6E0D          ld   l,(ix+13)  ;get address to
DD660E          ld   h,(ix+14)  ;load data into
E5              push hl         ;push address onto stack
DDE1            pop  ix         ;and get it into ix
3EFF            ld   a,255      ;indicate load data
        ;next instruction requires user to insert data
         length
11AF2B          ld   de,nn      ;insert data length
37              scf             ;indicate a LOAD
CD5605          call #0556      ;doit
DDE1            pop  ix         ;restore ix
C9              ret
        ;now follow details of desired f/name
03      head:   defb 3          ;indicates CODE block
54455354        defm "TESTPROG" ;f/name
00              defb 0          ;use for data length
00              defb 0          ;data-length hi-byte
48              defb 72         ;lo-byte for load
EE              defb 238        ;address of 61000
00              defb 0          ;used for BAS. prog only
00              defb 0          ; ditto
        ;now follows space for header loaded for
         checking
        head2:  defs 17
```

Again, no messages are printed to the screen during this operation. The routine will load a data block called TESTPROG to the address specified in the HEAD area of memory. The routine can be used to load files of a different name, and you could also add code to check that the type of file is correct. One thing to note is that these ROM routines, like the BASIC SAVE and LOAD operations, can be terminated by pressing the Break key.

We conclude this discussion of the save and load operations of the Spectrum OS with a short program that reads in file headers from tape and prints details about the file to the screen — such as the file's length, start address, and so on. The machine code routine is stored in the DATA statement, and it simply loads in a HEADER block as outlined above, and then examines the block from BASIC.

```
5 CLEAR 59999
10 FOR I=0 TO 19
20 READ A:POKE (60000+I),A
30 NEXT I
40 RANDOMIZE USR 60000
50 LET type=PEEK 60020
60 LET length=PEEK 60031+256*PEEK 60032
70 LET start=PEEK 60033+256*PEEK 60034
80 LET L$=""
90 FOR I=60021 TO 60030: LET L$=L$+CHR$(PEEK I): NEXT I
100 PRINT "Name: ";L$
110 IF type=0 THEN LET f$="BASIC"
120 IF type=1 THEN LET f$="Number Array"
130 IF type=2 THEN LET f$="String Array"
140 IF type=3 THEN LET f$="CODE"
150 PRINT "Type: ";f$
160 IF type=0 THEN PRINT "Auto Run line number: ";start
170 IF type<>0 THEN PRINT "Start address: ";start
180 PRINT "Length: ";length
190 GOTO 40
200 DATA 221,229,62,0,55,221,33,116,234,17,17,0,205,86,
    5,48,241,221,225,201
```

PRINTER

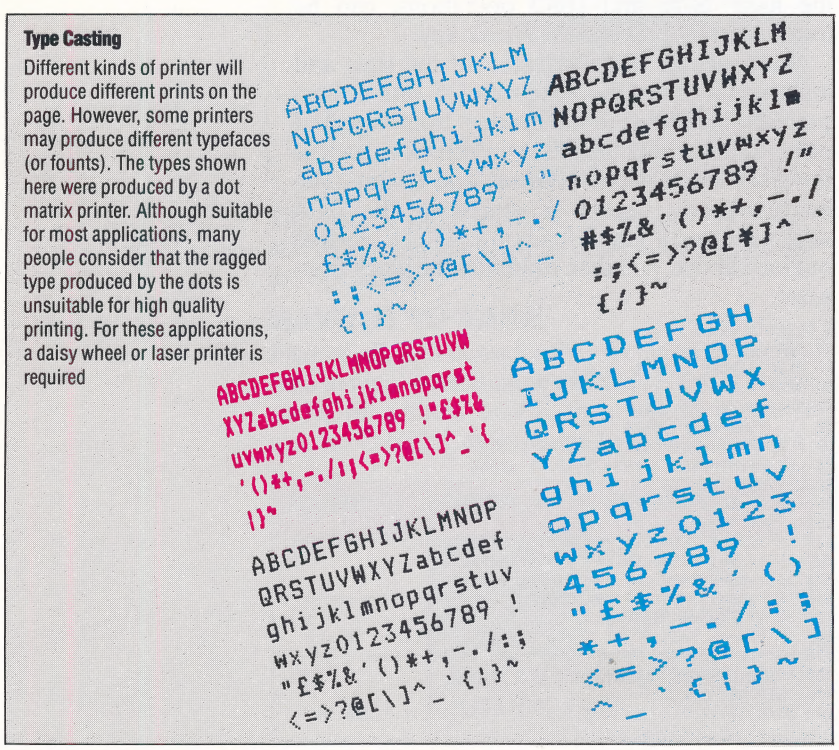
This is an output device that converts data from a computer or storage device such as a disk drive into typewritten letters or characters on a piece of paper. Typically, a *printer* will consist of an interface to a computer, a method of decoding the data (which usually arrives in the form of ASCII code), a buffer area (which allows the storage of pieces of data so that the computer can perform another function while printing progresses), a print head, a device to hold the paper and a means of moving the print head in both horizontal directions over the paper.

