PCN micropaedia

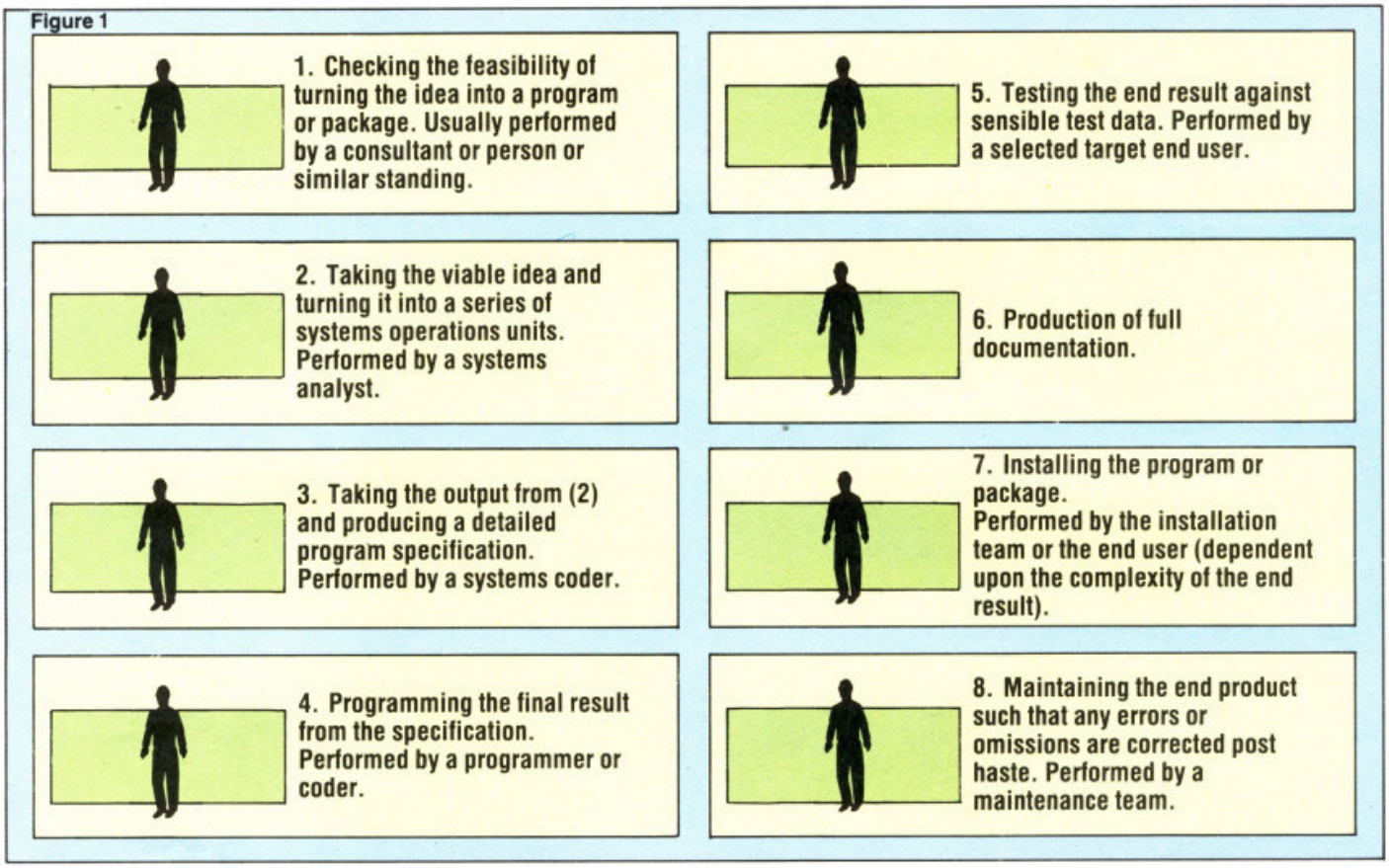
Vol 11 Part 1

EVERYTHING YOU WANTED TO KNOW ABOUT PROGRAMMING...

...but were afraid to ask

PULL OUT AND KEEP

Figure 1



TALKING TO YOUR MACHINE

Over the next eight weeks, this section of the programming course will be devoted to helping you put the questions that help you in developing good program design habits.

In this first part, for example, we show you how the simple process of getting up and going to school is actually a complicated process when you try and describe it in detail to a computer. There are many things the computer would need to know in order to recognise the idea of simply 'waking up' or 'taking a bath' — such as 'what is a bath', and 'what is a school'.

By the time you've finished this programming course you should be able to deal with more weighty questions than telling your computer how to take a bath. You should be able to approach the problem of, say, designing a Mastermind game and know that the first thing you have to decide is how to generate six random numbers, allow for the guesses of the player, check the guesses of the player, produce a message saying that the player has either failed or succeeded and then the appropriate action for each of those eventualities.

PROGRAMMING OUT PROBLEMS

If you have a problem, and that problem makes any logical sense, you should be able to program a computer to help you solve it. The problem can be everything from working out how to make a Space Invader appear on the screen of your micro to calculating the average monthly mileage of your car.

The method of solving the problem by computer is commonly known as a program. In the following eight weeks, we'll be looking at just what goes into developing workable programs. This study will come in three parts: Program Design — the business of how you go about putting a program together (the discussion of which begins further down this page); Program Structure — the best ways to actually order commands and instructions in a program; Program Tools — the aids which make design and structuring of a program easier.

A program, in its crudest sense, is a set of rules and guidelines to produce a solution to a problem. Commercial software companies, who produce programs for a living, often find that it helps to divide up the work when planning a program.

It is divided into clearly defined areas and appropriate staff are assigned to each part of the task in hand. But, what are those clearly defined areas, and who are the people responsible for them? (Figure 1).

