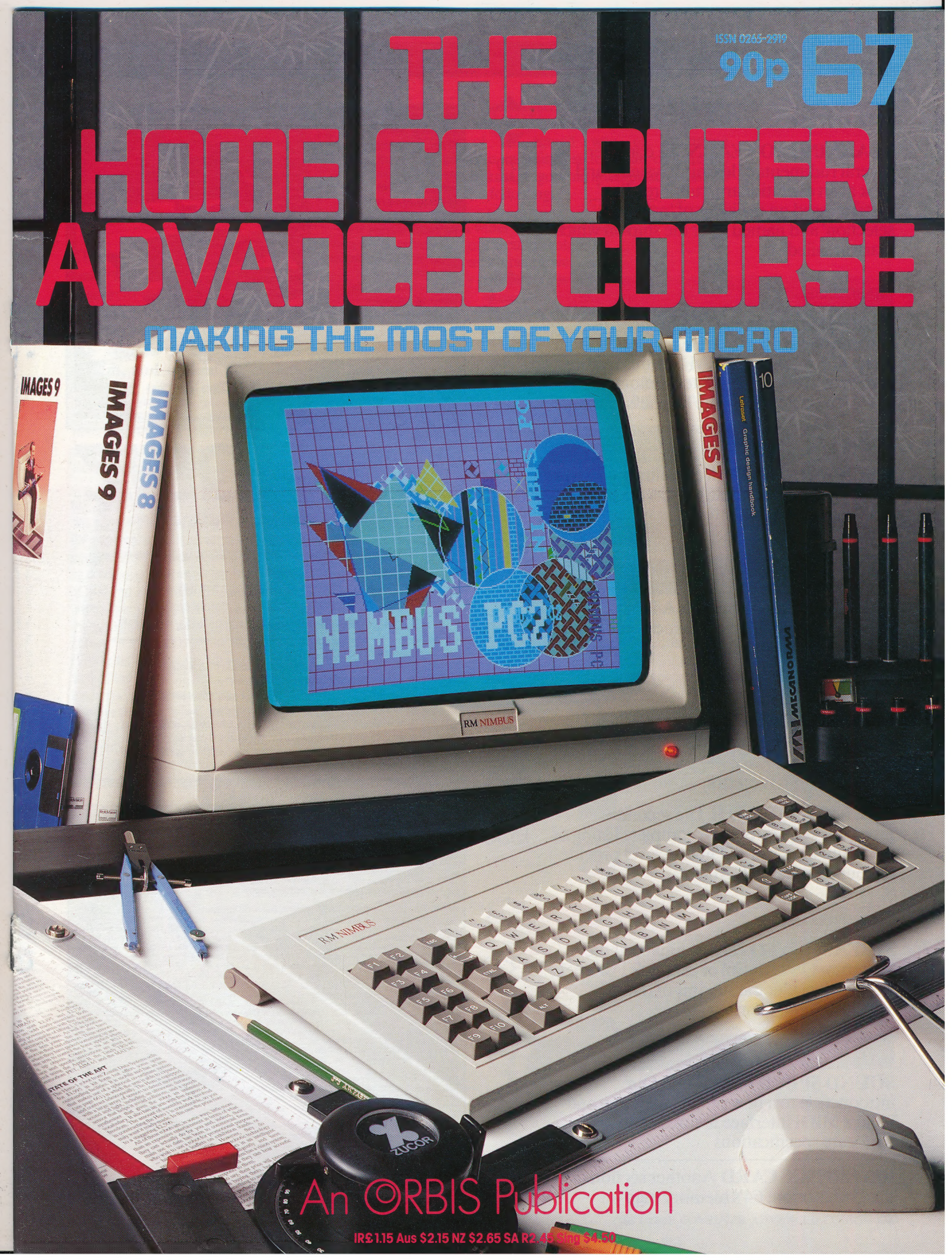
ISSN 0265-2919

90p 67

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

LEARNING TO LEARN We consider several rudimentary learning systems and look at an evolutionary 'rule-breeding' system based on the ideas of 'natural selection' and named after the ship on which Darwin sailed — 1321

HARDWARE

AHEAD OF ITS CLASS Produced by a company with a formidable reputation in the education market, the RM Nimbus features sufficient 'state of the art' technology to make it a contender in the business market — 1328

SOFTWARE

THE SAME AGAIN In examining CP/M's ability to duplicate itself, we discuss the role of the main components of the system — the CCP, BIOS and BDOS — 1324

COMPUTER SCIENCE

USEFUL SIDE-EFFECTS A look at how PROLOG incorporates 'side effects' and several 'extralogical features' — 1326

JARGON

FROM POKE TO PRINTED CIRCUIT A weekly glossary of computing terms — 1340

PROGRAMMING PROJECTS

THE NEW WORLD II The second instalment of a three-part complete listing of our simulation game. This week we provide flavours for the BBC Micro — 1338

MACHINE CODE

STREAM LINED Having looked at the Spectrum OS's control of channels last week, we now consider the streams of data associated with them — 1332

WORKSHOP

SOUND SPECIFICATIONS Our introduction to a new project to design and build a MIDI interface focuses on the hardware specifications — 1335

REFERENCE CARD Another section of the Commodore 64's memory map — INSIDE BACK COVER

Next Week

- **Developing a computer that can 'learn' from its mistakes is a significant aspect of AI research. We look at a method of achieving this goal.**
- **We publish the final part of our complete listing of the New World simulation game.**
- **Our CP/M series concludes with a look at how the program operates within the computer's memory.**



QUIZ

1) What does 'natural selection' mean when applied to AI?
2) What function is performed by an opto-isolator in an electronic sound system?
3) 'The Nimbus is compatible with the IBM PC because the two machines use different central microprocessor units.' Is this true or false, and why?

Answers To Last Week's Quiz

1) The 'Freddy' chip handles the bank switching required to access the additional 64 Kbytes of memory.
2) The Z80's CALL command will run a subroutine held at the address which is the argument of the op-code.
3) A 'wildcard' is a character inserted in a filename, which (when compared on the directory) can refer to any character in that position. This effectively means that the operating system will ignore the character in that position.

COVER PHOTOGRAPHY BY CRISPIN THOMAS

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

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: 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 stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

IMAGES ANNUALS COURTESY OF THE ASSOCIATION OF ILLUSTRATORS

LEARNING TO LEARN

There is a growing body of opinion that the only solution to many of the problems researchers are encountering in the field of AI is the development of a 'child' system — one that learns by itself. We examine here the criteria necessary to create such a system, and outline an evolutionary 'rule-breeding' system called BEAGLE.

If a computer system has improved its performance of a given task over a period of time without being re-programmed, it is valid to say that it has learned. It is important to note that this implies an agreed yardstick, or standard, against which the system can be measured. In the absence of an agreed way of evaluating progress, it is meaningless to speak of learning.

Thus a learning algorithm attempts to do one or more of the following:

- Cover a wider range of problems.
- Give more accurate solutions.
- Obtain answers at lower cost.
- Simplify codified knowledge.

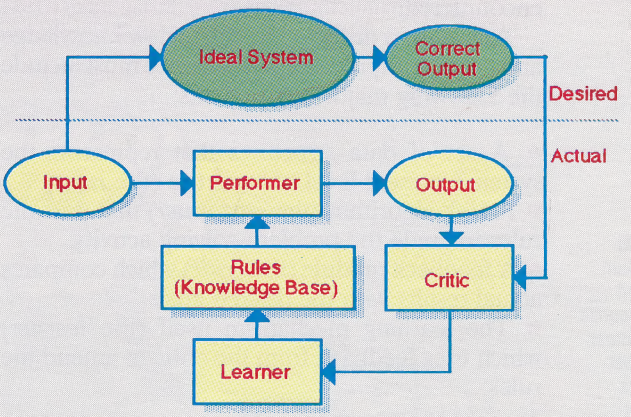
The last point assumes that the simplification of stored knowledge is valuable, even if it does not result in improving the performance of the computer's task. This may be true if the system begins with one set of rules and ends up with another, equally effective set that happens to be more easily comprehensible to us.

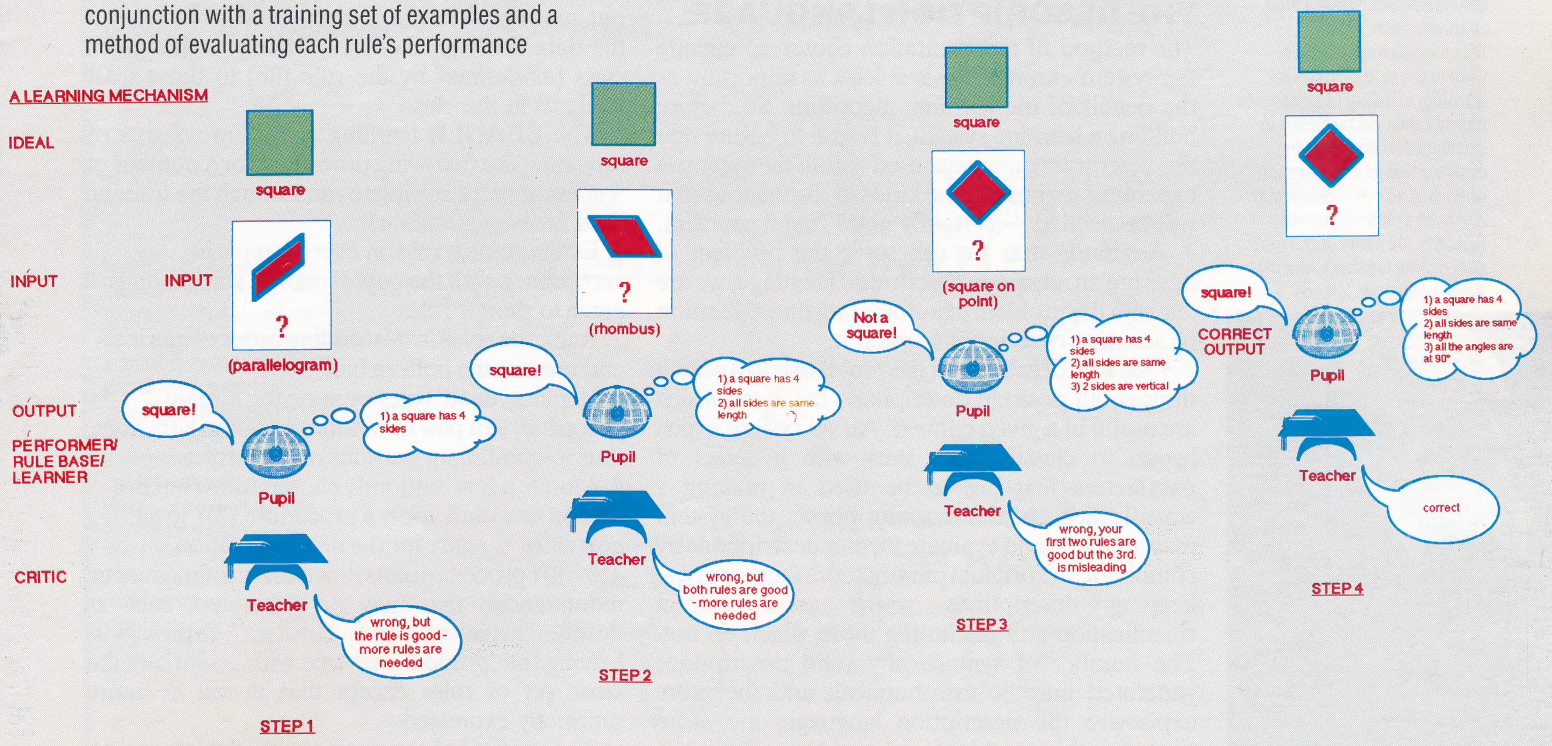
Machine learning can apply to many domains; but most successful learning systems have been applied to classification problems. The objective

IAN McKINNELL ON THE MACINTOS

Critical Intellect

The way in which a child learns is complex and not fully understood. However, psychologists have suggested that children learn by forming rule structures, known as 'schemas', often by a process of trial and error; testing hypotheses (the 'rules') and keeping those which give correct results. In many instances the learning process is guided by someone who has knowledge of the ideal results (the 'critic'). The critic, shown here as a teacher, assists the child (the 'learner') to evaluate and refine his or her internal set of rules that will help to improve the child's performance at a given task. AI learning systems attempt to replicate this by creating a knowledge base which can be used in conjunction with a training set of examples and a method of evaluating each rule's performance





of such a system is to look at input data and classify, identify or interpret it in some way.

Several methods of achieving automatic performance improvement have been attempted. two of the simplest — *rote learning* and *parameter adjustment* — were pioneered by Arthur Samuel in his classic studies of machine learning using the game of checkers. Even prior to this, Rosenblatt and Selfridge, among others, had devised pattern-recognition systems with rudimentary learning capabilities, such as the Perceptron (page 1221).

Parameter adjustment, a method in which program coefficients and parameters are repeatedly adjusted in order to improve performance, is simply a kind of optimisation technique. It has been exhaustively studied in the literature of applied mathematics, and is consequently understood relatively well. Rote learning is entirely 'uncreative', and purely a storage compression technique, whereas the methods we consider here are capable of generalising, and hence learning, the appropriate response in a situation not previously encountered.

Any system designed to create new knowledge and thereby improve its performance must include the following major components:

- A set of data structures that represents the system's present level of expertise (the *rules*).
- A task algorithm (the *performer*) that uses the rules to guide the problem-solving activity.
- A feedback module (the *critic*) which compares actual results with the desired goals.
- The learning mechanism itself (the *learner*) which uses feedback from the critic to amend the rules.

THE DESCRIPTION LANGUAGE

The method of representation chosen to encode the system's knowledge is at least as important as the details of the learning algorithm. So, before building a learning system, it is vital to ensure that the description language used within the system is capable of expressing the kinds of distinctions that will be needed — certainly not a trivial problem.

Assuming that we can solve the problem of devising an adequate description language, we are left with the problem of automating the generation of accurate descriptions within it.

