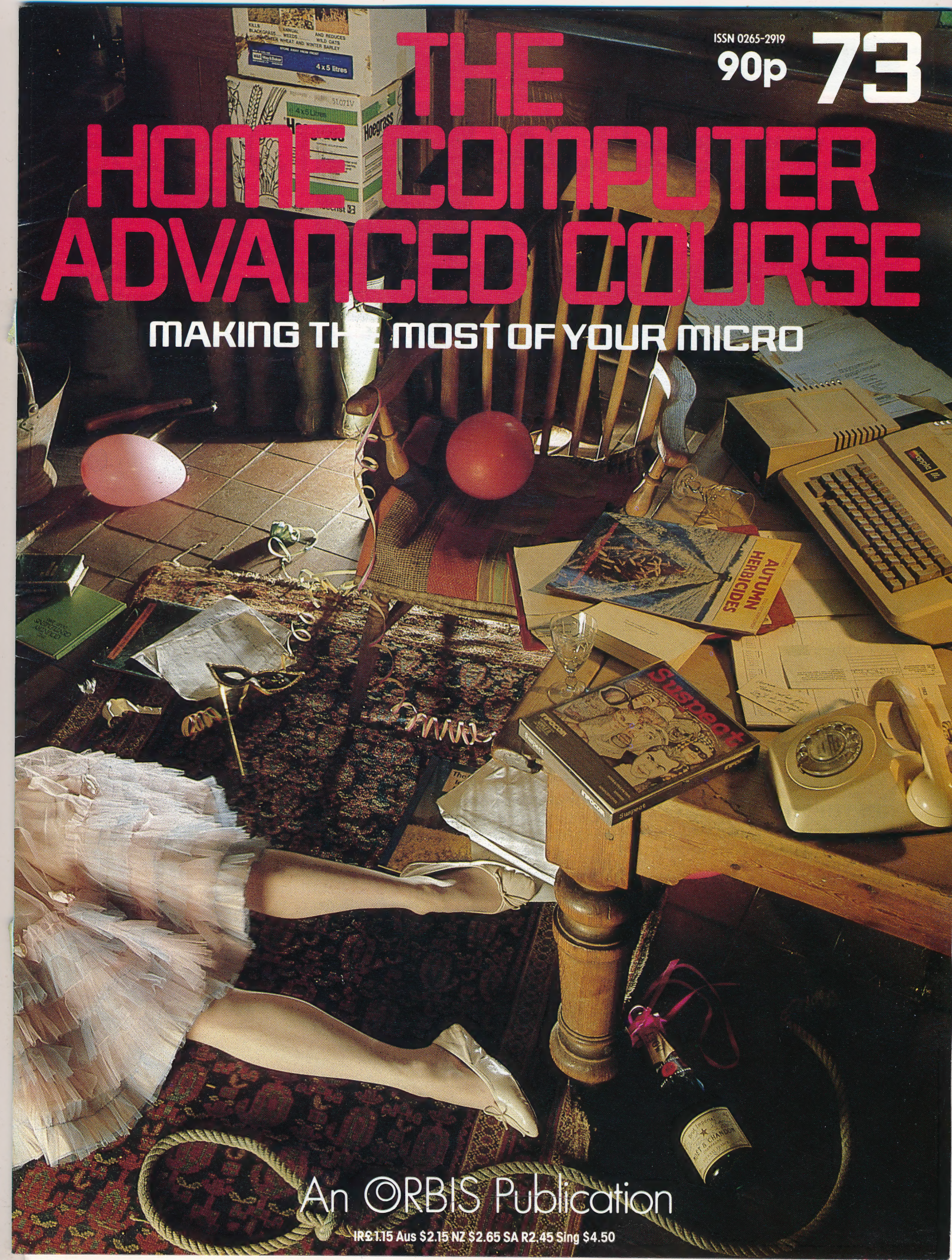
ISSN 0265-2919

90p

73

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

LANGUAGE MATTERS We look into the types of programming language most suited to artificial intelligence applications 1444

HARDWARE

PLUS ÇA CHANGE On the surface nothing's changed, so what's new about the BBC+? 1449

SOFTWARE

SUSPECT CHARACTERS The increasing sophistication of adventure programming is evident in the use of 'interactive' characters. We look at how this is achieved 1441

COMPUTER SCIENCE

WHAT'S IN A WORD? An awful lot if you're a FORTH programmer. We look at the centrality of words, symbols and a dictionary 1446

JARGON

FROM READ/WRITE HEAD TO RECOVERABLE ERROR A weekly glossary of computing terms 1460

PROGRAMMING PROJECTS

A STONE'S THROW The module that allows a stone to be placed on our Go board 1454

MACHINE CODE

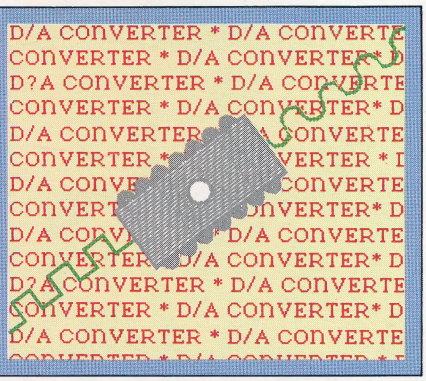
GET THE MESSAGE A look at the networking capabilities available to the Spectrum machine code programmer 1458

WORKSHOP

METER MANIFESTO We discuss the overall structure of our digital multimeter 1452

Next Week

- **We take a detailed look at the D/A converter chip at the heart of our Workshop multimeter.**
- **Our Go programming project tackles the task of 'teaching' the computer to choose a reasonable move for itself.**
- **In our artificial intelligence series we discuss the differences between data and knowledge.**



QUIZ

1) How does the BBC+ display the shadow screen while running a BASIC program that uses the same memory addresses?
2) What is the effect of placing the characters @ and ! after a word in FORTH?
3) What does a 'scout' do?
4) Explain the difference between a 'logical record' and a 'physical record'?

Answers To Last Week's Quiz

1) The clock track on the rule provides a code that tells the Omni-reader software which direction it is scanning.
2) When in MIDI free-run mode, the timer is being decremented until it reaches zero. Then an interrupt is generated and the register is reloaded to its initial value.
3) 50 FORWARD
4) When producing a long series of random numbers on a computer, the major problem is that after a while the numbers either fall to zero or begin to repeat themselves.

Coming Up

- A programming project to design a spreadsheet in BASIC for the Spectrum, Commodore 64, BBC Micro and Amstrad CPC 464/664.
- A look into GEM, Digital Research's 'operating environment' created for 16-bit machines such as the IBM PC.

REFERENCE CARD The final section of the Commodore 64's memory map INSIDE BACK COVER

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editors** Nik Lumsden, Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Steve Cooke, Joe Pritchard, Steve Malone, Steven Vickers, David Mudd; **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: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. 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 sterling. 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 BY MARCUS WILSON-SMITH

SUSPECT CHARACTERS

One of the major reasons for the popularity of adventure games has to do with the interactiveness between the player and the game's characters. We begin here a short series on the programming of such characters and illustrate the possibilities with a quick look at Infocom's Suspect.

Software that appears to give 'character' to a computer by analysing a user's natural language inputs and producing apparently meaningful responses never fails to generate interest and excitement. Whatever psychological issues may be involved, programs like ELIZA create a degree of user involvement out of all proportion to their relative simplicity and, from the programmer's point of view, there are few challenges more satisfying than that of programming a computer to 'listen' and 'reply' like a fellow human being.

We have already developed a simple ELIZA-type program, Private Consultation (see page 412) and discussed the principles of natural language processing in our series on artificial intelligence (see page 1361). In this series, we examine another application of these principles — the programming of 'interactive characters' in adventure games. We shall look at the role such characters can play in these games and then program our own character handler, which may be run as a stand-alone module or adapted to run in conjunction with our Haunted Forest and Digitaya programs (begun on page 766).

Adventure software has become increasingly sophisticated since Crowther and Woods developed Colossal Cave in FORTRAN on a mainframe computer using approximately 170 Kbytes of code. Nowadays, programs like The Hitch Hiker's Guide to the Galaxy (see page 1400) can not only accept complex inputs but can also boast characters that move around 'of their own accord' and who may be addressed (and even in some instances, commanded) by the player.

The introduction of characters to 'interactive fiction' (which adventure software is now being called) has enabled programmers to break away from the traditional adventure scenario, which was almost exclusively centred around the process of exploring different locations, collecting objects and solving the occasional puzzle. Interactive characters introduce new possibilities — first, of presenting situations that in fact cannot be solved by the player without the help of another computer-controlled character; second, and most importantly, of presenting a game that may be radically different each time it is played



The Finger Points At You
A murdered fairy queen, a werewolf, a vampire and many other characters are all waiting for you in Infocom's Suspect, a state-of-the-art adventure game featuring interactive characters and vivid location descriptions. Invited to a high society fancy-dress ball, you stumble across the scene depicted above and, to your dismay, discover that your hostess has been strangled with a lariat taken from your cowboy costume. Clearing your name involves extensive and complex interaction with your fellow guests

THE INTERACTIVE CHARACTER

Before we go any further, we need to define exactly what an interactive character is. An interactive character must have at least one of the following attributes:

- It must be able to move from location to location.
- It must be able to collect and drop objects without necessarily being instructed to do so by the player.
- It must be addressable by the player and respond meaningfully (even if it only replies 'I don't understand you').
- It must be able to address the player without being prompted to do so by the player.
- It must appear to be aware of, and responsive to, its environment.

Ideally, a character should combine all these attributes.

Perhaps the best example of early interactive character programming occurs in The Hobbit, a text and graphics adventure by Melbourne House. This game featured a number of characters, including, most notably, Thorin the Dwarf and Gandalf the Magician. The player could address

these characters using the SAY TO command, as in SAY TO THORIN 'GO NORTH'. The program then employed a random number-driven routine that would decide whether or not the character should obey. The characters in The Hobbit, however, were (by today's standards) comparatively primitive — issuing a command to them would frequently result in: THORIN SAYS NO or a similar response. Furthermore, Gandalf and his companions were not particularly well-spoken, if indeed they spoke at all. Thorin, for example, seems mostly limited to saying HURRY UP if the player waits too long between inputs, and has a habit of 'sitting down and singing about gold' with monotonous frequency.

Perhaps most significantly, the characters in The Hobbit have no personal history — that is, if you perform a particular action that might affect your relationship with them, the characters will not subsequently 'remember' it. Typical of most early adventures, each character is programmed to behave in a certain way throughout the game and, to a large extent, regardless of circumstances. Thus Thorin, for example, might sit down and start singing about gold in the path of an oncoming dragon.

An early attempt to provide characters with a 'history' was made in Valhalla, a program that received the British Microcomputer Game of the Year award in 1984, both for incorporating this feature and for its animated display. In this game, which takes place amongst the gods and heroes of Norse mythology, you would find that the 'good' characters were willing to obey your commands

House Of Horror
Suspect takes place at the exclusive Ashcroft residence, shown here. Some of the larger rooms appear in the game as several different locations — for example, Ballroom Near Band, Ballroom Near Fireplace and Long Hall North. The map shows the position of some of the principal characters shortly after the murder of Veronica Ashcroft in the office

Key:
1. Harlequin, watching television
2. Fairy Queen (dead)
3. Gorilla (the butler, Smythe)
4. Sheik, husband of the Fairy Queen
5. Harem Girl, dancing
6. Astronaut, moving west
7. Bartender
8. Vampire, leaving the Ball Room
9. The Player, dressed as a cowboy



(and vice versa) if you went around killing all the evil characters.

This presented, among other things, the interesting possibility of 'brainwashing' characters. For example, the Norse god Loki is initially an evil entity, but you can endear yourself to him by picking fights with, say, Thor (a virtuous god) and then order Loki to carry out good deeds, such as picking fights with his erstwhile evil companions. Unfortunately, despite the complexity of the 'history' aspect of the characters in Valhalla, the other attributes of the personalities involved are limited, for the most part to eating, fighting and drinking.

Let's now look at a far more advanced interactive adventure — Suspect, from Infocom (who also produced the Hitch Hiker's Guide to the Galaxy). Suspect is a perfect example of a game in which the characters are all-important; in fact, success or failure in the game depends entirely on observing, questioning, examining and eventually accusing other computer-controlled 'people'. As a result, and because of the random elements involved, the game can be played in a number of different ways.



SUSPECT

Suspect is set in a large country house (see map on this page) and features at least 12 major characters. The game covers the period from nine o'clock in the evening through to the early hours of the morning when the Ashcrofts — high-society American millionaires — are holding their annual fancy dress ball. During the course of the game, you discover (if you bother to look around the house) that Veronica Ashcroft has been strangled with the lariat that was included with the cowboy costume you are wearing. Suspicion, therefore, rests firmly on your shoulders, and unless you find the real culprit, you will soon end up in jail.

Throughout the game, the characters wander about, talking amongst themselves and leading their own lives. The program keeps you updated as to their immediate whereabouts. Our map shows the positions of some of the characters at a particular point in the game when, standing in the ballroom near the band, you receive the following on-screen description:

Ballroom, Near Band
This is the northern end of the ballroom. A raised area provides a band platform, and there is a stereo hook-up for use when there is no band. Elsewhere in this large ballroom, there are scores of other party guests dressed in all sorts of outlandish costumes. On the dance floor are some of the older dancers. The band is playing the 'Tennessee Waltz'.

On the periphery of the room small groups can be seen, discussing everything from politics to local scandals. Michael is near the North entrance. Alicia is on the dance floor. Ostmann is near the south entrance. The Astronaut is near the fireplace. Johnson is at the bar. The band is here, playing their set. The Astronaut is now near the south entrance.