Obviously, for those of us working at home on our small machines, we do not tend to have recourse to such a large team of people and, as such, we are expected to perform all the functions outlined above ourselves. For the purposes of this article we shall ignore functions 4 to 8 as it is assumed that you will be capable of these.

Let us look at the first three sections as mentioned above.

1. Feasibility and viability. Let us assume you have found or defined a problem, whether it be simple or complex, whether it be a game or a package. The first requirement is to look at the problem in a global sense. This means taking a step back from the absolute problem and breaking it into three distinct areas. What input is required, what output is required and how does the output relate to the input? You must remember that the most important person to take into consideration while developing any program is the end user. He or she will be the person spending most of their time and effort using the program, so make life easy for them.

A major consideration at this point is to decide whether or not the problem you have defined is capable of fitting the resources or equipment that you have access to. If not, then it is not viable, so give up or break the problem down even further — thereby opting for the package approach and repeat this procedure until each problem is

reduced to a manageable size.

2. Taking each problem in turn. It is now required to convert them into an overall picture of the final job, defining each major area of operation and their relationships to each other. This is needed so that the completed product remains consistent throughout.

3. The real work starts here! Normally, the functions of the system analyst and the systems coder are combined into one individual or team — purely for consistency and ease of communication. But it is still best to work on these functions as separate entities until you have gained enough experience in problem decoding and understanding. The requirement of the system coder is to take each major operating area and then translate them into an absolute detailed series of instructions for the programmer to program from.

In a software house, these instructions tend to take the form of a flowchart, definitions of the types of data to be input and also the formats of the output. Armed with such information, anyone with even the smallest amount of programming skill can churn out a program that will do the job. Variations in speed and efficiency are all down to individual skills of the programmer, but that is another topic altogether.

Let us look at a problem. It is something that we have all done, so we are all familiar with it. At the lowest level (ie. least detailed), it is as follows. (See Figure 2).

The options listed at the bottom of the box at right are only five of the possible decisions to be made following waking up. If your answers to the first four are 'yes' then you progress to the next action otherwise you may wish to go back to sleep, read a book, do some homework or a whole variety of other things. For any of these actions you have inserted a delay into the proceedings, therefore, you need to keep looking at the time (or use an alarm clock) to indicate when your next decision has to be made.

Each action must be uniquely defined, each decision must be identified and all possible answers must be catered for. Each result of an action or decision must be clearly labelled. Any calculations that have to be performed should be defined both in detail and also, should be wholly described so that errors do not occur. All of these things tend to be overlooked by the lone programmer, usually due to the fact that time and expedience is of the essence, and, anyway no one else is going to use the program are they? So what does it matter if a few errors slip through?

It does matter. This statement may seem somewhat contentious, but the reasoning behind it is that if you get into the habit of dismembering each problem as it arises then you will find that after a few statements, it starts to become the natural way of looking at problems.

And of course one side effect of doing things tidily is that you produce programs quicker and more efficiently whilst also allowing you the capability of being able to describe problems to others in a clear and concise fashion.

SOLVING IT STEP BY STEP

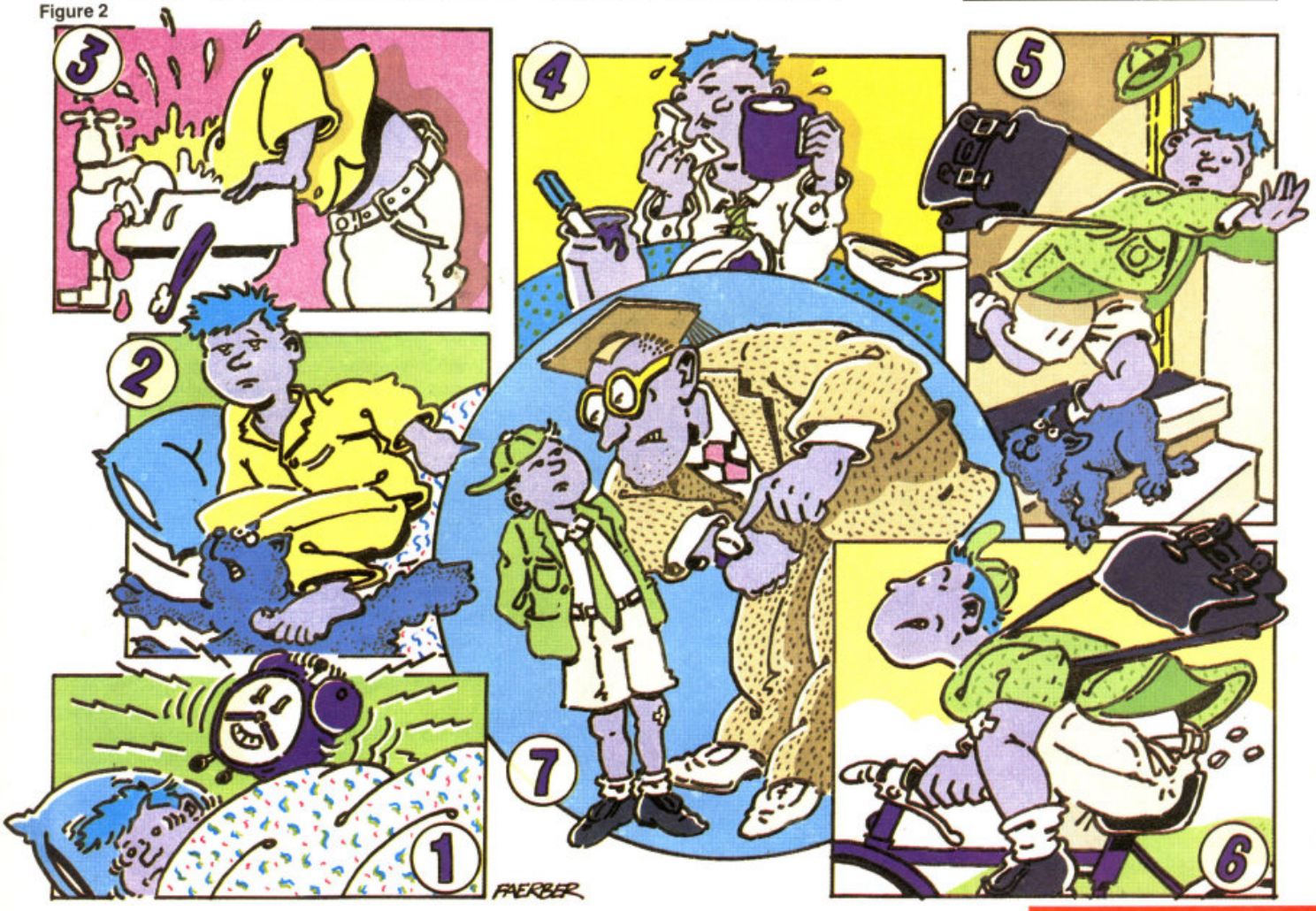
In the series of pictures below, you'll see the steps necesary to perform the action of going to school in the morning, and how you might describe this to a computer.