One way of looking at the problem is as a search through all possible descriptions for those which are useful in a given context. An AI system which learns to classify may start with a series of parameters that are to be used in making a classification. In the learning phase, the system may generate and evaluate further descriptions by combining the original parameters in some way — keeping descriptions which assist correct classification and discarding those which do not. The number of syntactically valid descriptions generated may be astronomical; and the more expressive the description language, the more explosive this combinatorial problem will be.

Clearly, some way must be found of guiding the search and thereby ignoring the vast majority of potential descriptions that are irrelevant.

A number of methods have worked well with 'noise-free' training instances, in which classifications are clear-cut, with few grey areas between classes. Dealing with 'noisy' data, however, is a more challenging problem. Let's now look at a comparatively simple method that appears to work quite well in this second instance.

BEAGLE (Biological Evolutionary Algorithm Generating Logical Expressions) is a computer system that produces decision-rules by induction from a database. As such, it addresses the problem — frequently sidestepped — of where the rules in a rule-based system come from.

BEAGLE works on the principle of 'natural selection', whereby rules that do not suit the data are killed off and replaced by 'mutations' of better rules, or by new rules created by mating two better adapted rules. The rules are Boolean expressions represented by tree structures.

The original software consisted of two PASCAL programs, namely HERB (Heuristic Evolutionary Rule Breeder) and LEAF (Logical Evaluator And Forecaster).

HERB requires three input files to be created by the user: a datafile, a payoff file and an old rule file, which may be empty. It produces, as output, a new rule file which is at least as good as the old one. The datafile contains the training set, for which the correct category membership is known. The user also has to furnish a payoff matrix, which defines the value or cost of correct and incorrect classifications.

LEAF is simpler than HERB. It merely takes a datafile in the same format as the training set and runs a rule file over it. It can be requested to print out, among other things, an ordered list of items in the data file from those most likely to fit a given class (as defined by the rule file) to those least likely to fit the class.

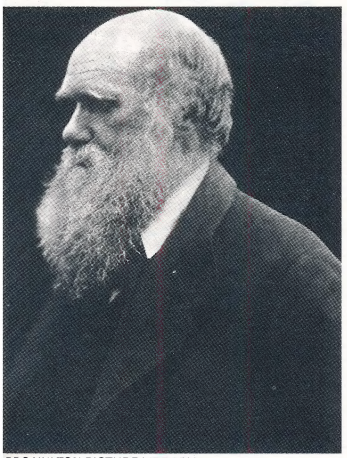
The BEAGLE learning algorithm consists of repeating the following procedure for a number of 'generations' (a complete run through the training data is one generation).

1. Evaluate each rule on every sample in accordance with the payoff matrix, with a bonus given to shorter rules.
2. Rank the rules in descending order of merit and remove the bottom half.
3. Replace 'dead' rules by applying a mate procedure to a pair of randomly chosen survivors, thus re-combining portions of good rules.
4. Mutate a few randomly chosen rules (but not the top one) and apply a procedure TIDY to all new rules — ready for the next generation.

The TIDY procedure cuts down on certain syntactic redundancies that may be generated, such as double negatives and constant expressions, leaving the 'pruned' rule-tree with essentially the same set of rules except that it will be more succinctly expressed.

This method of generating new rules is based on

Charles Darwin (1809-1882)
Darwin's theories of evolution hold that species adapt and improve themselves in response to their environment by natural selection; only the strongest and best adapted members surviving to breed the next generation and pass on their characteristics. This idea has been successfully used in the field of machine learning, allowing AI systems to improve their performance in classification problems by evolving sets of classification rules. It is no coincidence that one such system is named BEAGLE, the name of Darwin's ship on this famous expedition to the Galapagos Islands



BBC HULTON PICTURE LIBRARY

Darwinian ideas of evolution — only the fittest rules survive to breed the next generation. In this way, the rule file 'evolves' during successive training runs to produce a set of rules that can more accurately determine whether or not a given item belongs to the class.

In the second part of our look at machine learning, we will consider the problems of implementing a simple BEAGLE-like learning method in BASIC.

Weather Report

An example of a learning system is one that uses one day's weather record to predict whether it will rain on the next day. The machine must 'learn' a good rule for determining this. The system may start with a series of features that describe the weather — such as the rainfall, amount of sunshine, maximum wind speed and air pressure (all measured at noon, say). In order to learn the rules that will allow the machine to make accurate predictions, a training set of data records must be collected. For each record it will be known whether it rained on the following day. A typical record may be:

Rainfall	0
Sunshine	7.2
Wind	22
Pressure	1017

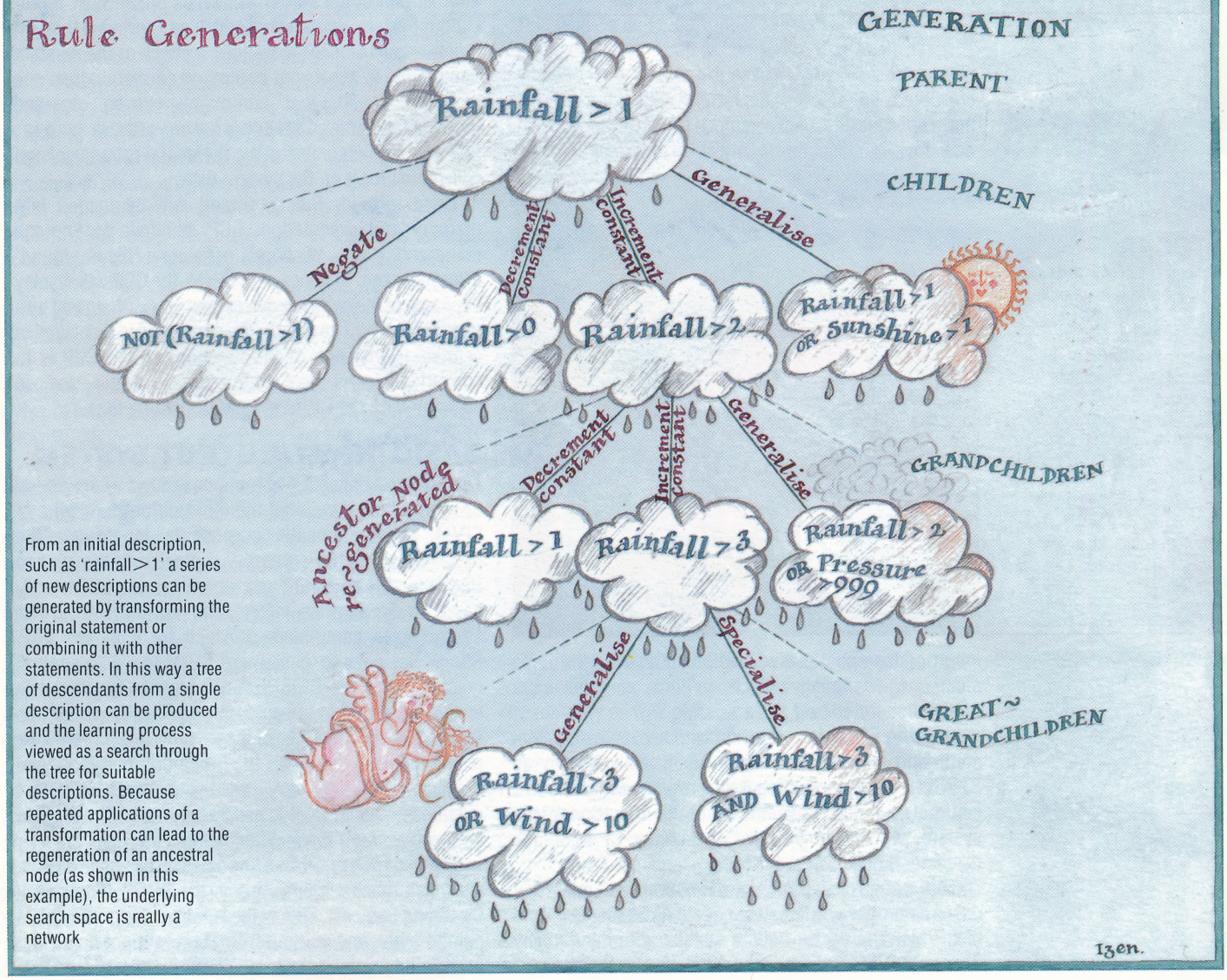
The system must produce rules (perhaps, in the first instance, randomly) and use the training set to test their validity. The rules may be held as components, such as:

Features	Wind
Constants	10
Comparison operators	> or <
Logical operators	AND, OR and NOT

and combine these into expressions like:

sunshine<4 AND pressure<1000

After each training run the good rules can be kept and the bad ones discarded, to be replaced by new rules 'bred' from the remaining good rules. By evolving a set of rules that improves its performance on each training run, the machine should improve its ability to predict tomorrow's weather



From an initial description, such as 'rainfall>1' a series of new descriptions can be generated by transforming the original statement or combining it with other statements. In this way a tree of descendants from a single description can be produced and the learning process viewed as a search through the tree for suitable descriptions. Because repeated applications of a transformation can lead to the regeneration of an ancestral node (as shown in this example), the underlying search space is really a network

GABRIELLE IZEN

THE SAME AGAIN

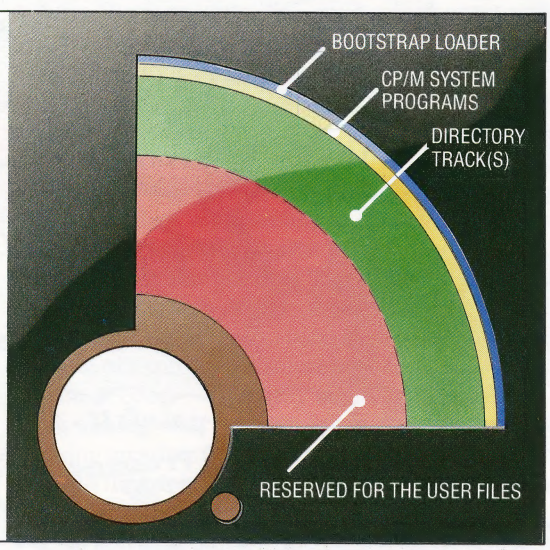
By issuing the transient command SYSGEN, we can enable CP/M to copy itself from one disk to another. We illustrate this process by following the flow of information through the system, and introducing the workings of the CCP, BIOS and BDOS functions.

So far in this series, we have examined the range of commands available to CP/M, and we are now able to perform all the functions that may be required in the home or office. We can load, edit, save and transfer the various types of files that are defined by CP/M as well as examine their properties. However, there is one vital function we have not yet performed — namely, copying CP/M itself. Remember, CP/M is designed to be 'transparent' to the user — it will not show up on any normal directory listing and so cannot be accessed by the standard transient command operations.

Obviously, it is extremely important that we are able to transfer CP/M onto other disks; not only to provide us with a security copy in case the master disk becomes corrupted, but also because many applications require that a specially installed version of CP/M be placed on the disk. An example of this is Dr LOGO on the Amstrad, which incorporates a customised version of CP/M on the same side as the language which, in turn, contains extra procedures for editing and colouring.

Making Tracks

Although most of a CP/M data disk is available for user files, the outermost tracks are reserved for use by the operating system itself. Track zero holds the bootstrap loader, which loads CP/M automatically from power up and reset, while tracks one and two hold the CP/M system itself. Beyond that are the directory tracks, which hold the File Control Blocks showing exactly where the records that make up a file are located



KEVIN JONES

Copying CP/M is quite simple. There is a transient command provided on the system disk known as SYSGEN, and by executing it the program will give you a series of prompts, depending on the configurations of the system, which can be used to transfer CP/M to the destination disk.

This sounds easy enough. However, the operation raises a number of interesting questions about the nature of CP/M. Why do we need a special command? Why doesn't CP/M and its associated resident (built-in) commands appear on a DIRectory listing? And furthermore, how does CP/M in fact read a directory? To answer these questions, let's turn our attention to the nuts and bolts of CP/M.

The operating system that is loaded into the computer can be divided into three parts, each of which handles a distinct part of disk operation. The part that you will be most familiar with is the Console Command Processor (CCP), which seems natural since this is the 'front end' of the system — the area sending information to the video monitor and interpreting commands that are typed in at the keyboard. Because of this, the CCP is the 'public face' of CP/M.

However, it does a great deal more behind the scenes. When information is typed in and entered at the keyboard, the CCP assumes that the information represents commands, and so they are transferred via another part of the system known as BIOS (Basic Input/Output System, which we will discuss shortly) to the CCP buffer area. The CCP then does a check of the resident commands to see if your command corresponds to one of them. If so, that command will be executed immediately. If the command is not one of those 'built-in', CCP will then assume that the command is transient and will search for it on the system disk.

Once located, the command will be loaded into memory and executed. If CCP still fails to find the command, it will, of course, generate a 'file not found' error message. In the case of CP/M, the CCP will display the name of the command in capital letters, followed by a question mark. The reason the command is displayed in capitals is that one of the functions of the CCP is to convert all lower-case commands to upper-case, to conform with the CP/M system of naming files.

BASIC INPUT/OUTPUT SYSTEM

The BIOS is a collection of routines (known as peripheral drivers) which handle all the input/output functions of CP/M. Its fundamental purpose is to manage the peripherals that are currently connected to the system (including the keyboard and monitor — the CCP merely sends and interprets information via the BIOS). The system uses parameters provided for it by the CCP. Should you wish to 'customise' CP/M to run on another machine, it is the BIOS that will have to be altered to enable the program to run correctly. The reason for this (aside from CP/M having certain hardware requirements), is that the method of peripheral management can vary widely between different machines. Therefore, it is necessary to alter the BIOS to take into account these differences.

The third part of CP/M is the section that deals directly with disk management. This is the BDOS (Basic Disk Operating System). This is the least 'visible' of the CP/M components, and without it CP/M would not be able to achieve its primary purpose. BDOS is the software that

manages the files on disk, allocates areas for storage and maintains the directory. As a prelude to a more detailed examination of how BDOS works, lets take a look at how storage space is arranged on a floppy disk.