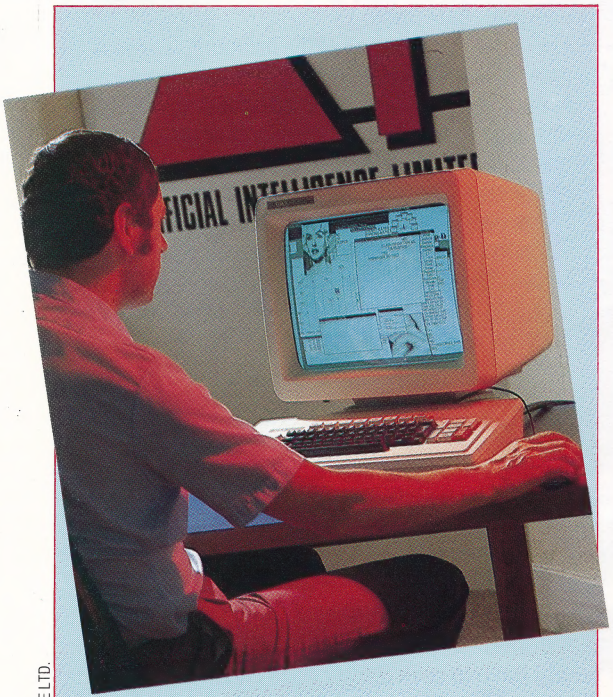
You will notice that the Astronaut is mentioned twice in the description — at first he is near the fireplace and then he is near the south entrance, so you can deduce that he is probably on his way out of the ballroom. The locations of other characters, such as Smythe the butler, are, of course, not mentioned — they are in other rooms and you would have to seek them out to be notified of their presence.

Playing a game like Suspect is an absorbing experience, and the presence of interactive characters adds many new dimensions to an area of games programming that is already extremely popular. In the next instalment, we will examine further examples of character interaction in the game, and prepare the way for writing our own routines to achieve similar effects.

LANGUAGE MATTERS

In the field of artificial intelligence, a lot depends on choosing the most suitable language for a particular application. In this instalment, we look at the different kinds of programming techniques that have proved valuable in AI research, and explain some of the features that make them so useful.

Most AI researchers use LISP or PROLOG. For this reason, these two programming languages have become known as *AI languages*. There are good reasons why AI workers choose languages with certain features, but the phrase 'AI language' can be misleading for two reasons. First, it suggests that LISP and PROLOG may be unsuited to conventional data processing. Secondly, and more importantly, it leads to the assumption that merely by writing a program in LISP or PROLOG, you are thereby involving yourself in AI. It also suggests that other symbol-processing languages, such as POP-11 and SNOBOL4, to name but two, are unsuitable for serious AI work.



COURTESY OF ARTIFICIAL INTELLIGENCE LTD.

AI Environment
Rank Xerox's Interlisp-D network forms a complete programming environment for AI researchers including interactive graphics, debugging tools and large on-line storage facilities in the form of a 29 Megabyte disk unit. The LOOPS knowledge programming system, when used in conjuction with Interlisp-D, allows the AI programmer to select a combination of different programming styles, such as object-oriented or rule-oriented coding, to build low cost knowledge systems

LISP is a 'veteran' among programming languages. It was initially developed by John McCarthy at MIT in the late 1950s and is therefore as old as COBOL (see page 288). The language stabilised in 1961 and its present form is very much the same. We have already presented a short introductory series on LISP (starting on page 1376), but one concept that we have not previously mentioned, which is of considerable importance for AI applications, is the 'property list'.

Each atom has its own property list, which consists of 'attribute-value' pairings. The property list is in effect a description of the atom, and provides easy access to a kind of database structure. The functions GET and PUTPROP are used to manipulate property lists. For instance:

```
(PUTPROP 'YESTERDAY 0.1 'RAINFALL)
(PUTPROP 'TOMORROW 'OVERCAST 'OUTLOOK)
```

have the effect of inserting 0.1 as the value of the RAINFALL attribute of the atom YESTERDAY and putting OVERCAST as the value of the OUTLOOK attribute for the atom TOMORROW. The attributes RAINFALL and OUTLOOK are like fields in a database record (see page 1097). The value of any given attribute can be retrieved by using GET. For example:

```
(SETQ RAIN (GET 'YESTERDAY 'RAINFALL))
```

would obtain the RAINFALL value from YESTERDAY's property list and assign it as the new value for the atom RAIN. (In fact, the value of an atom is just a special kind of property that is available without having to use GET and PUTPROP.)

Property lists can be used to build up complex and flexible data structures, and have been used to model the workings of human memory.

AI PROGRAMMING IN PROLOG

PROLOG is the main alternative to LISP for AI programming. In Europe it is becoming more popular than LISP, though LISP still prevails in the US. (For more information on PROLOG, refer to our series on the language beginning on page 1272.) Because PROLOG is a declarative language, it allows the programmer to specify facts and rules about objects and relationships. These facts and rules can then be used to answer questions about the objects and relationships involved.

Briefly, PROLOG answers questions by using a theorem-proving method based on the Resolution Principle, which negates the proposition to be proved and tries to refute it — a sophisticated version of the *reductio ad absurdum* technique. But as far as the user is concerned, it is just

searching a database.

PROLOG, like LISP, has facilities for building complicated data structures (including lists) and for input/output, calculation, file-handling and so on. In many ways, it fails to live up to the ideals of logic programming, but it is still a powerful and very flexible language.

However, LISP and PROLOG are not the only software tools suitable for AI programming. POP-11 (especially within the POPLOG environment developed at Sussex University) is also a candidate for AI work, and it is quite possible to use traditional languages, such as C and PASCAL — or even BASIC — for AI programming. It does, however, require more effort.

Because AI problems are complex, anything that makes AI programming easier is welcome. The right type of language helps, of course, but there are many other factors involved. One modern trend that will doubtless continue is the use of AI environments running on AI workstations.

Examples of such environments are LOOPS and POPLOG. LOOPS (LISP Object-Oriented Programming System) was developed at the Xerox Palo Alto Research Center, and is based on a dialect of LISP called INTERLISP-D; but it is much more than a LISP system. It provides a multiplicity of software tools — some conventional (such as window and icon management), and some less conventional (such as pre-packaged inference procedures for use in expert systems). The entire system runs on a special-purpose AI workstation that contains a processor optimised for running LISP. POPLOG runs on the VAX range of computers, so it does not need specialised hardware; but in software terms it is a similar idea to LOOPS. It provides screen-editing and other useful standard packages, as well as the ability to call either LISP or PROLOG from within POP-11 (its primary language), thus offering the 'best of both worlds'. Additionally, there is a library of pattern-matching and other procedures tailored specifically for AI applications.

What such systems offer, above all, is a software environment highly conducive to programmer productivity. Their main disadvantage is isolation: you can develop clever applications very quickly, but you cannot run them on the average personal computer.

It is very rare that an AI problem is completely understood when programming begins. Indeed, AI researchers often write programs as a means of gaining deeper insights into their problems. Therefore, the emphasis in AI programming must be on 'rapid prototyping' — a style of programming in which systems evolve gradually by means of many small additions and alterations. Traditional software tools and software engineering methodologies are inadequate to meet the demands of AI programming, which, as we now know, is hard work.

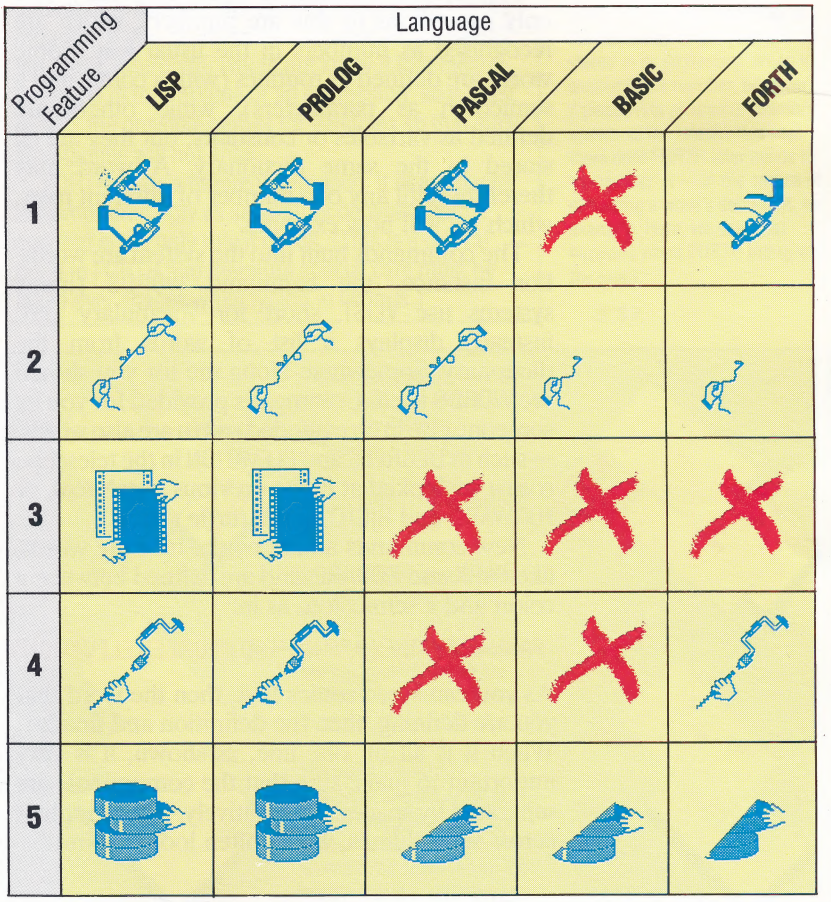
Fit For The Job

Over the years, a number of tools have been developed to make the AI programmer's task easier — which thus enable AI programmers to attempt more ambitious projects. Such tools share a number of important features

1 Recursion must be supported by the language.
2 The language should have good string-handling and symbol-manipulation facilities, allowing flexible data structures of arbitrary complexity to be built up.
3 It is important that there should be uniformity of program and data: both LISP and PROLOG represent programs and data in the same format.
4 The syntax should be extendable: as with LISP and PROLOG, it should be possible to build a new language on top of the original one.
5 Access to a built-in database of some sort is crucial: PROLOG has what amounts to a relational database at its heart, but LISP (with property lists) also provides a similar facility.

In addition, all the other workbench tools that make progamming easier (editors, graphical routines, syntax checkers, debuggers, compilers, documentation producers and so on) are needed in AI work at least as much as in other areas of computing.

Our chart shows the distribution of these features among four different high-level programming languages



ICONS BY IAN McKINNELL ON THE MACINTOSH

WHAT'S IN A WORD?

In the FORTH programming language, programs are built up from words (groups of symbols separated by spaces), which can be defined by the programmer. We look here at the language's dictionary, as well as examine how FORTH defines commands, constants and variables, and introduce the concept of the stack.

We would like all commands to be used the same way, whether they are built into the language or defined later as part of a program. So FORTH keeps all the commands in a dictionary. Like an ordinary dictionary this is a list of words and their definitions, although they're not stored in alphabetical order but, rather, in the order in which they were defined. Each definition tells the computer exactly what to do when that word is used — either because you've typed it in directly (interactively) or because it is part of the definition of another word that is being used.

In FORTH, everything that the system recognises is a word, with a definition in the dictionary. The only exceptions to this are numbers, which are recognised as numbers in the usual way. Some words are defined as routines (with a colon, and a semicolon as parameters), while others are defined as variables or constants; but they are all stored in the same dictionary. A word may therefore fulfil any of a number of different roles, which we will now examine.

The commands built into the system are words. For instance, the command WORDS (many systems use VLIST, short for Vocabulary LIST, instead) displays a list of words from the dictionary. Somewhere along the list you should see WORDS itself. Commands provided for you by someone else in an extended FORTH are also words. — such as DEGREES and ELEVATION in the telescope FORTH we looked at in the previous instalment, or FORWARD and RIGHT in the turtle FORTH.

New commands added by the user are words, like PARK and SQUARE, and are defined between a colon and a semicolon, as in:

```
: PARK 0 DEGREES AZIMUTH 90 DEGREES ELEVATION;
```

As you can see, : comes first, then the word that you are defining, then the definition and finally ;. When it is all on one line, as shown, it is very important to make sure that the components are separated by spaces. Alternatively, you can split it across several lines, which often looks clearer:

```
:PARK
  0 DEGREES AZIMUTH
  90 DEGREES ELEVATION
;
```

The entire program is a word. In fact, FORTH doesn't really think of anything as being *the* program. If your definitions build up to a single big command — you could call it RUN if you liked — then you might think of that as being the program, using the other words as subroutines; but there's no reason why you shouldn't have other independent commands/programs in the dictionary at the same time with different names. Subroutines are also words. They are just commands that happen to be used in another definition.

FORTH doesn't distinguish between commands, programs and subroutines. They are all defined using : and ;, and they can all be used either directly from the keyboard or indirectly from another definition. Because these definitions use a colon, they are called *colon definitions.*

Variables too are words. You declare a variable, LENGTH say, by typing:

```
VARIABLE LENGTH
```

The word LENGTH then has a definition in the dictionary that includes some memory space for its value. You have to declare it like this before you can use it. If you don't, there is simply no definition for it in the dictionary and it won't be recognised as a variable. You use variables with the characters @(meaning 'fetch') and ! ('store'):

```
LENGTH @
```

gives you the value of LENGTH, and:

```
26 LENGTH !
```

sets its value to 26 (like LET LENGTH = 26 in BASIC).

The @ is easy to forget, but you must use it to get the value of the variable. Without it you will get the *address* of the variable, rather than any value that might have been assigned to it.

