This book tells you how to use the Sinclair ZX80 microcomputer. Whether you have one already or are intending to buy one, you will find here a clear down to earth explanation of how to use and how to get the best out of the ZX80.

Robin Norman takes the mystery out of bits and bytes, binary and hexadecimal, but best of all, he tells you exactly what to do to get your ZX80 working for you. And when you have got to know the workings of the machine you can try some of the specially written programs (unavailable anywhere else).

ISBN 0 408 01101 7

N

Learning BASIC with your sinclair ZX80

Learning BASIC with your sinclair ZX80

Robin Norman

Learning BASIC with your Sinclair ZX80

Robin Norman

Newnes Technical Books

Newnes Technical Books
is an imprint of the Butterworth Group
which has principal offices in
London, Sydney, Toronto, Wellington, Durban and Boston

First published 1981

British Library Cataloguing in Publication Data

Norman, Robin
Learning BASIC with your ZX80.
1. Sinclair ZX80 (Computers) — Programming
2. Basic (Computer program language)
I. Title
001.64'24 QA76.8.S62/

ISBN 0-408-01101-7

Typeset by Tunbridge Wells Typesetting Services
Printed in England by Butler & Tanner Ltd., Frome and London

Preface

So much has been written in the last two years about the wonders of the silicon chip, that I hesitate to join the chorus! I learnt to write BASIC programs in the late 1960s, using a noisy teletype terminal connected via a GPO line to a large central computer which I never saw. Now, a microcomputer of comparable power is quiet, sits on a small desk and can be bought for the equivalent of a few days rental of the old terminal. After a holiday from computers, mainly spent teaching in a Middle School, I suddenly find that I can afford my own microcomputer, and I am relieved to find that the BASIC programming language is very little changed. I am back in business!

So what is the point of this book? Well it just happened that Newnes and I both came to the conclusion that microcomputers like the Sinclair ZX80 were very suitable for beginners to computing, but that the manuals supplied with them seemed to be intended for more advanced workers. In writing this book I have made these three assumptions about the reader (in each one for 'he' read 'she' if you prefer!)

1 He is a newcomer to computer programming (naturally, depending on experience, he can skip early sections of the book).
2 He has one particular microcomputer, the Sinclair ZX80, switched on, in front of him.
3 He wants to learn all the instructions available in ZX80 BASIC, using a structured course with a steadily increasing tempo.

Having worked through the book ('read' is not the right word), the reader should be able to make full use of his ZX80 and, later on, transfer easily to more sophisticated microcomputers.

I can't avoid a few acknowledgements. To Betty Clare, who typed tricky manuscripts calmly and efficiently. To Peter Chapman,

whose ideas for programs were often novel and usually translatable into BASIC. To my family, who learnt the hard way that computing can be addictive, combining the joys of puzzle solving with delusions of power. And finally to Clive Sinclair who lent one of his little machines. Having been reared on his programmable calculators, I was expecting something good and I was not disappointed. ZX80 has its limitations, that's economics for you, but it is very good value and a pleasant way to start computing. So, I hope you find this book helpful. Happy programming to you all!

R.N.

Contents

List of Programs

1

What is a Computer?

Not long ago I put this question to a class of 12-year-old children, and got quite a variety of answers. The replies nearest to the truth were, 'It's a machine that works for you' and, 'It does things for you.' Yes, friends, the computer is another labour saving tool, like an electric sewing machine or an automatic lathe. If you feed steel rods into the lathe, you can end up with lots of wood screws — very useful products. If you feed words and numbers into the computer, you can also get a useful product in the form of more words and numbers.

Can these machines work on their own? Well, yes and no. Imagine what would happen if you took a lathe straight from the factory and fed in steel rods. Probably all you would get would be a lot of pretty but useless steel spirals. Somebody has to fix all the settings on the lathe so that it can make the tapered shape, cut the screw thread and mill out the slot in the head to make a finished screw. In just the same way, someone has to tell the computer what data (words and numbers) to expect, what to do with this data, and what results to print. The instructions for a computer are what we call a *program,* and programs are what this book is all about!

It's the custom to spell computer programs with one 'm' since it helps to distinguish them from all the other meanings of the word 'programme'. You've just joined the ranks of the computer programmers and you must learn some new words, as well as some new meanings for old words. I'll try to keep these to a minimum, and I'll help you by listing them all in a glossary at the end of this book.

Here are two more jargon words that you must have come across. *Hardware* means all the physical parts of the computer with its bits and pieces — your Sinclair ZX80, the TV screen and the cassette recorder.

Software means all the programs and instruction books needed to make the computer work — your ZX80 operating manual, this book, the permanent programs put into the ZX80 by its designers, and the programs that you write.

Over the years, electronic ingenuity has been rapidly reducing the size and the cost of computer hardware, and the little ZX80 was the first to break the £100 barrier. Software costs have been falling at a much slower rate — if you want to know why, see how long it takes you to write even a fairly simple program. All the same, mass marketing should mean cheaper good software in the future.

2

Talking to Computers

With our human complement of ten fingers, we have got used to decimal numbers made up of the ten digits from 0 to 9. Computers, on the other hand, work with *binary numbers*. A binary digit, *bit* for short, can only have the values 0 or 1, and the computer is just about bright enough to tell the difference between these two!

One can write programs in binary numbers (in the early days of computers this was the only way), but humans find binary numbers clumsy to handle and hard to recognise. A better way is to use a *low level language*, and to write programs in machine code, a series of letters and decimal numbers. Machine code programs are fast to run, and are economical in the use of computer memory, but they are no way for beginners to learn programming. Most people converse with computers in a *high level language*, which uses decimal numbers and sets of recognisable English words. Some common high level languages are:

FORTRAN (FORmula TRANslation, mainly for science and engineering)
COBOL (COmmercial Business Oriented Language)
BASIC (Beginners All-purpose Symbolic Instruction Code)

The computer cannot understand these high level languages on its own, and so programs have to be written to translate them via machine code into binary numbers.

Computer Memories

The memories in a computer consist of a large number of 'boxes' or 'pigeon-holes', each containing an 8-bit binary number (called a *byte*). The size of computer memories is usually specified in terms of

'K', where 1K is a memory with a capacity of 1024 bytes. There are two types of memory in microcomputers such as the ZX80.
Read only memory (ROM) contains the software needed to run the computer and to translate the BASIC instructions into binary code. ROM is permanent, and so is not lost when the computer is switched off. The ZX80 uses BASIC in 4K of ROM.
Random access memory (RAM), also known as user memory, contains all the data and programs which you put in. It is not permanent, and if you switch off for a fraction of a second, the RAM contents are lost. Your ZX80 has 1K of RAM available, although a further 16K is available in a plug-in module (see Appendix 5).

Input and Output

We need to be able to put data and instructions into the computer memory and the ZX80 provides us with a reduced size version of the standard typewriter keyboard for this purpose. We also have to provide the means for the ZX80 to show its results, and to keep a check on our input — a standard commercial TV set is used for this.

We've seen that the RAM contents are lost when the ZX80 is switched off — this may be a precious program which has taken hours to write! We therefore need *back-up storage* in which to keep our programs and data permanently. A sure way is to write them into a notebook or file, but this still means a lot of tedious typing when we want to use them again. The ZX80 system uses a standard domestic tape recorder, and programs can be saved on a cassette, kept indefinitely and loaded back into the ZX80 when required.

Looking Into the Future

What about the long term developments — the crystal ball stuff? Well, I suppose it has all been said before, but here is a list of the features I should like on my future personal computer — maybe some of them will happen:

(1) Some agreement on standards, so that computers can communicate with each other more easily.
(2) Communication with the computer by voice — both for input and output.
(3) Cheap printed output — a domestic electric typewriter connected to the computer — word processing in the home.
(4) Cheap unlimited permanent memory for back-up storage.
(5) High definition output in colour, comparable with TV standards.

(6) A large range of cheap software — programs for business, home, learning and leisure — in simple plug-in form.
(7) Connection to large central computers, probably via the TV network, to give access to virtually unlimited information on any chosen subject.

3

Programming in BASIC

BASIC is one of the most widely used high level languages, especially for the present generation of microcomputers. Many different versions of BASIC exist, in rather the same way as there are many different dialects of English. However, do not lose heart! All versions of BASIC are easily recognisable as coming from the same original source (BASIC was developed at Dartmouth College, New Hampshire, USA), and when you have learnt one form of BASIC you can quickly transfer to another form on another computer. Sinclair ZX80 BASIC in 4K ROM is one of the simpler versions, and therefore it is ideal for beginners. On the other hand, it does have valuable features which are not found in many other BASIC dialects.

The First Computer Program?

Let's take our first look at programming with a light-hearted example — later on we'll write a real BASIC program for a mathematical model of it. *The Sorcerer's Apprentice* was originally written by Goethe, and most people know the musical version by Dukas. When I first came across it, Mickey Mouse was the apprentice in Walt Disney's *Fantasia*. The sorcerer goes out for the morning, leaving Mickey with the boring job of filling a great water tank with water from the well. Mickey is an enterprising lad, and he decides to program one of the kitchen brooms to do the job for him, while he has a crafty snooze.

Now, in writing a computer program, it's very important to get the instructions in the right order. With this in mind, every instruction is given a number, so that Mickey's first attempt at a program could have been like this:

1 Pick up bucket and go to well
2 Fill bucket with water
3 Carry bucket carefully to water tank (no spilling please)
4 Empty bucket into water tank

Clever stuff so far, but only one single bucketful of water has been shifted. Mickey could have repeated the same instructions over and and over again, numbering them 5, 6, 7, 8 and 9, 10, 11, 12 and so on. But no! He has read Chapter 9 in the spell book, and all he has to do is to add a fifth instruction:

5 GO TO 1

and now he has made a *program loop*. The broom follows the program exactly, and goes happily backwards and forwards filling and emptying buckets, while Mickey dozes off. . . .

* * * * *

. . . until he wakes with a start some time later to find his feet sloshing about in water. You've guessed it! He forgot to tell the broom when to stop! Panic stations — he chops the broom into sixteen bits, but each of these gets up, picks up a bucket, and carries on with the good work. Luckily the sorcerer arrives home in the nick of time. Being a skilled programmer of brooms, he knows that every loop must include a 'get out' test, or else it will go on for ever. We call this vital step a *conditional jump*, because it always contains the magic word 'IF'.

Now we add an IF statement, and renumber a little, and the final program looks like this:

1 Pick up bucket and go to well
2 Fill bucket with water
3 Carry bucket carefully to water tank
4 Empty bucket into water tank
5 IF water tank is not full THEN GO TO 1
6 Report 'Tank full'
7 Stop

Note that the IF statement must be *inside* the loop, so that every time it goes round the loop, the broom has to test whether the tank is full, and take action accordingly.

Childish stuff I know, but at least it raised four points which are going to be very important when we come to write real programs for the ZX80.

(1) A BASIC computer program is made up of a *series of instructions*.

(2) The instructions are all *numbered* so that the computer can carry them out in the order it is told to.
(3) You can make the computer do part of a program over and over again by using a GO TO instruction. We call this a *loop*.
(4) A loop must contain a *conditional jump*, which will stop the computer or send it out of the loop when the condition is fulfilled. The magic word is 'IF'!

4

Let's Get It Switched On!

Enough of theory! You'll need to read most of the rest of this book beside your switched-on Sinclair ZX80.

Connecting up the Hardware

To get going we need to connect up just three pieces of hardware.

(1) The ZX80 microcomputer (of course)
(2) The power supply for the ZX80

Sinclair specify a supply of 9 V d.c. at 600 mA unregulated. The 9 V output must terminate in a 3.5 mm jack plug, the tip of which must be positive. Sinclair's own unit is in a convenient box with pins which plug directly into a standard 13 A socket. It is worth making the point that the usual calculator power units will not provide enough amps, and must not be used.

(3) Any domestic UHF television set

A black and white set is preferable. The TV set enables you to read and check your programs as you type them in, and it is the means by which the ZX80 displays its output.

Connect up the UHF aerial socket of your TV set to the large central socket in the back of the ZX80 using the video cable supplied by Sinclair. Switch on the TV, let it warm up, tune to an unused UHF channel, and turn the sound right down.

Connect up your ZX80 to the power supply by plugging the 3.5 mm jack plug into the left hand one of the three sockets (it's farthest in from the edge) at the back of the ZX80 and switch on the power supply.

Take a deep breath — nothing has happened! You have to tune the TV set to the ZX80 frequency first. Some sets are tuned by

turning the push button on the channel selector, while others have a little door which opens to reveal a set of tuning knobs. When you have got the tuning right (it's about channel 36), the screen suddenly clears, and you see a little black square with a white 'K' in it at the bottom left of the screen — this is called the cursor. The cursor is there to show you where your next typed letter will appear, and as you type out your program lines, the cursor will move steadily ahead of your typing.

I know you are desperate to get started, but first we need to have a quick look at the ZX80 keyboard.

The ZX80 Keyboard

There are five different sets of characters obtainable from the keyboard, plus a few odds and ends.

(1) The set of numbers

The numbers 1 to 9 and Ø are obtainable at any time, regardless of the state of the cursor, by pressing the correct keys along the top of the keyboard. You will have to be very careful not to mix up the number 1 with the letter I, although the difference is fairly obvious on the screen. More subtle is the difference between the letter O (a square on the screen) and the number 0 (a hexagon on the screen). I am following the convention in the Sinclair ZX80 manual of printing the number as Ø, partly for clarity, and to agree with other computer manuals.

(2) The set of 'tokens'

These are the light yellow words and symbols above the number line. They are obtained by holding down SHIFT and pressing the appropriate number key at the same time. Their use will be explained later.

(3) The set of keywords

These keywords are printed in white above most of the letter keys, and are one of the great joys of the ZX80 system. Suppose you want the keyword *PRINT*. Well, the cursor must be reading [K] for keyword if a keyword is needed next — the ZX80 takes care of that. So, if you press the key 'O' which has the keyword *PRINT* above it, you will automatically type the whole word 'PRINT' on the screen, neatly spaced, and all with just one stroke of the key. At the same time the cursor will change from [K] to [L] (for letter), since ZX80 BASIC calls for keywords one at a time. It will change back to [K] whenever another keyword is required. Quite a few of the keywords are placed above their initial letters, which helps you to find them. The use of all the keywords will be explained as they are needed.

(4) The set of letters

The letters A to Z (and also full stop) are obtained by pressing the appropriate keys whenever the cursor shows [L] — the ZX80 system will make sure that this happens at the right time. Only capital letters are available in ZX80 BASIC.

(5) The set of upper case symbols

These are a mixture of punctuation marks, maths symbols, graphics blocks, and so on, and they are shown at the top right of each letter key. They are obtained by holding down the SHIFT and pressing the appropriate letter key at the same time, and they are not affected by the state of the cursor.

(6) Remaining odds and ends

SHIFT is at the bottom left; its use is just like that of the shift key on a typewriter. SPACE at the bottom right corresponds to the space bar of a typewriter. NEWLINE, EDIT, and BREAK will be explained in due course.

Clearing Out Old Programs

It is necessary to clear out any existing programs in the ZX80 memory before you type a new one. Alright, you have only just switched on and there should be nothing there, but let's practise it anyway. Press the 'Q' key, remembering that the cursor [K] is calling for a keyword.

The word NEW has appeared on the screen, and the cursor has changed to [L]. NEW is BASIC shorthand for 'clear out any old programs and variables and get ready for a new program'. It is 'on the screen', but now it must be passed to the computer so that it can obey the instruction. This is done by pressing NEWLINE on the right of the keyboard — do it now. NEW disappears (the computer has obeyed that instruction) and the cursor changes to [K] ready for another keyword. The ZX80 is cleared and ready!