SOFT AND HARD SECTORS

In hardware terms, a disk is divided into 40 tracks. Modern versions of CP/M will further divide the tracks into 'soft sectors' (so-named because it is the software that determines the sectors). Older types of floppy disk systems subdivided tracks into 'hard sectors', which were read mechanically by the hardware. Naturally, there will be more sectors available on the outer tracks of a disk than the inner ones. Most of the tracks on a disk will be used to store information.

However, there are three tracks (on a 5¼in disk) which are reserved for use by CP/M. These outermost tracks (numbered 0 to 2) will contain the short 'bootstrap loader' (also known as the 'resident monitor') program, which enables CP/M to load itself into the computer with no disk operating system present. Also contained on these tracks is the CP/M system itself. It is the information on these three tracks that is installed on the disk by the SYSGEN command.

The sectors (referred to in CP/M terms as 'records') contain 128 bytes of information. Records can be grouped together to form a 'unit'; one unit can contain up to 128 records. A CP/M file can hold up to 16 of these units. Thus a file can consist of a maximum of 128 by 128 by 16 bytes (256 Kbytes).

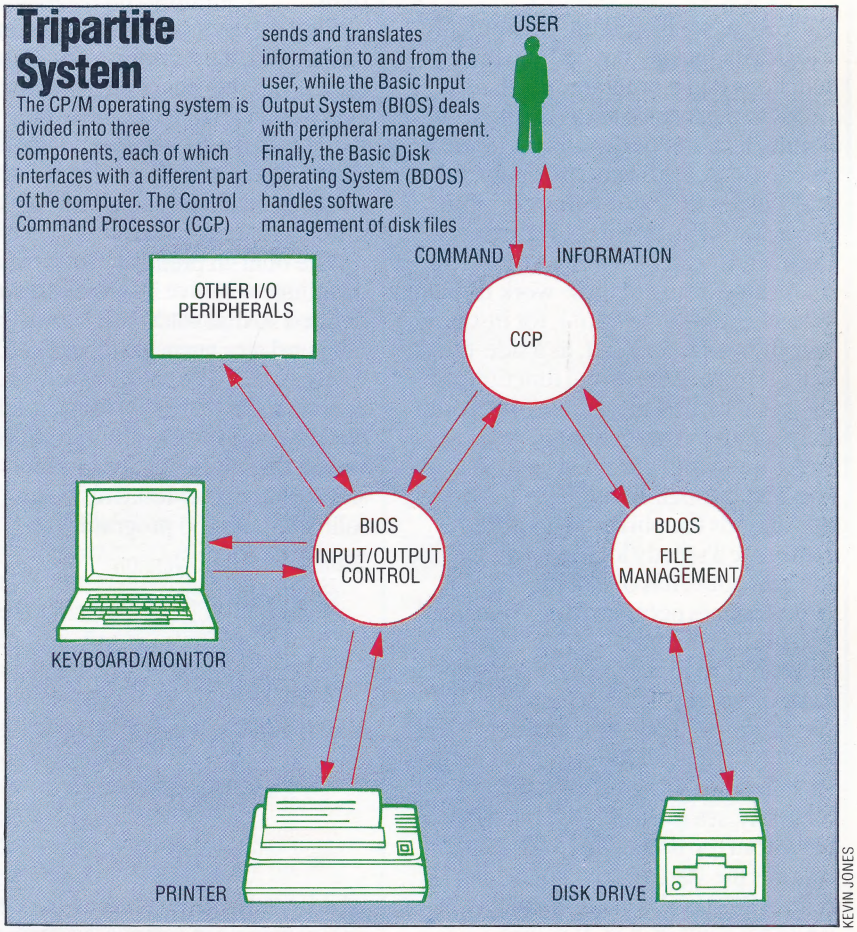
Unless the file is very small indeed, it will not be possible to hold it on a single record. Neither will it be possible, in practical terms, to arrange all the records of a file in sequential order on a disk. Thus a disk operating system must have some method of knowing which sectors on a disk are being utilised for particular files.

In CP/M, this information is known as a File Control Block (FCB). These FCBs are held on the fourth track (which is known as the directory track) of a 5¼in disk. This method of indexing files is read, written, altered and managed by BDOS. A File Control Block is made up of 33 bytes of information and holds all the data necessary to allow BDOS to identify and locate any file on a disk.

The FCB consists of, first of all, the 'entry type', a single byte that holds the identity of the disk drive in which the disk containing the file is currently resident. The next eight bytes contain the file name followed by three bytes that are reserved for the extension. Bytes 12 to 14 are used to hold the total extent of the file, while the fifteenth byte holds the record count (indicating how many records are in the file). The next 15 bytes contain the file's disk allocation map which indicates the sectors that are being occupied by that file. Finally, the last byte will hold the next record number to be accessed (should the file take up more than the 16 Kbytes that can be described by a single FCB).

When a file has been opened by the execution of a command, the CCP will create a version of the FCB in the area of memory usually reserved for transient commands. Extra information about the file — such as its length, number of records and sector allocation — will be supplied by BDOS when it finds the file with a name matching that created by CCP. The reason for placing an FCB in memory is so that the data held in it can be quickly updated by BDOS as the file is manipulated by CP/M commands. At the end of these operations, the final version of the FCB, taking into account the altered record length and sector allocation, will be written onto disk by BDOS.

We can now step back and take an overview of how CP/M retrieves its files. When the CCP receives a command from the terminal, it interprets the first word as the instruction. If the command is not resident in memory, the CCP will send the filename to the BDOS via the BIOS 'messenger' system (see page 1304). The BDOS will then search the directory track of the disk until it finds a matching occurrence of the filename in one of the FCBs. If the file is a COMmand file, the BDOS will examine the disk allocation map to discover which sectors on the disk are occupied by the file, find them on the disk and copy them into memory, at which point the command will be executed.



This may mean that BDOS will be required to find another file to be acted upon and will find and load that into memory as well. When a file is loaded into memory, the BDOS will also supply the appropriate FCB data to an empty version of an FCB that has been set up by the CCP. This FCB will be updated as execution progresses. At the end of the command program, the updated version will be copied back to the disk by BDOS.

In the next, and final, part of this series, we shall take a look at how the components of CP/M are arranged in the memory of the computer and how this space is organised to take maximum advantage of the limited space that may be available.

USEFUL SIDE-EFFECTS

Because of PROLOG's predicate logic, procedural operations cannot be performed without certain compromises. In this instalment, we'll examine how PROLOG incorporates 'side-effects', as well as several 'extralogical' features — such as the 'cut' and 'fail' operations.

PROLOG, as we have seen, is essentially a logical language, but certain functions that a programming language needs to perform, such as reading and writing files and performing arithmetic calculations, are essentially procedural and will not fit very comfortably into predicate logic. In order to cope with these, PROLOG has a number of built-in predicates. Some of these work through side-effects — write (Term), for instance, will always succeed but, as a side-effect, will perform some useful function (in this case, writing the value of Term onto the current output channel).

All input and output in PROLOG is handled through side-effects. Although this offends the purists, who would rather see a strictly logical language, the job gets done; it is one of the compromises necessary for a practical and useful language.

Writing to files in PROLOG is just the same as writing to the screen. A predicate, tell (Filename), instructs PROLOG to send all future output to the specified file; told closes the file. Similarly, see (Filename) tells PROLOG to read from the named file and 'seen' restores the status quo. PROLOG can send ASCII codes using 'put(Code), while get reads in ASCII values.

To perform arithmetic, PROLOG has a number of built-in predicates that are really functions and are evaluated by the interpreter. One such 'evaluable' predicate is is. This is used in the following way:

```
C1 is C + 1.
```

Which is equivalent to the BASIC:

```
C1 = C + 1.
```

Note that C is C + 1 would always fail in PROLOG , while BASIC's C = C + 1 works perfectly well. This is because languages like BASIC, FORTRAN, PASCAL, ALGOL and C use destructive assignment. That is, in assigning the result of the expression (on the right of an assignment statement) to the variable (on the left), any value that had been held in that variable will be over-written. So it does not matter that the same variable had appeared on the right.

PROLOG does not use assignment at all. Instead, it uses unification. That is, once a variable has been instantiated, every other instance of the same variable in the whole clause takes the same value. So saying that C is C + 1 would be like trying to give C two different values at once.

The built-in predicate not is used to introduce negative information and is defined so that not(X) will be true if X is false and vice versa, just like BASIC's Boolean NOT. For not(X) to be true, PROLOG must first show that X is false. It does this in a rather simple-minded way; a thing is false if it cannot be shown to be true! Imagine we had the following PROLOG program:

Standard PROLOG version:

```
language(pascal,hard)
language (cobol,hard)
language (basic,simple)
language (prolog,simple)
```

MICRO-PROLOG version:

```
(language Pascal hard)
(language Cobol hard)
(language BASIC simple)
(language Prolog simple)
```

which lists a few facts about the ease of programming in various languages. We could ask questions such as:

```
language(basic,DIFFICULTY).
```

where DIFFICULTY is a variable, and PROLOG will reply:

```
DIFFICULTY = simple.
```

But, if we ask:

```
language(basic,easy).
```

PROLOG tells us no. The reason is that the goal we have set cannot be proved because it is neither explicitly stated in the database nor derivable from other facts and rules. PROLOG, of course, does not understand the meaning of words, so it cannot tell that simple and easy mean the same thing. Furthermore, it assumes its database is a 'closed world'. That is, everything that can be known *is* known, so anything that cannot be proved to be true must be false.

This idea of 'negation as failure' is one of PROLOG's weakest features and is due to the logical proof method that the interpreter uses. This is called resolution and cannot handle negative information directly. For instance, consider this rule:

```
not(A):— B, C, D.
```

(which says that X is not the case if B, C and D are true). This cannot be written in PROLOG because a negative cannot be in the head of a clause.

EXTRALOGICAL FEATURES

PROLOG is not a purely declarative language. We have already seen how the ordering of clauses in the source code can affect the way the program works, especially for recursive procedures. In fact, the purity of its logic can be further diluted by the use of the so-called 'extralogical' features, fail and cut.

The first of these, fail, is quite easily explained. It is a predicate which, whenever it is encountered, will always fail. Why do we need fail? The main reason is to keep PROLOG working! If we are using our languages program and ask:

```
language(Lang,simple).
```

PROLOG would reply:

```
Lang = basic
```

and stop. It finds the first clause in its database that matches the goal we have set it and, having proved the goal, looks no further. Yet we may have wanted to get a list of all the simple languages. We can force PROLOG to do this if we add a rule such as:

Standard PROLOG version:

```
simplelist :- language(Lang,simple),
```

```
write (Lang), nl, fail
```

MICRO-PROLOG version:

```
((simplelist) (language X simple) (PX)
PP FAIL)
```

Now if we type simplelist, PROLOG will reply:

```
basic
prolog
no
```

What happened was that PROLOG tried to prove the goal simplelist by first proving language(Lang,simple). It found the fact language(basic, simple) and set the variable Lang to the value basic. So the first sub-goal was proved. It went on to the next, and with Lang being basic, it found write(basic). Because write automatically succeeds (writing basic as a side-effect) it passed on to the last sub-goal, which is fail. This, of course, failed, so PROLOG backtracked to write. Since write (and the other evaluable predicates) will not succeed again on backtracking, PROLOG backed up further to language(Lang,simple) and tried to find another way to prove it to be true.

The other feature, cut (written !), is also used to control backtracking. A cut is a predicate that always succeeds. However, as a side-effect, it prevents the interpreter from backtracking over it. This effectively freezes any choices about the values of variables taken earlier in a procedure. It also prevents alternative clauses in the same procedure from being tried. An important use of cut is in conjunction with fail, to ensure that a deliberately failed clause is not tried again. We might define a predicate missing using ! and fail as follows:

Standard PROLOG version:

```
missing(Clause):-Clause, !, fail.
missing(Clause).
```

MICRO-PROLOG version:

```
((missing X) (?X)/FAIL
(missing X)
```

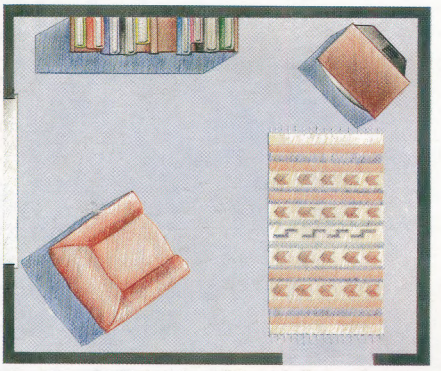
which is true if the clause given as its argument is missing from the database, and false otherwise. Trying this on the language program, we might ask:

```
missing(language(basic,simple)).
```

To prove missing(language (basic,simple)), we must prove language (basic,simple). This is true because it exists as a fact in the database. Moving on to the cut, this succeeds automatically, but then fail causes the entire clause to fail.

Without the cut, PROLOG would have backtracked and tried to find another way to satisfy Clause. However, the cut stands like an impenetrable wall in its way and, as there is no other way to satisfy the goal, it must all fail. Also, because of the cut, PROLOG will not look for alternative clauses, so missing(Clause) ends up false (because Clause was true) and we did not waste effort looking for alternatives.

This particular example illustrates a very important aspect of PROLOG: it can treat its own program as data. This makes it an ideal choice for artificial intelligence programming. This, and other advanced programming techniques in PROLOG, will be discussed in the next instalment.



KEVIN JONES

From Place To Place

In our final article on PROLOG we will devise a program to move a robot about the room illustrated above. Positions in the room are stored in the format place(door) etc. and the robot moves about using the procedure go(Place1,Place2). This first involves proving the sub-goal place(Place1) – i.e. establishing that Place1 exists and can be moved from. What other sub-goals would need to be proved before the program can begin planning a route from one place to another?