1. Wake up.
2. Get out of bed.
3. Bathe.
4. Have breakfast.
5. Leave home.
6. Travel to school (by any means).
7. Arrive at school.

Simple really isn't it?

Action 1 says 'wake up'. Is it really enough just to say that? Not really, in fact there are many actions and decisions to be made even for that one short statement:
Is it time to get up?
Is it a schoolday?
Are you well enough to go to school?
Are you going to go to school?
Have you overslept?
(For more see main story at right).

Figure 2



Introduction **– Just as most books begin with an introduction or preface, most programs will start by explaining in one or two Remark statements what the program is going to do.**

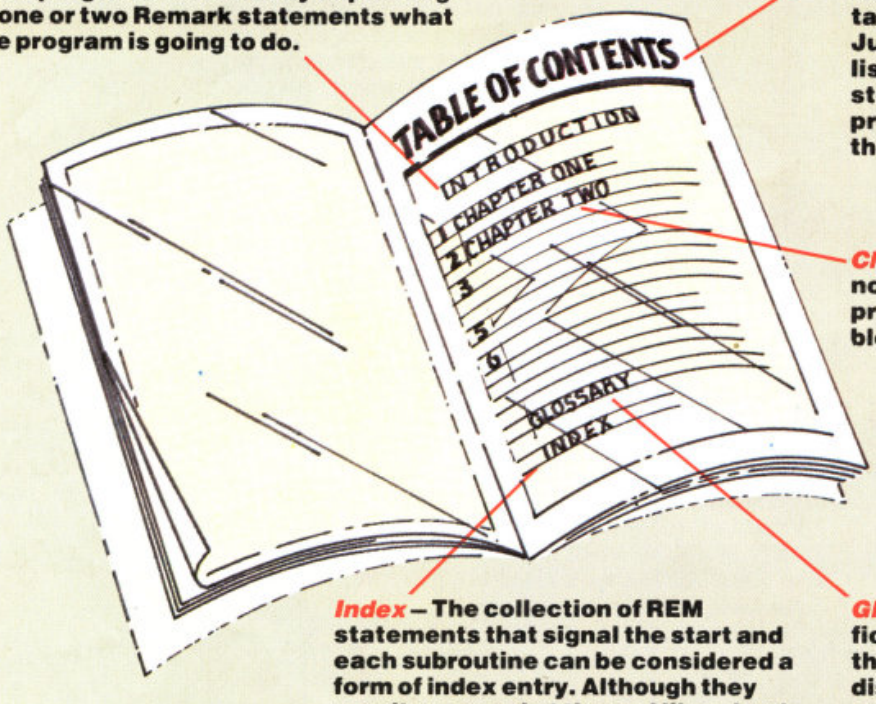


Table of Contents **– The beginning of a well-structured program should not look significantly different from the table of contents at the front of a book. Just as all the chapters in a book are listed at the beginning of a book, a structured program will have all its procedures or subroutines 'called' at the front end of the program.**

Chapters **– The chapters of a book are not unlike the subroutines or procedures that make up the building blocks of a computer program.**

Index **– The collection of REM statements that signal the start and each subroutine can be considered a form of index entry. Although they aren't arranged at the end like a book, these REM still perform the function of allowing you to find something quickly and easily.**

Glossary **– The glossary in many non-fiction or technical books carries out the job of defining the terms that are discussed in the book. Variable names are the 'terms' in a computer program and the section of a program that defines variables performs much the same function as a glossary.**

COURSE ART OF PROGRAMS

Structured programming means many things to many people, but all programmers would probably agree that it means at least an ordered approach to programming. In this section of our programming course, we will look in detail at program structure — starting with things that are common to all programs and progressing to the point where you'll feel confident enough to participate in the current debate regarding program structure and how it should evolve.

In the next couple of weeks, we'll be looking at how flowcharts, sketch code and other structuring aids can help you think about the best ways to structure your programs. You'll quickly see that the art of structuring a program comes down to not much more than making a series of yes or no decisions.

The 'Yes or No' philosophy of computing is otherwise known as Boolean logic, which we'll also examine in detail in later sections of this course.

STRUCTURE SURVEYED

If you own a micro, you will probably have tried writing programs yourself rather than just running the demonstration programs or commercial packages. You may have used the manual to familiarise yourself with the commands that your computer will recognise and respond to, like displaying a message on the screen, doing maths or using the sound and colour facilities. One thing you may not have done is to structure a program before you started typing it in.

Learning a programming language is similar to learning a spoken one. You can use your manual as a primer to find information about the words of the language and the rules of grammar and syntax with which to construct statements ('phrases' in my analogy) which your computer will understand and respond to correctly.