Several types of printer designed for use with microcomputers are available. These range from inexpensive thermal printers that can be bought for less than £100 (see page 1261) to laser printers, which can cost well over four-figure sums. Other types of printer are line printers, which print rather like an ordinary typewriter (one line at a time), and page printers, which can print a whole page at once. Printers can also be subdivided according to how they transfer the print onto the page. Impact printers produce the print by stamping the image of the character onto the paper. Examples of this type are dot-matrix (see page 304), in which the character is made up of a number of pins in a matrix, and daisy wheel printers (see page 364), which have a solid fount character like that on a typewriter.

Non-impact printers (see page 71) include laser printers (see page 948), which burn the characters onto the paper, and ink-jet printers (see page 829), in which electrically charged ink is shot out of a nozzle and directed onto the page by magnetic fields. However, these latter types are expensive and beyond the budget of most home computer owners.

Type Casting

Different kinds of printer will produce different prints on the page. However, some printers may produce different typefaces (or founts). The types shown here were produced by a dot matrix printer. Although suitable for most applications, many people consider that the ragged type produced by the dots is unsuitable for high quality printing. For these applications, a daisy wheel or laser printer is required



PROBABILITY

The study of *probability* is concerned with attempting to determine how likely an event is to happen. This is measured as a value between zero and one, where zero represents no likelihood of it occurring and one represents certainty.

Of course, we cannot predict which events will happen in the future with absolute certainty, but the probability can be assessed by observing and recording past events. A measure of probability can be determined by the number of times that an event occurs divided by the number of times it could happen. However, although we observe that the sun comes up in the morning, and we can say that there is a very high probability of it doing so tomorrow morning, we cannot assign a probability value of one to the event as we cannot be absolutely certain.

There are some events that have never happened, of course, and yet we can assign a probability value to them. In such cases we must apply other statistical methods. When throwing a dice, for example, the chances of throwing a six is one in six. To find out the chances of throwing three consecutive sixes we could throw the dice indefinitely to obtain an observed probability, but it is easier to calculate arithmetically that the chances are $(1/6)\times(1/6)\times(1/6)=0.0046$. When we perform such calculations we assume that none of the dice is 'loaded'.

The study of probability and predicting the likelihood of events is an application ideally suited to the computer. Because the probability of an event is largely determined by the analysis of large quantities of statistical data, the number-crunching capabilities of computer technology make the computer an invaluable tool in this area.

PROCEDURE

A *procedure* is an area of a program that has been assigned a unique name. The name, and hence the procedure, can be called without the command having to specify a line number. Procedures usually carry out a single task that will be contained within start and end statements. The pieces of information needed for the procedure to complete its task are known as the parameters.

It is possible to define local variables within a procedure. These are variable names unique to the procedure and have no meaning outside it, unless they are specifically passed as parameters.

Procedures are an integral feature of structured programming. Structured programs can be broken down into self-contained parts and organised as procedures that can be called from within a control loop. The use of procedures also renders the task of making further additions to the program much easier as an extra procedure can simply be tagged onto the end and called when it is needed. Furthermore, the ability of procedures to make use of local variables means that the user need not worry that any new variables that might be assigned within the procedure will conflict with any that may already exist.