Prolog Flavours

The dialect of PROLOG used throughout this series is called DEC-10 PROLOG (after the Digital Equipment Corporation computer on which it was first implemented), and it is widely recognised as the standard. Unfortunately, the most popular microcomputer implementation of PROLOG (MICRO-PROLOG) is one of the very few versions to differ significantly from this standard! The other major micro implementation (PROLOG-1) stays very close to the DEC-10 standard (as does its successor: PROLOG-2).

Programs in MICRO-PROLOG are written entirely in lists, in much the same way as LISP is written. However, the programmer is spared the effort of using this difficult notation by running a front-end program called Simple. Simple is written in MICRO-PROLOG and provides an alternative syntax.

There are many small differences between MICRO-PROLOG and the standard version, making the two dialects look quite different. MICRO-PROLOG puts the predicate as the first element of a list. For example:

```
home_of(mars,martian).
```

would appear as:

```
(home_of Mars Martian)
```

Which of these you prefer is a matter of taste. MICRO-PROLOG also drops all the connectives, such as ':-' and commas, so a clause like:

```
home_of(Planet,Creature):-
    born(Creature,Town),
    on (Town,Planet).
```

would look like:

```
((home_of x y) (born y z) (on z x))
```

which can be read as a list of three sub-lists, the first list being the head of the clause. The Simple program makes this rather awkward syntax easier to understand and, with its infix notation for predicates, the clause could look like this:

```
x home_of y if y born z and z on x
```

which would be even easier to understand if not for the fact that MICRO-PROLOG only allows x, y, z, X, Y or Z (or one of these followed by a digit) as variable names.

Other differences include MICRO-PROLOG using round brackets instead of square ones for lists; separating arguments and list elements with spaces and not commas; using a '/' instead of a '!' for cut; and giving built-in predicates in upper-case letters. The built-in predicates also have non-standard names and, in some cases, do not do quite the same things as their standard counterparts – but this is quite common in programming language dialects.

MICRO-PROLOG does have one useful feature that is not present in DEC-10 PROLOG (although other implementations include it – along with the standard syntax) in that it lets us create program modules. Modules allow programs to be written in functional units and then combined without worrying about predicate name clashes. This gives PROLOG a certain amount of structuring, but MICRO-PROLOG is still a long way from being a structured language

AHEAD OF ITS CLASS

Within the educational market, Research Machines Ltd has had much success with its 380Z and 480Z eight-bit micros. Now, with its 16-bit Nimbus — technologically advanced and exceptionally well-priced — the company hopes to break into the business market.

Conquering New Ground
The RM Nimbus is something of a departure for its manufacturers, Research Machines. Based around the Intel 80186 processor, which is capable of addressing up to a megabyte of memory, the machine suggests that the company is looking beyond its well established education base into the highly lucrative business field

Research Machines Ltd is a name not widely known outside educational circles. Over the past few years the company has established itself as a supplier of quality eight-bit microcomputers to schools and colleges. The expandable 380Z computer and the 480Z network machine each made a great impact during the government's Microcomputers in Education Project, providing Acorn's BBC Micro with its only serious competition.



CHRIS STEVENS

Despite the fact that computers like the 380Z were considered revolutionary at the time of their launches, the demand now is for greater processing power, and this can only be provided by 16-bit processors. In response to this demand, Research Machines has launched a new 16-bit micro known as the Nimbus.

Obviously, the industry has developed considerably since the appearance of the 380Z. The top end of the market is now dominated by the IBM PC and a multitude of 'compatibles', most of which are based around the 8088 processor. Although the 8088 is faster than the Z80, it is already beginning to look somewhat dated. It is to the company's credit that, in developing the Nimbus, it has not taken the easy option of producing yet another IBM clone. It has, rather, chosen to base the machine on the much faster 80186 chip, a true 16-bit procesor that can execute up to a million instructions per second. The result is a computer that leaves most of the competition far behind.

Although the Nimbus is not an IBM-compatible, the 80186 is a close relative of the 8088, and runs under the standard MS-DOS operating system. The keyboard can also be configured to the IBM format, as well as the more familiar Research Machines system used on the 380/480Z. Thus, the keyboard is fitted with 10 programmable function keys, a numeric keyboard and an ALTernate character set key. The keys themselves have an excellent feel to them, comparable to any on the market.

The version presented here is the Nimbus PC2. It is equipped with twin Sony-format $3\frac{1}{2}$in disk drives, although a single-drive version, known as the PC1, is available. These disks can each store up to 720 Kbytes of information and are currently being adopted by an increasing number of 'leading edge technology' products.

The decision to incorporate these drives, as well as several other advanced features, probably persuaded RM away from producing an IBM-compatible machine towards making a computer more at the 'leading edge'. The IBM machine uses the older $5\frac{1}{4}$in format, which is more delicate and has less storage capacity than its Sony counterpart. It would have seemed strange had RM installed disk drives that are becoming increasingly out of date and would have hampered the performance of its machine, considering its other advanced features.

This presents the problem of software compatibility between earlier Research Machines computers, which used $5\frac{1}{4}$in disks, and the Nimbus. This has been solved by the company

providing facilities to enable further disk drives to be added, including, of course, a 5¼in drive. This means that 5¼in drives in PC-DOS, MS-DOS or CP/M format can be read by the Nimbus, a tremendous selling point to owners of computers with different formats who'd rather not abandon all their current software.

Beneath the drives are two ROM cartridge sockets. These are compatible with the cartridge port fitted on the 480Z, although there is little ROM-based software yet available. To the left is a socket to enable a 'dongle' to be fitted. This is a software protection device that prevents users running a program without one, even though they may have a copy of the software.

The software provided with the Nimbus includes an MS-DOS systems disk, Microsoft's Wordplan and Multiplan, as well as RM's own LOGO and BASIC. When the software is loaded, you begin to appreciate what an excellent machine the Nimbus really is. In particular, the graphics facilities provided from BASIC are outstanding.

THE RM BASIC

The BASIC that is bundled with the Nimbus is a development of RM's Version 5. Bearing in mind that RM has its main base in the educational market, it is not surprising that the language has a number of commands designed to encourage structured programming, although not as many as the later Version 6 BASIC. The RM BASIC on the Nimbus contains command structures such as REPEAT . . UNTIL, PROC . . ENDPROC and GLOBAL, but lacks the later DO . . WHILE.

On the other hand, Research Machines has added some useful commands that are designed to be of use to games programmers. The optional joystick and mouse can be used by the JOYX, JOYY and MOUSE commands — the BUTTON command initiates the effects produced by these devices.

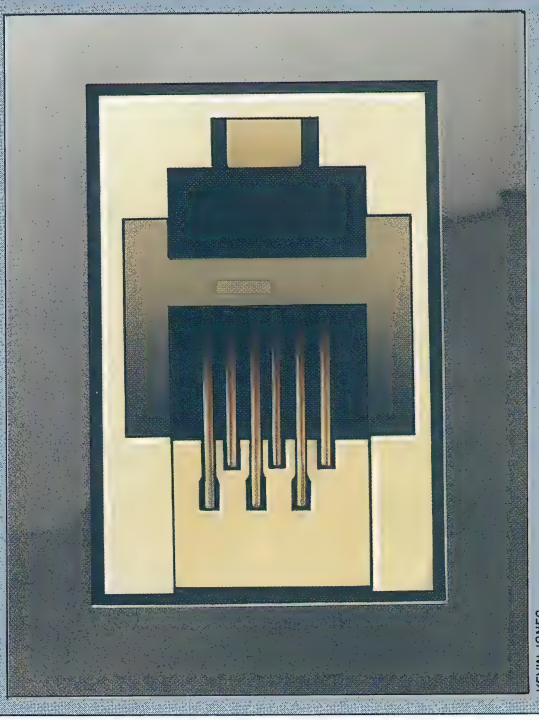
Graphics, sound and other attributes are selected from BASIC by using the SET command followed by any parameters that may be required. By using this instruction, it is possible to select foreground and background colours (defined by the PAPER, BRUSH and PEN commands), draw lines and circles and select character founts, directions and sizes. It is also possible to choose what the manual describes as 'styles' and 'dithering patterns'. These are essentially patterns that can be placed on the foreground or background, adding texture to a picture or, in the case of the 'dithering patterns', producing the effect of a colour that is a combination of the dots within the area.

Apart from the wide range of effects that can be produced on the Nimbus, the speed at which the graphics are drawn on the screen from BASIC is very impressive, and comparable to many 16-bit computers programmed in machine code.

The disks are organised into directories and sub-directories. This means, for example, that in order to enter BASIC, you have to enter the BASIC directory from the main one. This is performed from MS-DOS and BASIC itself by the CHange DIRectory command, or CD for short. Following this command, you must specify the pathnames that are required in order to obtain the required directory. This may sound a bit complicated, but it enables (and encourages) you to organise your directories in a systematic manner. New paths and directories can be created from the operating system and new pathways between existing directories can be established.

Keeping in line with its design philosophy for the Nimbus, Research Machines has opted for 'state of the art' technology rather than established standard peripheral ports. The Nimbus has three separate power-output lines from the main computer unit. A 2A output from the main power



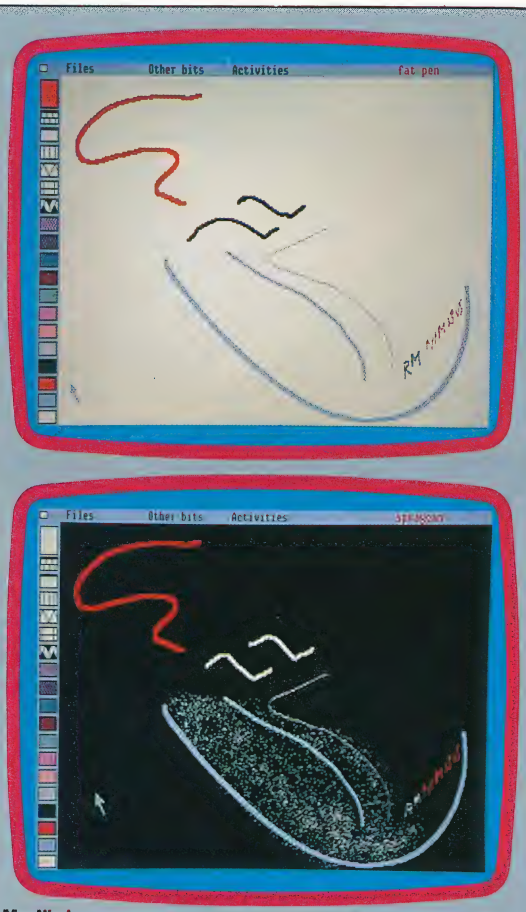


KEVIN JONES

Room To Expand

The RM Nimbus is well provided with interfaces. The computer has two monitor sockets, a mouse/joystick port, a Piconet interface, printer port and no less than three auxiliary power lines. The important Piconet and printer I/O ports use the recently introduced BT jack standard, which many users with older equipment may not be able to connect directly.

The BT bell jack (shown on the left) is a serial interface that is capable of handling both input and output signals. There are six lines available, and these can be arranged in a number of different configurations for transmission, handshaking and other applications



MacNimbus
Among the demonstration programs provided with the Nimbus is a mouse-driven graphics package. Like many mouse drawing programs the screen shows a remarkable similarity to that of the Apple Macintosh. Different 'brushes' and styles can be selected. However, unlike the Macintosh, up to 16 different colours can be displayed on screen at any time

IAN McKINNELL

transformer is positioned in the top left-hand corner of the computer, and two smaller outputs, 12v and 5v, are along the rear panel.

Among the other interfaces provided is a Piconet port, which is installed as a BT-standard telephone jack. The Piconet enables up to 30 separate peripherals, each connected via an external module, to be governed from a single port. The interfacing is serial and constructed to the RS422 standard. It is this new high-speed connection that enables the Nimbus to control the 30 devices.

Unfortunately, the RS422 is not directly compatible, in many cases, with the earlier RS232 standard, so if you want to use these devices, you will have to obtain suitable interfaces. Also provided is a network capability enabling the Nimbus to be connected to a network of other Research Machines computers via the co-axial jack socket.

PRINTER INTERFACE

A curious inclusion is the printer interface. This has been installed as a BT telephone jack identical to the Piconet port, and is somewhat non-standard. Furthermore, the port is serial only and

I/O Chip
Input/output management is controlled from this processor

Z80A Chip
In order to be able to read and process RM disks formatted for use with 380Z/480Z computers, Research Machines have included the familiar Z80A as a second processor

RAM Board
This particular version of the computer has extra RAM provided by this add-on board

RAM Chips
The standard version of the Nimbus comes equipped with 128 Kbytes of RAM

Soft Key Interface
Commonly known as a 'dongle' port, this interface ensures that unauthorised copying of programs is impossible when a dongle is not fitted

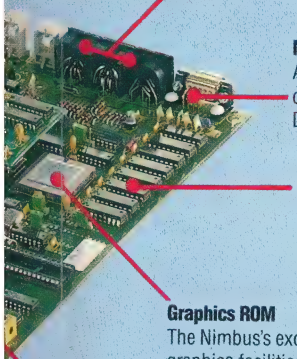
Floppy Disk Control
The disk drives are controlled by this board, which is fitted into one of the expansion slots

Sound Chip
The Nimbu
sound is p
specialised

Disk Drives
The Nimbus PC2 is fitted with a pair of Sony format disk drives



Monitor Sockets
There are two monitor sockets on the Nimbus — one for composite video, the other an RGB colour socket



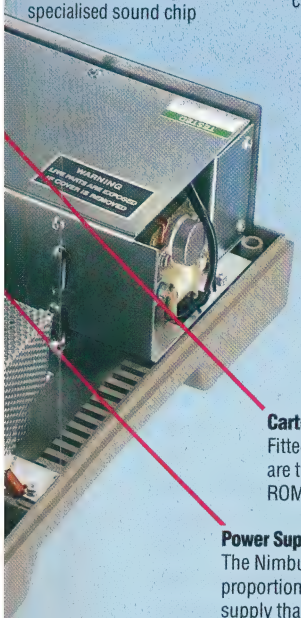
Mouse Port
An optional mouse or joystick controller plugs into this 9-pin D socket

Video RAM
The Nimbus has 64 Kbytes of RAM dedicated to screen memory

Graphics ROM
The Nimbus's exceptional graphics facilities are produced by this custom-built chip

CPU
The Intel 80186 processor at the heart of the Nimbus is one of the most advanced chips currently used

Sound Chip
The Nimbus's three channel sound is provided by this specialised sound chip



Cartridge Ports
Fitted beneath the disk drives are two cartridge ports for ROM-based software

Power Supply
The Nimbus has a proportionately greater power supply than most other computers, because of the number of auxiliary power outlets it possesses

CHRIS STEVENS

is designed to the RS422 standard.