But your manual will not give you very much information about how to start writing a program — that is, how to use the basic tools to build a complex process.

The majority of manuals serve only as introductions to the trade of programming. Consequently, many enthusiasts are unable to find useful sources of ideas and instruction in the arcane art, and teach themselves as best they can. And many do a very good job of it.

But just as you wouldn't expect to be able to write a decent letter in a foreign language using only a phrase book for guidance, so most would-be programmers fall short in the structure of their programs. For this you need a guide to the art of programming and these are few and far between.

Whether you are a complete novice or reckon yourself to be a good programmer, you will find these articles useful for producing programs which do exactly what you want them to do. They should not require much debugging (in terms of both effort and time), you will be able to modify them quite easily, and they should possess that elusive quality — 'elegance'.

Getting started

There are a few major concepts which you should take pains to bear in mind while you are developing a program. The first and most important of these is that a program should be considered as a set of modules, and not just a sequence of numbered statements. The second is that it should have a structure, rather like a story has an introduction, characters, chapters, footnotes perhaps, and an ending. Your program

should have an initialisation sequence, recognisable variables, subroutines and REM statements.

The next point to bear in mind is that you must control exactly what goes on in your program. This ranges from preventing the interpreter butting in with messages like 'ERROR — REDO', or even prematurely ending your program with a 'division by zero' error, to carefully designing the screen format of information displays and so on.

It may seem like a chore and even something of an anachronism to have to do all this on paper first, after all, the micro is supposed to do away with such primitive note-making. But you will find that the dividends that result from time spent at the design-stage will more than amply repay the effort you have taken.

There are at least two stages to go through in the initial design of a program. The first of these is defining the overall 'problem space' — the 'plot'. Is the program to be a game, an educational tool or a 'systems' utility? If it is to be an educational aid, what age and ability range is it to be aimed at? If a game, is it to be text or graphics oriented, are instructions to appear on-screen or in a separate document? Such questions may seem obvious but it is surprising how many programmers do not seem to have thought of them until half-way through the program and have then had to make clumsy patches to cope with the change in direction.

Once the basic framework has been sorted out you should draw up a block diagram to show the gross internal structure of the program. To begin with you will have to have an 'initialisation' sequence. This is where you declare the variables, arrays and so on that you are going to use. Declaring a variable simply means writing such lines as '10 LET X = 25'. In many dialects of Basic, you don't actually have to do this. You can introduce a new variable half-way through a program without any problems. If the variable has not been given a value before, the interpreter will automatically give it the value zero. Most other languages force you to state all the variables (and in some cases constants) that you are going to use, even if they are to have an initial value of zero or, in the case of a string, null. It is good practice to decide on such things as variable names, initial values and so on and to state them in the first few lines of a program. To continue the literary analogy, the variables are the characters of the program.

If your characters aren't consistent throughout a novel, then there's a good chance it won't be believable to a reader. The same situation applies to computer programs: variables should maintain their integrity throughout and should be different enough not to be easily confused (see box, above right).

The sections or subroutines in a program can be likened to the chapters of a book. In a properly structured program, the first few lines of the program should not be too dissimilar from the table of contents in a book (see diagram).

In some versions of the Basic programming language, the building block subroutines of a program are known as procedures and can be written as self-contained units which can be called up from the beginning of a program.

This allows you to break down various routines into common and useful procedures, saving yourself much time and effort.

PROGRAMMING: A SENSE OF ORDER

There is nothing worse than being faced with one of your old programs and finding that you can't remember much about it. It is a curious fact that while writing a program you can become completely at one with the detail, to the extent that you know exactly what value a variable has at any given point and what it represents in terms of the internal micro-world of your program or the outside world of the screen, yet only a short time later are completely stumped on reading lines like:

```
100 IF PQ <> CV AND HG – 5 * (RG–2)=96 THEN GOTO 675.
```

This brings you to two points which are of such importance that they bear mention both here and later so that you don't forget them:

1 Make full use of REM statements — in complex routines nearly every line needs explanation.

2 Use variable names or labels that bear some relation to what they represent. While this is not all that easy using most home micros' implementations of Basic (because you are limited to the first two characters) such labels as Z7 or Y3$ are psychologically meaningless before they are used.

Be careful here as it can be easy to confuse labels that are too similar. For example, NR for the 'Number of Records' in a data-base file could be confused with RN meaning 'Record Number'. This is yet another argument for deciding on the exact 'names' before you write or code the proper program. Some languages allow you up to eight characters, which makes life easier. Some extensions of the Forth language allow you up to 32 characters, although I imagine that entering them gets more than a bit tedious.

The first few lines of your program might be represented in a block-diagram quite simply as 'DEFINE VARIABLES'.

This conceals a multitude of ideas at the next level of design, which is an intermediate step between a concept and its coding in the programming language. It might conceal such ideas as:

- Set BS$ as the ASCII code for BackSpace — CHR$(8)
- Define Number of Goes as 10

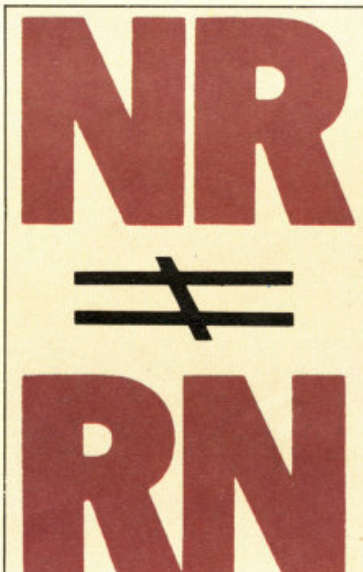
Set up an array 'L' to hold line numbers for PRINT

- AT commands

These ideas might later be encoded as:

```
10 LET BS$ = CHR$(8)
15 REM BACKSPACE
20 LET NG = 10
25 REM NUMBER OF GOES
30 DIM L(16)
35 REM USE 'I' AS A GENERAL PURPOSE
   LOOP COUNTER ANYWHERE
40 FOR I = 1 TO 16
50 L(I) = 32 * (I – 1)
60 NEXT
65 REM ********** 'PRINT AT L(X) . . .'
   PRINTS ON Xth LINE
```

Note that REM statements are put on separate lines.