Constants can also be words, naturally enough. If you use a number repeatedly with a certain meaning, you may prefer to give it a meaningful

Numbers As Words

In the definition:

```
0 CONSTANT 0
```

0 was previously a number that could be recognised under rule 2. Afterwards, it will have a definition in the dictionary and so will be recognised by rule 1 before rule 2 gets a look at it. Since it is defined as the number 0, there is no obvious difference, but it could possibly be more efficient this way in your FORTH — in fact, figFORTH defines 0, 1 and 2 in the dictionary because they are so often used.

A more malicious definition might be:

```
12 CONSTANT 13
```

so that when you type 13, you actually get 12. (You could still get the proper 13, however, by typing 013.)

name by defining a word as a constant. For example:

```
66 CONSTANT CLICKETYCLICK
```

After this the word CLICKETYCLICK simply means 66. You could, of course, have used a variable instead, but the advantage of a constant is that you don't need @. Of course, you can't use the store command (!) with it either.

Note that you can't use VARIABLE and CONSTANT declarations inside a colon definition — you'll see why when we look at how the definitions are stored in the dictionary. It's tempting, however, to try to define words like:

```
:INITIALISE
   VARIABLE SCORE    0 SCORE !
   VARIABLE WICKETS    0 WICKETS !
;
```

but it simply won't work. Another consequence of this is that all variables are *globals* (see page 593), as in BASIC. Local variables, such as you find in PASCAL, do not exist in FORTH.

FORTH includes +, −, *, / and many other arithmetic operators. Like all words, they must be separated from other words by spaces, and they expect to find their arguments already there. Thus:

```
2 3 +
```

is written instead of 2+3. This will be explained more thoroughly, with the workings of the 'stack', in the next instalment.

The colon, VARIABLE and CONSTANT are unusual, because they make new definitions in the dictionary; for this reason they are called *defining words*. But apart from that, they are still words in the dictionary, just like any others. It is actually possible in FORTH to define new defining words of your own.

Program structure symbols include DO and LOOP, which you have already seen (similar to FOR . . . NEXT in BASIC), ; (the end of a colon definition) and other program structures like IF . . . THEN and BEGIN . . . UNTIL, which are used in branching and looping. Again it is possible, though not always easy, to define you own novel program structure words.

Since we now know what goes in the dictionary, we can see how FORTH deals with your input. It treats each group of symbols without spaces as a potential word and then proceeds according to three main rules.

1. If it finds the word in the dictionary, it performs whatever the definition dictates. If it is defined more than once, the newest definition is used.
2. If it's not in the dictionary, FORTH checks to see if the symbol (or group of symbols) is a number and, if so, remembers it temporarily. (It puts it in a piece of memory called the 'stack'.)
3. If the group of symbols is found to be neither a number nor present in the dictionary, then it is unrecognisable, and FORTH lets you know about it.

These rules have to be modified slightly within

Words And Spaces

Quite simply, a word is any sequence of characters. Thus the following are all words — though not necessarily defined in the dictionary — in FORTH:

```
SAUSAGE
De-emulsifiedRubberTube
239
pH
25p
+
"merguez"
,!XXX)so-thereQWERTY
```

FORTH makes no attempt whatsoever to give special meanings to special characters, except the space. You would probably choose words like LENGTH and SCORE for your variables, but FORTH wouldn't mind at all if you called them 23PI/2 or FORI=1TO10.

You may wonder then why other languages are so much more restrictive. The name of a variable, for instance, in many languages must be a letter followed by any letters or digits. The answer really lies in their attitude towards spaces. Most languages treat spaces as a kind of decoration, which you can insert to make the program easier to read. But in those languages you can omit them if you like, and you probably will when writing expressions — in BASIC you wouldn't normally put spaces in the expression 2+3.

The languages must therefore be able to differentiate between the various components of the program. If you could name a variable 2+3 in BASIC, the BASIC interpreter would not know if that meant the name of the variable or an expression; so it must restrict the possible names for variables so that such ambiguities will not arise.

FORTH takes a different approach. It uses spaces uncompromisingly as separators between words, and that is the only rule it needs. (This is a critical point in FORTH and therefore special care must be taken over spaces when typing in listings.)

FORTH Flavours

In figFORTH, a number must precede VARIABLE, so that:

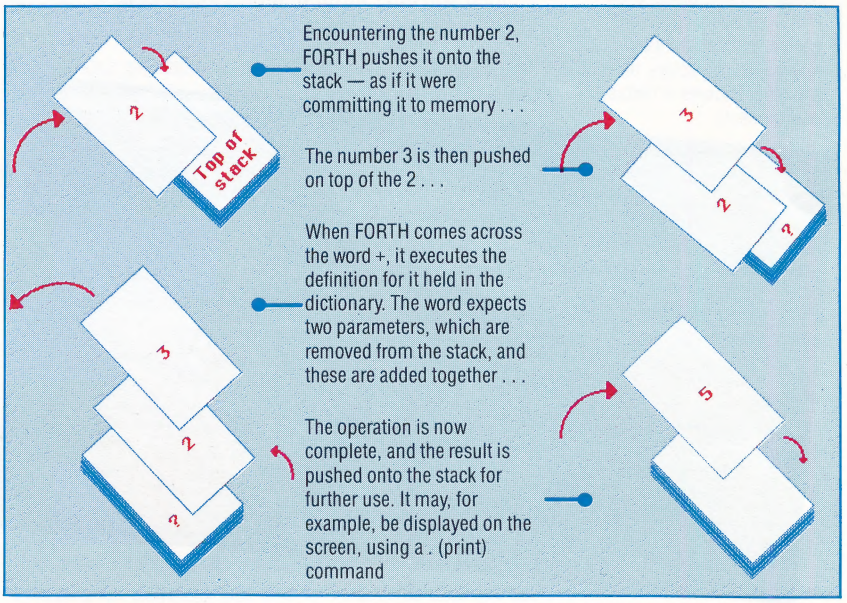
```
10 VARIABLE
   GREENBOTTLES
```

is the value that the variable has to begin with

It All Adds Up

When FORTH encounters a word, it proceeds according to the definition for that word stored in its dictionary. When it encounters a number, however, it places it in an area of memory known as the 'stack'. It is kept there until FORTH later needs to 'remember' it — in the course of executing a word's definition, for example. Here, we see what happens when FORTH interprets the input:

2 3 +



colon definitions because the appropriate actions are not performed straight away, but are remembered in the definition.

Obvious exceptions are words (like :, VARIABLE and CONSTANT) and these cause some difficulty. When you write:

```
VARIABLE SCORE
```

it would seem to generate an error under rule 3. But in fact VARIABLE deals with SCORE itself and the three rules never see it. Otherwise, we could never enter words into the dictionary.

We can now see how the dictionary gives FORTH its interactiveness and extendability. Interactiveness is ensured by the three rules, and extendability is due to the fact that all new definitions are put in the dictionary on equal terms with the old ones.

Back To Square One

Experience of machine code programming can make FORTH more immediately comprehensible, but for those who lack such experience or are finding the initial programming examples somewhat confusing, we present the following annotated example.

To define a word SQUARE as a subroutine that draws a square on the screen with sides of length SIDE, we would proceed as follows. First, we must declare the variable SIDE as:

```
VARIABLE SIDE
```

This instructs FORTH to reserve a space in memory where a numeric value can be stored. We can move values to and from this address using @ (fetch) and! (store). So, for example:

```
20 SIDE !
```

will assign the value 20 to the variable SIDE.

We can now define our new word SQUARE using the 'words' : and ; (just as we would use TO and END in LOGO) as:

```
: SQUARE
   SIDE !
   4 0 DO
      SIDE @ FORWARD
      90 RIGHT
   LOOP
;
```

This procedure expects two things. First, that the words RIGHT and FORWARD have already been defined and offer turtle-graphics facilities as in LOGO, so that, for instance, the instruction:

```
50 FORWARD
```

would move the turtle forward 50 screen units.

Secondly, the user is expected to enter a value for SIDE, so that entering:

```
50 SQUARE
```

would draw a square with sides of 50 units.

Let's now work through the subroutine and see exactly how it would work if we had in fact entered the command 50 SQUARE. FORTH examines your input and finds the first group of symbols are separated by spaces — in this case 50. FORTH then checks to see if this group has been entered in its dictionary. Assuming that you haven't previously defined the character symbol group 50 as a word, FORTH will fail to find it in the dictionary and will then check to see if it is a number, which of course it is. FORTH therefore takes the number, puts it in a memory store and checks the next group of symbols, which is SQUARE.

SQUARE is, of course, in the dictionary (we have just defined it), so FORTH calls up its definition and proceeds to evaluate that. The first terms it comes across are SIDE !. At this point, you may well think that something is wrong, since SIDE ! doesn't appear to have a value to assign to SIDE, but FORTH checks its memory store and finds the value that we entered (50) and uses that, thereby assigning 50 to the variable SIDE.

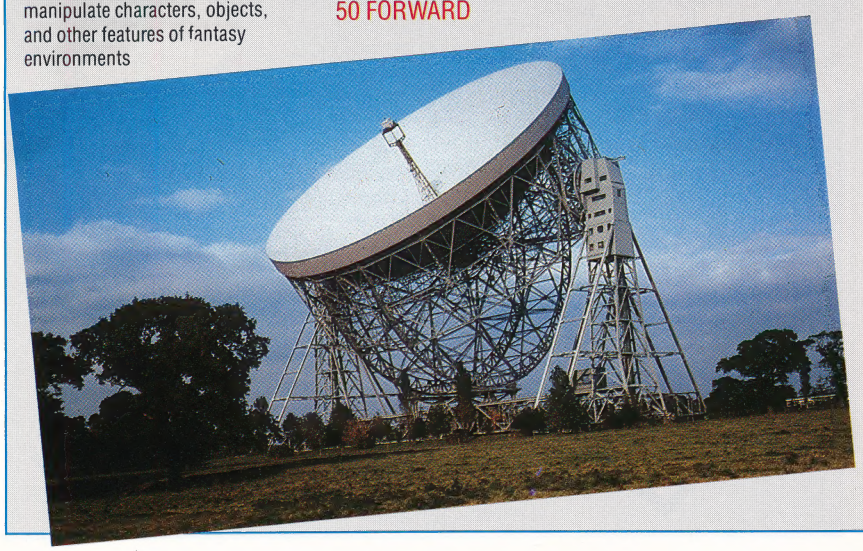
Note that FORTH, unlike LOGO, doesn't check up on this — it will simply assume that you have entered a number in the appropriate place, so if you simply called up the subroutine SQUARE with SQUARE instead of 50 SQUARE, you would get some unpredictable results rather than an error message.

The 4 0 DO . . . LOOP is fairly self-explanatory, but it's worth noting how within that loop the value of the variable SIDE is passed to the subroutine FORWARD by the words SIDE @ FORWARD. These first call up the address of SIDE and then (using @) fetch a value from that address, ready to be used by FORWARD. (Compare this with the method used by the SQUARE routine (see page 1437) which used DUPlicate to present a value for FORWARD.) The terms 90 RIGHT are, of course, necessary to turn the turtle through 90 degrees after drawing each side, thereby completing the square.

FORTH may appear somewhat confusing at first, particularly if you are not already an experienced programmer. It is, however, far more straightforward in its structure and behaviour than might seem apparent, and if you persevere, you will soon come to see why its supporters are so loyal

The Power Of The Word

Defining a word SQUARE illustrates just one aspect of the potential offered by the extendability of FORTH. Not only can one construct 'dictionaries' of words to control mechanical devices, like the telescope shown here, but one can also devise control systems for more abstract applications. For example, one British software house (Mastertronics) uses FORTH to create adventure games, defining words to manipulate characters, objects, and other features of fantasy environments



PLUS ÇA CHANGE

BBC+

PRICE
£499 inc VAT

MEMORY
76 Kbytes RAM, of which 64 Kbytes are available for BASIC programs. 32 Kbytes ROM, expandable to 192 Kbytes

CPU
6512 processor clocking at 2MHz

DISKS
The BBC+ comes fitted with an Acorn DFS system as standard, although the disk drives themselves are extra

DOCUMENTATION
The BBC Micro User Guide, already very thorough, has been updated to include additional information on the shadow RAM and other useful information

STRENGTHS
The extra hardware provision within the BBC+ counters many of the criticisms levelled at the original BBC Micro. There should now be sufficient user memory available to allow programmers to both support a high-resolution screen and write sophisticated programs in BASIC

WEAKNESSES
Despite being an improved machine, the BBC+ is still very expensive for most home micro users. The educational market now appears to be opting for MS-DOS machines, which further limits the possible markets for the BBC+

Since the BBC Micro was launched in 1981, developments in the industry have left the Acorn product lagging behind the front runners. The updated version, the BBC+, remedies the major weakness — lack of user RAM — but its price of £500 leaves it without a definite market to enter.

When the BBC Microcomputer contract was awarded in 1981 to Acorn, the machine was hailed by most of the industry as a tremendous advance in home computers. Its speed and versatility meant that it out-performed every other home micro on the market. Since then, however, the market has changed considerably. Although the range of interfaces and peripherals for the BBC Micro has remained unrivalled, the computer has revealed a few weaknesses with the passing of time.

The primary weakness, which has dogged the computer from its inception, has been the relative shortage of available memory. Some of the higher resolution graphics modes leave very little user RAM for program development. This is not because the BBC Micro is lacking in user RAM — it has the same 64 Kbytes as many other home micros — but because the advanced graphics features of the computer fit uncomfortably within the confines of the memory addressing capabilities of an eight-bit processor.

This problem became even more threatening to the long-term future of the machine by 1984. While other manufacturers took advantage of the world-wide drop in the cost of microprocessors, (and memory chips in particular) to cut their prices, Acorn maintained the price of the BBC Model B Micro at its original £399. Acorn was able to do this partly through the prestige value of the BBC contract and partly because of the government subsidy given to educational establishments if they purchased the computer through the Microcomputer Educational Programme (MEP). Furthermore, while the other manufacturers, such as Sinclair, were re-engineering their printed circuit boards to take advantage of the higher capacity, lower-priced chips that were becoming available, Acorn failed to do so. This meant that the BBC Micro's manufacturing cost became considerably higher than any of its rivals.