The remaining interfaces that are provided are more familiar. There is a nine-pin D socket that facilitates a joystick or, more likely, a mouse to be connected to the computer in order to run Macintosh-type applications (the demonstration programs contain a drawing program, which, apart from the colour graphics, looks remarkably like a Macintosh screen, though lacking many of the MacPaint facilities).

Two monitor sockets are provided, one of which, for RGB monitors, runs the Cub colour monitor that has been 'badge engineered' by RM. The other is a composite video socket intended for monochrome 'green screen' monitors. Finally, there is a DIN plug for the keyboard.

On the far left (looking at the back of the machine), are four expansion ports into which you can add extra boards. These would include additional banks of RAM, digital/analogue interfaces and so on. Unlike many other manufacturers who insist that this sort of work should only be carried out by authorised dealers, Research Machines has provided full instructions in the User Guide to enable owners to fit their own boards.

COMPLETE DOCUMENTATION

The manuals provided with the Nimbus are well up to the high standard the public has come to expect from RM. The information contained includes a beginners guide to MS-DOS and the disks, right through to a detailed look at the ports. The only obvious omission is an examination of the 80186 processor chip, which, although not likely to be of use to the vast majority of customers, could be to programmers intending to develop their own software on the machine.

It is, as yet, too early to say whom the Nimbus will appeal to most. It is definitely a high-quality machine which, at entry level, is almost half the price of its only serious rival, the IBM PC/AT. The computer will undoubtedly sell, but how well it sells depends on RM's marketing strategy.

Universities and other large educational establishments already having good relationships with RM will probably buy the machine, and will not worry a great deal about software support — they will probably wish to write their own and can transfer the RM software they already possess. The speed and easy programming facilities will obviously represent big selling points as well as the number of peripherals that can be fitted.

However, if the Nimbus is to fulfill its potential in the business market (which the computer obviously possesses) Research Machines will have to convince business users that suitable software will be forthcoming. RM may also have to tailor the interfaces to the Centronics and RS232-standard used in most offices. If Research Machines can break into the business sector, the Nimbus could prove a considerable challenge to the current market leaders — ACT/Apricot and IBM.

RM NIMBUS

PRICE
PC1 single-drive version £1,604, PC2 twin-drive £1,949 (both incl VAT). Educational customers receive 20% discount

DIMENSIONS
400x350x110mm

CPU
Intel 80186 running at 8 MHz

SCREEN
Maximum 80x25 text characters, 640x250 pixels in four-colour mode, 320x125 in eight-colour mode

INTERFACES
Network jack, Piconet port, printer port, 12v, 5v and 2A power outputs, composite and RGB monitor sockets

LANGUAGES AVAILABLE
RM BASIC, RM LOGO and RM PASCAL

KEYBOARD
83 Keys including numeric keypad and 10 programmable function keys

DOCUMENTATION
The manuals are thorough and contain most of the information a user may require. They also appear to be less dense and easier to read than previous documentation

STRENGTHS
At the price, the Nimbus has few competitors. Advanced technology, together with an ease of use, excellent graphics and good manuals, should make this a successful product for the company

WEAKNESSES
In the short term at least, the technology may be a little too advanced — the rest of the industry has not caught up with the Nimbus yet and thus certain interfaces may be in short supply

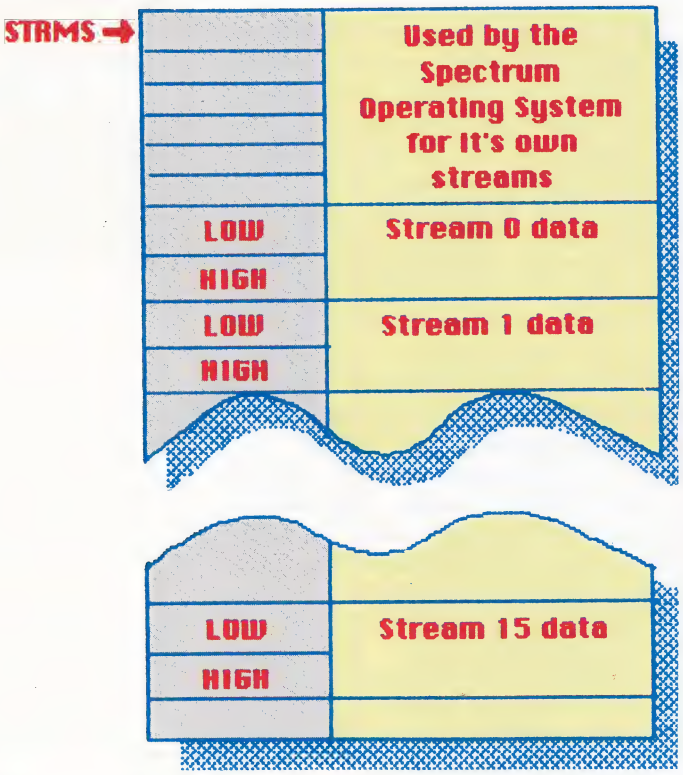
STREAM LINED

Having shown how the Spectrum's operating system selects and controls its input/output channels in the previous instalment (see page 1317), we now take a look at the streams of data that it sends down these channels to the printer and screen, or receives from the keyboard.

You will recall from our previous discussion that whenever we turn the Spectrum on or reset it, certain streams are associated with particular channels in the initialisation process. The following table shows these associations:

Channel	Stream	Channel Description
K	0	Keyboard and lower part of the screen
K	1	
S	2	Main part of screen
P	3	ZX printer

However, there are other streams available to us that are not used in a standard Spectrum. It should be noted that the following description refers to a Spectrum without the Interface 1 device fitted; everything is altered when we use local area networks, Microdrives or the serial interface.



These devices will be discussed in a later part of the course. The other streams are numbered 4 to 15, and they can also be associated with any I/O channels that we may want to interface.

The Spectrum operating system keeps a record of which streams are associated with each channel that is being used. This information is kept in a 38-byte table called STRMS, which can be found in the area reserved for system variables at address 23568. There are six bytes at the head of the table which are used by the Spectrum OS for its own purposes. Following these, each of the 16 streams is represented by a two-byte entry. The arrangement of the table is shown in our diagram.

The following piece of BASIC code will print out the contents of this area of memory for you to examine:

```
10 FOR I=23568 TO 23604 STEP 2
20 PRINT PEEK I+256*PEEK (I+1)
30 NEXT I
```

This prints out all the two-byte entries. The first three are the details of the Spectrum's internal streams, and then we see the entries for streams 0 to 15. The values in this latter portion of the table act as pointers to different entries in another table, located elsewhere in memory and pointed to by a system variable called CHANS. This is found at address 23631, and the address of the CHANS table can be found using:

```
PRINT PEEK 23631+256*PEEK 23632
```

This system variable should *never* be poked. The expression we have just used gives you the start of the *channel record table*, which holds various items of information for each channel. The value read from the STRMS table is used to index this channel record table. A STRMS table entry gives the (offset+1) from the beginning of the channel record table of the information about the channel that is to be used with a particular stream. Thus, the STRMS table entry for Stream 3 (to the printer) is 16. Subtract one from this and we obtain an offset of 15. Therefore, the data regarding channel P will be found at address:

```
(PEEK 23631+256*PEEK 23632)+15
```

(assuming you've just turned the computer on and haven't altered the stream and channel associations after initialisation).

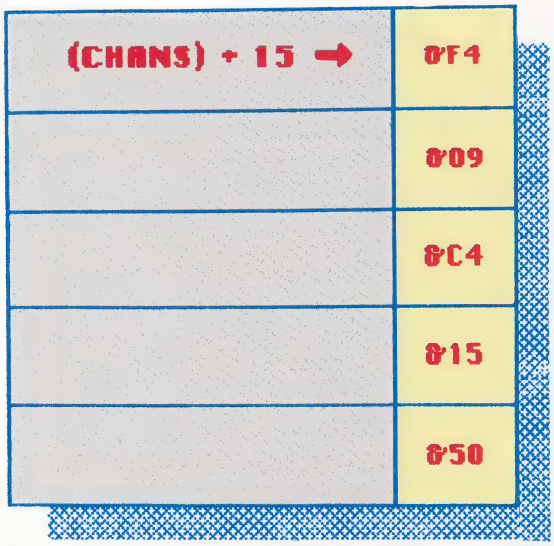
If the data in a given STRMS table entry is zero, then it indicates that the stream has not been associated with any channel. On an unexpanded Spectrum, this is the case with streams 4 to 15 after initialisation.

Each entry in the CRT is of the form:

lo-byte	Address of the output routine for the channel
hi byte	
lo-byte	Address of the input routine for the channel
hi-byte	
n	Single byte representing the channel code letter in ASCII format e.g. K, S or P

A STRMS table entry points to the second byte in each channel record table entry — hence the need to subtract one before we can use it. This table is at the heart of the Spectrum's input/output system. If a channel is not able to perform one or other of the I/O operations, then the appropriate address entry in the CRT table will point to a routine in the Spectrum ROM that generates the Invalid I/O Device error message. So for a channel capable of output only — such as channel S — the input routine address entry will point to this error routine. Otherwise, the appropriate ROM routine for output or input will be pointed to be these table entries.

Let's examine channel P's entry in the channel record table. Remember that under normal circumstances this is an output only channel, sending data to the ZX printer. The data for this channel in the CRT entry is:



The entry for channel S is at (CHANS)+5, and the channel K data is at (CHANS)+0 — (CHANS) is the address held in the CHANS system variable.

By using the OPEN# and CLOSE# commands, we can associate unused streams with one of the channels that we've looked at so far. But the real use of these two commands is to associate streams with other channels in an expanded system, such as those provided by Interface 1. However, we'll take a brief look at the use of these commands without Interface 1 connected. For example:

```
OPEN #4,"S"
```

will associate stream 4 with channel S. If you issue this command and then examine the STRMS table, you'll find that an entry has been made in the position that refers to stream 4. Variables are also allowed in such a statement, such as:

```
10 LET n=4
20 LET c$="S"
30 OPEN #n,c$
```

which will perform exactly the same operation as the preceding OPEN command. Once we've opened a channel in this way, the PRINT# and LIST# commands will enable you to send data to the channel via this new stream. So you can add:

```
40 PRINT #4;"Hello"
```

which will print to the screen. LIST#4 will list the program on the screen in the usual way. If a stream had been attached to a channel that permitted input as well as output, then the INPUT# command could also be used. These extended PRINT and INPUT statements are only really useful when the additional channels provided by Interface 1 are available — as we shall see later in the course.

Once we've finished with a particular combination of stream and channel, and we want to 'dissolve the partnership' between them, we use the CLOSE# command. Thus, after our previous OPEN command, a CLOSE #4 command would dissociate stream 4 from channel S. Once this is done, any attempt to send data to this 'closed' stream will generate an error message.

The PRINT#, LIST#, INPUT# and INKEY$# commands can also be used with streams 0 to 3. These streams are effectively permanently open — as you'll find out if you issue these commands:

```
10 CLOSE #2:REM should close channel S
20 PRINT #2;"Hello"
```

No error message is generated — the OS makes sure to open the channel again before passing information to it. Now let's consider the LIST# command.

```
LIST #3
```

is the same, under normal circumstances, as LLIST — the program is listed to stream 3, which as we have seen is usually associated with channel P. LIST #0 and PRINT #0 are quite interesting commands, as these print to the lower part of the screen (which is normally 'out of bounds' to PRINT statements). Note that anything printed to this part of the screen will be overwritten when the Spectrum OS prints a prompt or error message. PRINT#3 is identical to LPRINT under normal circumstances.

Finally, it is possible to modify the behaviour of a channel on the Spectrum to make it do something else rather than what it has been initialised to do. For example, channel P is often modified to enable a Centronics printer to be driven by the Spectrum and a suitable interface. If we alter the 'output routine' entry of the appropriate channel record table entry, then LPRINT and LLIST will now send the data to a routine that handles the Centronics printer. Of course, it need not be a printer; it could be an output port or any other output device.

Hi-Res Print Program

The Program Hi-Res Print given here intercepts data sent along the P channel on the Spectrum and diverts it to a routine that provides the user with a high-resolution print facility. The listing can be entered using a standard assembler (CHAMP, DevPac or Picturesque, for example). If you do not have an assembler then you will have to enter the hex coding directly, either by using a monitor or the Hex Loader given here. After you have typed in the Hex Loader program and run it, you will be asked to type in the start address (which is 65068). You will now be expected to type in hex coding, and this can be entered only two digits at a time. For example, the line of code:

2130FE

would need to be typed in as:

21 (ENTER)
30 (ENTER)
FE (ENTER)

When all the code has been entered, type S and the program will finish. You will then be prompted for the checksum (which is 29155), and this will indicate whether you have made a typing error.