It may seem obvious that RN doesn't equal NR, but when you're naming variables in a computer program the obvious has to be made more than obvious.

For instance, it may seem a good idea to call the variable that keeps track of the number of records in a file NR — and it could seem an equally good idea to call the identity number for a record as RN.

The two variables — RN and NR — are keeping track of two different things in the program, but they look so much alike that you could quite easily confuse them when trying to debug or list a program.

A better way to handle this problem would be to make sure variables names are easily recognisable as relating to the appropriate variable, and do not look or sound so much like one another that you're likely to confuse them.

So, in the case of our RN and NR example, you could rename them as RCN (record number) and NMR (number of records).

LANGUAGE AT YOUR COMMAND

The first two sections in this programming course show you how to go about designing and structuring programs, while this section helps you to become familiar with the tools to carry out those tasks.

In most cases, the tools will be computer languages which interpret the commands and ideas from 'keywords' specific to a language such as Basic into the series of 0s and is (known as binary numbers) that make up the 'machine code' language that's understood by your computer.

On the following three pages, are examples of some of the popular computing languages to give you an idea of how diverse they can be. Over the next eight weeks, we'll take a close look at many of the languages and give lots of program examples for you to try for yourself.

The most popular home computing language in Basic (Beginner's All-purpose Symbolic Instruction Code) and it will be well represented in the examples for you to type into your computer. Basic resides in most computers as information stored on a silicon chip built into the machine. When the computer is 'powered up', it immediately recognises the presence of Basic and puts a message on the screen telling you that you're operating in Basic.

PROCESSOR CONTROL

To perform any task a computer needs a sequence of instructions to follow.

The task itself must be broken down into smaller and smaller parts until a level is reached where the parts of the task are equivalent to the program words available on the computer. The breaking down of the problem into simple tasks that the computer can act upon is one of the major jobs of anyone who wants to program a computer.

Fortunately, the computer itself is able to help a great deal. It does this by starting from the opposite end of the problem from the programmer. The computer system is able to execute preset sequences of instructions that allow it to perform fairly complex tasks. For instance, printing a character to the VDU is a complex procedure but since it is used in most programs there is always a print or write command in every high-level language.

Long series of short commands can be sequenced to give the basis of a programming language. The computer and the programmer therefore meet at a mid-point, which should be optimised to save the time taken to write the program, the speed with which the program runs, and the space that the program occupies in the memory. This optimisation is done by choosing the appropriate programming language for the job, *ie* the correct software tool.

A programmer should not stick to one language. He should choose the most appropriate one for the job. As will be seen, some languages are more suited to certain kinds of work than others.

All computer systems are, at their lowest level, a set of switches, logic gates and memory. These all need to be controlled in the correct sequences to enable operations to be performed. For instance, to add a number into a register, various switches must be set, gates opened, and flip-flops flipped to allow the operation.

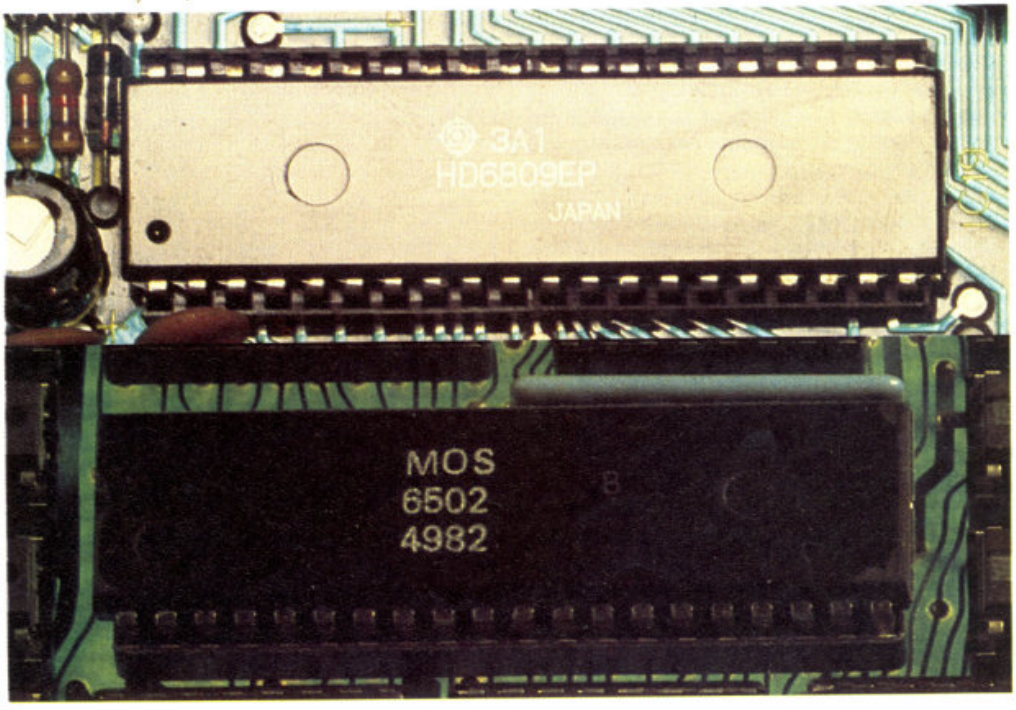
The basic procedures of a processor are controlled by a very low level language known as micro-programming. The micro-program instructions are as basic as: open this gate, clock that counter, etc. These instructions are then grouped together by the manufacturer into machine code operations.