Finally, the BBC Micro's eight-bit technology was in danger of becoming outdated. Sinclair Research, a close rival of Acorn's for the BBC contract, made no secret of the fact that its 16-bit computer, the QL, was designed to compete at exactly the same price as the BBC Micro.

Since then, the danger of 16-bit processors making their eight-bit counterparts obsolete has receded, albeit temporarily. The slump in computer sales and the financial crises at both Acorn and Sinclair has made the industry much more cautious. The result has been that rather than introducing new machines with all the attendant risks, nearly all the major home micro manufacturers have chosen to simply upgrade their existing products. To be fair, this is a response to what the public wants. The market is growing much more sophisticated and the customer appears not to be as interested in the latest hardware innovations as much as in the quality and quantity of the software base. Hence the development of more powerful computers that are compatible with the software bases that previous machines have developed.

THE SOFTWARE BASE

The BBC+ is a response by Acorn to the demand from customers requiring large existing software bases, but also increased memory to run larger and more sophisticated programs. The computer is fitted with 32 Kbytes more of RAM than its predecessor. Because it is based around an eight-bit processor, the maximum amount that can be directly addressed is obviously limited to 64 Kbytes. Therefore, like Atari and Commodore, Acorn has utilised the technique of 'bank switching', which enables it to accommodate the additional areas of memory.

The bank switching technique allows the processor to 'look' at either one area of memory or another. Two banks of memory are arranged so that they occupy the same memory addresses. As far as the processor is concerned, there is only a single memory location, but, depending on which area of memory is 'banked in', the address can contain very different contents.

The additional 32 Kbytes of RAM are set at addresses 12288 (3000 hexadecimal) and 45055 (AFFF hexadecimal). This takes in most of the BASIC program area and the area of memory occupied by the paged ROMs (the paged ROMs are the BASIC ROM and those which users may have installed themselves, like View or LOGO).

The shadow RAM itself is divided between these two sections and we will now look at them separately. As we have already seen, the major problem with memory space on the BBC Micro is the large amount of RAM that is required to support a high resolution graphics screen. Thus the vast bulk of the additional RAM has been given over entirely for use as video RAM. In fact,

the entire 20 Kbytes 'shadowing' the BASIC program area is set aside for this purpose — which is enough memory to support the most detailed screen. In redesigning the printed circuit board in order to accommodate the extra RAM chips, Acorn has taken the opportunity to make some other changes. One of the most noticeable is that the new machine runs on a 6512 processor instead of a 6502. This is to standardise some of the information exchange between the CPU and the 6522 peripheral chip with which Acorn has had some trouble in the past. The bank switching address itself (FE34 hexadecimal) is held on one of the redesigned operating system ROMs (IC36). Because the BBC Micro runs to a large extent on interrupts, the addition of this address has not required any fundamental change in the system.

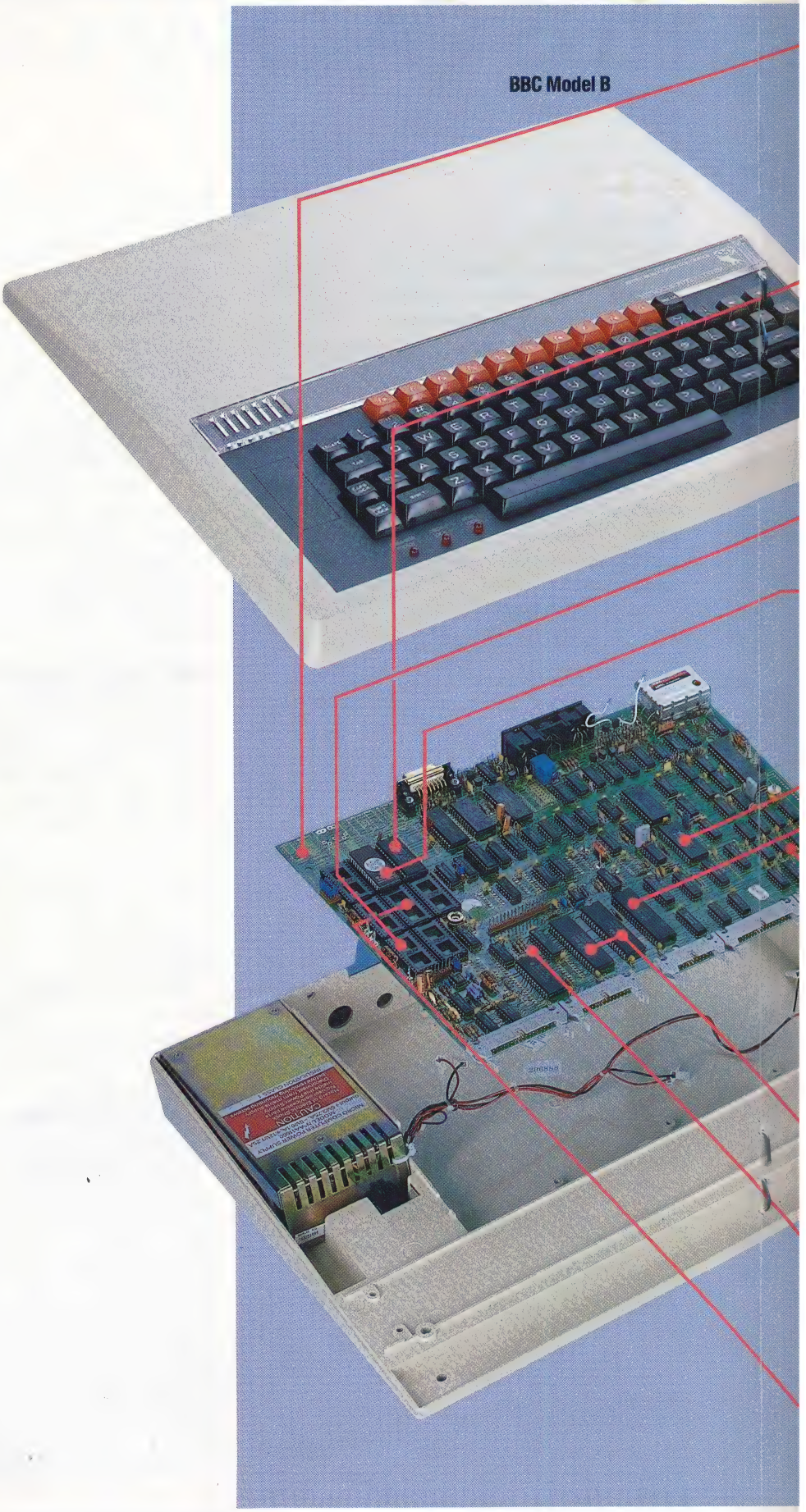
The remaining 12 Kbytes of memory that has been provided for the BBC+ is in the memory address area between 32768 (8000 hexadecimal) and 45055 (AFFF hexadecimal). This is the area reserved for use by the paged ROMs. Once again, the bank switching enables time to be shared by the ROMs and the shadow RAM underneath.

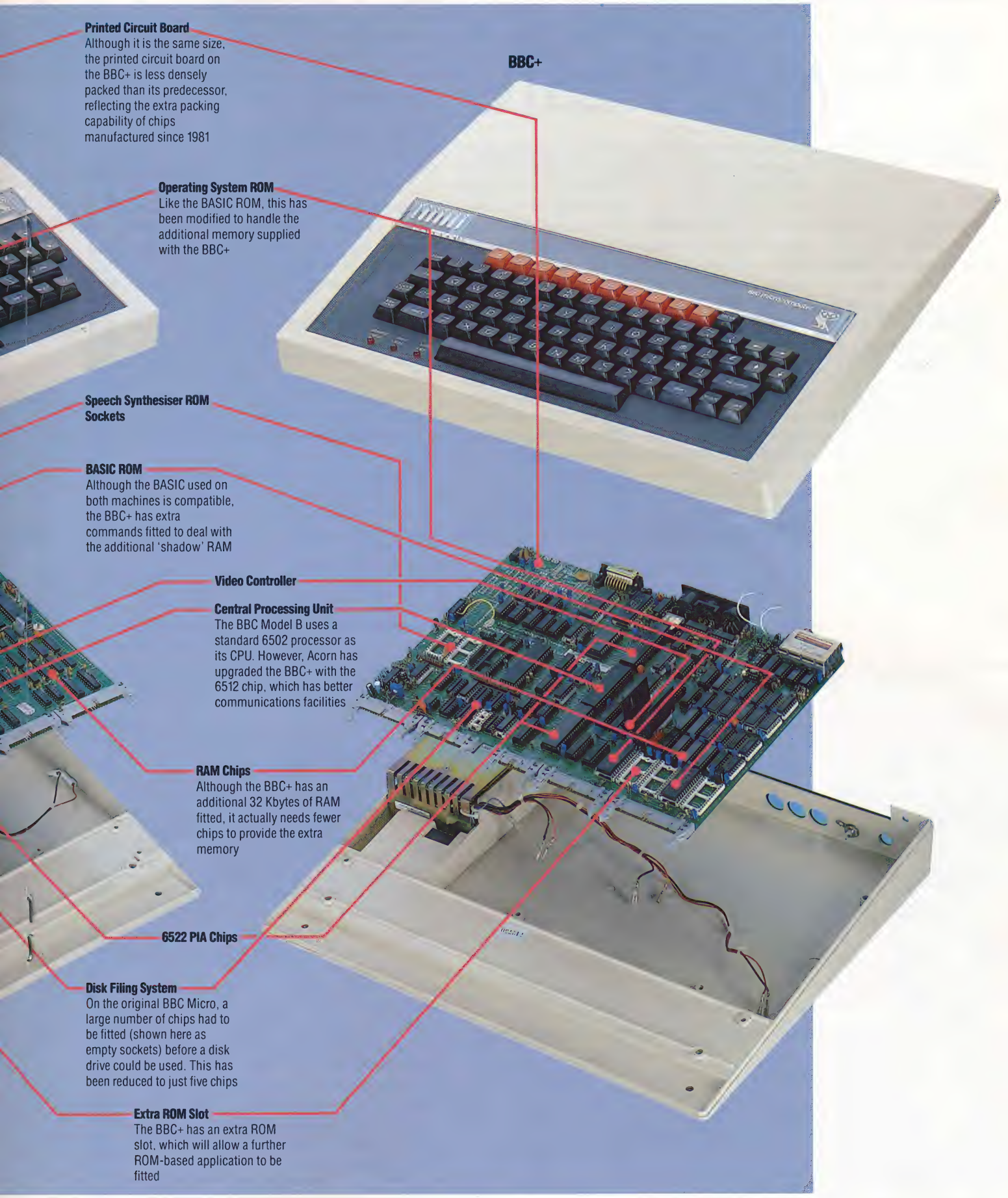
REDESIGNED BASIC AND OS

BBC+ owners have an advantage over their Atari and Commodore counterparts, who have to use POKE commands in order to access the extra banks of memory, because Acorn has also redesigned its BASIC and machine operating system (MOS) ROMs to provide software support for the bank switching facilities.

The shadow screen memory can be accessed in several ways. Using the MODE command, followed by a number between 128 and 135, will produce a default shadow screen corresponding to the eight screen modes between 0 and 7. However, subsequently MODE commands not in the 128 to 135 range will default to the normal RAM area, which is obviously a fatal error. The *SHADOW command will permanently fix the screen into shadow RAM, preventing you from overwriting any BASIC programs. In order to break out of this, the command *SHADOW 1 has to be executed. Naturally, like all BBC * commands, there are corresponding *FX instructions; in this case, *FX114 and *FX114,1, respectively. Similarly, the MODE commands can be emulated by VDU 22, (128 + n), in which n stands for the required mode.

In upgrading the BBC Micro, Acorn has also chosen to fit a disk filing system (DFS) as standard equipment. This is useful to those who wish to buy a computer and intend to use disk drives with it from the start — the alternative is to have a DFS fitted for an extra £100 or so. But for most users, who want only to use a cassette drive with the machine, this extra provision will merely increase the entry price for the BBC+. This seems a pity since the extra cost will also do nothing to silence the critics who maintain that the BBC range of micros is overpriced, and that Acorn should cut its prices in line with other manufacturers.





Printed Circuit Board
Although it is the same size, the printed circuit board on the BBC+ is less densely packed than its predecessor, reflecting the extra packing capability of chips manufactured since 1981

Operating System ROM
Like the BASIC ROM, this has been modified to handle the additional memory supplied with the BBC+

Speech Synthesiser ROM Sockets

BASIC ROM
Although the BASIC used on both machines is compatible, the BBC+ has extra commands fitted to deal with the additional 'shadow' RAM

Video Controller

Central Processing Unit
The BBC Model B uses a standard 6502 processor as its CPU. However, Acorn has upgraded the BBC+ with the 6512 chip, which has better communications facilities

RAM Chips
Although the BBC+ has an additional 32 Kbytes of RAM fitted, it actually needs fewer chips to provide the extra memory

6522 PIA Chips

Disk Filing System
On the original BBC Micro, a large number of chips had to be fitted (shown here as empty sockets) before a disk drive could be used. This has been reduced to just five chips

Extra ROM Slot
The BBC+ has an extra ROM slot, which will allow a further ROM-based application to be fitted

METER MANIFESTO

We begin here a new series in Workshop that will outline the requirements for an easy-to-build digital multimeter. In this introductory instalment, we will consider some of the problems of analogue-to-digital conversion and develop a strategy for a suitable design.