HAVE YOU ORDERED YOUR VOLUME SIX BINDER YET?

FOLD 2

IF YOU HAVE NOT ASKED FOR THE BINDERS TO BE SENT TO YOU AUTOMATICALLY, DO SO NOW, AND ENSURE THAT YOUR COPIES OF THE HOME COMPUTER ADVANCED COURSE ARE KEPT PROPERLY BOUND AND IN GOOD CONDITION FOR YEARS TO COME.

BY TICKING THE BOX OPPOSITE, YOU WILL BE SENT BINDER NUMBER 6.

☐ PLEASE SEND ME MY VOLUME 6 BINDER NOW. I ENCLOSE A CHEQUE/POSTAL ORDER FOR £3.95 (WHICH INCLUDES POSTAGE AND PACKING).

FOLD 3

BY TICKING THE OTHER BOX *AS WELL*, YOU WILL BE SENT SUBSEQUENT BINDERS FOR YOUR COLLECTION.

☐ I WOULD ALSO LIKE TO RECEIVE FUTURE BINDERS AS THEY ARE ISSUED. I UNDERSTAND THAT I WILL RECEIVE A PAYMENT ADVICE FOR £3.95 (WHICH INCLUDES POSTAGE AND PACKING) WITH EACH BINDER. IF I AM NOT SATISFIED WITH THE BINDER, I CAN RETURN IT TO YOU, WITHIN 14 DAYS, AND OWE NOTHING.

NO STAMP NECESSARY.

JUST FOLD UP THE PAGE AS INDICATED, REMEMBERING TO ENCLOSE YOUR CHEQUE/POSTAL ORDER MADE PAYABLE TO ORBIS PUBLISHING, AND SEND TO US TODAY.

FOLD 4

FOLD 4

I enclose a cheque/postal order made payable to: Orbis Publishing Ltd. for a total of £____ which I understand includes the cost of postage and packing.

NB: Please allow 28 days for the delivery of your binders.