Commands and Statements

We just used the instructions NEW and NEWLINE, and these are called *commands*. Commands do not form part of a program — they are separate instructions to the computer from outside the program, though they generally have some effect on the program. Some examples of commands are:

NEW LOAD SAVE

We are also about to meet a *statement*. Statements are included in

a numbered program line, and form part of the program. Some examples are:

FOR TO NEXT IF

Some instructions can be used either as *commands* or *statements*, for example:

PRINT GO TO CLS

Your First Program

'And not a moment too soon', I hear someone say! We learnt that a BASIC program consists of lines of instructions, each line starting with a number. In ZX80 BASIC you must limit yourself to one instruction per line. Let's start by writing a single line program.

Your cursor is requesting a keyword, but first we must type in a line number. Type '1Ø', and notice how the cursor jumps ahead of your typing — it always shows where the next typed character will appear. The keyword we want is *PRINT*, short for the BASIC instruction 'print any characters which follow on this line and are contained within quotation marks'. Which key are you going to press for *PRINT*? Well done, 'O' is correct, you're getting the idea! Do it now, and you have this on the screen:

1Ø PRINT [L]

the [L] cursor showing that letters or upper case characters are to be typed in now. Type a quotation mark (SHIFT Y) and you will see:

1Ø PRINT "[L][S]

Why did the [S] appear? This is because ZX80 BASIC *makes* you type lines which have correct syntax. Most other computers will let you type a single quotation mark by mistake, and you won't discover it until you try to run the program. The [S] on your screen now is saying, 'I've noticed your first quotation mark, and I won't let you enter this line until you have typed a second one to go with it'. Try it — press NEWLINE and nothing happens!

Now finish off your line, typing very carefully to give:

1Ø PRINT "THE FIRST RULE IN ZXØØ BASIC" [L]

Note how the [S] disappeared when the second quotation mark went in. This means that the syntax is good and the line is suitable to be entered — do it now by pressing NEWLINE. The line immediately vanishes from the bottom of the screen, to be replaced by [K], ready for the next line of the program. The first line reappears

at the top of the screen — the top three-quarters of screen is reserved for displaying all the lines of the program up to the one you are currently typing (or as much of it as it can fit in). On the left of your line is the *current line pointer* ⧁, which marks the last line you have typed in. Don't worry about this for the present, we'll be meeting it again in Chapter 12.

No more program lines for the moment — instead try running the program as it stands. Press keyword *RUN* and wait . . . Hmmm! RUN [L] certainly appeared on the screen, but little else seems to happen. What did we forget to do? Oh yes, enter the command *RUN* by typing NEWLINE. Do it now, and then shout 'Eureka!!!'. ZX80 did what it was told! It printed all the words, letters, numbers and spaces, but left out the PRINT, quotation marks and cursor — after all these were only there to tell it what to do. At the bottom right of the screen is the *error code* Ø/1Ø, which tells you that the program has run without any errors, finishing at line 1Ø.

What the program has just done is to make the ZX80 print a *literal string* — a *string* of characters of any kind which is printed *literally* as it occurs in the program line. Want to do it again? Right — press any key to put the ZX80 into command mode again, and to bring back the [K], and then press *RUN* and then NEWLINE again. Repeat until you've had enough, and then turn to Chapter 5 to add more to the program.

At the end of this and every chapter from now on, I'll give a short summary of any new points learned in the chapter.

We Learnt These in Chapter 4

Commands

NEW to clear out old programs and variables.
RUN to make the ZX80 run the program and carry out its instructions.

Statements

PRINT to make the ZX80 print a literal string on the screen.

Anything else

Connecting up hardware.
The ZX80 keyboard.

What the cursor does.
Numbering your program lines.
Literal strings — collections of any characters which are printed literally as they occur in the program line.
Quotation marks which are needed at the beginning and end of a literal string.

5

We Want Style in Our Programs!

A Second Line for Your Program

Let's finish off the first program and make it look as neat as possible. First of all type in this line very carefully, letter for letter.

```
2Ø PRINT "EVERY LINE MUST HAVE A NUMBERR" ⏎
```

but don't enter it yet. Oh-oh, two Rs in number, we'll have to correct it. No rubber needed, simply press RUBOUT twice (it's SHIFT Ø) and the cursor moves backwards and rubs out the last two characters. Now you have to replace the missing quotation mark, and enter the new line by typing NEWLINE. Run the program exactly as you did before, and all being well you have on the screen:

```
THE FIRST RULE IN ZX80 BASIC
EVERY LINE MUST HAVE A NUMBER
```

So far I've been carefully printing in the cursor whenever it occurred in a program line. Now I'm going to assume that you've got used to the cursor, and I will not bother to print it unless there is a special reason.

Tidying Up

Your program will give a neater result if we insert a space between the two lines of output. Try this — type:

```
15 PRINT
```

and then enter it by pressing NEWLINE. What on earth are we trying to do? PRINT what? Well, run the program as usual and see what happens. . . . Did it work? It did! When the ZX80 comes to a PRINT

statement it moves to the next line on the screen and prints what it is told to. In line 15 it was told to print nothing, and nothing was what it printed!

Finally, it would be nice to write a little remark at the beginning of the program to say what it is all about. Type this:

5 *REM* + + + MY FIRST ZX80 BASIC PROGRAM + + +

and enter it with a NEWLINE. Why all the + + + + + +? Showing off, that's all. The vital part of line 5 is the statement *REM*, which is saying to the ZX80, 'anything on this line is a remark by the programmer and you must ignore it'. Care to check up? Run the program as usual, the result should be just the same.

Our first program is now complete. Many versions of BASIC require an END statement at the end of their programs, but ZX80 says 'no need'.

Numbering and Listing

With most computers you have to *ask* for a list of all the lines in your program. ZX80 gives you one automatically as soon as you go into command mode. I'm sure you have been noticing everything that has been going on in the top part of the screen while we have been typing in the program. The ZX80 automatically sorted the lines into the correct order, although we typed them in the order: 1Ø, 2Ø, 15, 5.

Now, why have we numbered our first two lines 1Ø and 2Ø? Well done, you've got it! ZX80 does not care what the line numbers actually are, all it is interested in is the *order*. So we usually make jumps of ten between line numbers, to make it easier to insert extra lines later on. ZX80 offers a choice of line numbers from 1 to 9999, so there's no shortage.

Getting Rid of Whole Lines from a Program

Suppose we want to delete a whole line from the program — what can we do? RUBOUT will not work, not once you have entered the line into the program with NEWLINE. All you need to do is this. Type the number of the line and then NEWLINE. The ZX80 wipes out the old line completely and replaces it with the empty line. In the same way you can change a line by typing the line number, then the new version of the line as usual, and finally entering it with NEWLINE. You can change a line like this as often as you like, ZX80 will always erase the old line and replace it with the new.

Now for a couple of exercises to practise what you have learnt in the last two chapters.

Exercise 5.1

Get into command mode and then delete lines 5, 15 and 2Ø of your present program. Check that the lines have really gone, by running the program, and then change the remaining line to read:

THREE LINES GONE, ONE LEFT

Exercise 5.2

Delete the rest of your program with *one single keyword* (plus NEWLINE of course). If you have forgotten which keyword, look back to the end of Chapter 4.

Now write a new program to make the ZX80 print this:

SINCLAIR ZX80 MICROCOMPUTER
  MADE BY SCIENCE OF CAMBRIDGE

Put a space between the two lines, and two spaces before 'MADE'. See what you can do with that one, and just to check that it can be done, try making jumps of a thousand between the line numbers.

We Learnt These in Chapter 5

Statements

PRINT to produce a line space.
REM for remarks — ignored by the ZX80 when it runs the program.

Anything else

RUBOUT to get rid of mistakes, one character at a time, while typing in a line of program.
Automatic listing of the program while you are in command mode.
Leaving gaps in line numbering, so that extra lines can be inserted later on.
Delete a whole line from the program after entering it, by typing the line number and then NEWLINE.
Change any entered line by typing its number, then the new version, then NEWLINE.

6

Will It do Sums?

First, let us be honest and say that the answer is a qualified 'YES' for the basic ZX80.

The trouble with the simple ZX80 is that it uses an INTEGER BASIC which only recognises integers (whole numbers). If an answer should contain a decimal fraction part, this is simply chopped off and lost. For example:

Expected answer	*ZX80 answer*
4	4
7.2	7
12.5	12
15.99	15
0.999	0

Note that there is no question of rounding off, the decimal part, however large, simply disappears.

Mathematical operators

Back to school now, to remind ourselves of the four maths operators. You'll easily find two of them on the ZX80 keyboard:

+ (plus) at SHIFT K
– (minus) at SHIFT J

You will look in vain for × (times) because the ZX80, like most computers, uses * (at SHIFT P) for this. Similarly, for ÷ (divided by) the ZX80 uses / (at SHIFT V); remember how we use this to write fractions like 2/3, which are really division sums.

There is one more useful operator available, for calculating powers of numbers, and this is ** (SHIFT H). For example:

7^2 (or 7×7) would be written as 7**2
2^4 (or $2\times2\times2\times2$) as 2**4 and so on.

We now have all the tools — let us see how to use them.

The LET Statement

BASIC uses a most powerful statement, which allows a computer to do a calculation, put the answer in a memory box, and label it so that it can be used later in a program.

Clear your ZX80 by typing NEW, and then type in this line, using the keyword LET,

```
10  LET A=123*45
```

(So far, I have been reminding you to type NEWLINE after each command and program line, but now you are on your own!)

What our line 10 is saying is:

'Work out the answer to 123×45, and put it in a memory box and label it A'.

Now RUN the program — the code 0/10 at the bottom of the screen tells you that it ran successfully, stopping at line 10. There's precious little in the way of output though — WHY NOT?

Pause for deep thought . . .

What is the statement that makes the ZX80 print things on the screen? Yes, PRINT was the one. Type:

```
20  PRINT A
```

and RUN it again. Eureka! It worked — the answer 5535 came up on the screen.

Exercise 6.1

Why did we not type:

```
20  PRINT "A"
```

The answer is at the end of the book, or to save time, try doing it.

Since 5535 is rather an anonymous number, let's give it a name tag:

```
15  PRINT "A="
```

RUN, and check that the output is

```
A=
5535
```

Integer Variables

Our memory box now contains the value 5535 and has a label A. More correctly we say that *the integer variable A has the value of 5535.*

Very many different integer variables are possible in ZX80 BASIC, limited only by the memory available. You must follow two simple rules in naming them:

(1) Names *must* start with a *letter.*
(2) They *may* contain *letters and numbers,* but *nothing else* (no spaces, punctuation, etc.)

So, here are a few simple examples.

A to Z
AA to ZZ
A1 to A9
Z∅ to Z2∅

We usually choose names to act as a mnemonic for the contents (T for total, A for area, and so on).

Using LET with integer variables

As we have seen, this very important statement in BASIC takes the form:

LET integer variable name = . . .

Now let's see what we can put on the right of the = sign. (Let's start calling integers 'numbers' from now on — not quite correct but a bit more homely.)

(1) A number
e.g. LET B = ∅
LET C = 99
(2) An expression using numbers
We have already used LET A = 123*45

Similarly, LET D = 156/13
LET E = 67 + 89
LET F = 98−76
LET G = 3**4

You can make these expressions as complicated as you like.

(3) An expression using other variables, with or without numbers.
e.g. LET H = 2*G
LET I = F + 2
LET J = A + B

Here we need to be very careful, for it is no use telling the ZX80 that H = 2*G, unless you have already said what G is. *You must define your variables before you use them,* even if they are ∅.

(4) An expression using the same variable as the one on the left of the = sign.

e.g. LET K = K + 1∅

Algebra was never like this! Remember we are not using the = sign in quite the usual way. We are saying to the ZX80, 'take the contents of memory box K, add 1∅ to it, and put this new value back in box K in place of the original contents'.

Note, once again K must have been defined previously in the program.

There has been a lot to think about in this chapter, now let's try some practical work. Try writing some short programs to define variables, using all the above methods, and to print the values of the variables. Especially, try a program like this:

```
1∅ LET K = 9
2∅ PRINT "ORIGINAL K = "
3∅ PRINT K
4∅ LET K = K + 78∅
5∅ PRINT "FINAL K = "
6∅ PRINT K
```

Exercise 6.2. Exchange Rates

The bank offers you 69 Belgian Francs for £1. Write a program to calculate and print out the number of Francs you get for £275, and similarly how many pounds you must pay in to get 5382 Belgian Francs.

We Learnt These in Chapter 6

Statements

LET to define a variable
PRINT to print the value of a variable

Anything Else

ZX80 uses integer BASIC (no fractions and no decimals).
Four maths operators, +, −, * and /.
** for powers of numbers.
Rules for naming variables.

7

Useful Odds and Ends

Punctuation

That exchange rate program in Chapter 6 — rather a messy print-out, wasn't it? How much neater if we could print strings and variables on the same line when we want to. No problem! The program needs a little punctuation, that's all. Try this new version of the first part of Exercise 6.2:

```
1Ø  LET L=275
2Ø  LET B=69*L
3Ø  PRINT B;"  BELGIAN FRANCS FOR £";L
```

and RUN it. That looks a lot neater. The *semi-colons* are saying 'don't move to a new line, I want to print the next item right after the last'. Notice that we needed a space between the value of B and 'BELGIAN', so we had to include that space in the string.

Exercise 7.1. Miles per Gallon

Your car manages to cover 258 miles after being filled with 8 gallons of petrol. Write a program to calculate the petrol mileage and print it in the form:

PETROL MILEAGE = ×M.P.G.

As a matter of interest, check your answer with a hand calculator.

Sometimes we want to leave a gap between two PRINT items. We can use a string of spaces as above, but quicker and more economical in precious memory is the *comma*. ZX80 BASIC divides each line on the screen into four equal zones, and a comma in your program says, 'move to the next zone before PRINTing the next

item'. You'll see how it works in this program:

```
10 PRINT "FEE", "FI", "FO", "FUM"
20 PRINT
30 PRINT "I SMELL THE BLOOD OF AN"
40 PRINT,, "ENGLISHMAN"
```

Notice how we used ,, in line 40 to skip on two PRINT zones.

Now we know how to PRINT items immediately after one another, or with spaces between, or on successive lines, and our program print-outs should be looking much prettier. Try your hand at this:

Exercise 7.2. Family Transport

Write a program to print a table of the way your family gets around, like this:

Name	Colour	Make	Type
Dad	White	Austin	Car

and so on.

Error Codes

You must have been wondering about the numbers which appear at the bottom left of the screen after RUNning a program. They are error codes, and are listed on page 99 of the ZX80 Manual. 'Error' is not quite the right word, because one of the codes tells you that your program has finished running *without* apparent error. For instance, the 'FEE, FI, FO, FUM' program will have ended with 0/40, showing that the program has RUN and finished at line 40.

Try this program:

```
100 PRINT "LINE BEFORE ERROR"
200 LET A=99*999
300 PRINT "LINE AFTER ERROR"
```

The ZX80 accepts it quite happily, but will it RUN? No — the error code 6/200 has come up, and the ZX80 Manual will tell you that this means an arithmetic overflow at line 200 (we tried to put a number greater than 32767 into A). At line 200, the program stops running, and no instructions on or after this line are carried out.

More on error codes later.

We Learnt These in Chapter 7

Punctuation (, and ;) to control where items are printed on a line.
Error Codes to indicate whether or not your program contains errors which prevent it from running.