The machine code language is a little higher up on the scale as it allows operations such as addition to take place within only one instruction.

The way in which micro-program instructions are grouped together is what makes a Z80 processor (used in the Sinclair Spectrum computer) different from a 6502 processor (used in the BBC Micro). The more complex processors, such as the 68000 and the Z8000, allow even higher level operations to be performed at machine-code level. In these larger processors, it is not unusual to have machine-code instructions for multiplication, division and even string-handling.

From the machine-code level, it is up to you to specify how the system operates. Programs can either be written directly in machine code, or in one of the many standard high-level languages such as Forth, Basic, Pascal or Fortran.

Which one is selected depends on the task to be performed and the various other constraints discussed in the introduction.



The process is the workhorse of the computer and more or less defines the identity of the machine. Popular computers generally use 8-bit or 16-bit processors, the first type being the mainstay of home computers and the latter being more common in expensive business machines. The pictures show the 6502 and the 6809 processors — both 8-bit type — used in popular micros. The 6502 is the processor in the Apple, Atari and BBC micros and many other machines. The 6809 is less widely used but makes an appearance in the Dragon and Tandy Color Computer. In a Basic system, the processor takes information from the Basic interpreter and then acts on it.

Example Assembler program for BBC micro assembler.

```
10 DIM PROG 200
20 DIM BUFF 20
30 DIM SPTR 1
40 DIM EPTR 1
50 DIM PLEN 1
60 DIM CTR 1
70 INPUT"ENTER WORD " A$
80 $BUFF=A$
90 ?PLEN=LEN(A$)
100 FOR T%=0 TO 3 STEP 3
110 [ OPT T%
120.PROG
130 LDA PLEN ;GET LENGTH
140 LSR A ;DIVIDE BY TWO
150 STA CTR;STORE IN CTR
160 LDA£0
170 STA SPTR ;ZERO SPTR
180 LDA PLEN
190 STA EPTR
    ;PUT LENGTH IN EPTR
195 DEC EPTR
200 .CPAR
210 LDX SPTR
220 LDA BUFF,X ;GET CHAR
230 LDX EPTR
240 CMP BUFF,X ;CMPAR
250 BEQ NEQ  ;IF NOT THE
    SAME THEN END
260 LDA CTR
270 BEQ IEQ    ;IT IS A
    PALINDROME AT IER
280 DEC CTR
290 DEC EPTR
300 INC SPTR
310 JMP CPAR
320 .NEQ
330 LDA £ASC("N")
340 JSR &FFEE    ;OUTPUT
    CHAR TO SCREEN FOR NO
350 RTS
360 .IEQ   ;IS EQUAL
370 LDA £ASC("Y")
380 JSR &FFEE ;OUTPUT
    CHAR TO SCREEN FOR YES
390 RTS
400 ]
410 NEXT T%
420 CALL PROG
430 END
```

On the first electronic computers, all programming was done in machine code, all of which was worked out by hand. Fortunately, it was soon realised that a computer was capable of doing much more of the legwork involved in putting together a machine code program. From this, the idea of assemblers was born.

Assemblers use mnemonics, easy-to-remember names, for each of the machine code instructions available on a particular processor. They are also used to calculate the number of bytes separating points in the program that would otherwise have to be worked out by hand.

Places in the program are normally marked by labels. These are words that tell the assembler to mark places in the program to jump to or store information. Assemblers are also used to perform error-checking on the mnemonics to aid the programmer in long pieces of code.

Assemblers allow easier contact to be established between the programmer and the machine. This is mainly because most people find it easier to remember and make patterns from words than they do with numbers.

The main advantages of assembly languages are:-

- The program generally runs fast.
- The program is compact compared with high level languages.
- It is very efficient for the computer to run assembled programs.

The main disadvantages are:-

- Writing complex programs can be difficult and time-consuming.
- Assembly language programs are generally specific to one processor. They will not normally work on other processors and so are not portable.
- For complex programs they are fairly inefficient from a programmer's point of view, because of the above disadvantages.

An improvement on normal assemblers is the macro-assembler.

```
FORTH EXAMPLE PROGRAM.
(Written on Acornsoft
FORTH but should work
on most forths)

( PALINDROME )

: EQUAL C@ SWAP C@ = ;

VARIABLE TPOINT

VARIABLE BPOINT

: PALIN ( ADDR LEN )

2DUP + 1- TPOINT !

2/ CTR ! BPOINT !
      BEGIN

BPOINT @ TPOINT @ EQUAL

CTR @ 0= NOT

AND

WHILE

1 BPOINT +! -1 TPOINT
+! -1 CTR +!

REPEAT

CTR @ 0= IF

." IS " ELSE .

" IS NOT " THEN

." A PALINDROME "
```

Forth is a high-level language that can masquerade as a low-level language if necessary. It is a rigorously structured language, that is, structure is more or less forced upon the programmer by the language.

When writing a program in Forth, it is necessary to break it down into more and more detailed parts. These small parts are defined from system words, words already defined in the system. The definitions are used in the definition of more definitions and so until the program is complete.

The program is run by executing one defined word which executes the words that were used to define it, then these words are executed in the same way and the whole program is executed. The method of calling predefined subroutines is what makes Forth so fast and so compact.

Forth has difficulties with handling strings of characters. See the example given, which works out whether the string given to it is a palindrome, then compare this with the Basic and machine code versions of the same program. There is quite a difference.

Forth's main strength is its ability to access memory locations quickly and easily, making it the ideal language for interfacing a computer to the real world.