Computers usually work completely digitally. Even the 'human' interfaces of keyboard input and video output rely exclusively on digital circuitry. The usefulness of computers can be extended considerably if they can communicate with the 'outside world'. But the outside world tends to be an analogue world of infinitely variable voltages, temperatures, heights, weights and so on. Consequently, when computers need to collect data from, or control devices in, the outside world, conversion between an analogue and a digital environment is necessary.

A computer attached to a thermo-sensitive device, for example, will need to work with binary numbers that correspond to the measured temperatures in order to decide what it should do. This will involve what is called analogue-to-digital, or A/D, conversion (see page 223). Later, we will look at some of the techniques used to convert analogue (continuously variable) measurements into digital (binary) representations.

The converse of this situation occurs when a digital device, such as a computer, is used to alter the value of something in the real world. As an example, consider a computer that has been programmed to produce music. The computer creates a binary digital value, but this must be converted into an acoustical frequency. In this kind of situation, conversion from digital to analogue signals is required (see page 714).

DIGITAL METERS

A multimeter is an instrument for measuring conductivity (measured in ohms), current (measured in amps) and potential (in volts) in an electrical system — see page 86. In this series, we will be building a digital multimeter, which will measure these to a high degree of accuracy and display the results digitally using an LCD or LED display. Developing such a device would be a major undertaking if it were not for the advent of specialised large-scale integrated circuits that combine many analogue and digital stages onto a single chip. As we develop our DVM (digital voltmeter), we want to design a circuit that can both work as a stand-alone digital multimeter, and also interface directly with a computer to enable voltage and other values to be read and processed by it.

A conventional voltmeter uses a pointer that swings against a scale to give a direct reading in volts, amps or ohms. Combinations of shunt and series resistors enable such meters to give readings in terms of voltage, current or resistance, but fundamentally they all work the same way. Ultimately, a current flows through a coil suspended within a magnetic field, and the strength of the current determines how far the coil will turn against the force of a spring. A needle connected to the coil shows how far the coil has turned, and the scale is calibrated in terms of the units being measured

When an unknown voltage is to be measured and displayed digitally, the problems are much greater. Before going into how this is done, it is worthwhile to consider the inverse problem of converting digital values into analogue values.

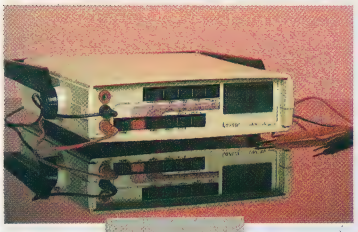
Digital-to-analogue (D/A) conversion is a fairly straightforward operation. Suppose you want to convert an eight-bit word (a unit consisting of eight binary digits) into an analogue voltage of between 0v and 1v. An eight-bit word can represent any value between 0 and 255 (binary 00000000 and 11111111). That one-volt range can therefore be resolved into voltage steps of 256 stages, or 0.0039 volts.

Binary values can be converted into analogue ones by using the binary signals to switch on increasingly higher currents. The diagrams show both the simplest type of D/A converter, and the more common R2R type. In both cases, the mechanical switches represent electronic switches actuated by binary signals, with binary 1 meaning 'closed' and binary 0 meaning 'open'. Each positive binary digit throws an electronic switch that simply allows more current to flow.

Converting analogue signals (voltages, for example) to their digital equivalents is far more

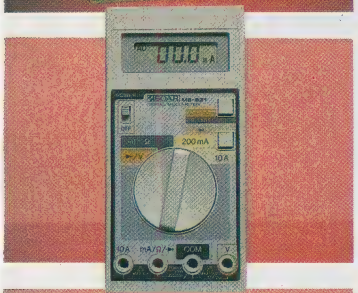
Meter Market

We compare the features offered by three typical digital meters that are currently available:



Lascar LMM 100 Bench Meter

Display:	3½-digit LCD
Range:	25 ranges
Accuracy:	0.1% (DC volts)
Power:	Battery
Other Features:	Hold, polarity, battery and overrange indication
Price:	£104.95



Soar ME-531 Autoranging Hand-held Meter

Display:	3½-digit LCD
Range:	Auto-ranging
Accuracy:	0.8% (DC volts)
Power:	Battery
Other Features:	Polarity, overload indication, diode check
Price:	£63.95



Maplin Precision Gold M-5010 Basic Hand-held Meter

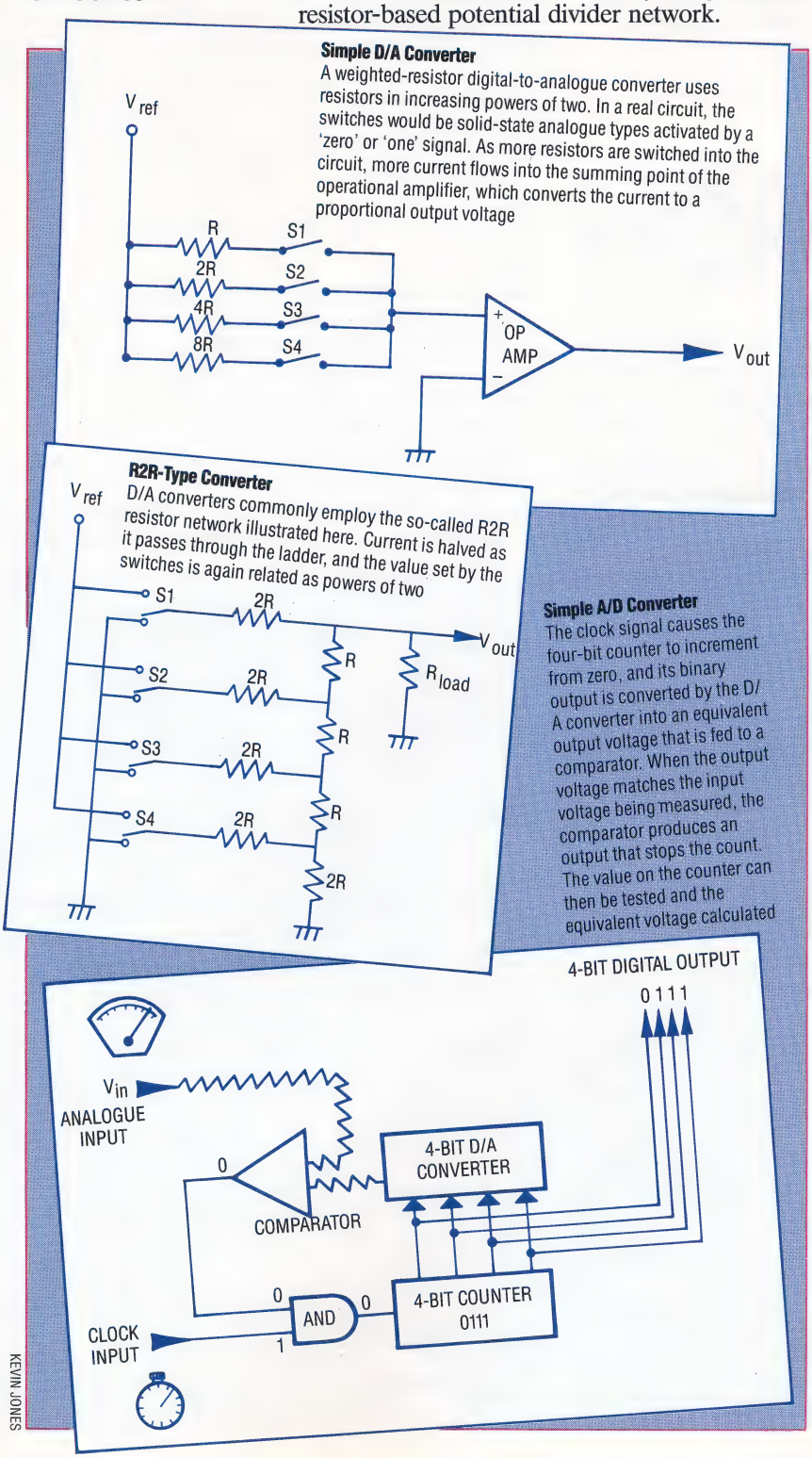
Display:	3½-digit LCD
Range:	29 ranges
Accuracy:	0.25% (DC volts)
Power:	Battery
Other Features:	Audible continuity indication, overrange and polarity indication, diode check
Price:	£42.50

MARCUS WILSON-SMITH

difficult, and there are many techniques available for doing it. As with D/A conversion, the number of bits used determines the resolution of the measurement; it has nothing to do with the range of voltages that can be measured. For higher resolution, more bits will be necessary, but for the purposes of our DVM, eight bits will be more than adequate.

Our design will have a basic range of 0v to 2v, so with eight-bit resolution, the measurable steps will be 0.0078 volts. Measuring ranges of 0v to 20v and 0v to 200v will be achieved by using a simple resistor-based potential divider network.

Conversion Circuits



KEVIN JONES

Analogue-to-digital conversion is a good deal more complex and expensive than D/A conversion, and depends partly on how many bits of resolution are required and how fast the conversions need to be made. If audio signals with frequency ranges of up to 15KHz are to be converted to digital signals, at least 30,000 samples of the signal will need to be converted every second. If video signals are to be digitised, the sampling rate will need to be very much higher.

Fortunately, a DVM needs only to measure constant (DC) voltages or alternating (AC) voltages of low frequency, so low conversion speed will be alright; even so, eight bits of resolution will be required for reasonable accuracy.

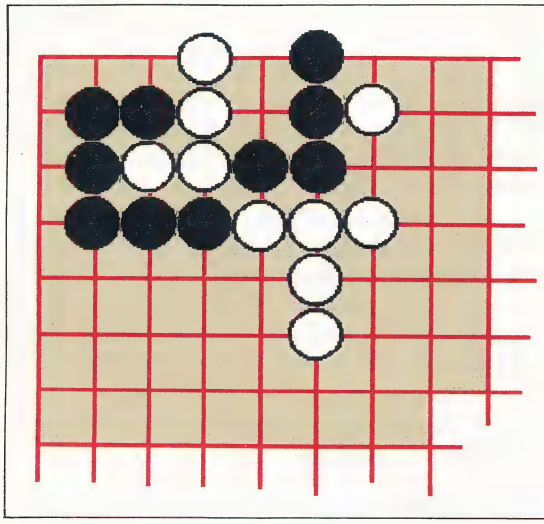
A/D CONVERSION TECHNIQUES

The method generally used is to compare a known voltage with the unknown voltage (the one being measured). Integrated comparator chips (a kind of operational amplifier, see page 1058) are ideal for this, as they can be configured to give an output when (and only when) the two input voltages are the same. A binary counter is initially set to zero and counts up, giving a voltage output (using either of the D/A techniques referred to earlier) that will eventually equal the unknown input voltage being measured. When the unknown voltage and the voltage determined by the D/A converter are equal, the comparator produces an output that stops the binary counter and allows the voltage to be calculated. A simplified circuit for this type of A/D converter is shown.

Although there are many single-chip ICs that can directly drive LCD or LED displays specifically designed for DVM applications, they are not normally easy to interface with computers, since the outputs are only suitable for driving digital displays. We have therefore opted to base our design on the Analog Systems ICL7135 digital panel meter chip. This chip gives binary coded decimal outputs that can be easily converted into a form suitable to drive seven-segment displays or be interfaced with a computer.

The chip features very high input impedance, a differential input that allows both positive and negative voltages to be measured. It also allows a true zero reading for 0v input, under-range, over-range and polarity indication, logic I/O for interfacing with a microprocessor and simple clock and power requirements. Although it is an expensive chip, costing about £16, it is a lot cheaper than the equivalent circuit built out of discrete components, is easier to use, and stands a far better chance of working right first time.

Only two things are needed to make a highly accurate meter: close tolerance resistors for the input attenuator and the use of a friend's accurate DVM for setting the reference voltage. Even without these, highly satisfactory results can still be obtained. In our next instalment, we will describe in a little more detail the actual circuit we will be using.



SEMEAI

Looking Bad

A common method of capturing stones near the edge of the board is to smother them — a tactic known as SEMEAI. The number of liberties remaining to each group involved in a semeai battle is important. For example, here things look bad for the L-shaped white group, as it has only three liberties while the corresponding black group has four

A STONE'S THROW

As we continue our Go programming project, we present the module that will allow you to place stones on the board. This includes several checks by the computer to ensure that the move is a legal one, appropriate messages, and an update of the board display.

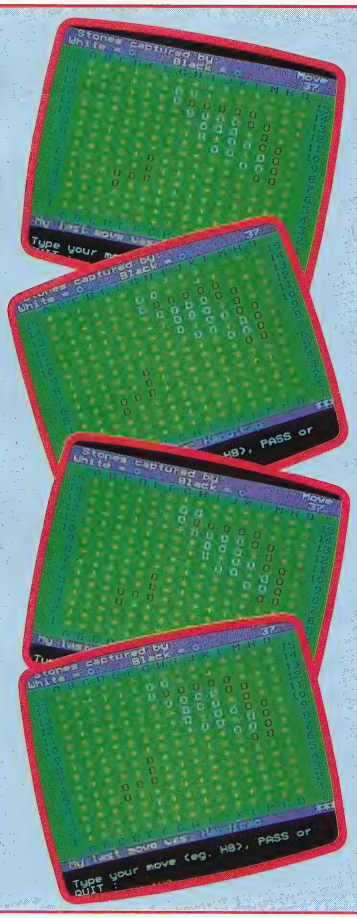
Due to the structured and generalised way in which the program has been developed, it is extremely easy to alter the main program that calls the routines (lines 10 to 130) to allow you and a friend to play each other. We'll see how to do this later, so that you will then be able to use the program for a two-player game, with the computer acting as referee. As before, the program description relates to the BBC Micro version; the line numbers and program structure are maintained in the other three.

The first routine that we'll consider is FNlegality (lines 3890 to 4000). This function will accept a move, P%, by colour C% and return a value incidating whether or not the move is legal. This check ensures that:

1. The intersection (P%) is vacant.
2. The player does not capture a stone in Ko (see page 1367).
3. The played stone is not committing Suicide (see page 1367).