8

The Order is Important

We saw in Chapter 7 that an answer greater than 32767 stops a program with an error code. Given a complicated expression with several operators and numbers, the ZX80 (like a human being) has to do one step of the calculation at a time, and *at no stage in the calculation must the answer go over 32767* (or under −32767).

You will remember how important it is in maths to do different operations in the right order. It is just the same for the ZX80, which has to follow these standard rules of priority.

High priority	x**y	(x to the power y)
	−x	(x made negative)
	x*y	(x times y)
	x/y	(x divided by y)
	x+y	(x plus y)
Low priority	x−y	(x minus y)

It's useful at this stage to know that we can do calculations in command mode, without affecting any program in the ZX80 memory. For example, type:

PRINT 2**3*6+9 NEWLINE

The answer (57) comes up immediately, having been calculated in these stages:

$2^3 = 8$
$8 \times 6 = 48$
$48 + 9 = 57$

Try doing more calculations in command mode, and make sure that you get the answers you expect, following the given priority rules.

Suppose we want to change the order — remember the good old bracket? Things *are* what they used to be, and the ZX80 gives *absolute priority* to calculations within a pair of brackets. Let's change the last calculation to:

PRINT 2**3*(6+9)

This time the stages are:

$6 + 9 = 15$
$2^3 = 8$
$8 \times 15 = 120$ (answer)

Even more complicated expressions? Maybe you need brackets within brackets within. . . . Remember that the ZX80 starts at the innermost pair of brackets and works its way out. You don't need telling that brackets come in pairs — nor does the ZX80; it will not allow you to enter a line with an odd number of brackets. Try this expression:

PRINT (2*(3+16))+9

The answer is worked out in these stages:

$3 + 16 = 19$ (inner brackets)
$2 \times 19 = 38$ (outer brackets)
$38 + 9 = 47$ (answer)

Now for a more subtle point. Take an expression like:

$$\frac{24 \times 27}{10}$$

This has to be worked out in two stages, and humans can take any two of the three numbers for the first stage, and get the same answer (6). Now calculate the answer in two ways using the ZX80.

PRINT 24*27/10 (this gives 6 as expected)
PRINT (24/10)*27 (gives us 0)

See what happened? We fell foul of the dreaded integer basic! 24/10 gives an answer a little over 0.2, and as we saw in Chapter 6, integer basic reads 0.2 as 0. Hence the answer to the whole sum is given as 0.

Now you know the full problem. Like Ulysses steering his ship between the monster Scylla and the whirlpool Charybdis, you have to set a course to avoid your calculations going over +32767, under −32767 or between 0 and 1. The only advice which one can give is to keep cool and use brackets to steer one way or the other.

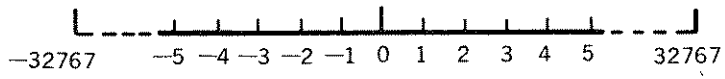
Exercise 8.1

Find the best way of working out these two expressions, and check your answers with a calculator.

$$\frac{36\times 9\emptyset}{54} \quad \text{and} \quad \frac{45\times 73\emptyset}{25}$$

Positive and Negative Numbers

ZX80 BASIC works with sets of positive and negative integers, as shown on this number line



The − sign of a negative integer is always shown, but + signs of positive integers are assumed. We can easily convert from positive to negative, or vice-versa, by using an instruction like:

```
1ØØ  LET A=-A
```

We can obtain the absolute value of a number by using the statement ABS.

```
2ØØ  LET A=ABS(A)
```

Remember that ABS is one of the 'integral functions' which has to be typed out in full. The ABS function converts all negative integers to positive, and leaves positive integers unchanged.

```
e.g. ABS(-1Ø)=1Ø
     ABS(1Ø)  =1Ø
     ABS(Ø)   = Ø
```

Try working with some of these absolute and negative values, using PRINT in command mode. Remember that ABS must always be followed by brackets containing a number, a variable, or an expression.

Now we have covered all the rules and instructions needed to do calculations in integer BASIC. Here are some simple problems which you will manage very easily.

Exercise 8.2. Temperature Conversion

Temperatures are increasingly being given in degrees Celsius (°C).

Write a program to take a Fahrenheit temperature, convert it to Celsius, and print out the result. Remember that:

$$°C=(°F-32)\times\frac{5}{9}$$

Exercise 8.3. Volume and Weight of Cuboid

A lead brick is 14 cm long, 9 cm broad and 6 cm deep. Write a program to calculate the volume (=length×breadth×depth) and display the result in cubic centimetres. If 1 cubic centimetre of lead weighs 11 grams, work out and display the weight of the brick.

We Learnt These in Chapter 8

Expressions

ABS (n) The absolute value of n.

Anything else

In BASIC, arithmetical operations are carried out according to standard rules of priority.
The priority can be changed, if required, by using brackets.
Calculations can be done in command mode, without affecting any programs in memory.

9

Over and Over and Over Again

Remember the sorcerer's apprentice in Chapter 3? Here is a simple mathematical model of a broom filling a 150 gallon water tank at the rate of 4 gallons of water per trip. Type in this program:

```
1Ø  LET W=Ø
2Ø  LET W=W+4
3Ø  PRINT W,
4Ø  GO TO 2Ø
```

Line 1Ø sets the water in the tank (W) at Ø at the start.
Line 2Ø adds 4 gallons to the tank.
Line 3Ø prints the total number of gallons added to the tank (note the comma).
Line 4Ø contains an important new statement saying 'go to line 2Ø and continue running the program from that line'. In other words, 'take another trip to the well for more water'.

Can you predict the output of this program? Well, now RUN it and see if you were right.

GO TO is certainly a cheap way of generating numbers! We have made a *loop* in lines 2Ø to 4Ø, and each time round the loop we are adding 4 gallons of water to the tank. It stopped at 368 gallons, but only because the screen was full — the error code 5/3Ø tells us that. No wonder your feet are feeling wet!

In Chapter 3 we saw that we need to include a 'conditional jump' in the loop to check whether the tank is full. Type in this program — the first three lines are as before — and RUN it:

```
1Ø  LET W=Ø
2Ø  LET W=W+4
3Ø  PRINT W,
4Ø  IF W<15Ø THEN GO TO 2Ø
5Ø  PRINT
6Ø  PRINT
7Ø  PRINT ''THE TANK IS FILLED, O MASTER''
```

That worked pretty well, apart from the last 2 gallons which slopped over. Line 4Ø is the vital one, which is saying, 'check the present value of W, if it is less than 15Ø then go to line 2Ø and round the loop again, but if W is not less than 15Ø then go to the next line (5Ø)'. BASIC is a nice concise language.

Relational Operators

The general form of our statement in line 4Ø is:

IF something is true THEN do something.
(e.g. W<15Ø) (e.g. GO TO 2Ø)

We *must* always follow the IF keyword with a statement using one of the three *relational operators* which are used to compare two items:

= (equals)
< (is less than)
> (is greater than)

On either side of the relational operator are the two items being compared. These may be variables, numbers, or expressions using variables and/or numbers, as shown in these examples:

```
IF A=Ø THEN . . .
IF B>99 THEN . . .
IF C<D THEN . . .
IF 2*E<50 THEN . . .
IF F+999>13*G THEN . . .
```

We can also use the logical operator NOT with the relational operators, like this:

IF NOT A=75 THEN . . . means 'if A is not equal to 75 then do. . . .'

Similarly:

```
IF NOT B>1ØØ THEN . . .
IF NOT C<66 THEN . . .
```

and so on.

IF Something is True THEN What?

We dealt with IF, so now for THEN. THEN is a 'token' (SHIFT 3) which always produces the K cursor, calling for a keyword statement to represent 'do something'. ZX80 will accept any keyword here, but only the following make sense:

PRINT
GO TO
LET
INPUT
STOP
POKE
GOSUB
CLS
RET

} These will be covered in later chapters.

Here are some examples of lines containing conditional statements:

```
10  IF Z>21 THEN PRINT "OVER 21 AND BUST"
20  IF Y=200 THEN GO TO 1000
30  IF X>300 THEN LET X=300
```

GO TO Where?

Whether our GO TO is compulsory or conditional, it must be followed by a valid line number! In this way you can direct the ZX80 to go to any line in your program, either before or after the GO TO line. You may write the line number either as a line number, or a variable, or an expression (of course, any variables used must have been defined).

Note that if you say GO TO a non-existent line, the ZX80 will GO TO the first line following that one.

What About STOP?

With all this GO TOing in the program, it's as well to know how to stop! Type in this simple number testing routine, which could form part of a longer pontoon program.

```
100  LET T=18
200  IF T>21 THEN GO TO 500
300  PRINT "YOUR SCORE IS ";T
500  PRINT "OVER 21 AND BUST"
```


RUN it — there is obviously something needed to stop the ZX80 going charging on and doing both lines 300 and 500 — so type in 400 STOP (keyword S). The error code 9/400 simply says that the program has STOPped at line 400. Now you can try putting in different values of T in line 100 — make sure you get the answers you expect.

Finally, here are two problems, each to be solved by a loop containing an IF . . . THEN statement.

Exercise 9.1. Inflation

You earn £80 a week now, but at the end of each year your pay is increased by 20% to keep pace with inflation. How many years will it take you to reach £1000 a week?

(New pay = old pay *12/10)

Exercise 9.2. Chess Prize

A man so pleased the king of his country by teaching him chess, that he was offered any gift he wanted. He chose to have gold coins put on his chess board, 1 on the first square, 2 on the second, 4 on the third, and so on, doubling each time. How many coins were needed for the first 15 squares? Why not the first 16 squares?

We Learnt These in Chapter 9

Statements

GO TO n sends the ZX80 to the line numbered n (or if line n does not exist, to the next line after n).

IF . . . THEN. IF tests whether some statement is true; if it is, THEN orders the ZX80 to do something (e.g. PRINT, GO TO, LET). If the statement is not true, the ZX80 continues with the next line of the program.

STOP makes the ZX80 stop the program here.

Anything else

Relational operators

A=B (A equals B)

C>D (C is greater than D)
E<F (E is less than F)
A statement like this always goes between IF and THEN.

NOT is a logical operator which can be used with IF
e.g. IF NOT X=1ØØ THEN GO TO 25Ø.

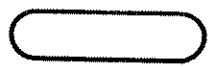
10

Flowcharts

We are able to write quite complicated programs, now that we have learnt about loops and conditional branching. At this stage, it is worth reminding ourselves about *flowcharts* as an aid to good programming.

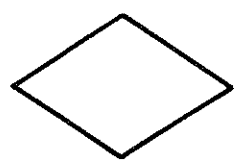
Suppose you have some operation for which you want to write a program — let's use the sorcerer's apprentice idea from Chapter 3 as an example. The idea of a flowchart is to split the operation up into separate stages, to write each stage in a box, and to join the boxes by arrows to show the order in which the stages have to be done. We use boxes of these shapes:

Beginning or end.



'Processing block' — one stage of the operation which needs no decision.

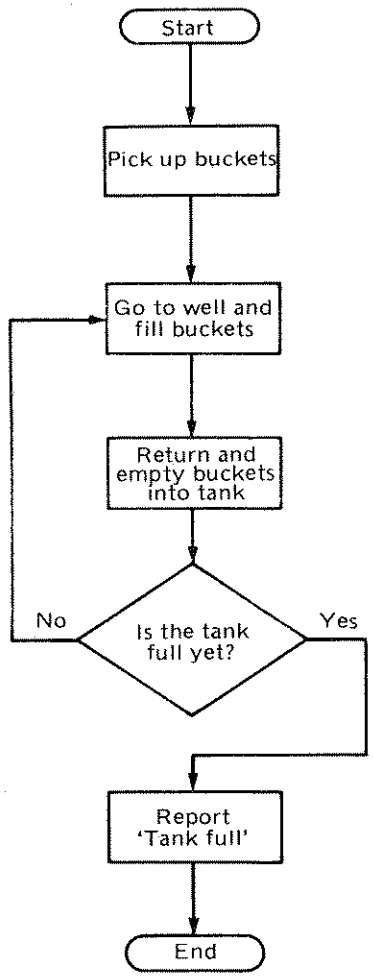
'Decision diamond' — here a question is asked and the flowchart branches to either side depending on the answer.



Now we can draw up a flowchart for filling the water tank from the well. Compare it with the original program in Chapter 3, and with the mathematical model in the last chapter. Notice how the place of the decision diamond is taken by the IF . . . THEN . . . statement.

Some people can carry a flowchart in their heads and type out a program direct. However, most of us will benefit from drawing up a flowchart on paper first. We'll see more examples of flowcharts for ZX80 programs later. Also read through Chapters 5 and 7 of the ZX80 Operating Manual.

Broom filling water tank from well



11

Data Please

Let's go back to our program for converting temperatures, listed at the end of the book as the answer to Exercise 8.2. It worked well, but with the snag that to convert a different temperature we need to re-type line 1Ø and RUN again. Surely we can do better than that!

Well, try this one:

```
4Ø  INPUT F
5Ø  LET C=(F-32)*5/9
6Ø  PRINT F;" DEGREES F=";C;" DEGREES C"
```

We have replaced LET F=77 by a new statement INPUT F. Now RUN the program — do you see the double cursor [L][S] on the otherwise empty screen? The cursor is saying, 'enter a number now, and then press NEWLINE'. Do it now, type in 77 and then NEWLINE. You will get the immediate output: 77 DEGREES F=25 DEGREES C. The program has accepted and used the value for F which you INPUT, just as though there was a LET F=77 statement. RUN the program again, and INPUT different values for F.

Let's add more lines to make a better version:

```
1Ø  PRINT "FAHR. TO CELSIUS CONVERSION"
2Ø  PRINT
3Ø  PRINT "TYPE IN NEXT FAHR. TEMP. NOW"
4Ø  INPUT F
5Ø  LET C=(F-32)*5/9
6Ø  PRINT F;" DEGREES F=";C;" DEGREES C"
7Ø  GO TO 3Ø
```

This time we have added a title (line 1Ø) and a 'prompt' (line 3Ø) which tells you what data is to be INPUT. Finally we have put in a GO TO in line 7Ø to make the program loop back for more data.

RUN the new program, and type in lots of different temperatures.

Still something lacking, isn't there? Up to ten temperature conversions are shown, but in rather a squashed up way, and then the program stops with a 'screen full' error code. Try adding these lines:

```
45 CLS
65 PRINT
```

Much better now — a single answer printed and then a request for more data. It's all thanks to CLS, which is an instruction to *clear the screen*. We INPUT our next value for F, and then line 45 clears the screen ready to receive the next answer printed by line 6Ø. You'll find that, in programs with lots of output, CLS is vital to prevent the screen filling up and stopping your program.

Breaking Out of Loops

If you carry on using this program, you will notice that you are in a bit of a fix. What happens when you want to stop converting temperatures? Sure, you can unplug the power supply, but then your whole program is gone. You are in a *number INPUT loop*, and the way to get out of it is to input a letter (one which is not a variable name in the program) instead of a number — this stops the program with a 2/4Ø error code, meaning 'variable name not found'.

Now a different kind of loop. Change line 4Ø to read 4Ø LET F = 77, and RUN again. INPUT has gone now, and we are in an *infinite loop* caused by bad programming. The poor old ZX80 is buzzing round and round lines 3Ø to 7Ø, and while this is happening there is no output, only a grey screen. Press the BREAK key — it's at the bottom right of the keyboard — to get out of the loop. BREAK is a certain way of stopping the ZX80 while it is working with a blank screen.

Exercise 11.1. Running Average

You remember how to work out the average (arithmetic mean) of some numbers — add them all up and divide by the number of items. Write a program in which you INPUT items one at a time and calculate the running average (the average of all the items entered so far).