The new command provided by the program has the format:

LPRINT AT X,Y;"MESSAGE"

where X must be between 0 and 168, and Y must be in the range 0 to 248. The "MESSAGE" can be a string variable or a number. Note that when the string being printed using the new command runs over the edge of the screen, it will 'wrap round' but will not move down a line. The X and Y co-ordinates work in exactly the same way as they would for the Spectrum's PLOT command

Hex Loader

```
  10 CLEAR 63999
  15 POKE 23658,8
  17 DEF FN H(H$)=16*(CODE (H$)-48-7*(H$
(1)>"9"))+(CODE (H$(2))-48-7*(H$(2)>"9")
)
  20 INPUT "Start address ";a
  25 LET Z=A
  30 INPUT (a), LINE a$
  35 IF A$="S" THEN  GO TO 90
  40 PRINT a,a$: POKE 23692,255
  50 FOR Z=1 TO LEN A$/2
  60 POKE a,FN H(a$(((z*2)-1) TO )): LET
 a=a+1
  70 NEXT z
  80 GO TO 30
  90 INPUT "CHECKSUM:";D
 100 LET C=0: FOR B=Z TO A-1: LET C=C+PE
EK B: NEXT B
 110 IF C=D THEN  PRINT "The code is OK!
": STOP
 120 PRINT "Oh dear , the code is incorr
ect": STOP
```

Assembly Listing

```
                ORG   65068
          PIXAD EQU   22AAH
          UDG   EQU   23675
          CHARS EQU   23606
                ;     "Make the data in CHANS point to our routine"
2A4F5C    ENABL LD    HL,(23631)
010F00          LD    BC,15
09              ADD   HL,BC
013AFE          LD    BC,DO-IT
71              LD    (HL),C
23              INC   HL
70              LD    (HL),B
C9              RET
                ;     "Preserve regesters and call new print routine"
E5        DO-IT PUSH  HL
C5              PUSH  BC
D5              PUSH  DE
F5              PUSH  AF
CD46FE          CALL  DOIT1
F1              POP   AF
D1              POP   DE
C1              POP   BC
E1              POP   HL
C9              RET
                ;     "Check to see if a holdsan AT control code"
F5        DOIT1 PUSH  AF
3A1DFF          LD    A,(ATFLG)
FE00            CP    0
200B            JR    NZ,GETXP
F1              POP   AF
FE16      ATCHQ CP    22
201D            JR    NZ,CRCHQ
3EFF            LD    A,255
321DFF          LD    (ATFLG),A
C9              RET
FEFE      GETXP CP    254
2809            JR    Z,GETYP
F1              POP   AF
3216FF          LD    (XPOSI),A
211DFF          LD    HL,ATFLG
35              DEC   (HL)
C9              RET
F1        GETYP POP   AF
3217FF          LD    (YPOSI),A
3E00            LD    A,0
321DFF          LD    (ATFLG),A
C9              RET
                ;     "Check for end of print data (signified by 13)"
FE0D      CRCHQ CP    13
2001            JR    NZ,VCHRQ
C9        SKIPC RET
                ;     "If the code in a is    between 32 & 128 or if"
                ;     "it is a UDG code then  find its data in memory"
FE20      VCHRQ CP    32
380C            JR    C,PRNT?
FE80            CP    128
381C            JR    C,FCHR
FE90      UDGCQ CP    144
3804            JR    C,PRNT?
FEA5            CP    165
3804            JR    C,FUDGC
                ;     "For all other codes a ?will be printed"
3E3F      PRNT? LD    A,63
1810            JR    FCHR
                ;     "Find the data for the  UDG or character in mem"
D690      FUDGC SUB   144
210000          LD    HL,0
6F              LD    L,A
29              ADD   HL,HL
29              ADD   HL,HL
29              ADD   HL,HL
EB              EX    DE,HL
2A7B5C          LD    HL,(UDG)
19              ADD   HL,DE
180C            JR    PRNIT
210000    FCHR  LD    HL,0
6F              LD    L,A
29              ADD   HL,HL
29              ADD   HL,HL
29              ADD   HL,HL
EB              EX    DE,HL
2A365C          LD    HL,(CHARS)
19              ADD   HL,DE
010700    PRNIT LD    BC,7
09              ADD   HL,BC
221BFF          LD    (CHRAD),HL
                ;     "Check to see that the X& Y positions are valid"
3A16FF          LD    A,(XPOSI)
FEF9            CP    249
D231FF          JP    NC,ERRB
3A17FF          LD    A,(YPOSI)
FEA9            CP    169
3078            JR    NC,ERR5
ED4B16FF        LD    BC,(XPOSI)
CDAA22          CALL  PIXAD
321AFF          LD    (PIXPO),A
2218FF          LD    (DFADD),HL
                ;     "Get the first pixel slice of char being printed"
0608            LD    B,8
C5        PRNLP PUSH  BC
2A1BFF          LD    HL,(CHRAD)
7E              LD    A,(HL)
2B              DEC   HL
221BFF          LD    (CHRAD),HL
6F              LD    L,A
                ;     "If the pixel slice doesn't need moving within"
                ;     "the display byte then  jump forward"
3A1AFF          LD    A,(PIXPO)
FE00            CP    0
CAE6FE          JP    Z,PUTIT
                ;     "Move the slice along in the display bytes to"
                ;     "the correct pixel position"
47              LD    B,A
2600            LD    H,0
CB3D      ROTLP SRL   L
CB1C            RR    H
A7              AND   A
10F9            DJNZ  ROTLP
                ;     "Put the slice at the   right screen location"
ED5B18FF  PUTIT LD    DE,(DFADD)
1A              LD    A,(DE)
AD              XOR   L
12              LD    (DE),A
3A1AFF          LD    A,(PIXPO)
FE00            CP    0
CAFAFE          JP    Z,PST
13              INC   DE
1A              LD    A,(DE)
AC              XOR   H
12              LD    (DE),A
1B              DEC   DE
2A18FF    PST   LD    HL,(DFADD)
CD1EFF          CALL  ULINE
2218FF          LD    (DFADD),HL
C1              POP   BC
10C4            DJNZ  PRNLP
                ;     "Add 8 to the X positionso that the next char"
                ;     "is printed in a clear  space"
3A16FF          LD    A,(XPOSI)
C608            ADD   8
                ;     "If the end of the line is reached go back to"
                ;     "the start BUT DO NOT   move down a line"
FEF9            CP    249
DA12FF          JP    C,STAIT
3E00            LD    A,0
3216FF    STAIT LD    (XPOSI),A
C9              RET
00        XPOSI DEFB  0
00        YPOSI DEFB  0
0000      DFADD DEFW  0
00        PIXPO DEFB  0
0000      CHRAD DEFW  0
00        ATFLG DEFB  0
                ;     "Move the address in HL one pixel up the screen"
F5        ULINE PUSH  AF
7C              LD    A,H
25              DEC   H
E607            AND   7
200A            JR    NZ,END
7D              LD    A,L
D620            SUB   32
6F              LD    L,A
3804            JR    C,END
7C              LD    A,H
C608            ADD   8
67              LD    H,A
F1        END   POP   AF
C9              RET
                ;     "Create various errors"
CF        ERRB  RST   8
0A              DEFB  10
CF        ERR5  RST   8
04              DEFB  4
          FINIS END
```

SOUND SPECIFICATIONS

We begin a new project in Workshop: designing and building a MIDI interface for the BBC Micro and Commodore 64, which will allow them to communicate with many digital electronic instruments. We begin by outlining the MIDI concept and explaining the MIDI hardware specification.

The Musical Instrument Digital Interface (MIDI) specification outlines the hardware and software necessary for a system that allows digital instruments equipped with the interface to communicate with each other. As the communication is of a digital nature, we can also introduce a home micro into the MIDI system to control other instruments or to store data in its memory — on tape or on disk. The hardware specification of the interface allows us to build an inexpensive (around £20) and easy to construct, plug-in circuit board.

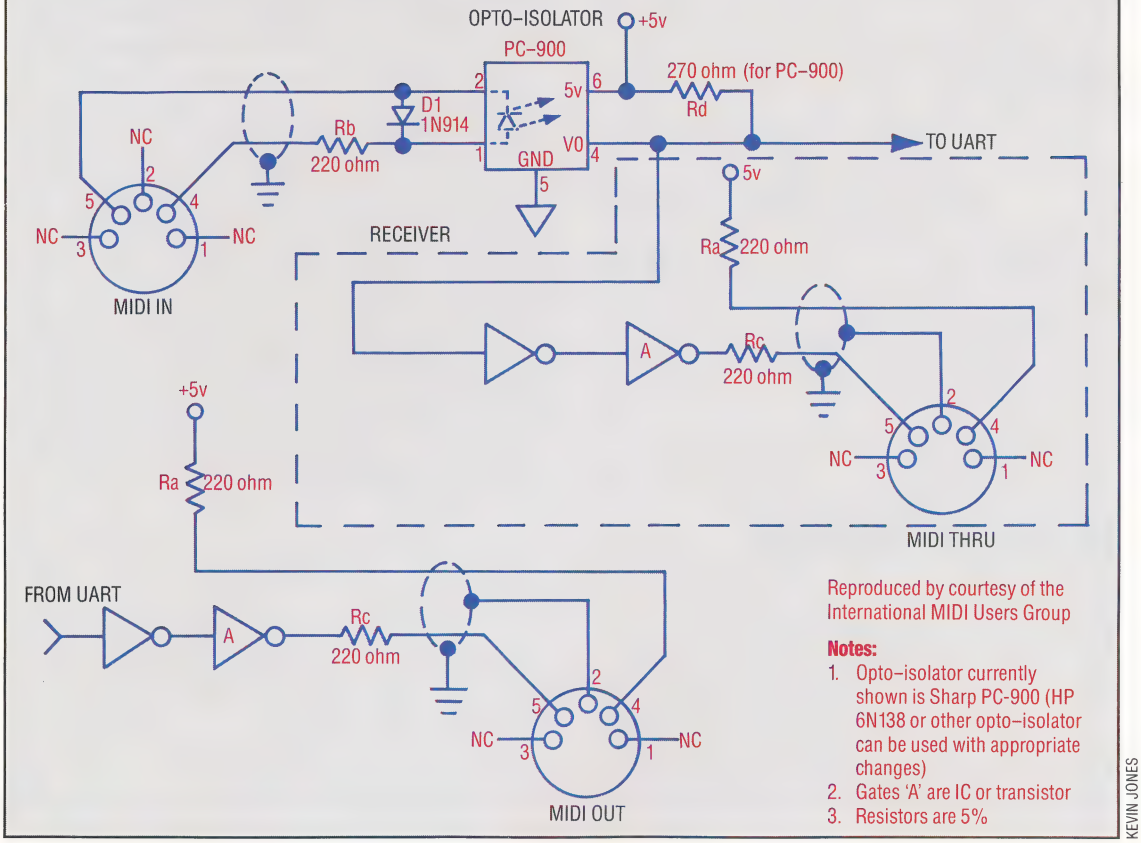
Once the interface has been built, the computer can then be programmed to interact with other devices on the MIDI system by writing the appropriate software. We shall be including such listings later in this project.

Since the introduction of the first MIDI specification in August 1983, the electronic music world has been filled with confusion and misapprehension as to what the MIDI interface can and cannot do. This is because the nature of communication between pieces of electronic equipment is of no concern to musicians, most of whom are (rightly) interested only in what a particular system can offer them in terms of musical performance.

In the course of the project we will explain the MIDI specification in terms more familiar to programmers than musicians, and hence provide you with the same basic information that is used by

MIDI Standard Hardware

The interface usually consists of two sections: one for receiving data and one for transmitting. Each has its own connectors, labelled MIDI IN and MIDI OUT, respectively. Optionally, there is a third connector, labelled MIDI THRU, which simply sends out an exact copy of the data appearing at MIDI IN. This is to facilitate connecting multiple receivers in chains without the need for multiple MIDI OUTs on the transmitter. The circuit diagram of the interface hardware is shown here



Reproduced by courtesy of the International MIDI Users Group

Notes:

1. Opto-isolator currently shown is Sharp PC-900 (HP 6N138 or other opto-isolator can be used with appropriate changes)
2. Gates 'A' are IC or transistor
3. Resistors are 5%

KEVIN JONES

equipment designers who intend their products to be compatible with the standard.

First of all, we need to get some idea of the nature of the data communicated by MIDI. It is best understood by regarding the various parts of an electronic music system as computer peripherals. The distinctions between these parts may not be immediately obvious — for example, a typical modern polyphonic synthesiser is really two separate parts, although they are usually contained in the same box. The two components are the keyboard (the input device) and the sound generation circuitry (the output device). The normal mode of operation has the input device connected directly to the output device — depressing a key immediately produces a sound output.

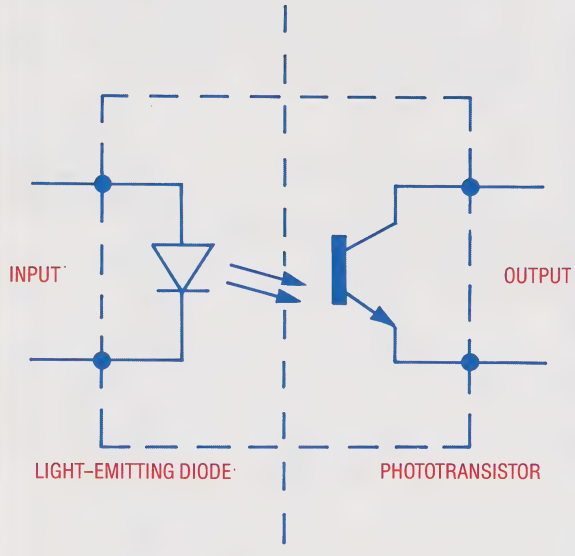
This is a similar situation to that of a sophisticated electronic typewriter, in which the two components — the keyboard and the printer mechanism — can be connected together so that the input data will be immediately reproduced. However, the keyboard and printer are easily recognisable as potential peripherals to a central computer, which will read the input data and *process* it before sending it to the output device. This is, of course, familiar to all of us by now as a word processor.

In the case of the synthesiser, it is important to note that the (internal) interface between the keyboard and sound generators is purely digital. The keyboard simply acts as a controller for the synthesiser in the same way the typewriter keyboard acts as a controller for the printing mechanism. The link can similarly be broken, with a computer inserted between the two devices, forming a 'music processor' system.