Example BASIC program.

```
10 REM PALINDROME PROVER

20 INPUT "ENTER WORD
   " P$

30 FOR I=1 TO INT
   (LEN(P$)/2)

40 IF   MID$(P$,I,1)
   <>MID$(P$,LEN(P$)
   -I+1,1)     THEN
   I=INT(LEN(P$)/2):NEXT
   :PRINT"NO":END


50 NEXT

60 PRINT"YES"

70 END
```

One thing to look for in a language is structure. If the language contains the facilities to cope with procedures and functions then for implementing large and complex programs it can be useful.

These two structures allow the task to be broken down into blocks that are smaller and thus easier to deal with. The small blocks, once they work, can be used to execute larger blocks. This process continues until the whole program can be run.

The *lingua franca* of microcomputers, Basic, does not generally provide for a great deal of structuring, although some attempts have been made to introduce more structure into the language, BBC Basic being the most famous (or infamous).

The lack of forced structuring makes many Basic programs a nightmare to understand, sometimes even by the original programmer.

```
PASCAL PALINDROME
TESTER

PROGRAM PALINDROME
(INPUT , OUTPUT);

TYPE
STRING=PACKED ARRAY
[1..30] OF CHAR;

VAR
COUNT,I,A        :INTEGER;
WORD             : STRING;

BEGIN
WRITELN(' ENTER
SUSPECTED PALINDROME ');

COUNT:=1;

REPEAT

READ   ( WORD[COUNT]);

COUNT:=COUNT+1;

UNTIL WORD[COUNT-1]=13;

I:=0;

A:=COUNT;

REPEAT
        A:=A-1;

        I:=I+1;

UNTIL (NOT (WORD[I]) =
WORD[A])) OR A=0;
IF A=0 THEN WRITELN
('THIS IS A
PALINDROME');
END
```

Pascal was originally written as a teaching language but, because of its flexibility and logical structure, it is now widely used as a commercial programming language.

It is a partially compiled language and its main programming strengths lie in its structuring and data typing, laying data out in ways that make it easily accessible. As a tool, it is ideal for most types of number-crunching and data-processing. The main disadvantage of Pascal is that it takes a lot of memory to implement a system, and every time an alteration has to be made to the program the whole thing has to be recompiled.

As a tool it is not much good for interfacing to the real world although, as with most languages, almost anything can be done if enough time and effort is devoted to it.

```
LISP PALINDROME PROVER
(DEFUN REVERSE   (X  (W))

(LOOP
(WHILE X W)
(SETQ W  (CONS
(CAR X)   W))
(SETQ X  (CDR X))))
(DEFUN EQUAL  (A B)
(COND
((NULL A)  (NOT B))
((ATOM A)  (EQ A B))
((ATOM B)  F)            (T
(AND
(EQ  (CAR A)(CAR B))
(EQUAL   (CDR A)
(CDR B))))))
(DEFUN PALINP  (L)
(EQUAL
(EXPLODE L)  (REVERSE
(EXPLODE L))))
```

The language Lisp has been around for a long time and is being used in artificial intelligence research. It is an extremely flexible language with which almost any task can be performed. It is ideal for writing database systems, as its data handling facilities are excellent. It is also very good for writing compilers and interpreters.

Lisp's main disadvantage in its compiled form is that it is fairly slow but it is interactive, so programming is easy. As a software tool this language tends to be a bit neglected — it can seem somewhat complicated at first sight.

It is obvious that there are many programming languages available — so many, in fact, that there is not enough room to cover them all. From those discussed, it is easy to see what advantages and disadvantages make a language suitable or unsuitable for doing a particular job. The chances are that you'll never find the ideal language for your program.

The answer is either to make do with whatever languages are around or write your own language to do the job perfectly, possibly using one of the other high-level languages to write it with.

Macro assemblers allow a programmer to set up predefined routines which perform specific operations.

Here is a simple example:

```
MACRO.POP  %R1,%R2
        PLA
        STA  %R1
        PLA
        STA  %R2
EMACRO.
```

This routine allows two bytes to be popped off the processor's stack, with one call. When assembled, the whole section of code is placed into the program. This makes assemblers almost high-level languages. In fact, the original Fortran was no more than a souped-up macro assembler with the predefined 'macros' making up the language.

As macro assemblers became more complex, they evolved into compilers. Compilers are similar to assemblers in that they convert a set of predefined instructions into machine code. They are also capable of performing error-checking and program optimisation.

Once a set of instructions has been compiled, the compiled code can be run as a program.

An alternative to compiling the instructions is to interpret them. With an interpreter, the set of source instructions or program steps is taken one at a time and executed. Because of this step-by-step nature, most interpreted languages such as Lisp or Basic run more slowly than their compiled equivalents.

The advantage of an interpreted language over a compiled one is that it is interactive. This means that it does not need to be recompiled every time a change is made to the program. This is the main reason why most Basics are interpreted. Any alterations to the program can be made and the program can immediately re-run, without all that tedious fiddling with compiler space.

Present-day computers are able to support many different languages, some of which are compiled (Pascal, Fortran), some of which are interpreted (Basic, Lisp — for small applications), and some of which are both (Forth).

Contributors: Nigel Cross, Kenn Garroch, Bryan Skinner and Geof Wheelwright
Design: Nigel Wingrove
Illustrations and diagrams: JOHN HALLET AND KEVIN FAERBER

NEXT WEEK

In Part 2 of Everything You Wanted to Know About Programming we look at grammar and fluency in program design (in which you play the part of a computer-age Eliza Doolittle), introduce the concept of flow-charting with examples of how they help in structuring your programs, and take a first look at one of the most popular programming tools — the Basic language.

And we take each of those subjects a bit further in the following weeks as the programming course covers more complex topics in the quest to give you a complete view of the programmer's art.