We Learnt These in Chapter 11

Commands

BREAK stops the ZX80 while it is working.

Statements

INPUT to stop the program and insert numerical data.
CLS to clear the screen and make room for more output.

Anything else

INPUT loop. Makes the ZX80 repeatedly stop to accept data and then process it.
Enter a letter to break out of a numerical INPUT loop.

12

First Edition

We know two ways of correcting mistakes so far. We can use RUBOUT while we are typing a line or an INPUT, or we can delete or replace an entered line by typing its line number plus a new version.

If we need to make a small change in a long, entered line, the first method will not work and the second takes a lot of time. The answer is to EDIT the line.

The Current Line Pointer

Let's look at the program first. Assume that you have the 'Average' program from Chapter 11, listed in the answer to Exercise 11.1. We want to make changes to one of the lines in the program:

Line 40 Insert THE between INPUT and NEXT
Delete ITEM and insert NUMBER in its place.

If you look at the program on the screen you will see that one of the line numbers is followed by a black cursor with a white arrowhead — the current line pointer. Unless you have moved it, it will be at the last line you typed in.

The first essential is to move the current line pointer to line 40, and we can do this in different ways:

(1) Use the key ⇧ (SHIFT 7) to push it up, line by line.
(2) Type LIST 40 NEWLINE.
(3) Send the current line pointer to (theoretical) line 0 by typing LIST NEWLINE or HOME (SHIFT 9), then use key ⇩ (SHIFT 6) to push it down, line by line.

Try all of these methods, until you really can make that current line pointer go where you want.

EDITing a Line

Now type EDIT (SHIFT NEWLINE) and you will see line 40 appear at the bottom of the screen with the usual K cursor after the line number.

Press the ⇨ (SHIFT 8) key, and see the cursor skip past the keyword PRINT and change to L. Keep pressing ⇨ repeatedly until the cursor is just beyond the T of INPUT, and then type in your addition which is:

space THE

Now press ⇨ until the cursor has got to the end of ITEM — if it goes too far, bring it back with ⇦ (SHIFT 5). Press RUBOUT four times to get rid of ITEM and then type NUMBER in its place.

Finally press NEWLINE, and your EDITed line goes into the program in place of the original version.

Renumbering Lines

We'll change line 100 to become line 105. Hard to do on many computers, but child's play for the ZX80.

Type LIST 100 and then EDIT, to put line 100 on the chopping block. Press RUBOUT once, to get rid of the second 0, type 5 in its place, and then NEWLINE to put the renumbered line in the program. Old line 100 is still there — you'll need to type 100 NEWLINE to erase that.

Another use for LIST

When a program takes up more than 22 lines of screen, the ZX80 can only show the last line entered and some of the other lines. With a very long program, the height of screen available shrinks so that even less can be seen. Picture the top part of your screen as a window which is only high enough to view part of the program at a time. Then you can move the window up and down by typing LIST (to see the beginning of the program) or LIST (line number) to see that line plus as many other adjacent lines as possible.

Exercise 12.1. Editing

EDIT your present program as follows:
Change line 105 to read:

```
105  PRINT N;" NUMBERS SO FAR"
```

Renumber line 110 to become 130
Type two new lines:

```
110  PRINT
120  PRINT "AVERAGE=";A
```

RUN the program to make sure it works, but don't lose it — we will need it in Chapter 13.

We Learnt These in Chapter 12

Commands

EDIT brings the current line to the bottom of the screen for editing.
LIST shows the beginning of a program with up to 22 lines in all, and sets the current line pointer at 0.
LIST n shows line number n of the program, with up to 22 adjacent lines, and sets the current line pointer to that line number.

Anything else

Vertical arrows to move the current line pointer up or down.
Horizontal arrows to move the EDITing cursor backwards or forwards on a line.
EDITing and renumbering lines.

13

A Program Worth Saving?

The ZX80 has a nominal 1K of RAM, which can hold 1024 bytes of information. Any one of these can be produced immediately on demand. This is where the program which you are currently using is stored. If you put in another program, or if you switch off the ZX80, all the contents of the RAM are lost. Hence you must have *backing storage*, in which you can keep finished programs and data until you need them again.

Most domestic cassette recorders can act as backing storage for the ZX80, and since your longest programs will take less than a minute of recording time, costs are low.

How to SAVE a Program

Ideally, the cassette recorder should be of good quality, with clean recording and playback heads, and with 3.5 mm jack sockets for microphone and earpiece/extension speaker. If it has a round 5-pin DIN socket for the microphone, you will need an adaptor from a radio parts shop. A tape footage counter will be of tremendous value for finding programs on a long tape.

The following routine has given me good results, but bear in mind that cassette recorders vary and you may have to experiment a little.

(1) Wind your tape to the beginning, set the counter to zero, and then wind the tape on to a suitable space.
(2) Most tape recorders now have automatic recording level control. If not you will have to find a suitable recording level setting.
(3) Make a written note of the counter reading and the program to be recorded. If no counter, record the program title on the tape, using the microphone on your tape recorder.

(4) Use one of the Sinclair double jack plug connectors to join the ZX80 'MIC' socket to the 'MIC' socket of your recorder.

(5) Start the tape recorder on 'RECORD', press SAVE and NEWLINE on the ZX80.

(6) For 5 seconds you will see a grey, even pattern all over the screen. Then you will see a rapid succession of contrasty, black and white lines — these show that the program is being recorded. When the recording is finished, the program listing reappears on the screen.

(7) Wait about 5 seconds, without stopping the recorder, and then press SAVE and NEWLINE again to make a second recording.

I have found that my first recordings are hard to LOAD, while second ones are alright, perhaps due to some trouble with automatic recording level. In any case, it's good practice to SAVE a program twice, 'just in case'.

How to LOAD a Program

Some time you'll want to put your program back into the ZX80 to RUN it again or improve it. This is the way to do it.

(1) Wind your tape to the point where you started recording the program, using the counter or the voice identification as a guide.

(2) Set the TONE control to MAX, or TREBLE to MAX and BASS to MIN. Set the volume control to MAX.

(3) Connect the ZX80 'EAR' socket to the 'EAR' socket of your tape recorder. You can also use the extension loudspeaker socket, but note that the DIN socket is no good, as it does not give an amplified signal.

(4) Press LOAD and NEWLINE, and immediately start the recorder on 'PLAYBACK'.

(5) For 5 seconds you will see the grey even pattern, and then this will change to 'heavy driving rain' falling from top right to bottom left of the screen, showing that the program is LOADing.

(6) After a successful LOAD, the screen will clear and some or all of the new program will be seen listed on the screen.

(7) If this does not happen after a minute, or if there are other signs of a bad LOAD, you will have to stop LOADing by pressing BREAK, or else by unplugging the ZX80 power supply. Rewind the tape and try again, perhaps at a lower volume setting.

Saving Variables

If your program has been RUN before SAVEing and some variables have been INPUT, all of these will have been SAVEd with the program. If you want to avoid losing these variables next time you use the program, type GO TO 1 instead of RUN. With some long programs you can save RAM space in this way by generating variables and then erasing program lines to make room for more.

We Learnt These in Chapter 13

Commands

SAVE to record a program from the ZX80 for future use.
LOAD to put a program from tape into the ZX80.
GO TO 1 (instead of RUN) to run a program without losing variables which are already in memory.

14

Over and Over . . . Ten Times Exactly

In Chapter 9 we met *the loop,* and saw how important it was to include an IF . . . THEN statement to jump out of the loop when some given condition was met. You could easily design a program to take exactly ten trips round a loop . . . couldn't you?

```
1Ø  LET J=Ø
2Ø  LET J=J+1
3Ø  PRINT J;" TIMES ROUND THE LOOP"
4Ø  IF NOT J=1Ø THEN GO TO 2Ø
5Ø  PRINT
6Ø  PRINT "STOPPED FOR A REST"
```

Type it and RUN it.

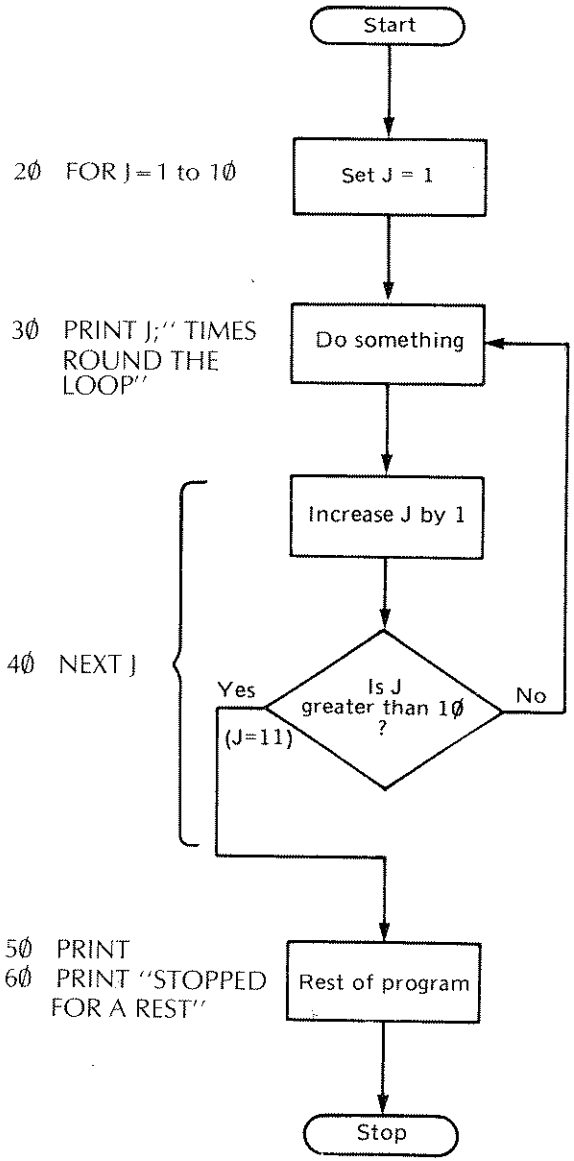
No trouble at all — but wait — ZX80 BASIC has a special set of instructions to do just the same job in a more economical and flexible way.

Delete line 1Ø
Rewrite line 2Ø FOR J=1 TO 10
(FOR is keyword F, TO is token 4)
Rewrite line 4Ø NEXT J
(NEXT is keyword N)

Now RUN it again — the output is identical to the first — FOR/NEXT is a real winner! The complete program is shown opposite, together with a flowchart. Here are a few general points:

(1) J is the *loop control variable,* which can be given any single letter name from A to Z, though you must not use any variable names which are already in your program.
(2) FOR J=m TO n
n must be greater than m, otherwise you will only go once

Ten times round a FOR/NEXT loop



round the loop. Apart from this, m and n may be any number within the range ±32766, any variable (previously defined) or any expressions using variables, numbers or both. All these are legal, for instance:

```
FOR J=Ø TO 1ØØ
FOR J=-1Ø TO -5
FOR J=A TO 25
FOR J=B TO B+19
FOR J=23+B TO A*17
```

(3) NEXT J increases J by 1 and then compares it with n. If J is equal to or less than n, then we go back and through the loop once more. If J is greater than n we go on to the line after NEXT J. Note that J ends up as n+1.

(4) Inside the loop may be any number of lines with any of the usual BASIC statements. You can even include an IF . . . THEN statement to jump out of the FOR/NEXT loop before it is finished.

(5) You can have a FOR without a NEXT — the program will ignore it. However NEXT without FOR will stop the program with a 1/n or 2/n error code.

Now try these examples, using FOR/NEXT loops in your answers.

Exercise 14.1. Savings

You are saving up for a new ZX80 at £99.95, and you put a different amount into your piggy bank each month for a year. Write a program to INPUT each monthly amount, and calculate and display the total saved at the end of each month.

Include a reminder to send for your ZX80 when you have saved enough money.

Exercise 14.2. Compound Interest

You put £100 into a building society which pays 9% compound interest annually, and leave it there for 7 years. Write a program to work out the capital at the end of each year, and print out a heading and a table showing YEAR and CAPITAL. The vital maths is:

$$\text{After 1 year, capital} = \frac{1ØØ \times 1Ø9}{1ØØ} = 1Ø9$$

$$\text{After 2 years, capital} = \frac{1Ø9 \times 1Ø9}{1ØØ} \text{ etc.}$$

We Learnt These in Chapter 14

Statements

FOR . . . TO . . . NEXT to send the ZX80 round any loop a fixed number of times.

15

Big Fleas Have Little Fleas . . .

Although our programs in Chapter 14 all had a single FOR/NEXT loop, you are by no means restricted to one. Type in this program and RUN it.

```
 1Ø  FOR J=1 TO 5
 2Ø  PRINT "LOOP 1"
 3Ø  NEXT J
995Ø  PRINT
996Ø  FOR J=32760 TO 32766
997Ø  PRINT "LOOP 2"
998Ø  NEXT J
```

There is usually no objection to using the same variable to control more than one loop — it is reset to the correct number when it meets a new FOR statement. You may find that it helps in programming to mark the loops as above.

Loops Within Loops

Now take a look at this program and see if you can predict what the printout will be:

```
 1Ø  FOR J=1 TO 4
 2Ø  PRINT "OUTSIDE LOOP"
 3Ø  FOR K=1 TO 3
 4Ø  PRINT, "INSIDE LOOP"
 5Ø  NEXT K
 6Ø  NEXT J
```

Type it and RUN it — were you right? This is the simplest example of *nested loops*, which need a little careful organisation.

(1) Your loops must not overlap — inner loops must be entirely within outer loops.
(2) You must use different control variables for nested loops.
(3) Subject to the above rules, you may nest up to 26 loops if you wish. At this point you run out of variables!

You must be getting good at predicting outputs by now — try your hand at this one.

```
 1Ø  FOR J=1 TO 3
 2Ø  PRINT "OUTSIDE LOOP"
 3Ø  FOR K=1 TO 3
 4Ø  PRINT, "1ST INSIDE LOOP"
 5Ø  NEXT K
 6Ø  FOR K=1 TO 2
 7Ø  PRINT,, "2ND INSIDE LOOP"
 8Ø  NEXT K
 9Ø  NEXT J
```

And now for a tricky problem.

Exercise 15.1. Number Square

Here is one of the number squares that you used at school:—

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Write a program to draw one of these squares. Clue — use two nested loops and make the maximum use of J and K.

Exercise 15.2. Chessboard

Write a short program using nested loops to draw a picture of a chess board. Not a very good picture, because you'll have to use W for a white square and B for a black square. Spread out your letters with spaces in between, and remember that the bottom right square is white.

```
W B W B W . . .
B W B W . . .
W B W B . . .
B W B . . . etc.
```

We Learnt These in Chapter 15

A program may have any number of separate FOR/NEXT loops. Loops may be nested within each other, provided that certain rules are obeyed.

16

The ZX80 Gets Friendly

We know how to handle numbers in flexible ways by making use of *numerical variables*, defined either by LET or INPUT, and we can do the same with *strings*.

Literal Strings

We met these in Chapter 4. We simply use PRINT followed by the string enclosed in quotation marks, and the string may contain any characters except ". Note that, though a string may consist of a number we cannot do any arithmetic with it — it looks like a number but behaves like a word.

```
1Ø PRINT "1Ø";
2Ø PRINT " LORDS A-LEAPING"
```

The output is:

```
1Ø LORDS A-LEAPING
```

Literal strings are very useful, provided that we only want to use them once in a program (or once each time round a loop which contains them).