If you look back to the routine that reads messages (lines 390 to 560 in the BBC Micro version, page 1393), you'll notice that these three conditions coincide with messages two, three and four (three, four and five in the Spectrum version), respectively. Therefore, FNlegality has been designed to return either one of these numbers, or zero if the move is legal.

Seeing if the position is vacant is handled by checking for a zero in the appropriate board byte,



Module Three

BBC Micro:

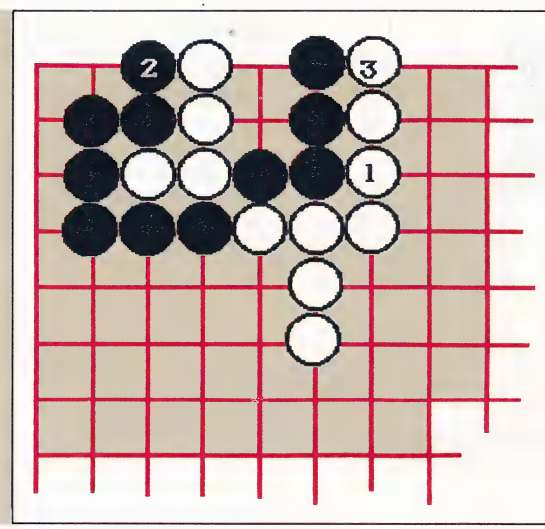
```
 1380 ko%=FALSE:end%=FALSE
 2310 :
 2320 DEF PROCwhite_move
 2330 LOCAL C%,L%,P%,X%,Y%,A$,C$,Y$
 2340 atari2$="      "
 2350 A$=FNinput(21,8,4)
 2360 IF A$="PASS" THEN ko%=FALSE:PROCpr
int_board:GOTO 2490
 2370 IF A$="QUIT" THEN end%=TRUE:ENDPRO
C
 2380 X%=ASC(LEFT$(A$,1))-64
 2390 Y$=MID$(A$,2)
 2400 FOR C%=1 TO LEN(Y$)
 2410   C$=MID$(Y$,C%,1)
 2420   IF C$<"0" OR C$>"9" THEN PROCmes
sage(23,1,A$):GOTO2350
 2430   NEXT
 2440 Y%=VAL(Y$)
 2450 IF X%<1 OR X%>15 OR Y%<1 OR Y%>15
THEN PROCmessage(23,1,A$):GOTO 2350
 2460 P%=16*Y%+X% : L%=FNlegality(P%,whi
te%)
 2470 IF L%>0 THEN PROCmessage(23,L%,A$)
:GOTO 2350
 2480 PROCmake_move(P%,white%)
 2490 PROCmessage(23,0,"")
 2500 ENDPROC
 2510 :
 2520 REM*****************************
 3620 :
 3630 DEF PROCmake_move(P%,C%)
 3640 LOCAL A%,L%,N%
 3650 board%?P%=C%
 3660 FOR L%=1 TO 4 : A%=P%+dir%(L%) : I
F board%?A%<>colour%-C% THEN 3700
 3670   PROCcount(A%,colour%-C%):IF clib
%=0 THEN ko%=A%:N%=N%+cstn%:PROCremove(A
%,colour%-C%)
 3680   IF clib%=1 AND C%=black% THEN at
ari1$="Atari":atari2$="      "
 3690   IF clib%=1 AND C%=white% THEN at
ari2$="Atari"
 3700   NEXT
 3710 IF N%<>1 THEN ko%=FALSE
 3720 capture%(C%)=capture%(C%)+N%
 3730 PROCprint_board
 3740 ENDPROC
 3750 :
 3760 REM*****************************
 3770 :
 3780 DEF PROCremove(P%,C%)
 3790 IF board%?P%<>C% THEN ENDPROC
 3800 board%?P%=0
 3810 PROCremove(P%+dir%(1),C%)
 3820 PROCremove(P%+dir%(2),C%)
 3830 PROCremove(P%+dir%(3),C%)
 3840 PROCremove(P%+dir%(4),C%)
 3850 ENDPROC
 3860 :
 3870 REM*****************************
 3880 :
 3890 DEF FNlegality(P%,C%)
 3900 LOCAL A%,K%,L%,S%
 3910 IF board%?P%<>0 THEN =2
 3920 board%?P%=C%
 3930 FOR L%=1 TO 4
 3940   A%=P%+dir%(L%)
 3950   IF board%?A%=colour%-C% THEN PRO
Ccount(A%,colour%-C%):IF clib%=0 THEN K%
=K%+cstn%
 3960   NEXT
 3970 IF K%=0 THEN PROCcount(P%,C%):IF c
lib%=0 THEN S%=4
 3980 board%?P%=0
 3990 IF P%=ko% AND K%=1 THEN =3
 4000 =S%
 4010 :
 4020 REM*****************************
```

at line 3910. To check the other two possibilities, the stone to be played is temporarily placed on the board. The loop, L%, then counts the liberties of the surrounding enemy groups (using PROCcount, developed in the last module — see page 1426). Within this loop, K% keeps track of the number of stones that would be killed by the move P%. The final checks can now be made:

- If no stones are killed (K%=0) and the stone to be placed has no liberties (clib%=0), then the stone must be committing Suicide.
- If one stone is killed (K%=1) and the stone to be played is in the Ko position (P%=ko%) then the rule of Ko is being violated.

The Ko check may be a little hard to follow. When a single stone is captured by a player, ko% is set equal to the position of the captured stone by the make_move routine in lines 3670 and 3710. If the new stone to be played is positioned on this point, and this stone is capturing only one opposing stone, then it must be on the Ko point and violating the Ko rule. The stone it is capturing must be the last stone played by the opponent.

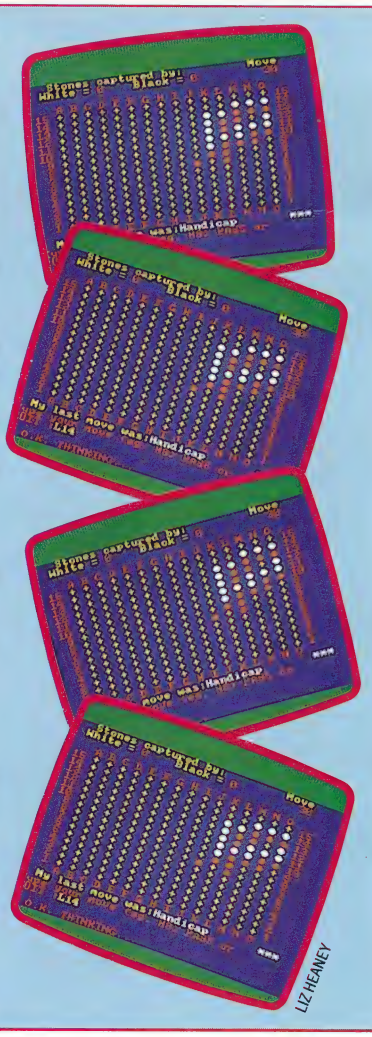
The next routine is one that will update the board for any particular move. PROCmake_move begins by placing a stone of the appropriate colour on the board and a loop then checks the status of each of the groups surrounding it. If the new stone fills the last remaining liberty of an opposing group, then that group will be removed from the board by calling PROCremove. Alternatively, if it has a single liberty, then it is in danger of being captured on the next move, so the appropriate atari string is updated. You'll remember that atari is a customary warning, given when you are about to capture one, or more, of your opponent's stones. Finally, the number of stones captured by the

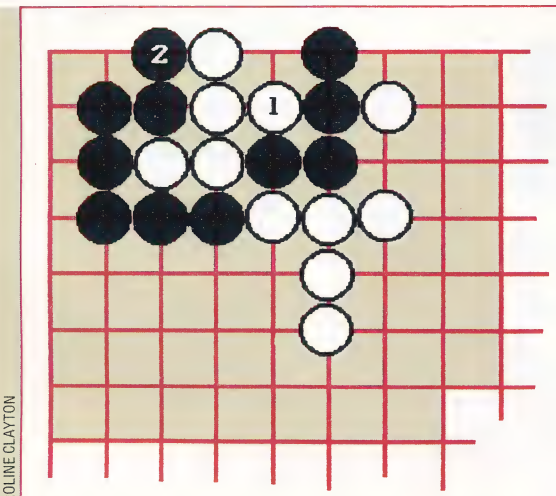


Going For Seki
Semeai battles nearly always end either in the capture of the weaker group or in the stalemate situation called seki (see page 1406). If White attacks the black group from outside, then seki can be achieved. In the situation shown here, White has secured its position as it shares its two remaining liberties with the black group. Neither side can play into this area without laying their group open to capture

Amstrad CPC 464/664:

```
60 MOVE%=MOVE%+1:GOSUB 2320
1380 KO%=0:FIN%=0
2310 :
2320 REM WHITE-MOVE ROUTINE
2340 A2$="      "
2350 IP%=22:IM%=8:IW%=4:GOSUB 1990
2360 IF A$="PASS" THEN KO%=0:GOSUB 1730:
GOTO 2490
2370 IF A$="QUIT" THEN FIN%=-1:RETURN
2380 X%=ASC(LEFT$(A$,1))-64
2390 Y$=MID$(A$,2)
2400 FOR WC=1 TO LEN(Y$)
2410 C$=MID$(Y$,WC,1)
2420 IF C$<"0" OR C$>"9" THEN MP%=24:MM%
=1:O$=A$:GOSUB 2160:GOTO 2350
2430 NEXT
2440 Y%=VAL(Y$)
2450 IF X%>0 AND X%<16 AND Y%>0 AND Y%<1
6 GOTO 2460
2455 MP%=24:MM%=1:O$=A$:GOSUB 2160:GOTO
2350
2460 WP%=16*Y%+X%:LP%=WP%:LC%=WHITE%:GOS
UB 3890
2470 IF LL%>0 THEN MP%=24:MM%=LL%:O$=A$:
GOSUB 2160:GOTO 2350
2480 MP%=WP%:MC%=WHITE%:GOSUB 3630
2490 MP%=24:MM%=0:O$="":GOSUB 2160
2500 RETURN
2510 :
2520 REM*********************************
3620 :
3630 REM MAKE-MOVE ROUTINE
3640 N%=0
3650 POKE BOARD+MP%,MC%
3660 FOR K=1 TO 4:A=MP%+DIR%(K):IF PEEK(
BOARD+A)<>COLOUR%-MC% GOTO 3700
3670 CP%=A:CC%=COLOUR%-MC%:GOSUB 4040:IF
 CLIB%<>0 GOTO 3680
3675 KO%=A:N%=N%+CSTN%:RP%=A:RC%=COLOUR%
-MC%:GOSUB 3780
3680 IF CLIB%=1 AND MC%=BLACK% THEN A1$=
"ATARI":A2$="      "
3690 IF CLIB%=1 AND MC%=WHITE% THEN A2$=
"ATARI"
3700 NEXT
3710 IF N%<>1 THEN KO%=0
3720 CAPTURE%(MC%)=CAPTURE%(MC%)+N%
3730 GOSUB 1730
3740 RETURN
3750 :
3760 REM*********************************
3770 :
3780 REM REMOVE ROUTINE
3790 IF PEEK(BOARD+RP%)<>RC% THEN RETURN
3800 POKE BOARD+RP%,0
3805 SK%(STACK%)=RP%:STACK%=STACK%+1
3810 RP%=SK%(STACK%-1)+DIR%(1):GOSUB 378
0
3820 RP%=SK%(STACK%-1)+DIR%(2):GOSUB 378
0
3830 RP%=SK%(STACK%-1)+DIR%(3):GOSUB 378
0
3840 RP%=SK%(STACK%-1)+DIR%(4):GOSUB 378
0
3845 STACK%=STACK%-1:RP%=SK%(STACK%)
3850 RETURN
3860 :
3870 REM*********************************
3880 :
3890 REM LEGALITY ROUTINE
3900 LL%=0:LK%=0
3910 IF PEEK(BOARD+LP%)<>0 THEN LL%=2:RE
TURN
3920 POKE BOARD+LP%,LC%
3930 FOR K=1 TO 4
3940 LA%=LP%+DIR%(K)
3950 IF PEEK(BOARD+LA%)<>COLOUR%-LC% GOT
O 3960
3955 CP%=LA%:CC%=COLOUR%-LC%:GOSUB 4040:
IF CLIB%=0 THEN LK%=LK%+CSTN%
3960 NEXT
3970 IF LK%=0 THEN CP%=LP%:CC%=LC%:GOSUB
 4040:IF CLIB%=0 THEN LL%=4
3980 POKE BOARD+LP%,0
3990 IF LP%=KO% AND LK%=1 THEN LL%=3
4000 RETURN
4010 :
4020 REM*********************************
```





CAROLINE CLAYTON

Black Captures
If White makes the mistake of playing on these internal liberties earlier in the semeai battle, as shown here, then it lays itself open to capture. After Black has played stone 2, only one liberty remains to the white group and the battle is effectively lost

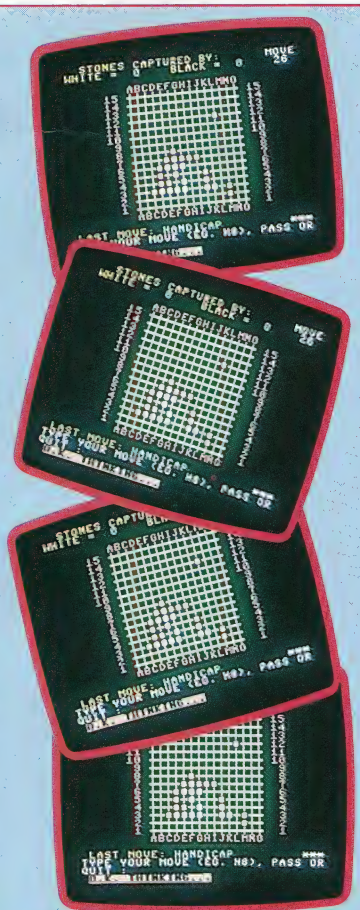
current player is updated and the screen is reprinted using the print_board routine.