The music in this case is not represented as a pattern of air pressure variations (as produced by a gramophone record or compact disc on which the music is digitally encoded), but as a sequence of 'events' inputted at the music keyboard and subsequently transmitted to the output device. It is the function of the sound generation circuits to convert these events into electronic signals that will be reproduced as sound by an amplifier/speaker system. As a consequence, MIDI introduces for the first time the serious prospect of non-keyboard controllers, such as string or wind instruments, provided their respective outputs can be made to comply with the MIDI specification.

A good example of a well known, albeit crude,

Opto-Isolator Circuit

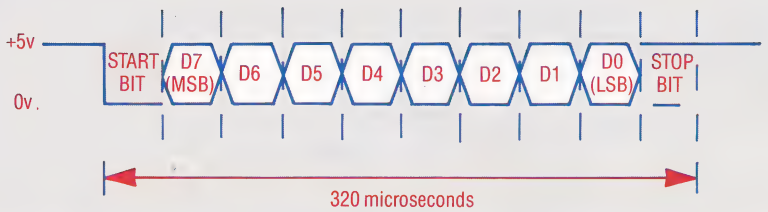


The second requirement for MIDI hardware specifies that the input line should be isolated from the receiving equipment to prevent ground loops in the system. Ground loops are to be avoided since they tend to manifest themselves in the form of a very audible hum at the mains fequency (50Hz). This problem can be overcome with a device called an 'opto-coupler', or 'opto-isolator'. As its name implies, the device is an integrated circuit with an optical (rather than electrical) connection between its input and output. A diagram of the unit is shown.

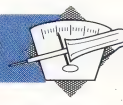
The two input pins on the opto-islolator chip are connected to an internal LED, similar to the red indicator lights used on the front of disk drives. An LED emits light when a voltage is applied in the positive direction. In the opto-coupler, the LED is not externally visible, but instead illuminates a phototransistor, which allows current to pass through when light falls on it. The logic signal (low or high) at the input is thus transferred to the output without an electrical connection.

MIDI specifies that the opto-coupler should require less than 5mA of current to turn on the LED, and the rise and fall times at the output should be less than two microseconds

MIDI Serial Data Byte



KEVIN JONES



music processing system dates back to long before the modern computer age. The pianola was a special type of piano that could play a piece of music recorded as a pattern of holes in a roll of paper. As the paper was rolled through the piano at a constant speed, a sensing mechanism detected the sequence and combination of the holes, each of which represented a particular note. The piano's mallets then struck the appropriate strings. Thus the music was encoded on the paper as a sequence of keypress events.

The distance of a hole *across* the paper represented the pitch of the note, and the distance *along* the paper represented the time at which the note was to be played. It was then conceivable to *edit* the music! For example, if a wrong note was punched, it could be located and covered, while the correct note could be punched into the roll. Encoding data in this way has many direct analogies to the first computer data storage systems that used punched cards.

A modern music processor can be seen as an updated version of the pianola system. The paper is replaced by computer memory and magnetic storage media, and the piano is replaced by a synthesiser with a computer interface. Editing is now simply a matter of processing the data in the computer's memory by means of a suitable program.

Before we can do this — indeed even before we can get the data into the computer — we need to know two fundamental things about MIDI. The first is the type of data transmission used and the second is the format of the data. These two factors correspond to the hardware and software components of the interface specification; this instalment will deal with the hardware required for our project.

SERIAL DATA TRANSMISSION

The first thing we need to know is how a byte of data is transmitted over the link. MIDI operates as an asynchronous serial interface running at 31.25 Kbaud. This means that only one bit of data can be transmitted at a time, and that the receiver and transmitter are not synchronised to each other by common clock signals. The disadvantage of this method of data transmission is its low speed. However, because the link consists of just two wires, it requires very simple and inexpensive cables and connectors.

MIDI specifies that standard 180° five-pin DIN connectors be used (with three-pin XLR professional audio connectors as an option). This helps keep down the cost of the interface, encouraging its incorporation in low-cost instruments.

The main criticism of MIDI is in fact the low serial transmission speed and, as we shall see, transmission delays can become quite noticeable in a large system. On the positive side, however, the transfer rate is low enough for the limited processing speed of home computers to cope.

MIDI data is sent in groups of 10 bits, each representing one byte of data plus a start bit and a stop bit. The line is normally set at a high voltage (+5v for MIDI) when no data is being sent. The start of a transmission is signalled by the line going low (0v) for a one-bit period (the start bit). The eight data bits are then sent-MSB (most significant bit) first-followed by a one-bit period at a high level to signal the end of the data. Each byte therefore takes a total of 10 one-bit periods to transmit, which is equivalent to 10/31.25K = 320 microseconds. This is a very important number for budding MIDI programmers to bear in mind, as we shall see later in the project.

When the interface is idle, the line is at a high voltage level. The receiver continuously monitors the line for the first occurrence of a low level — the start bit of a transmission. The receiver then knows that a byte of data is being sent and can set its counter, which counts out one and a half bit periods from detection of the start bit. After this time period has elapsed, the receiver will be half-way through the bit period containing the MSB of the data and can thus reliably detect whether the bit is a one or zero. The remaining bits are received by successively counting seven one-bit periods and detecting the level each time.

Finally, the stop bit is checked for a high level to verify that the data was correctly 'framed' by the start and stop bits. Using this communication method, the clocks in the receiver and transmitter need not be precisely synchronised, further simplifying the interface. However, the difference should not be so large as to cause timing errors towards the end of the byte, and so MIDI specifies a tolerance of one per cent on the standard clock frequency.

The interface is relatively fast considering its serial nature (RS232 runs at a maximum rate of 19.2Kbaud) and thus the connecting cables should be a maximum length of 15m to prevent excessively slow rise and fall times.

TRANSMISSION DELAYS

At this point, it is convenient to discuss the fundamental limitation of MIDI — namely, transmission delays between instruments. In general, a note on/off message consists of a total of three bytes. The time taken to send this message is therefore 3 × 320 microseconds = 0.96ms. Now consider the situation of a multi-channel sequencer system where, say, 20 of these messages may need to be sent at one time. This means that the last note of the group is nearly 20ms later than intended.

Many sequencers using real-time recording utilise timing resolution down to three or four milliseconds to reproduce adequately the timing nuances of a human performance. Thus a 20ms delay could result in noticeably impaired timing when the recording is replayed.

In a forthcoming part of this series, we shall look at the software of the MIDI specification, and give the meaning of the byte values transmitted over MIDI.

THE NEW WORLD II

We continue our round-up of the New World simulation game by giving the second part of the complete listing. We also discuss the alterations required to allow the program to run on the BBC Micro.

The New World simulation game is written in minimal BASIC and as such will run with minor alterations on the BBC Micro. The main alterations that need to be made to the listing given involve the command that clears the screen and the method of pausing for a keypress. Where the command PRINT CHR$(147) is found in the listing, BBC Micro owners should replace this with CLS.

The code used to await a keypress in the Commodore 64 version of New World (which is the version listed here) involves setting up a loop to GET characters from the keyboard. The loop is terminated when a character other than the null string is received. This type of construction occurs in many places within the program listing as follows:

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

BBC BASIC has a special command that has the same effect as the above, but is much simpler. BBC Micro owners should replace occurrences of this construct with the following:

<line no> I$=GET$

Different display modes on the BBC Micro have different memory requirements. In order to release the maximum amount of BASIC memory for the program, BBC Micro owners should select mode 7. This is the mode automatically selected when the computer is first turned on. However, to make sure, BBC Micro owners may wish to add an extra line at the beginning of the program:

```
5 MODE 7
```

```
5300 REM END OF WEEK REPORT
5305 PRINTCHR$(147)
5310 S$="          CAPTAINS LOG*":GOSUB9100
5312 S$="          -------- ---*":GOSUB9100
5314 GOSUB9200
5316 PRINT:PRINT"          END OF WEEK";WK
5318 GOSUB9200
```

This loop checks for any crew members who have died during the week and reduces the crew number, CN, accordingly

```
5320 X=0
5325 FORT=1TO16
5330 IFTS(T,2)<>-999THEN5350
5335 X=X+1
5340 TS(T,1)=0:TS(T,2)=0
5345 CN=CN-1
5350 NEXT
5355 IFX=0THEN5400
5358 PRINT:PRINT
5360 S$="DURING THE PAST WEEK*":GOSUB9100
```

If all the crew have died (signified by CN=0) then the game is over

```
5365 IFCN>0THEN5390
5367 S$="ALL THE REMAINING CREW DIED*":GOSUB9100
5369 GOSUB9200:PRINT:GOSUB9200
5373 S$="YOUR SHIP DRIFTS HELPLESSLY*":GOSUB9100
5374 GOSUB9200
5376 S$="ON THE EMPTY OCEAN.....*":GOSUB9100
5378 GOSUB9200
5380 PRINT:PRINT
5382 S$="       GAME OVER*":GOSUB9100
5384 PRINT:PRINT
5386 GOSUB9200:END
5388 GOTO5386
5390 PRINTX;
5392 IF X=1 THEN S$="CREW MEMBER HAS DIED*"
5394 IFX<>1 THEN S$="CREW MEMBERS HAVE DIED*"
5396 GOSUB9100
5398 GOSUB9200
5400 PRINT:PRINT
```

Have any of the ship's provisions run out? If so, an appropriate message is printed

```
5405 S$="YOU HAVE NOW RUN OUT OF "
5410 FOR T=1TO4
5415 IFPA(T)<>-999THEN5440
5420 PRINTS$;P$(T)
5425 PA(T)=0
5428 S$="                    AND "
5440 NEXT
5450 GOSUB9200
5455 REM CALC NEW JL
```

The total crew strength is calculated by scanning the array TS(,) and if it is less than half of the full strength, extra time is added to the voyage

```
5460 X=0
5465 FORT=1TO16
5470 X=X+TS(T,2)
5475 NEXT
5480 IFX>799THEN5494
5481 PRINT:PRINT
5482 S$="THE CREW IS BELOW FULL STRENGTH*":GOSUB91
00
5483 S$="SO JOURNEY MIGHT TAKE LONGER*":GOSUB9100
5489 EW=EW+(800-X)/800
5490 REM JL=JL+INT(EW)
5491 JL = 8 + INT(EW)
5492 PRINT
5494 S$=K$:GOSUB9100
5496 GETI$:IFI$=""THEN5496
5499 RETURN
```

In order to generate unpredictable events during the voyage a random number is selected and used to call a routine, using the ON . . . GOTO command

```
5500 REM RANDOM EVENT GENERATOR
5502 IFRC=RMTHENRETURN
5503 REM EXIT IF ALL RANDOM EVENTS DONE
5504 PRINTCHR$(147)
5505 GOSUB9200
5510 X=INT(RND(1)*RM)+1
5515 REM GENERATE RANDOM NO. BETWEEN 1 AND RM
5520 IFRR(X)=1THENRETURN
5522 REM TRY AGAIN IF THIS EVENT ALREADY DONE
5523 RR(X)=1:RC=RC+1
5524 REM SET IND. TO SHOW THIS EVENT DONE AND INCR
EMENT EVENT COUNT
5525 ON X GOTO 5540,5570,5570,5570,5570,5600,5700,
5800,5850,5900,5950,6000,6050
5528 REM GO TO APPROPRIATE CODE FOR THIS EVENT
5530 PRINT:S$=K$:GOSUB9100
5535 GETI$:IFI$=""THEN5535
5539 RETURN: REM RETURN TO MAIN JOURNEY LOOP
```

Some events are bad . . .

```
5540 REM EVENT 1 - MAN OVERBOARD!
5542 PRINT
5543 S$="  DURING THE WEEK*":GOSUB9100
5545 PRINT:GOSUB9200
5546 S$=" 1 PERSON WAS LOST OVERBOARD*":GOSUB9100
5548 S$=" IN A STORM*":GOSUB9100
5550 PRINT:GOSUB9200
5552 S$="YOUR CREW IS NOW REDUCED TO*":GOSUB9100
5554 PRINTCN-1;"MEMBERS"
5558 FORT=1TO16
5559 REM SEARCH FOR CREW MEMBER TO LOSE
5560 IFTS(T,2)=0ORTS(T,2)=-999THEN5566
5562 TS(T,2)=-999:REM DEAD
5564 T=16
5566 NEXT
5568 GOTO5530
5570 REM EVENTS 2 TO 5 - PROVISIONS LOST
5572 X=X-1:REM X NOW POINTS TO PROVISION(1-4)
5574 IFPA(X)=0 OR PA(X)=-999 THENRETURN
5576 REM NO ACTION IF THIS PROVISION ALREADY EXHAU
STED
5578 PRINT
5580 S$="  DURING THE WEEK*":GOSUB9100
5582 PRINT:GOSUB9200
5584 PRINT" SOME OF YOUR ";P$(X)
5586 S$=" WAS WASHED OVERBOARD*":GOSUB9100
5588 PRINT:GOSUB9200
5590 S$="YOU NOW HAVE APPROXIMATELY*":GOSUB9100
5592 PA(X)=PA(X)-INT(PA(X)/(INT(RND(1)*3)+2))
5593 REM REDUCE PROV AMOUNT BY 1/2 1/3 OR1/4
5594 PRINTINT(PA(X)/(CN*PN(X)));
5595 PRINT"WEEKS ";P$(X);" LEFT"
5599 GOTO5530
```