String Variables

Type this program and RUN it:

```
1ØØ LET A$="NINE"
11Ø LET B$=" LADIES DANCING"
12Ø PRINT A$;B$
```

Here we have two string variables. The first is "NINE" (it could equally well have been "9") and is called A$, the second is " LADIES DANCING" and is called B$. These are kept in the ZX80 memory (until we change them or switch off) and we can use them as often as we like. All the usual punctuation rules apply, and we can use string variables in loops. Try adding these lines to your program:

```
13Ø  PRINT
14Ø  PRINT B$, A$
15Ø  PRINT
16Ø  PRINT A$
17Ø  PRINT
18Ø  PRINT B$
```

RUN the program and then add these lines:

```
115  FOR J=1 TO 9
125  NEXT J
```

and RUN again.

It's quite an art, handling strings — you need plenty of practice, but you also need to know these rules:

(1) A string variable consists of any characters — numbers, letters, graphic blocks, punctuation (but not "), and spaces — enclosed within a pair of quotation marks.
(2) 26 names are available for string variables, consisting of a letter followed immediately by $ (you can use, for instance, a string variable A$ as well as a number variable A in one program).
(3) You can define string variables with the statement LET, and you must define them before you use them.
(4) String variables are output in the usual way by using PRINT, and follow all the usual punctuation rules. One single PRINT line may contain any mixture of string variables, number variables and literal strings.
(5) If a space is required between a string variable and anything else, it must be included within the quotation marks.

This loop program illustrates many of these rules:

```
1ØØ  LET A$=" GREEN BOTTLE"
11Ø  LET B$=" HANGING ON THE WALL"
12Ø  FOR J=Ø TO 1Ø
13Ø  PRINT 1Ø-J;
14Ø  PRINT A$;
15Ø  IF NOT 1Ø-J=1 THEN PRINT "S";
16Ø  PRINT
17Ø  PRINT , B$
18Ø  NEXT J
```


The next exercise is equally daft, but it does give practice in playing with strings.

Exercise 16.1. Songwriter

Write a program to print this verse:

```
5 MEN WENT TO MOW
WENT TO MOW A MEADOW
5 MEN
4 MEN
3 MEN
2 MEN
1 MAN AND HIS DOG
WENT TO MOW A MEADOW
```

If you are feeling really confident, turn it into an INPUT loop, ask how many men (1 up to 12), and print the appropriate verse.

INPUT for String Variables

In Chapter 11 we used INPUT to stop the program and enter a number variable. We can do the same with a string variable. Try this program:

```
1Ø  PRINT "TYPE IN YOUR NAME NOW"
2Ø  INPUT N$
3Ø  CLS
4Ø  PRINT
5Ø  PRINT "THANKS VERY MUCH, "; N$;","
6Ø  PRINT
7Ø  PRINT "WHAT A PRETTY NAME"
```

Now you can join in the ancient sport of computer-baiting, otherwise known as 'typing silly names'. Notice that you don't have to type quotes round your name, they were there waiting for you with the cursor "[L]". Now if we add three more lines to the program:

```
8Ø  PRINT
9Ø  PRINT "TYPE NEXT NAME NOW"
1ØØ  GO TO 2Ø
```

we have a string input loop, and you can go on typing names for as long as you like. What then? These loops are very tricky to get out of, because whatever we type in is going to be a legal string.

Whatever you do, don't type EDIT, the result is a complete seize-up! Press BREAK and NEWLINE simultaneously and repeat this until you are out of the loop (the theory is that eventually you press BREAK while the ZX80 is working, and this stops the program).

Exercise 16.2. Form Filling

Write a program to make the user INPUT his Christian name, surname, age in years and home town. Write a sentence on two lines displaying all this information, and then thank him very much.

We Learnt These in Chapter 16

Statements

LET to define a string variable (e.g. LET A$="ABC").
INPUT stops the program to allow the user to insert a string (e.g. INPUT B$).
PRINT to print a string variable (e.g. PRINT C$).

17

Natural Breaks

One of these four things will stop a program running:

(1) It has come to an end.
(2) It has reached a STOP statement.
(3) It has filled up the screen, or produced some other error.
(4) It has reached an INPUT statement.

The last, INPUT, is the only convenient way of stopping in the middle of the program, so that it can be easily restarted. Here is an example:

```
10Ø  LET P=Ø
11Ø  CLS
12Ø  LET P=P+1
13Ø  FOR J=1 TO 84
14Ø  PRINT "PAGE ";P,
15Ø  NEXT J
16Ø  PRINT "PRESS N/L FOR NEXT PAGE"
17Ø  INPUT A$
18Ø  GO TO 11Ø
```

RUN it, and press N/L (short for NEWLINE) several times. These are the important stages of the program:

(1) Clear the screen.
(2) Collect up to one screenful of output.
(3) Give a prompt, and then pause for INPUT so that the current screenful is displayed.
(4) INPUT something so that the program resumes and goes back to 1.

We can INPUT any string we like at Stage 4, but it is enough just to press NEWLINE. This INPUTS 'nothing', technically known as the *null string*.

Branching at an INPUT Pause

Another way of using INPUT is to give the user a choice of two or more branches to different parts of the program:

```
10Ø  PRINT "TYPE YES OR NO"
11Ø  INPUT A$
12Ø  IF A$="YES" THEN GO TO 2ØØ
13Ø  PRINT "YOU TYPED NO"
14Ø  STOP
2ØØ  PRINT "YOU TYPED YES"
```

RUN the program twice and type "YES" and "NO", and make sure it works. Now for the snag — RUN again and type "CODFISH" (or anything else you like). This program depends on having nice obedient users who will follow instructions, but beware! The world is full of vandals whose only idea is to make computer programs 'crash'. Here is an example of a vandal-proof program to do the same job:

```
10Ø  PRINT "TYPE YES OR NO"
11Ø  INPUT A$
12Ø  IF A$="YES" THEN GO TO 2ØØ
13Ø  IF A$="NO" THEN GO TO 3ØØ
14Ø  GO TO 11Ø
2ØØ  PRINT "YOU TYPED YES"
21Ø  STOP
3ØØ  PRINT "YOU TYPED NO"
```

You can't get round that one, can you? At the cost of two extra lines, you have been *made* to obey the instruction, and it's worth looking at the flowcharts for the two programs to see how they work. Remember that it's an imperfect world, though, and with our 1K RAM we can't always spare the space to make our programs crash-proof.

We Learnt These in Chapter 17

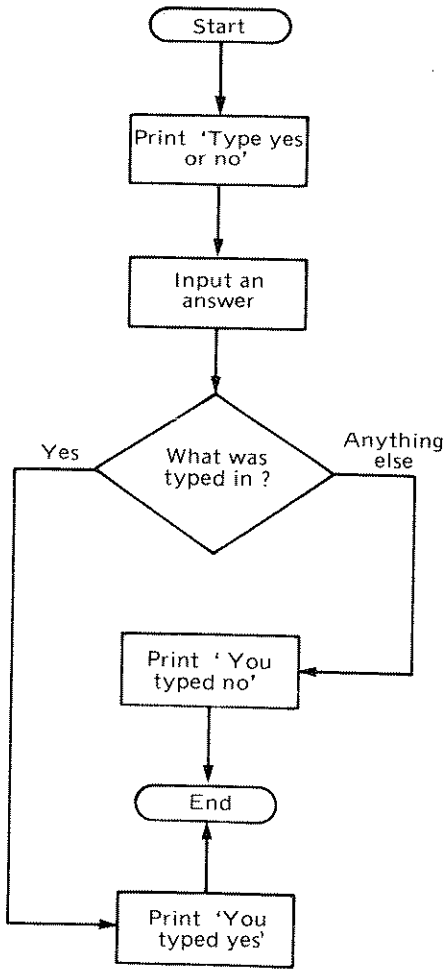
Statements

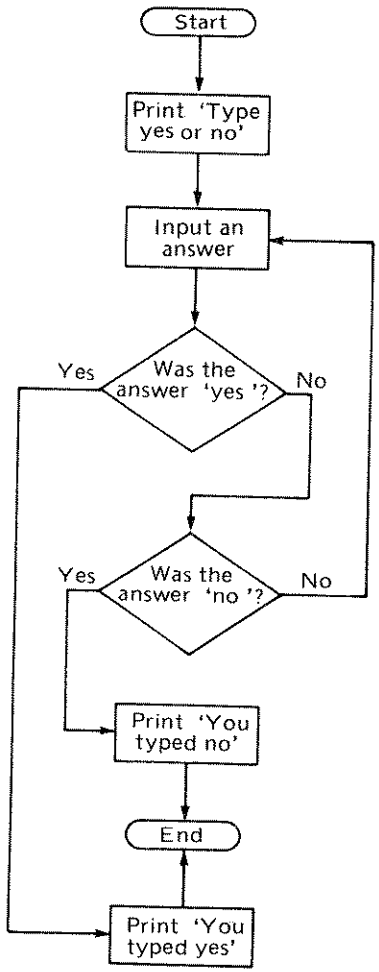
INPUT to make the program stop and display screen contents.

Anything else

Making the program branch, under control of a string INPUT by the user.
'Crash-proof' programs.

The YES/NO program 1. Subject to the attentions of vandals





18

A Matter of Chance

Random Numbers

What are they? Let's consider a random number generator we have all used, the dice. We know that it can only give numbers from 1 to 6, and also that unless it is loaded or very misshapen, each number is equally likely to appear. Since it is an unintelligent cube of wood or plastic it is not affected by anything that has happened before. We can turn these observations into general rules for random numbers:

(1) A random number is one number drawn from any given set of numbers.
(2) Each number in the set is equally likely to be drawn.
(3) The draw is completely unaffected by previous draws.

The ZX80 has a random number function, and we'll use this to throw a dice 12 times:

```
10Ø FOR K=1 TO 12
11Ø PRINT RND(6);" ";
12Ø NEXT K
```

(There is no keyword for RND, it has to be typed in full). RUN the program and write down the 12 numbers, then try a few more RUNs if you like. Certainly they are all between 1 and 6, and they look reasonably random.

Now unplug the power supply for a moment, re-type the program and RUN again. Numbers seem familiar? They are identical to the previous batch! The problem is that the ZX80 is using a fixed 'seed' number and cleverly calculating a list of numbers from this. Since our rule 3 has been broken, we call these *pseudo-random* numbers, and these are what most microcomputers deal in.

Games based on a series of numbers that everyone knows are a bit one-sided, but luckily ZX80 has the answer. Add this line to your program:

```
1Ø  RANDOMISE
```
(RAND = keyword J)

Every time the ZX80 comes to the RAND statement, it picks a new random seed number from its own 'works', and all the pseudo-random numbers in the program come from this seed. RUN again, and you start with a new seed number, and a new set of pseudo-random numbers.

In passing, let us note that you can choose your own seed number (n) between 1 and 32767, by typing RANDOMISE(n).

Let's extend our program now to print out more dice throws:

```
 1Ø  RANDOMISE
 9Ø  FOR J=1 TO 2Ø
1ØØ  FOR K=1 TO 12
11Ø  LET D=RND(6)
12Ø  PRINT D;" ";
13Ø  NEXT K
14Ø  PRINT
15Ø  NEXT J
```

This time we have defined a variable as RND(6), using a LET statement. Of course, we are not limited to numbers from 1 to 6. We could change to different games:

RND(2)	Penny tossing (1 = tail, 2 = head)
RND(2Ø)	Dartboard
RND(36)	Roulette

You'll find that RND(1) is not a great help — try it!

Random Branching

This is a useful little trick to make your program branch in two different ways under control of a random number. The vital statement is of the form:

```
1ØØ  IF RND(2)=1 THEN GO TO 2ØØ
```

RND(2) can only be 1 or 2, so that there is an even chance of going on to line 2ØØ or simply to the next line after 1ØØ. Here is a more elaborate program using this idea:

```
 7Ø  PRINT
 8Ø  PRINT
 9Ø  PRINT
1ØØ  PRINT "YOU SET THE ODDS ON GETTING HOME"
11Ø  PRINT "THE ODDS ARE 1 IN ----"
12Ø  INPUT A
13Ø  CLS
135  LET X=RND(A)
14Ø  IF X=1 THEN GO TO 2ØØ
15Ø  PRINT A-1;" TO 1 AGAINST GETTING HOME"
16Ø  PRINT
17Ø  PRINT "YOU DID NOT MAKE IT"
18Ø  GO TO 7Ø
2ØØ  PRINT "YOU GOT HOME"
21Ø  PRINT
22Ø  PRINT "THE ODDS WERE "; A-1; " TO 1 AGAINST"
23Ø  GO TO 7Ø
```

There is a flowchart for a simpler version of the program using odds of 1 in 3 shown on the next page.

Probability

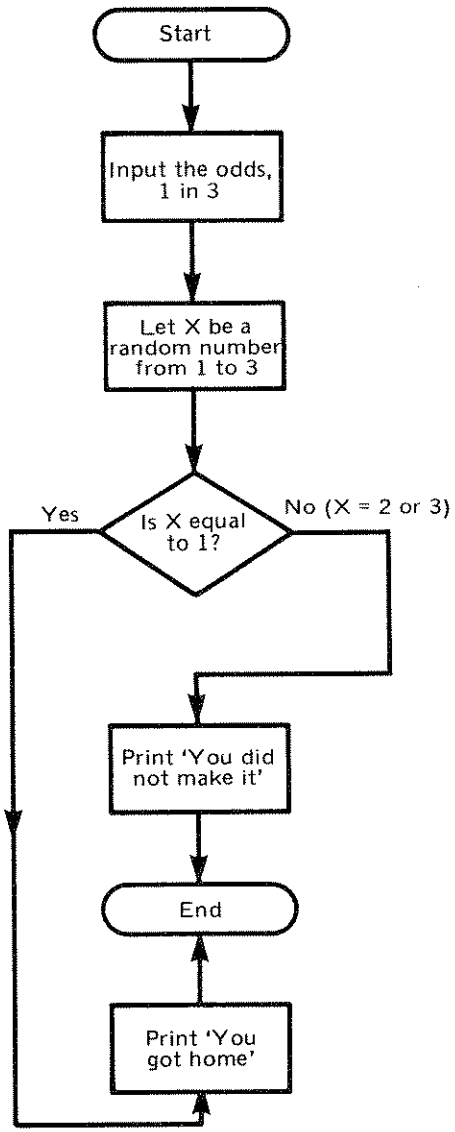
With random numbers under your belt, you can really go to town on programs to study probability. You may want to take the dice throwing programs further, perhaps by calculating and printing out the frequency of all six possible results (there are two dice throwing programs at the end of this book). On the other hand, you could turn your attention to penny tossing, and here is an exercise to start you off.

Exercise 18.1. Penny Tossing

You are tossing pennies two at a time, and score 1 for a tail and 2 for a head. You can get four possible results:

T T (score = 2)
T H (score = 3)
H T (score = 3)
H H (score = 4)

Write a program to toss your pennies 100 times, and add up the number of times you score 2, 3 and 4. Print out the results neatly and see if they are as you expect.



Project

This is a longer-term project with no answer in this book, to design a game of golf for the ZX80. Start with a single hole of random length between 120 and 550 yards. Make the player pick a suitable club say from No. 1 (driver, up to 250 yards) down to No. 10 (putter, up to 5 yards). Give each club a range consisting of a fixed part plus a random part. Fix bunkers if you like, with some kind of penalty. After each shot, loop back for another choice of club for the next shot. When you have the single hole perfected, build another loop around it so that you have an 18 hole course.

We Learnt These in Chapter 18

Statements and Expressions

RND(n) to give a random number from 1 to n inclusive.
RANDOMISE to set the seed number (for generating random numbers) to a random number.

Anything else

Computers usually deal in pseudo-random numbers.
Program branching under control of random numbers.

19

It Draws Pictures as Well?