SUBSCRIPTION ORDER CARD

I would like to subscribe to Personal Computer News. **VOL 1 NO. 30**

☐ 1 year (51 issues) UK £22.50
☐ 1 year rest of the world (51 issues) £35.00
☐ I enclose my cheque made payable to Personal Computer News
☐ Please charge my Access/Visa/Diners/American Express card (delete where not applicable)

Account No. _____

Signed _____

What type of Micro do you use? _____

Do you use it for home/work/education? _____

Name:
Address:

Please use block capitals

Please send me my free PCN T-shirt.
I am … Small size (36″) … Medium size (38″)…. Large size (40″)

NEWSAGENT ORDER CARD

If you would like to make sure your newsagent keeps you a copy of Personal Computer News every week, then just complete this card and pass it to your local newsagent today.

Please reserve me a weekly copy of Personal Computer News until further notice.

☐ I'll collect it

☐ Please deliver it to the address below.

Name _____

Address _____

PROGRAM TOOLS

```
PASCAL PALINDROME
TESTER

PROGRAM PALINDROME
(INPUT , OUTPUT);

TYPE
STRING=PACKED ARRAY
[1..30] OF CHAR;

VAR
COUNT,I,A        :INTEGER;
WORD             : STRING;

BEGIN
WRITELN(' ENTER
SUSPECTED PALINDROME ');

COUNT:=1;

REPEAT

READ   ( WORD[COUNT]);

COUNT:=COUNT+1;

UNTIL WORD[COUNT-1]=13;

I:=0;

A:=COUNT;

REPEAT
        A:=A-1;

        I:=I+1;

UNTIL (NOT (WORD[I]) =
WORD[A])) OR A=0;

IF A=0 THEN WRITELN

('THIS IS A

PALINDROME');

END
```

Pascal was originally written as a teaching language but, because of its flexibility and logical structure, it is now widely used as a commercial programming language.

It is a partially compiled language and its main programming strengths lie in its structuring and data typing, laying data out in ways that make it easily accessible. As a tool, it is ideal for most types of number-crunching and data-processing. The main disadvantage of Pascal is that it takes a lot of memory to implement a system, and every time an alteration has to be made to the program the whole thing has to be recompiled.

As a tool it is not much good for interfacing to the real world although, as with most languages, almost anything can be done if enough time and effort is devoted to it.

It is obvious that there are many programming languages available — so many, in fact, that there is not enough room to cover them all. From those discussed, it is easy to see what advantages and disadvantages make a language suitable or unsuitable for doing a particular job. The chances are that you'll never find the ideal language for your program.

The answer is either to make do with whatever languages are around or write your own language to do the job perfectly, possibly using one of the other high-level languages to write it with.

interactive. This means that it does not need to be recompiled every time a change is made to the program. This is the main reason why most Basics are interpreted. Any alterations to the program can be made and the program can immediately re-run, without all that tedious fiddling with compiler space.

Present-day computers are able to support many different languages, some of which are compiled (Pascal, Fortran), some of which are interpreted (Basic, Lisp — for small applications), and some of which are both (Forth).

Contributors: Nigel Cross, Kenn Garroch, Bryan Skinner and Geof Wheelwright
Design: Nigel Wingrove
Illustrations and diagrams: JOHN HALLET AND KEVIN FAERBER

NEXT WEEK

In Part 2 of Everything You Wanted to Know About Programming we look at grammar and fluency in program design (in which you play the part of a computer-age Eliza Doolittle), introduce the concept of flow-charting with examples of how they help in structuring your programs, and take a first look at one of the most popular programming tools — the Basic language.

And we take each of those subjects a bit further in the following weeks as the programming course covers more complex topics in the quest to give you a complete view of the programmer's art.

G
D
BOUT

PROGR MMING...

...but were afraid to ask

PULL OUT AND KEEP

R THE BBC MICRO ■ SI
PAN AND PERS

60 PROC

(LESS THAN THE

new force in computer publishing. You can be sure that these new books which introduce a new range of paperbacks are the best – they're published by Britain's bestselling paperback publisher and the country's most successful and authoritative computer magazine – Pan and Personal Computer News.

The most successful software writers have been commissioned to write a completely new collection of programs for each of the most popular and fast-selling microcomputers. **Robert Erskine** is a programmer whose software has topped bestseller lists. **Humphrey Walwyn** is a broadcaster, magazine

FREE T-SHIRT!

When you subscribe to the No. 1 Weekly Micro Magazine

Subscribe to Personal Computer News today and we'll send you a free T-shirt! Ideal kit to wear while trying out PCN Programs on our special program cards. Each T-shirt has the name of your favourite magazine emblazoned on the front and back.
All you have to do is fill in the subscription card in this issue and send it with your remittance to Personal Computer News. In return, we'll send you a free T-shirt and a copy of Personal Computer News each week!

SUBSCRIBE TODAY!

GET MORE OUT OF YOUR MICRO WITH...

60 PROGRAMS

POST NOW, NO STAMP NEEDED To Pan Books Ltd., FREEPOST, P.O. Box 109, 14-26 Baker Street, High Wycombe, Bucks HP11 2TD.
YES Please send me the following **60 PROGRAMS**... paperbacks at £4.95 each plus 35p for the first book ordered plus 15p for each additional book to a maximum charge of £1.25 to cover postage and packing.

Name (Mr/Mrs/Miss/Ms) _____ ☐ BBC Micro ☐ Sinclair ZX Spectrum

address _____ ☐ Oric 1 ☐ Dragon 32

_____ post code _____

I enclose my cheque/postal order for £_____ payable to Pan Books Ltd *or* debit my Access/Barclaycard/Visa/Trustcard

VISA no. _____

Signature _____

Allow up to 15 days for delivery. This offer available within UK only.

Pan Books Ltd Registered in England Registration No. 389591

Pan PERSONAL COMPUTER COMPUTER NEWS LIBRARY PCN