When a stone, or group of stones is captured, PROCremove is called to remove them from the board. This routine works in a similar recursive way to PROCsearch (see page 1426). It initially removes the stone in position P% from the board. It then calls itself recursively to remove any stones of the same colour that are immediately north, east, south or west of the current position. Again, in the Commodore 64, Spectrum and Amstrad version, the 'stack' is used to simulate recursion.

Now we turn to the routine that will allow you to enter and make moves — PROCwhite_move. Since we've already got a number of useful routines that we can call, most of the code concerns itself with checking the validity of the input. The routine FNinput (at line 2350) will allow you to enter any four characters, so it has to check that you have entered PASS, QUIT or a legal board position.

PASS merely requires the Ko flag to be reset before returning, whereas QUIT sets the end-of-game logical value to TRUE, so that on return, the program exits the main move loop from line 60 to 90. If the input is neither PASS nor QUIT, the program code breaks the input into horizontal (A to O) and vertical (1 to 15) co-ordinates which are then combined into a board position, P%. You'll notice that we can't use the function VAL here very easily. This is because the input could be two or three characters long and VAL(MID$("A1?",2)) would still give the legal result of one, even though there is a spurious question mark at the end of the string. If the input is valid and FNlegality confirms that the move is legal, the routine calls PROCmake_move, resets the message area by calling PROCmessage with the O.K. THINKING message — ready for the computer to play — and then terminates.

Throughout this program, we have used the constants black% and white% instead of the actual numbers one and two. The only exception to this is in PROCprint_board, where the elements of the capture%() array show the number of stones



Commodore 64:

```
  60 mve%=mve%+1:GOSUB 2320
  90 IF NOT over% THEN 60
1380 ko%=0:over%=0
2310 :
2320 REM **** white move routine ****
2340 atari2$="      "
2350 ip%=21:im%=8:iw%=4:GOSUB 1990:REM i
nput
2360 IF a$="PASS" THEN ko%=0:GOSUB 1730:
GOTO 2490
2370 IF a$="QUIT" THEN over%=1:RETURN
2380 x%=ASC(LEFT$(a$,1))-64
2390 y$=MID$(a$,2)
2400 FOR c%=1 TO LEN(y$)
2410 c$=MID$(y$,c%,1)
2420 IF c$<"0" OR c$>"9" THEN mp%=25:mm%
=1:o$=a$:GOSUB 2160:GOTO 2350
2430 NEXT c%
2440 y%=VAL(y$)
2450 IF x%<1 OR x%>15 OR y%<1 OR y%>15 T
HEN mp%=25:mm%=1:o$=a$:GOSUB 2160:GOTO 2
350
2460 lp%=16*y%+x%:lc%=white%:GOSUB 3890:
REM check legality
2470 IF ll%>0 THEN mp%=25:mm%=ll%:o$=a$:
GOSUB 2160:GOTO 2350
2480 mmp%=lp%:mmc%=white%:GOSUB 3630:REM
 make move
2490 mp%=25:mm%=0:o$="":GOSUB 2160
2500 RETURN
2510 :
3620 :
3630 REM **** make move routine ****
3640 n%=0
3650 POKE (board+mmp%),mmc%
3660 FOR k%=1 TO 4:a%=mmp%+dir%(k%):IF P
EEK(board+a%)<>colour%-mmc% THEN 3700
3670 cp%=a%:cc%=colour%-mmc%:GOSUB 4040:
IF clib%=0 THEN ko%=a%:n%=n%+cstn%:rp%=a
%:rc%=colour%-mmc%:GOSUB 3780
3680 IF clib%=1 AND mmc%=white% THEN ata
ri2$="Atari"
3700 NEXT k%
3710 IF n%<>1 THEN ko%=0
3720 capture%(mmc%)=capture%(mmc%)+n%
3730 GOSUB 1730:REM print board
3740 RETURN
3750 :
3760 REM ************************
3770 :
3780 REM remove routine
3790 IF PEEK(board+rp%)<>rc% THEN RETURN
3800 POKE (board+rp%),0
3805 s(stack%)=rp%:stack%=stack%+1
3810 rp%=s(stack%-1)+dir%(1):GOSUB 3780
3820 rp%=s(stack%-1)+dir%(2):GOSUB 3780
3830 rp%=s(stack%-1)+dir%(3):GOSUB 3780
3840 rp%=s(stack%-1)+dir%(4):GOSUB 3780
3845 stack%=stack%-1:rp%=s(stack%)
3850 RETURN
3860 :
3870 REM ************************
3880 :
3890 REM legality routine
3900 ll%=0:lk%=0
3910 IF PEEK(board+lp%)<>0 THEN ll%=2:RE
TURN
3920 POKE (board+lp%),lc%
3930 FOR k%=1 TO 4
3940 la%=lp%+dir%(k%)
3950 IF PEEK(board+la%)=colour%-lc% THEN
 cp%=la%:cc%=colour%-lc%:GOSUB 4040:IF c
lib%=0 THEN lk%=lk%+cstn%
3960 NEXT k%
3970 IF lk%=0 THEN cp%=lp%:cc%=lc%:GOSUB
 4040:IF clib%=0 THEN ll%=4
3980 POKE (board+lp%),0
3990 IF lp%=ko% AND lk%=1 THEN ll%=3
4000 RETURN
4010 :
4020 REM ************************
```

captured by each side. One advantage of this method is that it allows us easily to change the form in which board bytes represent colours.

We can now use this flexibility to make a two-player game by changing the following lines (see the Basic Flavours box for versions for other machines):

```
60 move%=move%+1:black%=1:white%=2:
   PROCwhite_move
80 move%=move%+1:black%=2:white%=1:
   PROCwhite_move
```

This merely changes the real values of black% and white% calling PROCwhite_move each time. Since all the routines have been designed to allow for either value, the computer will allow Black and White to play alternately, with both using PROCwhite_move to check the legality of each move and display them on the board. We'll have to change these lines in the next module so that the computer is able to play. For the time being, however, this minor alteration will allow you to have two-player games.

Basic Flavours

By inserting these lines in the program listings given, you can set the game up for two players, with the computer acting as an intelligent board, checking legal moves and keeping scores.

Commodore 64

```
60 MOVE%=MOVE%+1:BLACK%=1:WHITE%=2:
   GOSUB 2320
80 MOVE%=MOVE%+1:BLACK%=2:WHITE%=1:
   GOSUB 2320
```

Spectrum

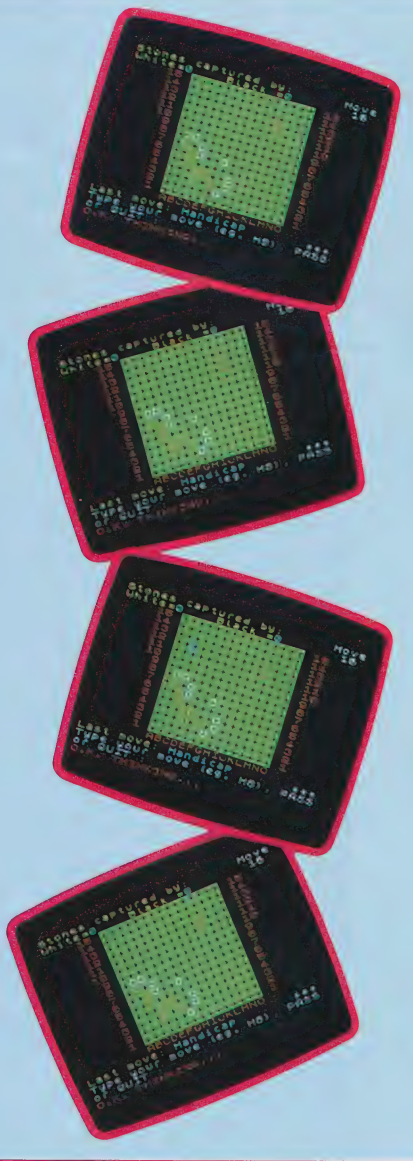
```
60 LET move=move+1:LET black=1:LET white=2:
   GO SUB 2320
80 LET move=move+1:LET black=2:LET white=1:
   GO SUB 2320
```

Amstrad CPC 464/664

```
60 mve%=mve%+1:black%=1:white%=2:
   GOSUB 2320
80 mve%=mve%+1:black%=2:white%=1:
   GOSUB 2320
```

Sinclair Spectrum:

```
  60 LET move=move+1: GO SUB 232
0
1380 LET ko=0: LET end=0
2310:
2320 REM white-move routine
2340 LET y$="      "
2350 LET ip=19: LET im=9: LET iw
=4: GO SUB 1990
2360 IF a$="PASS" THEN  LET ko=0
: GO SUB 1730: GO TO 2490
2370 IF a$="QUIT" THEN  LET end=
1: RETURN
2380 LET x=CODE (a$(1))-64
2390 LET z$=a$(2 TO )
2400 FOR c=1 TO LEN z$
2410 LET c$=z$(c)
2420 IF c$<"0" OR c$>"9" THEN  L
ET mp=21: LET mm=2: LET o$=a$: G
O SUB 2160: GO TO 2350
2430 NEXT c
2440 LET y=VAL z$
2450 IF x<1 OR x>15 OR y<1 OR y>
15 THEN  LET mp=21: LET mm=2: LE
T o$=a$: GO SUB 2160: GO TO 2350
2460 LET lp=16*y+x: LET lc=white
: GO SUB 3890
2470 IF ll>0 THEN  LET mp=21: LE
T mm=ll: LET o$=a$: GO SUB 2160:
 GO TO 2350
2480 LET mmp=lp: LET mmc=white:
GO SUB 3630
2490 LET mp=21: LET mm=1: LET o$
="": GO SUB 2160
2500 RETURN
2510:
2520 REM ************************
3620:
3630 REM make-move routine
3640 LET n=0
3650 POKE board+mmp,mmc
3660 FOR k=1 TO 4: LET a=mmp+d(k
): IF PEEK (board+a)<>colour-mmc
 THEN  GO TO 3700
3670 LET cp=a: LET cc=colour-mmc
: GO SUB 4040: IF clib=0 THEN  L
ET ko=a: LET n=n+cstn: LET rp=a:
 LET rc=colour-mmc: GO SUB 3780
3680 IF clib=1 AND mmc=black THE
N  LET x$="Atari": LET y$="
"
3690 IF clib=1 AND mmc=white THE
N  LET y$="Atari"
3700 NEXT k
3710 IF n<>1 THEN  LET ko=0
3720 LET c(mmc)=c(mmc)+n
3730 GO SUB 1730
3740 RETURN
3750:
3760 REM ************************
3770:
3780 REM remove routine
3790 IF PEEK (board+rp)<>rc THEN
  RETURN
3800 POKE board+rp,0
3805 LET s(stack)=rp: LET stack=
stack+1
3810 LET rp=s(stack-1)+d(1): GO
SUB 3780
3820 LET rp=s(stack-1)+d(2): GO
SUB 3780
3830 LET rp=s(stack-1)+d(3): GO
SUB 3780
3840 LET rp=s(stack-1)+d(4): GO
SUB 3780
3845 LET stack=stack-1: LET rp=s
(stack)
3850 RETURN
3860:
3870 REM ************************
3880:
3890 REM legality routine
3900 LET ll=0: LET lk=0
3910 IF PEEK (board+lp)<>0 THEN
 LET ll=3: RETURN
3920 POKE board+lp,lc
3930 FOR k=1 TO 4
3940 LET la=lp+d(k)
3950 IF PEEK (board+la)=colour-l
c THEN  LET cp=la: LET cc=colour
-lc: GO SUB 4040: IF clib=0 THEN
  LET lk=lk+cstn
3960 NEXT k
3970 IF lk=0 THEN  LET cp=lp: LE
T cc=lc: GO SUB 4040: IF clib=0
THEN  LET ll=5
3980 POKE board+lp,0
3990 IF lp=ko AND lk=1 THEN  LET
 ll=4
4000 RETURN
4010:
4020 REM ************************
```



GET THE MESSAGE

The Spectrum Interface 1 is most commonly used to control Microdrives or provide an RS232 serial connection. However, the device also allows you to access local area network (LAN) facilities. We discuss ways in which the machine code programmer can make the best use of the most inexpensive network systems available.

We have already discussed ways in which the machine code programmer can use the facilities of the Interface 1 to control Microdrives (see page 1418) and to access the RS232 serial port (see page 1438). In the course so far, we have looked at the general principles of networking (see page 321) and given a simple network game for the Spectrum (see page 346). Let's now look at the facilities available to the machine code programmer. In our discussion, we will use the following convention when naming the machines involved in the network: SELF is your machine, and IRIS is another machine on the network with which you wish to communicate.

There are two ways to transmit programs and data over the network. The first of these is the *broadcast*. This can be 'heard' by any machine on the network that is 'tuned in' when the broadcast is made. It thus provides a very efficient method of distributing one program to a variety of machines quickly. The second is a *directed transmission*, in which SELF specifies the IRIS that is to receive the transmitted information. If the IRIS specified is not listening, the SELF machine will repeatedly attempt to establish communication.

The method by which machines communicate via the network is rather interesting. A machine that wants to transmit data first of all waits until the network is quiet, or 'resting'. To cut down the risk of competition for the network, which would lead to data being 'mixed up', each machine waits for a random amount of time after checking that the net is clear before it tries to claim it for itself. If the net is active, then it waits again.