The Characters

Type this short program and RUN it:

```
1ØØ  FOR J=Ø TO 63
11Ø  PRINT CHR$ (J);"    ";
12Ø  IF (J/8)*8=J THEN PRINT
13Ø  IF (J/8)*8=J THEN PRINT
14Ø  NEXT J
```

Notes: CHR$ must be typed in full.
Lines 12Ø and 13Ø are simply to leave a blank line after each line of 8 characters.

The program will give you a neat print-out of most of the characters available from the ZX80 keyboard. Every character has a code number, and the statement:

```
PRINT CHR$ (J)
```

is saying, 'PRINT the character which has the code number J'. Ø is the code number for a space, and 1 is for the null string — they don't show up. Some are missing:

```
1ØØ  FOR J=212 TO 256
```

would give you all the rest, including keywords and token words.

Now change line 11Ø to read

```
11Ø  PRINT CHR$ (J+128);"    ";
```

(J being still Ø to 63) and RUN again. This time you have the *inverse* of your first 64 characters, white on black instead of black on white.

Your ZX80 Manual shows a complete list of characters, together with their codes, on pages 75 to 77.

Graphic Blocks

Have a look at page 78 of the manual. Ten of these blocks (codes 2 to 11) are available from the keyboard, and the inverse of these (codes 13Ø to 139) via the CHR$ function. Add to these the inverse space (a black square, code 128) and any other characters (inverse or otherwise) which seem useful, and you have the basis for all sorts of simple pictures.

Let's look at some ways of drawing chessboards:

```
1Ø  LET A$="CHESSBOARD I"
2Ø  PRINT A$
3Ø  PRINT
4Ø  FOR J=1 TO 4
5Ø  PRINT "  ▧  ▧  ▧  ▧"
6Ø  PRINT "▧  ▧  ▧  ▧  "
7Ø  NEXT J
1ØØ  PRINT
11Ø  PRINT
12Ø  PRINT A$;"I"
13Ø  FOR K=Ø TO 7
14Ø  FOR L=Ø TO 7
15Ø  IF NOT ((K+L)/2)*2=K+L THEN GO TO 18Ø
16Ø  PRINT "  ";
17Ø  GO TO 19Ø
18Ø  PRINT "▧";
19Ø  NEXT L
2ØØ  PRINT
21Ø  NEXT K
3ØØ  PRINT "PRESS N/L FOR MORE"
31Ø  INPUT B$
32Ø  CLS
33Ø  PRINT A$;"II"
34Ø  PRINT
35Ø  FOR M=1 TO 4
36Ø  PRINT ,"▀▄▀▄▀▄▀▄"
37Ø  NEXT M
39Ø  PRINT
4ØØ  PRINT ,, "AND IV"
41Ø  PRINT
42Ø  PRINT ,,, CHR$(137)
```

Chessboard II (lines 13Ø to 21Ø) is the most useful one, since the squares are drawn one by one, and we can use conditional branches to print other characters on the squares when we choose. This is the basis of the Fox and Hounds game at the end of this book.

The Screen

We'd better have a look at the space that we've got to draw on. A very simple program will fill the screen:

```
1Ø  PRINT "ZX8Ø ";
2Ø  GO TO 1Ø
```

Count along the top line — eight ZX80s equals 32 characters altogether. There are 23 lines from top to bottom, plus the 'screen full' code, but we really ought to leave space free for an INPUT to continue the program. Let's see how much we can squeeze in:

```
1Ø  LET A=5
2Ø  FOR J=1 TO A
3Ø  PRINT "ZX80 ";
4Ø  NEXT J
5Ø  PRINT
6Ø  PRINT "HOW MANY "; CHR$(212); "ZX80 "; CHR$(212);"S
    DO YOU WANT?";
7Ø  INPUT A
8Ø  CLS
9Ø  GO TO 2Ø
```

You'll find that 176 is the maximum — 22 lines of 32 characters. The ; at the end of line 6Ø is needed to squash the input cursors on to the bottom line. Note that we did the 'impossible', printing " on the screen by using CHR$(212).

This is the deepest screen available. Remember that a long program borrows RAM from the screen display department, and reduces the depth of screen accordingly. You will often have to compromise between a long program and a nice display.

The next program is called 'The pyramids were not built in a day':

```
1ØØ  LET N=RND(16)
2ØØ  FOR R=1 TO N
3ØØ  FOR B=1 TO N
35Ø  PRINT CHR$(128);
4ØØ  NEXT B
5ØØ  PRINT
6ØØ  NEXT R
7ØØ  PRINT "PRESS N/L FOR MORE"
71Ø  INPUT A$
72Ø  CLS
73Ø  GO TO 1ØØ
```

Run it and press NEWLINE a few times. By the way, you are now in a string input loop, and we saw in Chapter 16 that you can get out of these by pressing BREAK while the ZX80 is working with a blank screen.

'I had something a bit more pointed in mind' (Pharaoh Cheops speaking).

Right, let's try:

```
3ØØ  FOR B=1 TO N-R+1
```

'You've got it the wrong way up!'

```
3ØØ  FOR B=1 TO R
```

'Much better. Let's have a look at the back'. Add these lines:

```
25Ø  FOR S=1 TO N-R+1
26Ø  PRINT "    ";
27Ø  NEXT S
```

'Keep on trying. I think you're getting nearer to it'. Over to you now, I have not worked out the answer to a real pyramid shape. I do know that you could tidy up the staircase by using CHR$(132) at the beginning of each line of blocks.

Exercise 19.1. Areas of Rectangles

Easy graphics, illustrating a computer-assisted learning program for young sister or brother! Write a program to draw a rectangle with random length and breadth, show its dimensions, ask the user to input area, and check (and if necessary correct) the answer.

We Learnt These in Chapter 19

Expressions

CHR$(n) is equivalent to the character which has the code number n.

Anything else

Inverse characters (white or black) obtainable by using CHR$(n).
Graphic blocks, ten on the keyboard and ten obtainable by using CHR$(n).
Screen size, 22 lines of 32 characters each, plus one line for an INPUT prompt.

20

Playing with Strings

Some programs put words together. You may have read about the theory that, given (a) a number of monkeys with typewriters and (b) infinite time and paper, you would eventually get the complete works of Shakespeare. Let's try it.

```
 1Ø  RANDOMISE
 9Ø  CLS
1ØØ  FOR J=1 TO 8Ø
2ØØ  FOR K=1 TO RND(8)
21Ø  LET A=RND(26)
22Ø  PRINT CHR$(A+37);
3ØØ  NEXT K
31Ø  IF RND(15)=1 THEN PRINT ".";
35Ø  PRINT "     ";
4ØØ  NEXT J
48Ø  PRINT
49Ø  PRINT
5ØØ  PRINT "PRESS N/L FOR ANOTHER PAGE"
51Ø  INPUT A$
52Ø  GO TO 9Ø
```

RUN and press N/L a few times. You may recognise an odd word, but I have seen nothing of literary merit yet!

Heads and Tails

After that diversion, we'll pull a few words apart. Type and RUN this program:

```
 1Ø  PRINT "TYPE ANY WORD NOW"
 2Ø  INPUT A$
 3Ø  LET A=CODE(A$)
 4Ø  PRINT
 5Ø  PRINT "A$=";A$
 6Ø  PRINT
 7Ø  PRINT "CODE(A$)=";A
 8Ø  GO TO 1Ø
```

INPUT the words 'ABLE', 'BAKER', 'CHARLIE'. Check page 76 of the ZX80 Manual where you will see that 38, 39 and 40 are the codes for A, B and C. The expression CODE (string) always gives you the code number for the first character in that string. We can use CODE with a string variable as above, or with a literal string (which must be in quotation marks).

Now we'll change our program to use another expression, TL$:

```
 1Ø  PRINT "TYPE ANY WORD NOW"
 2Ø  INPUT A$
 3Ø  PRINT "A$=";A$
 4Ø  LET B$=TL$(A$)
 5Ø  PRINT "TL$=";B$
 6Ø  PRINT
 7Ø  GO TO 1Ø
```

Once you have RUN this program, you'll be well aware what TL$ does to a string! TL$ (string) chops off the first character and discards it, leaving the rest of the string (TL = tail). Try INPUTting some single character strings, and the null string (press NEWLINE to INPUT nothing). All of these give TL$ = the null string.

Here is a program using CHR$, CODE and TL$ to chop up a word into separate letters and give the code number for each letter:

```
 1Ø  PRINT "TYPE ANY WORD NOW"
 2Ø  INPUT A$
 3Ø  CLS
 4Ø  PRINT A$
 5Ø  PRINT
 6Ø  PRINT CODE(A$), CHR$ (CODE(A$))
 7Ø  LET A$=TL$ (A$)
 8Ø  IF CODE (A$)=1 THEN GO TO 1Ø
 9Ø  GO TO 6Ø
```

Note how we print the code of the first letter, and then the actual letter by using CHR$. Then we discard the first letter by using TL$, and loop back to 6Ø to deal with the next letter. When nothing is left (null string, code = 1) line 8Ø sends you back to the start.

Dealing with Numbers

It is sometimes useful to take a number and turn it into a string, so that we can use these string handling expressions on it. We use the expression STR$ to do this, e.g.

```
1Ø  LET A$=STR$ (A) where A is a number variable
2Ø  LET B$=STR$ (999)
```

(LET B$="999" would give the same result). A$ and B$ are now strings, though if PRINTed they look like numbers. Note that we cannot convert these strings back directly into numbers. However, if we separate them into individual characters by the methods described previously, we find that the codes of these characters are exactly 28 more than the numbers themselves, so that we could laboriously put the number together again. Here is a short program in which you can check what is happening:

```
 1Ø  PRINT "INPUT A STRING OF 10 DIGITS"
11Ø  INPUT A$
12Ø  PRINT
13Ø  PRINT "THE STRING WAS ";A$
14Ø  PRINT
15Ø  PRINT
16Ø  PRINT "CHAR.","CODE OF",,"CODE — 28"
17Ø  PRINT,"THAT CHAR."
18Ø  PRINT
2ØØ  FOR J=1 TO 1Ø
21Ø  LET N=CODE (A$)
22Ø  PRINT CHR$ (N),N,,N-28
23Ø  LET A$=TL$ (A$)
24Ø  NEXT J
```

We Learnt These in Chapter 20

Expressions

CODE (string)	produces the code number of the first character in the string.
TL$ (string)	is equal to the tail obtained by discarding the first character from the string.
STR$ (number or number variable)	changes the number to a string having the appearance of the number.

21

Line 'Em Up!

We are going back mainly to numbers in this chapter.

Dummy Variables

If your program has a simple loop in which a variable (number or string) is generated or INPUT, then each time the ZX80 goes round the loop it overwrites that variable and it is lost.

```
1ØØ  PRINT "TYPE IN A NUMBER"
11Ø  INPUT A
12Ø  CLS
2ØØ  PRINT "YOU TYPED IN ";A
29Ø  PRINT
3ØØ  GO TO 1ØØ
```

Sometimes it happens that we want to keep the old value of the variable to compare with the next one. In that case we make a *dummy variable* of it. Add these lines to the program:

```
 1Ø  LET B=Ø
21Ø  IF A=B THEN GO TO 25Ø
22Ø  PRINT "THATS A CHANGE"
23Ø  PRINT "IT WAS ";B;" LAST TIME"
24Ø  GO TO 27Ø
25Ø  PRINT "SAME AS LAST TIME"
27Ø  LET B=A
```

Now we have built in a short memory for A in the form of the dummy variable B, and in line 21Ø we compare A with B (the previous value of A). We can use dummy string variables in just the same way.

Arrays

ZX80 BASIC allows up to 26 *single dimension arrays*. An array provides a permanent memory for a set of numbers, and we can call up any member of the set at will and do any of the usual operations on it. This is how they work:

```
1Ø  DIM N(3)
5Ø  PRINT N(2), N(1), N(3), N(Ø)
1ØØ LET N(Ø)=12
11Ø LET N(1)=34
12Ø LET N(2)=56
13Ø LET N(3)=78
2ØØ PRINT N(2), N(1), N(3), N(Ø)
```

This gives:

```
 0        0        0        0
56       34       78       12
```

as output. Nothing exciting yet, we could have done the same with ordinary number variables, but notice the DIM statement in line 1Ø, which is saying 'make room for a single dimension array containing four elements and set them all at Ø to begin with'. Each member of an array has the single letter title of the array (N in this case) followed by the subscript number in brackets, which shows which member it is, e.g.

X(3Ø) is the thirty-first member of the X(n) array (remember X(Ø)).

We don't have to use all the members of an array, though there is a small waste of variable space if we don't do so.

Arrays begin to look a bit more promising when we use them in FOR/NEXT loops. Here's a program to illustrate the point.

```
1Ø  DIM S(15)
2Ø  DIM C(15)
1ØØ FOR J=Ø TO 15
11Ø LET S(J)=J**2
12Ø LET C(J)=J**3
13Ø NEXT J
3ØØ PRINT "NUMBER", "SQUARE", "CUBE"
31Ø PRINT
32Ø FOR J=Ø TO 15
33Ø PRINT J, S(J), C(J)
34Ø NEXT J
```

This time we have used the FOR/NEXT control variable J to create

two arrays S(J) and C(J). We have printed the two arrays, and they are still available for use later in the program.

Arrays of Words

Unfortunately we do not have arrays of string variables available in our 4K ROM BASIC, but we can handle an array of single characters by using their code numbers. In this program we INPUT any word, split it into individual letters, and store their code numbers in an array. We also put the original word into a dummy variable (B$) in case we need it later, and make a note of the number of letters in the word (N).

```
2Ø  DIM C(3Ø)
1ØØ PRINT "TYPE IN ANY WORD"
11Ø INPUT A$
12Ø LET B$=A$
2ØØ FOR J=1 TO 3Ø
21Ø LET C(J)=CODE(A$)
22Ø LET A$=TL$(A$)
23Ø LET N=J
24Ø IF CODE(A$)=1 THEN GO TO 35Ø
3ØØ NEXT J
```

Now we can print out the letters of the original word in various ways. Try this:

```
35Ø CLS
38Ø PRINT B$
39Ø PRINT
4ØØ FOR K=1 TO N
41Ø PRINT CHR$(C(K))
42Ø NEXT K
```

Notice that in this second loop we are using C(K) to call for the members of the array. It is only the C and the number in the bracket which matters — how the number gets there does not matter. Try varying the punctuation in line 41Ø to change the printout, and then see if you can predict how this will print your name:

```
41Ø PRINT CHR$ (C(N-K+1)+128)
```

Finally, here is a useful program to pick the items out of an array at random. If you tack it on to the last program (lines 2Ø to 3ØØ) it will write an anagram of your original word and invite a second player to guess it.

```
 3Ø  LET L=Ø
35Ø  CLS
38Ø  PRINT "ANAGRAM"
39Ø  PRINT
4ØØ  LET X=RND(N)
42Ø  IF C(X)=1 THEN GO TO 4ØØ
43Ø  PRINT CHR$ (C(X));"  ";
44Ø  LET L=L+1
45Ø  LET C(X)=1
46Ø  IF L<N THEN GO TO 4ØØ
5ØØ  PRINT
51Ø  PRINT
52Ø  PRINT
53Ø  PRINT "MAKE A GUESS AT THE WORD"
54Ø  INPUT C$
55Ø  PRINT
56Ø  PRINT C$;
57Ø  IF C$=B$ THEN GO TO 6ØØ
58Ø  PRINT " IS WRONG. ANOTHER GUESS?"
59Ø  GO TO 54Ø
6ØØ  PRINT " IS RIGHT. WELL DONE"
```

The program is best explained by the flowchart on the next page.