. . . but others are beneficial. For example, it is possible to supplement the meat ration with fish

```
5600 REM EVENT 6 - CATCH SOME FISH
5605 X=0
5610 FORT=1TO16
5615 IFTS(T,2)=-999THENX=X+1
5620 REM COUNT DEAD THIS WEEK
5625 NEXT
5630 IFCN-X<1THENRETURN
5635 REM NO ACTION IF ALL CREW DEAD
5640 PRINT
5645 S$="  DURING THE WEEK*":GOSUB9100
5646 PRINT:GOSUB9200
5650 S$="ONE OF THE CREW CAUGHT*":GOSUB9100
5655 X=INT(RND(1)*10)+11
5660 REM 10 TO 20 KILOS
5662 PRINTX;"KILOS OF FISH"
5665 PRINT:GOSUB9200
5670 S$="YOUR MEAT SUPPLY IS NOW*":GOSUB9100
5675 S$="ENOUGH FOR APPROXIMATELY*":GOSUB9100
5678 IFPA(3)=-999THENPA(3)=0
5680 PA(3)=PA(3)+X
5685 PRINTINT(PA(3)/(CN*PN(3)));"WEEKS"
5690 GOTO5530
```

If the ship is running short of water then the stocks may be replenished by this routine. A number between 10 and 20 — representing the barrels of water which will be added to the stock held in PA(4) — is randomly generated at line 5735

```
5700 REM EVENT 7 - CATCH SOME WATER
5705 PRINT
5710 S$="  DURING THE WEEK*":GOSUB9100
5715 PRINT:GOSUB9200
5720 S$="A RAINSTORM REFILLED YOUR*":GOSUB9100
5725 S$="WATER BARRELS*":GOSUB9100
5730 PRINT:GOSUB9200
5735 X=INT(RND(1)*10)+11
5736 REM 10 TO 20 BARRELS
5740 S$="YOUR WATER SUPPLY IS NOW*":GOSUB9100
5745 S$="ENOUGH FOR APPROXIMATELY*":GOSUB9100
5748 IFPA(4)=-999THENPA(4)=0
5750 PA(4)=PA(4)+X
5755 PRINTINT(PA(4)/(CN*PN(4)));"WEEKS"
5760 GOTO5530
5800 REM EVENT 8 - GOOD WINDS
5805 PRINT
5810 S$="STRONG FOLLOWING WINDS ALL WEEK*":GOSUB91
00
5815 PRINT:GOSUB9200
5820 S$="YOU HAVE MADE GOOD SPEED*":GOSUB9100
5825 S$="AND YOUR JOURNEY TIME IS*":GOSUB9100
5830 S$="REDUCED BY 1/2 A WEEK*":GOSUB9100
5835 EW=EW-.5
5839 GOTO5530
```

Good weather can improve the crew's strength. This is achieved by scanning through the crew strength array — TS(,)— and increasing individual strength ratings by a number between 5 and 15

```
5850 REM EVENT 9 - GOOD WEATHER
5855 PRINT
5860 S$="GOOD WEATHER ALL WEEK*":GOSUB9100
5865 PRINT:GOSUB9200
5870 S$="THE CREW IS FEELING HAPPIER*":GOSUB9100
5875 GOSUB9200
5880 S$="                AND HEALTHIER!*":GOSUB9100
5882 FORT=1TO16
5884 IFTS(T,2)=0 OR TS(T,2)=-999THEN5888
5886 TS(T,2)=TS(T,2)+INT(RND(1)*11)+5
5888 NEXT
5889 GOTO5530
```

The next three routines can, if called, diminish the ship's cargo. Guns can be attacked by rust, bales of cloth by mice, or medicine bottles smashed

```
5900 REM EVENT 10 - LOSE MEDICINE
5905 IF OA(1)=0 OR OA(1)=-999THENRETURN
5910 PRINT
5915 S$="YOU DISCOVER THAT HALF YOUR*":GOSUB9100
5920 S$="MEDICINE BOTTLES HAVE BROKEN*":GOSUB9100
5925 OA(1)=INT(OA(1)/2)
5930 PRINT:GOSUB9200
5935 S$="YOU NOW HAVE ONLY*":GOSUB9100
5940 PRINTOA(1);"BOTTLES LEFT"
5945 GOTO5530
5950 REM EVENT 11 - RUSTY GUNS
5955 IF OA(2)=0 OR OA(2)=-999THENRETURN
5960 PRINT
5965 S$="YOU DISCOVER THAT HALF YOUR*":GOSUB9100
5970 S$="GUNS HAVE GONE RUSTY*":GOSUB9100
5972 S$="AND NO LONGER WORK*":GOSUB9100
5975 OA(2)=INT(OA(2)/2)
5980 PRINT:GOSUB9200
5985 S$="YOU NOW HAVE ONLY*":GOSUB9100
5990 PRINTOA(2);"GUNS LEFT"
5995 GOTO5530
6000 REM EVENT 12 - LOSE CLOTH
6005 IF OA(4)=0 OR OA(4)=-999THENRETURN
6010 PRINT
6015 S$="YOU DISCOVER THAT HALF YOUR*":GOSUB9100
6020 S$="BALES OF CLOTH HAVE BEEN*":GOSUB9100
6022 S$="GNAWED BY MICE*":GOSUB9100
6024 S$="AND ARE NOW WORTHLESS*":GOSUB9100
6025 OA(4)=INT(OA(4)/2)
6030 PRINT:GOSUB9200
6035 S$="YOU NOW HAVE ONLY*":GOSUB9100
6040 PRINTOA(4);"BALES LEFT"
6045 GOTO5530
6050 REM EVENT 13 TO     - ALBATROSS
6055 PRINT:A$="Y"
6060 S$="AN ALBATROSS FLIES OVER THE SHIP*":GOSUB9
100
6062 GOSUB9200
6065 S$="THIS IS A GOOD OMEN*":GOSUB9100
6068 S$="AND THE CREW IS HAPPY*":GOSUB9100
6070 PRINT:GOSUB9200
6075 IFPA(3)<(CN*PN(3)*(JL-WK+1))THEN6090
6080 REM NOT SHORT OF MEAT
6085 GOTO6122
6090 S$="YOU'RE RUNNING SHORT OF MEAT*":GOSUB9100
6095 S$="AND THE BIRD WEIGHS 10 KILOS!*":GOSUB9100
6100 PRINT:GOSUB9200
6105 S$="WOULD YOU LIKE TO CATCH IT?*":GOSUB9100
6110 INPUTI$
6112 PRINT:GOSUB9200
6115 IFLEFT$(I$,1)="Y"THEN6133
6120 S$="PROBABLY JUST AS WELL!*":GOSUB9100
6122 PRINT:GOSUB9200
6125 S$="THE ALBATROSS FLIES AWAY....*":GOSUB9100
6130 GOTO5530
6133 IFOA(2)=0OROA(2)=-999THEN6130
6135 S$="A SHOT IS FIRED.............*":GOSUB9100
6138 GOSUB9200:GOSUB9200
6140 IFRND(1)<.5THEN6150
6145 S$=".........BUT MISSES!*":GOSUB9100
6148 GOTO6122
6150 S$="AND THE BIRD FALLS TO THE DECK!*":GOSUB91
00
6155 PRINT:GOSUB9200
6160 IFPA(3)=-999THENPA(3)=0
6162 PA(3)=PA(3)+10:B$="Y"
6165 S$="YOU NOW HAVE 10 MORE KILOS*":GOSUB9100
6167 S$="OF MEAT......*":GOSUB9100
6170 S$="BUT YOU MAY NOT HAVE MUCH*":GOSUB9100
6172 S$="GOOD LUCK FROM NOW ON!!*":GOSUB9100
6174 GOTO5530
6180 S$="YOU CAN'T - YOU HAVE NO GUNS*":GOSUB9100
6190 GOTO6122
```

The events given above have a minor effect on the voyage, but there are also six more drastic events that can occur, randomly selected by the following routine

```
6500 REM MAJOR CONTINGENCIES
6508 X=INT(RND(1)*10)+1
6510 ON X GOSUB6530,6700,6800,6900,7000,7050
6520 RETURN
```

POKE

POKE is a command issued from a high-level language, such as BASIC, which loads an absolute number into an address in memory — usually a register. To implement this command, you type POKE, followed by an address and then the number to be inserted.

POKE is used for a wide variety of functions. One of its principal uses is to perform operations for which there are no corresponding commands in the high-level language. The Commodore 64, for example, despite excellent sound and graphics capabilities, has no BASIC commands to utilise them. Thus, in order to use these hardware features, the addresses on the VIC (video) and SID (sound) chips have to be POKEd with the chosen parameters.

POKEs can also be of use when we wish to enter machine-level language via a high-level language. The machine code will be held within data statements that can then be read and POKEd into the appropriate addresses.

POLISH NOTATION

This is a method of formulating mathematical expressions devised by a Polish mathematician named Jan Lukasiewicz. *Polish notation* (also known as 'prefix notation') places the operator before the operands, for example +xy instead of the more familiar x+y. The reason for notating in this manner is that computers operate in the same way, with the operator (+) preceding the data to be operated on. Furthermore, it accelerates the process of error checking as well as the operating speed: after decoding the + sign, the computer 'knows' that what follows should be real numbers that are to be added.

Related to Polish notation is 'reverse Polish notation' (known as RPN or 'postfix notation' — see page 649), in which the operator is placed at the end of the expression. This system was popular in early calculators and in early computer languages, such as FORTH, which were designed to match closely the computer's internal processes.

POLYNOMIAL

This is a mathematical expression in which a number of powers and multiples, each based around a single constant variable, are added together. An example of a *polynomial* series is:

$$5X^3+2X^2+2X+8$$

where the variable X is known as the 'indeterminate'.

Polynomial techniques are widely used in computing. For example, they can be used to code and decode data in order to detect transmission errors.

PORT

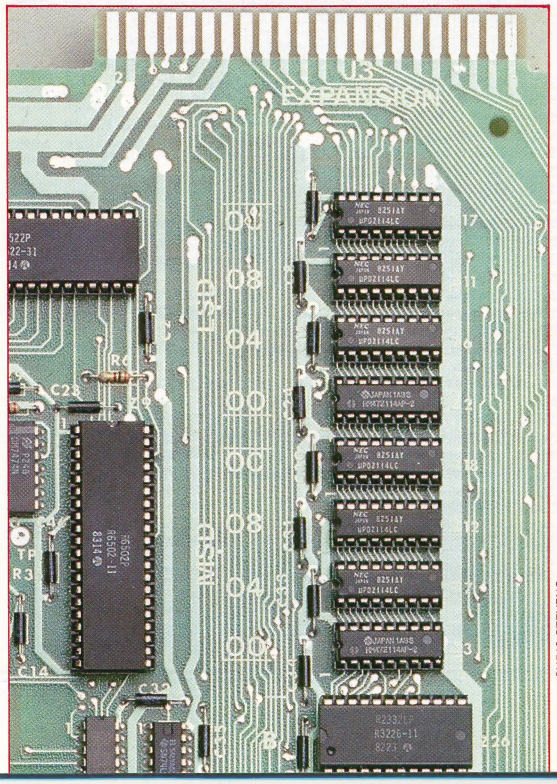
This is the connecting socket through which a signal from a peripheral input/output device can be fed into a computer. Although the *port* is generally connected directly to the computer's circuitry, the signals may need to be modified before they can be acted upon by the processor. Thus a chip known as a PIA (peripheral interface adaptor) is usually associated with the CPU (see page 1280), and this 'translates' signals passing between the processor and the peripheral. The type of functions a PIA chip might perform are converting parallel data transfer into a serial system as well as increasing and decreasing the baud rate of transfer.

Ports are, of course, necessary for a computer to communicate beyond its own circuitry. Ports on microcomputers include RF modulators to support a television signal, and Centronics and RS232C interfaces.

PRINTED CIRCUIT

A *printed circuit* is the board-mounted wiring pattern used for a particular electrical circuit. It is constructed by etching a circuit pattern onto a sheet of copper, which is then fixed onto a hard supporting surface such as fibreglass. Holes can then be drilled through the board into which solid state and other components can be soldered (on the front or back), either manually or mechanically, by dip soldering, to complete the circuit. Often circuits will be attached to both sides of the board with connections made between them. These are known as double-sided printed circuit boards (PCBs).

The invention of the PCB initialised the mass production of electronic devices. It is doubtful that we would have the inexpensive transistor radio, let alone the home microcomputer, without its emergence. However, development of PCB technology has not yet ceased. Modern techniques of etching printed circuits onto plastic have enabled several layers of circuit to be connected together. These methods have made possible the 'credit card' calculators that are now available.



CHRIS STEVENS

Miniature Art
The development of 'printed circuit' technology has enabled a far greater track density than would have been possible using older methods. Because printed circuits are designed as artwork and then photographed onto the board, it is possible to scale the image down to fit the area designated

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of the Commodore 64's memory map.

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
PNTR	00D3	211	Cursor Column on Current Line
QTSW	00D4	212	Flag: Editor in Quote Mode, $00 = NO
LNMX	00D5	213	Physical Screen Line Length
TBLX	00D6	214	Current Cursor Physical Line Number
	00D7	215	Temp Data Area
INSRT	00D8	216	Flag: Insert Mode, >0 = # INSTs
LDTB1	00D9–00F2	217–242	Screen Line Link Table / Editor Temps
USER	00F3–00F4	243–244	Pointer: Current Screen Color RAM loc.
KEYTAB	00F5–00F6	245–246	Vector: Keyboard Decode Table
RIBUF	00F7–00F8	247–248	RS-232 Input Buffer Pointer
ROBUF	00F9–00FA	249–250	RS-232 Output Buffer Pointer
FREKZP	00FB–00FE	251–254	Free 0-Page Space for User Programs
BASZPT	00FF	255	BASIC Temp Data Area
	0100–01FF	256–511	Micro-Processor System Stack Area
	0100–010A	256–266	Floating to String Work Area
BAD	0100–013E	256–318	Tape Input Error Log
BUF	0200–0258	512–600	System INPUT Buffer
LAT	0259–0262	601–610	KERNAL Table: Active Logical File No's.
FAT	0263–026C	611–620	KERNAL Table: Device No. for Each File
SAT	026D–0276	621–630	KERNAL Table: Second Address Each File
KEYD	0277–0280	631–640	Keyboard Buffer Queue (FIFO)



© 1983 COHEN, S.—ADVANCED TECHNOLOGY SYSTEMS