When you have completed the order form fill in your name and address in the space provided.

Then cut along the dotted line to detach the page, enclose your cheque/postal order, and fold the page carefully – following the instructions to complete the reply paid envelope.

NO STAMP NECESSARY.

FOLD 1

IF YOU ALREADY HAVE AN ACCOUNT NO. FOR YOUR BINDERS PLEASE FILL IT IN HERE.

Complete the section below with one letter or figure per space.

		INITIALS	SURNAME
MR			
MRS			
MISS			

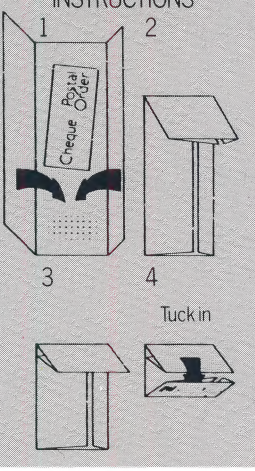
ADDRESS & POST CODE

TELEPHONE NO. INCLUDING STD CODE OR EXCHANGE NAME

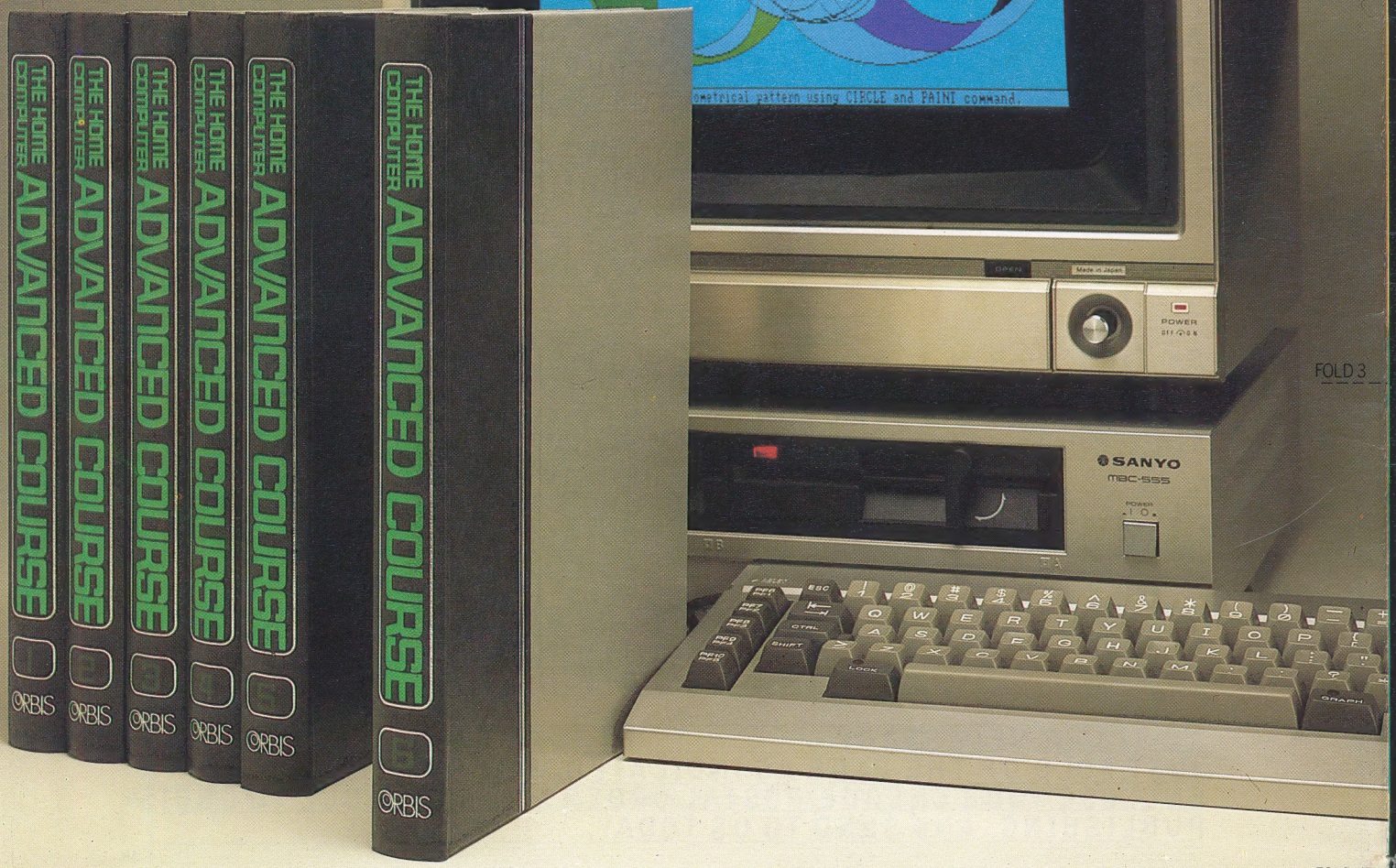
FOLD 1

FOLD 4

FOLDING INSTRUCTIONS



ARE YOU LOOKING AFTER YOUR COPIES.

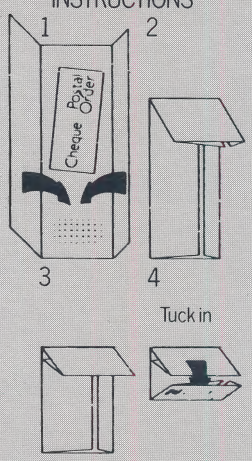


FOLD 2

FOLD 3

FOLD 4

FOLD 4
FOLDING INSTRUCTIONS



FOLD 1

Postage will be paid by licensee

Do not affix Postage Stamps if posted in Gt. Britain, Channel Islands or N. Ireland.

BUSINESS REPLY SERVICE
Licence No. SW4035.

The Home Computer
Advanced Course Binders
Orbis House, 20-22 Bedfordbury
London WC2N 4BR

FOLD 1

GUARANTEE

If you are not entirely satisfied with your binder, send it back immediately and it will be either exchanged, or, if you prefer, your money will be refunded in full.

GUARANTEE