Here are a couple of exercises on arrays.

Exercise 21.1. Moving Average

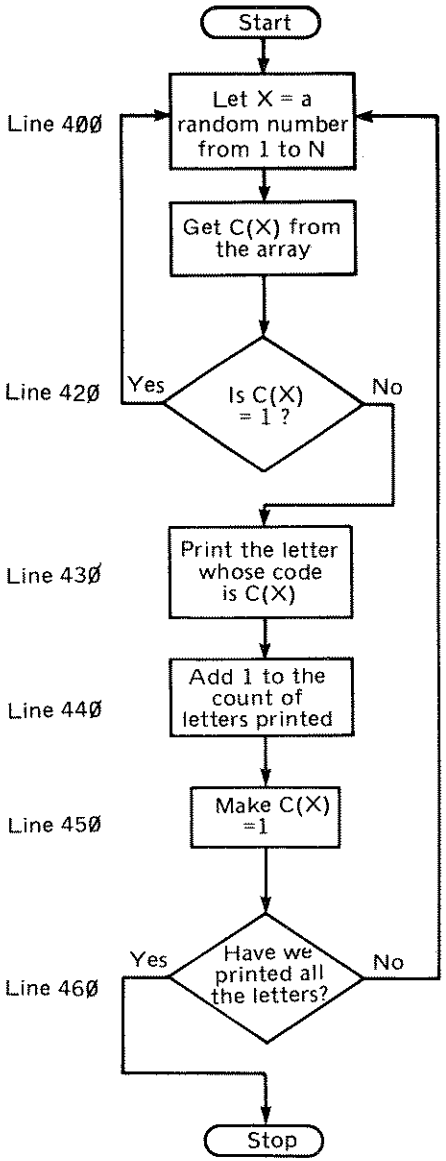
The statistician's delight! Your program has got to allow the input of six items of data one by one — they could be monthly sales figures, for example. The six items must be displayed, and their average calculated and displayed. Now we have to input a seventh item, and since we are only interested in the six most recent items, we have to throw out the first. Tricky stuff this, so here's a clue — put the seven items in an array, and then move each item one place down in the array. You can do it very simply by using a FOR/NEXT loop.

Now that you have your five old items plus one new one in your array, display them, calculate the new average and display it again. Continue with the eighth item, the ninth and so on. . . .

If you want to make the program really useful, add some lines so that you can vary the number of items which you are averaging on each run. Now you really have the ZX80 working for you.

Anagram — random sorting of an array

C(J) is a numerical array containing N numbers. Each member of C(J) is the code for a letter in a word of N letters.



Exercise 21.2. Simple Bulls and Cows

The Bulls and Cows program at the end of this book is rather wastefully programmed using numerical methods. Now is your chance to make a start on improving it. Write a program to do these operations:

Generate four random digits between 1 and 6
Store them in an array
Ask the player for his guess
Input the guess as a string variable
Split the string into characters
Store their codes as another array
Compare the two arrays and score a bull for each correct digit
Print out the result

Take great care, you'll be dealing with numbers as numbers, numbers as strings and numbers as codes of strings!

There is one way in which we can produce the equivalent of a string array.

```
10Ø  PRINT "WHICH GIRL, 1, 2 OR 3?"
11Ø  INPUT A
12Ø  CLS
13Ø  PRINT "GIRL NO. ";A;" IS CALLED ";
14Ø  GO TO 1ØØØ+1Ø*A
15Ø  PRINT A$
16Ø  STOP
1Ø1Ø  LET A$="LIZA"
1Ø15  GO TO 15Ø
1Ø2Ø  LET A$="BETSY"
1Ø25  GO TO 15Ø
1Ø3Ø  LET A$="BESS"
1Ø35  GO TO 15Ø
```

Here the three names are firmly linked to the numbers 1, 2 and 3, which could be members or subscripts of an array.

We Learnt These in Chapter 21

Statements

DIM X (n) — Reserves space for a single dimension numerical array with n + 1 members, and sets all the values in the array to Ø.

Anything else

Dummy Variables — used to keep the value of some variable when we want to give that variable a new value.
Working with strings — producing the equivalent of character and string arrays.

22

Just off to the Shops — Back Soon

By now, you will have got the message that a BASIC program can be broken down into a number of separate operations, like the boxes in a flowchart. Suppose we want to do one of these operations several times in the course of a program, in such a way that we cannot include it in a loop. In that case we turn the operation into a *subroutine*, so that we can go and do it as often as we like. It is traditional, and very much tidier, to put all your subroutines at the end of your program. It's also a help to label them with a REMark if you have memory to spare.

Here is a very simple program using two subroutines. We'll write it in stages so that you can see how they work.

```
10  PRINT "SUBR DEMO"
11  PRINT
12  PRINT "JUST OFF TO SUBR 2000"
13  GO SUB 2000
20  PRINT "ON MY WAY TO SUBR 3000"
21  GO SUB 3000
22  PRINT
30  FOR J=2 TO 3
31  PRINT "LEAVING THE LOOP FOR SUBR ";J*1000
32  GO SUB J*1000
33  NEXT J
(GO SUB is keyword V)
```

RUN it now to see what happens. Lines 10 to 12 are obeyed, producing the printout you see. Line 13 has said, 'go directly to line 2000 and do whatever is there'. Since there was nothing at or after line 2000, the program stopped there. We'll put the subroutines in now:

```
2000  PRINT "THAT WAS SUBR 2000"
2010  PRINT
3000  PRINT "I HAVE BEEN AT SUBR 3000"
3010  PRINT
```

RUN again — this time it executes *both* subroutines once, stopping at line 3010. Still more needed — take note of this:

Every *subroutine* needs a *RETURN*.

So we must type in:

```
2020  RETURN
3020  RETURN
(RETURN is keyword B)
```

Try again — we're nearly there. It did each subroutine twice, as expected, then an odd one at the end, and stopped with a 7/2020 error code. What was wrong? We crashed into our subroutines — that's what. After doing lines 10 to 33, and GO SUBing and RETURNing according to instructions we went on to 2000 and 2010, found an unexpected RETURN at line 2020 and stopped with a 7/n error (RETURN without GO SUB). One line:

```
1000  STOP
```

will put that right, and the program is now complete. RUN it to make sure. The 9/1000 error code shows that the program has stopped at a STOP Statement in line 1000.

We have seen that GO SUB behaves in much the same way as GO TO plus an automatic return to where it came from. Let's write down some rules for GO SUB . . . RETURN.

(1) On GO SUB n the ZX80 goes immediately to line n (or to the line following if no n exists). n may be a number, a variable or an expression.
(2) The ZX80 executes the subroutine just as though it was part of the main program.
(3) A subroutine must include a RETURN statement, which sends the ZX80 back to the *line following* the original GO SUB n.
(4) You may jump out of one subroutine into another by using GO SUB n, but don't do it with GO TO, that will cause chaos!
(5) A valid and useful statement is

    IF something is true THEN GO SUB n.

(6) Use STOP as a fence between your main program and your subroutines, to avoid 'RETURN without GO SUB' errors.

Now for a simple program to calculate the volume of a water tank.

One of two subroutines is called up, depending on the shape of the tank.

```
 10  LET V=Ø
 9Ø  CLS
 1ØØ  PRINT ''VOLUME OF A WATER TANK''
 11Ø  PRINT
 12Ø  PRINT
 13Ø  PRINT ''WHAT SHAPE IS IT?''
 14Ø  PRINT
 15Ø  PRINT, ''CYLINDER (CYL FOR SHORT)''
 16Ø  PRINT, ''CUBE''
 17Ø  INPUT A$
 18Ø  CLS
 2ØØ  IF A$=''CYL'' THEN GO SUB 1ØØØ
 21Ø  IF A$=''CUBE'' THEN GO SUB 11ØØ
 22Ø  IF NOT V=Ø THEN GO TO 3ØØ
 23Ø  PRINT ''DONT KNOW '';A$;'' SHAPE''
 24Ø  GO TO 12Ø
 3ØØ  PRINT
 31Ø  PRINT
 32Ø  PRINT ''VOLUME OF '';A$;'' = '';V; '' CUBIC CM''
 9ØØ  STOP
1ØØØ  REM **VOL OF CYL**
1Ø1Ø  PRINT ''HEIGHT IN CM? '';
1Ø2Ø  INPUT H
1Ø3Ø  PRINT H
1Ø4Ø  PRINT
1Ø5Ø  PRINT ''DIAMETER IN CM? '';
1Ø6Ø  INPUT D
1Ø7Ø  PRINT D
1Ø8Ø  LET V=(314*(D/2)/1ØØ)*(D/2)*H
1Ø9Ø  RETURN
11ØØ  REM **VOL OF CUBE**
111Ø  PRINT ''EDGE LENGTH IN CM? '';
112Ø  INPUT E
113Ø  PRINT E
114Ø  LET V=E**3
115Ø  RETURN
```

It is worth noticing how, after PRINTing the prompt (e.g. in line 111Ø) we PRINT the INPUT value on the same line.

We Learnt These in Chapter 22

Statements

GO SUB n sends the ZX80 to execute a subroutine at the end of the program, at line n.

RETURN. When the ZX80 reaches a RETURN statement in a subroutine, it immediately goes back to the line after GO SUB.

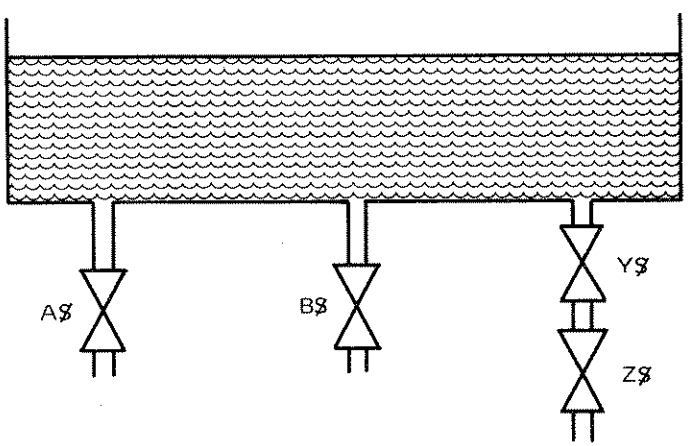
STOP. Prevents the ZX80 from crashing into subroutines after executing the main program.

23

It's Ever so Logical

We used a lot of logic in Chapter 9, with IF . . . THEN and NOT. There's much more to come!

Here is a picture of a water tank with some weird plumbing; there are four water taps labelled A$, B$, Y$ and Z$.



It's a chemical engineering problem, in which we have to write a program to ring an alarm bell to tell us when we have left a tap open. Let's deal with A$ first:

```
10 PRINT "SET YOUR TAPS NOW"
20 PRINT, "O=OPEN S=SHUT"
30 PRINT
40 PRINT "A$ IS? ";
50 INPUT A$
60 PRINT A$
200 IF A$="O" THEN GO TO 1000
500 PRINT
510 PRINT "EVERYTHING O.K."
600 PRINT
610 PRINT "PRESS N/L FOR MORE OR S FOR STOP"
620 INPUT N$
630 IF N$="S" THEN STOP
640 CLS
650 GO TO 10
1000 PRINT
1010 PRINT "CLANG-A-LANG"
1020 PRINT
1030 PRINT "WATER RUNNING AWAY"
1040 GO TO 600
```

RUN it, and open and close A$ to make sure you get the right answers. Now we'll add B$:

```
70 PRINT "B$ IS? ";
80 INPUT B$
90 PRINT B$
```

We need the equivalent of line 200 for B$, but wait . . . we can include it *in line 200*. With A$ and B$ we have an either/*or* situation, only one of them needs to be open to ring the alarm. Change line 200 to read:

```
200 IF A$="O" OR B$="O" THEN GO TO 1000
```

Did it work? Like a charm! The alarm rings if either A$ or B$ is open (and note that it also rings if both are open).

Now for Y$ and Z$. We'll write in the inputs first:

```
100 PRINT "Y$ IS? ";
110 INPUT Y$
120 PRINT Y$
130 PRINT "Z$ IS? ";
140 INPUT Z$
150 PRINT Z$
```

Now have a look at the right hand pipe. What happens if Y$ is opened? Precisely nothing! The same applies to Z$. It is a 'belt and braces' situation, because both Y$ *and* Z$ must be open for the alarm to ring. Type in the test for this occurrence:

```
210 IF Y$="O" AND Z$="O" THEN GO TO 1000
```

RUN the program and test it, and then combine lines 2ØØ and 21Ø in this new line 2ØØ:

```
2ØØ  IF A$="O" OR B$="O" OR Y$="O" AND
     Z$="O" THEN GO TO 1ØØØ
```

(line 21Ø deleted)

which works just as well. There is a flowchart for this program opposite.

Priorities

These long logical statements need very clear thinking. They depend on the fact that the ZX80 tests all the statements in a specific order:

NOT (higher priority)
AND
OR (lower priority)

This is rather like the performance of the arithmetical operations in a fixed order (Chapter 8). Again we can change the order or emphasise it, by using brackets. For instance, this line:

```
2ØØ  IF A$="O" OR B$="O" OR (Y$="O" AND Z$="O")
   THEN GO TO 1ØØØ
```

has exactly the same effect as before, but is perhaps easier to understand.

Exercise 23.1. The Water Tank

We are scrapping our water tank (never did like that plumbing system!) and putting in a new one with one outlet pipe (branched) fitted with three taps, A$, B$ and C$. Change the input lines to fit these three taps, and then type in this logic line:

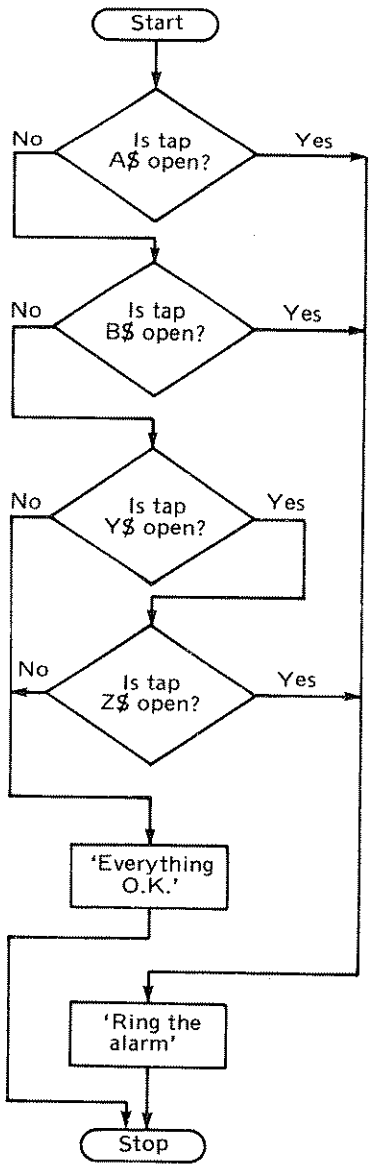
```
2ØØ  IF A$="O" AND (B$="O" OR C$="O") THEN GO TO
     1ØØØ
```

RUN the program with various combinations of open taps and deduce the new layout of pipes and taps.

Exercise 23.2. The Cashbox

Our town council is run by two pairs of brothers, Bob and Tom Jones, and Bill and Dick Brown. The town funds are in a large steel

Water tank — is the water running away?



chest, and since the two families trust each other not one inch, they fixed it with four combination padlocks, one of which could be opened by each man.

This turned out to be highly inconvenient since they were hardly ever present all at once. So, they called on the logical town blacksmith to work out a system of hasps and padlocks, so that the chest could be opened by either of the Jones family *plus* either of the Brown family (the padlocks and combinations were unchanged). Write a program to call the council roll and announce whether the chest can be opened. Can you sketch the arrangement which the blacksmith might have made?