Once a clear net is found, the SELF machine that is transmitting the data sends a *scout* out over the net. This consists of the machine's station number, transmitted in such a way that it can be read back off the net by SELF. If this is achieved, then SELF is said to have *claimed* the network.

A data *packet* can now be sent. This consists of a header, which holds data about the data block to follow, and a 255-byte data block. What happens next depends upon the type of transmission. If it's a directed transmission, then SELF waits one millisecond for a response signal from the desired IRIS. If such a signal is not received, then after an eight millisecond delay, SELF sends out the scout and header again. The data block is not sent until a response is obtained. If the transmission is a broadcast, then no response is expected and the data block is sent straight out on to the net.

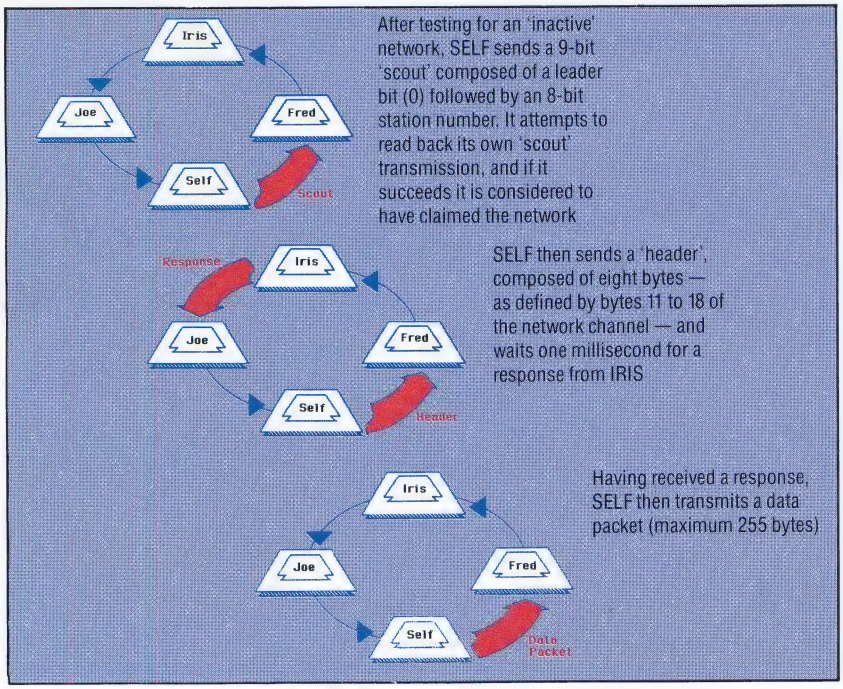
Now, let's consider actually using the net from machine code. There are four hook codes involved, and two important Interface 1 system variables:

- NTSTAT: Is found at address 23749 and is set up with the station number of the machine. The FORMAT command, when used with the network, sets up this system variable. The default value is one.
- DSTR1: We've already met this variable when we looked at the Microdrives. It is at addresses 23766 and 23767 and is set up to hold the station number of the IRIS with which SELF wishes to communicate. Setting this to zero will give a broadcast transmission of data.

As to actually using the hook codes, we are back to a system similar to that used with the Microdrives, in that a network channel has to be opened before we can do anything with it. A net channel is set up by hook code 45 (its structure is shown in the diagram). The channel is 276 bytes long, and is inserted in the Channel Data Area of memory like any other channel.

The header sent into the network consists of all the bytes from NCIRIS to NCHCS inclusive. The data

Station To Station
Network operation involves first 'claiming' the network, followed by the transmission of data (preceded of course, by the necessary header information). The diagram shows data flow on a network of four stations, where the station SELF is making a 'directed' transmission to the station IRIS



CAROLINE CLAYTON

block sent over the network consists of bytes 21 to 275 of the channel space. Bytes 19 and 20 of the channel data are only relevant when the channel is being used to read data from the network. As with the Microdrive channel system, reading and writing from the channel can be easily done by giving its address to CURCHL and then by using RST #10 and CALL #15E6 (as with the Microdrives). Here, though, the data and header are put onto the network when we attempt to write the 256th byte to the Channel Data Area. Now, the hook codes:

- *Hook Code 45:* This opens up a network channel. To use it, NTSTAT is set up with the SELF station number, and DSTR1 is set up with the IRIS station number. This applies if the channel is being opened for reading or writing; but if it is reading, then it is clear that IRIS will transmit the data to SELF. Again, you should ensure that the Interface 1 system variables exist. The hook code can then be used in the usual fashion, and on return the IX register holds the start address of the channel that has been set up.

As has already been mentioned, bytes can be sent down the network using RST #10 and received using CALL #15E6. However, when these are used with the network, the IX register can be corrupted. As the contents of this register, or at least the address of the channel, will be needed to close the channel once we've finished with it, it's a good idea to preserve the contents of IX before using these routines.

- *Hook Code 46:* This closes the network channel, and is called with the start address of the channel required in the IX register. If the channel was being used for writing, then any data still in the Channel Data Area is written to the net. Once a channel is closed, then the memory between the end of the channel and STKEND is moved down.

- *Hook Code 47:* This allows a specific packet to be read from the network. Channel variable NCNUMB should have the appropriate block number in it — it is assumed that NCSELF and NCIRIS will also be holding the appropriate values (these are usually set up by the act of opening a channel). Once the hook code has been used, an attempt to receive the packet will be made. On return, the C flag will be set to one if a 'time out' occurs, or if the data was in some way corrupt and the checksums didn't tally. Thus, by checking the Carry flag's status you can repeatedly request a certain packet from the network. If all goes well, and the packet is received, then an acknowledgement is sent if needed and NCNUMB is incremented.

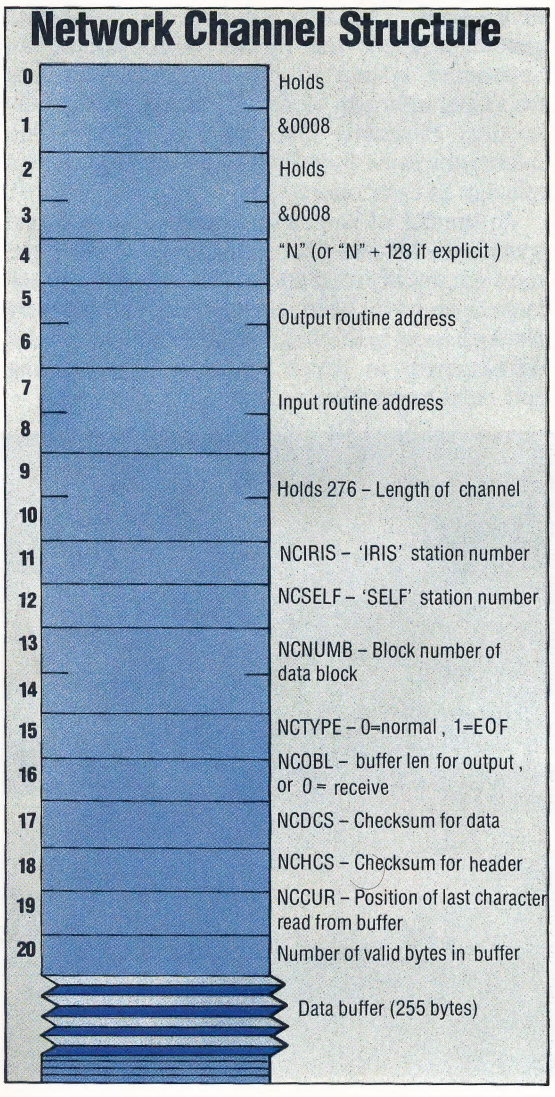
- *Hook Code 48:* This code transmits a packet of data across the network, and so should be regarded as the main routine to use for data transmission. It is used in conjunction with an open channel, the address of which is passed to the routine in the IX register. The A register is set to hold the file type of the data block being sent from the channel. If it is set to one, the block being sent is an End Of File data block; otherwise, for a normal data block, it should be set to zero.

The data is sent as a broadcast or as a directed transmission, depending upon the value in DSTR1 when the channel was opened. At the 'receiving end', if the data is accepted, the header will be read into the header area of the channel and the number of bytes sent will be available in NCOBL, and whether or not the packet recieved was an End Of File packet will be indicated by NCTYPE.

If a successful transmission is achieved, the channel variable, NCNUMB, will have one added to its value.

When using the LAN network, therefore, you can choose between the two different ways of sending and receiving data, depending on what your application is. If you are on a small net, with maybe only one other machine, the method made available by hook codes 48 and 47 may be a little 'over the top', and so simple communication using RST #10 and CALL #15E6 will be quite adequate.

That completes our examination of the Interface 1's ability to interact with other computers or peripherals. In the next instalment, we will examine the use of the Interface 1 to add new commands to the Spectrum's BASIC.



Network Channel Structure
Hook code 45 sets up a network channel of 276 bytes, of which 20 bytes are used for header information and 255 bytes are a buffer for data to be transmitted. Our diagram shows the byte allocation for the header area of the channel

R

READ/WRITE HEAD

This is the part of a tape machine or disk drive that translates the magnetic field on the storage medium (usually disk or cassette) into electrical signals that can be understood by a computer, or vice versa. In order to be able to read the data sequentially, the *read/write head,* or more usually the magnetic media, must be moving. This is done by spinning either the tape spools or the disk. As the magnetic field passes beneath the read/write head, the changing magnetic field held on the disk will induce a current in the head that will then be sent to the central processing unit.

REAL-TIME

A *real-time* system is one in which the computer is required to give an immediate response to user input — anything from flight simulators, which need to reproduce as closely as possible real flying conditions, to cash dispensers.

Obviously, time delays in these types of systems are critical and great effort is put into reducing them to a minimum; however, much depends on the application being used. For instance, while an airline booking system can comfortably accommodate a delay of a few seconds, radar-tracking computers can allow no more than a few milliseconds. The latter is regarded as a real-time system, whereas the former is considered an 'interactive system', in which an instantaneous response, although desirable, is not specifically needed. Programs that are used in real-time operations must therefore be extremely fast and efficient in their operations.

A number of languages have been developed specifically for real-time applications. Among the most commonly used are ADA and MODULA. These languages have been adopted as the primary research tools in the field of robotics — an area of development in which real-time programming and decision-making are vital.

Real To Real
Shown here is a flight simulator used to train airline pilots. In order to provide an accurate simulation it is necessary to make the controls as responsive as they would be in a real aircraft, as well as co-ordinate their readings with the image that the trainee sees through the cockpit window. In order to gain the required real-time response speed, the designers will program the computer with a compiled language such as PASCAL



RECORD

Like many other terms within computing, *record* has come to have a variety of separate but related meanings. The most common use of the term is when it is referring to an element in a file. Here, the file is said to consist of a number of records, which in turn are made up of a number of fields. Thus a record is a number of related data elements.

Similarly, data held on a disk or tape back-up system is also organised into records. The difference here is that a record is defined as the amount of data that is transferred from the storage medium to the system as a single unit. As an example, CP/M records are organised on disks as 128-byte sectors. These types of records are known as 'physical records', since they can be thought of in terms of the hardware limitations. On the other hand, a unit that is organised according to software specifications is called a 'logical record'. These depend on the number of associated data elements that make up the record.

It should be noted that elements which form a logical record are not necessarily physically adjacent to each other in the storage medium.

RECOVERABLE ERROR

Often, when a computer is reading data from a magnetic tape, the error-checking routine, usually a parity bit or checksum, will register a mistake. This may just be a single bit in a file of many Kbytes, so rather than re-reading the entire file, many computer systems include a method of attempting to reload that particular data. Information that can be reloaded in this manner is known as a *recoverable* (or *transient*) *error.*

There are two main methods of dealing with recoverable errors. The first is generally used only in mainframe and minicomputers, in which the CPUs have direct control over the tape machine motors. In this system, the computer will stop the tape when an error is detected, rewind and then attempt to reload from a point before the error was found. This process is continued until the error is rectified. Of course, there may be a fault in the tape or read head and the computer will be programmed to perform the operation a certain number of times before it brings the error to the attention of the operators. Obviously, the more times the computer attempts to rectify the error, the greater the chance of recovery. The BBC Micro approximates this method of recovery, displaying the number of blocks as they are loaded and asking for the tape to be rewound when an error is detected.

The other method of producing recoverable errors is amply demonstrated in the Commodore 64 system. Here, the tape is recorded with a large degree of redundancy (each byte of data is recorded several times). If the cassette deck read/write head fails to read a byte correctly, it should be able to pick up one of the copies further along. The problem with this method, of course, is that the magnetic tape ends up being several times longer than it would be otherwise.

DATABASE

Here, courtesy of Commodore Business Machines, we complete our detailed analysis of the Commodore 64's memory map

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
ISAVE	0332–0333	818–819	KERNAL SAVE Routine Vector
	0334–033B	820–827	Unused
TBUFFR	033C–03FB	828–1019	Tape I/O Buffer
	03FC–03FF	1020–1023	Unused
VICSCN	0400–07FF	1024–2047	1024 Byte Screen Memory Area
	0400–07E7	1024–2023	Video Matrix: 25 Lines × 40 Columns
	07F8–07FF	2040–2047	Sprite Data Pointers
	0800–9FFF	2048–40959	Normal BASIC Program Space
	8000–9FFF	32768–40959	VSP Cartridge ROM—8192 Bytes
	A000–BFFF	40960–49151	BASIC ROM—8192 Bytes (or 8K RAM)
	C000–CFFF	49152–53247	RAM—4096 Bytes
	D000–DFFF	53248–57343	Input/Output Devices and Color RAM or Character Generator ROM or RAM—4096 Bytes
	E000–FFFF	57344–65535	KERNAL ROM—8192 Bytes (or 8K RAM)



© 1983 KOREIN, SHAPIRO, PYLE, YAGER, BADLER—
UNIV. OF PA.