Science of Cambridge now offer a plug-in expansion pack giving an extra 16K of RAM (see Appendix 5).

Logical Values

There is nothing in this last section which can not be done by the use of IF, THEN, AND, OR and LET, so you can skip it for the present if you wish. However you'll need to come back to it when you are writing long programs and beginning to run short of RAM space.

Type this program:

```
10  PRINT "LOGICAL VALUES"
20  PRINT
30  PRINT
40  PRINT "A=";
50  INPUT A
60  PRINT A
100 PRINT
110 PRINT "(A=10) IS "; A=10
120 PRINT "(A<50) IS "; A<50
130 PRINT "(A>90) IS "; A>90
200 PRINT
210 PRINT "(A=10 AND 77) IS "; A=10 AND 77
220 PRINT "(A<50 AND 98) IS "; A<50 AND 98
230 PRINT "(A>90 AND 123) IS "; A>90 AND 123
300 PRINT
310 PRINT "(A=10  OR 13) IS "; A=10 OR 13
320 PRINT "(A<50 OR 44) IS "; A<50 OR 44
330 PRINT "(A>90 OR 999) IS "; A>90 OR 999
600 PRINT
610 PRINT
620 PRINT "PRESS N/L FOR MORE"
630 INPUT A$
640 CLS
650 GO TO 10
```


Now try running the program and inputting various values of A to satisfy the different conditions:

A=10 A<50 A>90

Each of these three expressions can be *true* or *false*, depending on what the value of A is. Look what happens when we put the value of 10 into A.

Expression	True or false when A=10	Logical Value
A=10	True	−1
A<50	True	−1
A>90	False	0

We can also combine expressions like these with a number, using AND or OR, giving effects which are shown in the program above. Let's summarise all these logical values.

```
True  =−1
False=0
True  AND x=x
False AND x=0
True  OR x=−1
False OR x=x
```

A useful way to use these is in a LET statement:

```
LET Z=(A=10)
```

This is equivalent to saying:

```
IF A=10 THEN LET Z=−1
IF NOT A=10 THEN LET Z=0
```

Similarly:

```
LET X=(A<50 OR 99)
```

which is equivalent to:

```
IF A<50 THEN LET X=−1
IF NOT A<50 THEN LET X=99
```

If your head is spinning, I suggest that you note the fact that these logical values exist, and come back to this chapter later. It will save you a lot of program space!

We Learnt These in Chapter 23

Logical statements NOT, AND, OR to use with our IF . . . THEN statement.

Priorities (NOT>AND>OR) used in working out logical statements.
Logical values
True = −1 False = Ø
True AND x = x False AND x = Ø
True OR x = −1 FALSE OR x = x

24

Thanks for the Memory

Binary Arithmetic

Computers always work in binary arithmetic, in which every digit must be 0 or 1. The ZX80 memories consist of a large number of 'pigeon-holes' or *bytes*, each containing 8 binary digits or *bits*. So, one single byte can contain numbers from:

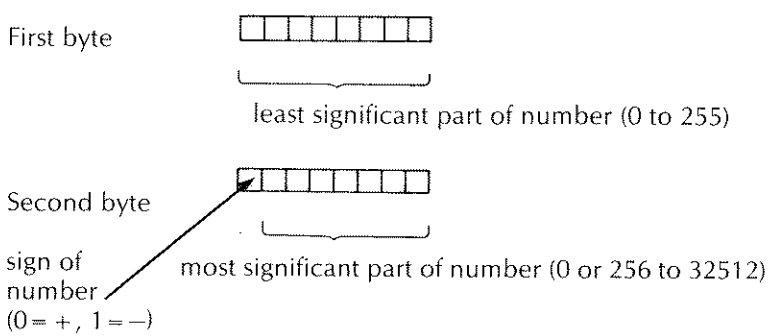
0 0 0 0 0 0 0 0 (0 in decimal system)

up to:

1 1 1 1 1 1 1 1 (255 in decimal system).

Now you see why the ZX80 character set contains 256 items! Each one of them can be stored as a code number in one single byte of memory.

Numbers need two bytes of storage:



Hence, two bytes will hold a number in the range ± 32767, already mentioned in Chapters 7 and 8.

Every byte in a computer memory is numbered so that it can easily be found and its contents inspected or changed. This number is called the *address* of the byte.

ZX80 Memories

The first slice of memory consists of about 4000 bytes of *read only memory* (4K of ROM). This contains the BASIC interpreter, the character set, the operating system and monitor, all needed to run the BASIC programs that are typed in. ROM can be read and used, but it cannot be changed in any way.

The next slice of memory is about 1000 bytes of *random access memory* (1K of RAM). This is the part which accepts the program and variables as they are typed in, and unlike ROM, all the contents are lost when when the ZX80 is switched off. The layout of the RAM is described in some detail in the ZX80 Operating Manual. The first 40 bytes contain the system variables (temporary records needed by the ZX80, see Appendix III of the Operating Manual). The programs start at address 16424, and immediately following the program come the various spaces taken up by the variables, the working space, the display file and the stack. As more program is typed in, so the spaces move along to make room for it. Eventually, the variable space or the display file spaces become so cramped that the program stops with an error code 4/n (no room to add more variables) or 5/n (no more room on screen).

What's In a Byte?

PEEK is an aptly named expression — it lets you look into any byte of memory and discover what is stored there. Its general form is PEEK (n) where n is the 5 digit address of the byte being PEEKed. PEEK (n) is a number from 0 to 255, and it can be used just like any other number or variable, e.g.

```
PRINT PEEK (n)
LET X=PEEK (n)
IF PEEK (n)=29 THEN . . .
PRINT CHR$ (PEEK (n))
```

Here is an example of the use of PEEK, using the fact that address 16420 contains the position on the line of the next character to be printed, and 16421 the position of the current line on the screen. It is a subroutine which will print a title word anywhere you like on the screen.


```
10 PRINT "LINE NO. (1 TO 22)?"
20 INPUT L
30 PRINT "CHARACTER NO. ON LINE (1 TO 32)?"
40 INPUT S
50 CLS
60 GO SUB 1000
70 PRINT "CAT"
900 STOP
1000 IF PEEK (16421)+L=24 THEN GO TO 1030
1010 PRINT
1020 GO TO 1000
1030 IF PEEK (16420)+S=34 THEN RETURN
1040 PRINT " ";
1050 GO TO 1030
```

PEEKing into the middle of your program is no use, because it only takes one change in a program line to shift the whole program a byte or two along in memory. However, here is a program which PEEKs the first line of a program to give the equivalent of a string array. Type line 1 very carefully with the spaces, numbers and punctuation exactly as shown.

```
1 REM 1TOM,2DICK,3HARRY,4JOHN,5CHARLIE,
20 DIM P(5)
50 PRINT "MAX. MARK ? ";
60 INPUT MX
70 PRINT MX
80 PRINT "ENTER MARK FOR EACH BOY"
90 PRINT
100 PRINT
210 PRINT "NAME","PER CENT"
220 PRINT
230 FOR N=1 TO 5
240 GO SUB 1000
250 INPUT M
260 LET P(N)=M*100/MX
270 PRINT,P(N)
280 PRINT
290 NEXT N
900 STOP
1000 LET A=16426
1010 IF PEEK(A)=CODE(STR$(N)) THEN GO TO 1100
1020 LET A=A+1
1030 GO TO 1010
1100 LET A=A+1
1110 LET A$=CHR$(PEEK(A))
```

```
112Ø  IF A$="," THEN RETURN
113Ø  PRINT A$;
114Ø  GO TO 11ØØ
```

It's a gift for busy teachers! Line 1 contains the class list as a REM statement. The FOR/NEXT loop in lines 23Ø to 29Ø fixes N, the subscript number of the boy, and goes to subroutine 1ØØØ to PEEK the boy's name from line 1, character by character, and PRINT it. RETURN to the loop, pause to input the mark, convert it to a percentage and PRINT it.

POKEing into Memory

As well as PEEKing into memory, you can POKE your own choice of value into any chosen memory byte. BEWARE!! POKEing into the wrong places can upset the ZX80, so that you have to switch off to clear the RAM. The general statement is:

POKE n, m

where n is the 5 digit address of the byte to be POKEd, and m is a number between 0 and 255, the value we are POKEing.

The ZX80 Operating Manual shows a reaction timer program on page 88, which uses the T.V. frame counter at addresses 16414/16415 and sets it to zero by two POKE statements. Here is a program which uses the same idea to set up a tables test. You decide how many seconds you need to answer each question — and if you can get down to three seconds you're not doing badly!

```
1Ø  LET S=Ø
2Ø  LET R=Ø
3Ø  LET W=Ø
1ØØ  PRINT "TIMED TABLE TEST"
11Ø  PRINT
12Ø  PRINT "HOW MANY SECS FOR EACH QUESTION?"
13Ø  INPUT A
19Ø  FOR J=1 TO 1Ø
2ØØ  LET B=RND(12)
21Ø  LET C=RND(12)
22Ø  PRINT
23Ø  PRINT
24Ø  PRINT
25Ø  PRINT
27Ø  PRINT B;" X ";C;" = ?"
28Ø  POKE 16414,Ø
29Ø  POKE 16415,Ø
```



```
3ØØ  INPUT P
31Ø  CLS
32Ø  IF (PEEK(16414)+PEEK(16415)*256-4)>A*5Ø THEN GO
     TO 5ØØ
33Ø  IF P=B*C THEN GO TO 4ØØ
34Ø  PRINT "WRONG",B;" X ";C;" = ";B*C
35Ø  LET W=W+1
36Ø  GO TO 52Ø
4ØØ  PRINT "RIGHT"
41Ø  LET R=R+1
42Ø  GO TO 52Ø
5ØØ  PRINT "TOO SLOW"
51Ø  LET S=S+1
52Ø  PRINT
53Ø  PRINT
54Ø  NEXT J
6ØØ  CLS
61Ø  PRINT "YOU GOT ";R;" RIGHT OUT OF 1Ø"
62Ø  PRINT
63Ø  PRINT
64Ø  PRINT W;" WRONG"
65Ø  PRINT,"AND ";S;" TOO SLOW"
7ØØ  PRINT
71Ø  PRINT"PRESS N/L FOR MORE OR S TO STOP"
72Ø  INPUT A$
725  IF A$="S" THEN STOP
73Ø  CLS
74Ø  GO TO 1Ø
```

We Learnt These in Chapter 24

Statements and Expressions

PEEK (n)	to determine the contents of the memory byte at address n.
POKE n, m	to insert the value m into the memory byte at address n.

25

Debugging Your Programs

It is an achievement of note if you can write a program of any length which runs first time without any errors. It is always likely that there are some errors or 'bugs' to get rid of before it will run properly.

Syntax Errors

Generally the ZX80 will not allow this kind of mistake. A quotation mark left out, an odd number of brackets, a number variable equated with a string variable, and [S] will come up and prevent the line from being entered. Note that leaving out *both* quotation marks is not a syntax error unless the proposed string has a space in it, it will be entered as a variable. It is always worth keeping an eye on the screen to make sure that lines *do* enter when you press NEWLINE, otherwise you will waste time trying to tack your next line on to the end of the previous line with a syntax error.

Error Codes

All the lines of the program may have been entered successfully, but it may stop running because of other errors. The ZX80 helps you by showing the line number which caused the crash, and telling you what kind of error it is.

1/n *Wrong NEXT*

The loop control variables at each end of a FOR/NEXT loop do not match, or there is a NEXT without a corresponding FOR.

e.g.
```
1Ø  LET M=5
1ØØ FOR J=1 TO 1Ø
2ØØ NEXT M
```


With or without line 1ØØ this gives a 1/2ØØ error code. If M is not defined as in line 1Ø, a 2/2ØØ error code will be given.

2/n *Variable not defined*

Before it is used, a variable must be defined by one of these statements:

INPUT A or LET A=n (numbers)
INPUT B$ or LET B$="s" (strings)
DIM C(n) (arrays, sets all members to Ø)
FOR D=n TO m (loop control variables)

3/n *Subscript errors*

Errors to do with the subscripts in an array. The most common error is to try to use a subscript outside the range defined by your DIM statement.

e.g.
```
1Ø  DIM X(2) allows for X(Ø), X(1), X(2)
2ØØ LET X(4)=5Ø
```
gives a 3/2ØØ error.

A(O) for A(Ø) or A(I) for A(1) also give this error.

4/n *Variable space full*

This occurs at a line which is trying to define a variable as in 2/n above, and shows that the variable space in RAM is full up. Here are some ideas to make room for more or bigger variables:

(1) Check that you are making full use of your DIM statements, including use of the X(Ø) members of your arrays.
(2) See whether you can cut down on the number of variables — maybe there are dummy variables which are unnecessary.
(3) Shorten your string variables — use abbreviations.
(4) Use only one string variable for all your INPUT pauses.
(5) Cut down the general length of your program (see next section) to make room for more variables.

5/n *Screen full*

There are two possible remedies for this error:

(1) Reduce the screen contents (if you have the full depth of screen available this is the only thing you can do). You can cut out empty or unnecessary PRINT lines, break up your display into smaller slices by using INPUT pauses followed by CLS, or reduce the depth of your graphics display.
(2) Increase the screen depth. If your available screen is too small for the display you want, you'll have to chop your program to make more room on the screen. Here are some ideas:

*Remove REM lines
*Cut out or abbreviate literal strings and string variables
*Make sure you are not wasting any variable space (see last section)

*Look for any repetitions in your program — do these as loops or subroutines.

6/n *Arithmetic overflow*
Maybe one of your final answers is too big — in this case you'll have to change your program or limit the range of INPUT variables. If it is an intermediate value which has gone over the top, you may be able to juggle with the order of operations to keep it within ±32767.

7/n *RETURN without GO SUB*
Generally caused by accidentally entering a subroutine — remember that you must fence off your subroutines with STOP or something else.

Errors Which Do Not Stop the Program

You may have a program which seems to run perfectly, but which prints out rubbish — it all depends on the instructions *you* have put in! With many programs it is obvious if the output is not correct or sensible, others are less obvious and need careful checking. If the output is incorrect, you will have to find out *why*. Here are some ideas:

(1) Check your program where possible by putting in data with a known answer.
(2) Check your answers with a hand calculator.
(3) Check punctuation carefully, especially when you are having trouble with tables of results or graphics displays.
(4) Check any conditional statements by putting in data which first satisfies, then does not satisfy, the condition.
(5) Follow the course of your loops carefully (especially nested loops), preferably using a flowchart.
(6) Put in temporary PRINT lines (use line numbers ending in 9 so that you can easily spot and remove them later), so that you can see the values of your variables at different points in the program.
(7) Check different parts of your program separately.
(8) Put in temporary STOP statements and then use command PRINT to find the values of your variables.

(9) Use command CONTINUE after a STOP statement to make the program continue running.
(10) Check later parts of your program by using RUN n, which starts to RUN the program at line n. RUN or RUN n clear out all variables, so you may need to insert values for the variables and then use command GO TO n (equivalent to RUN n but does not clear variables).
(11) If you are doubtful as to what the values of your variables *are*, use command CLEAR to empty the variable space completely.

We Learnt These in Chapter 25

Commands

CLEAR completely clears all variables (also used as a statement). RUN or RUN n automatically CLEAR as well.
RUN n starts the program at line n after clearing variables.
GO TO n starts the program at line n without clearing variables.
CONTINUE (CONT, keyword T) restarts the program after a STOP statement.

Appendix 1

ZX80 BASIC in 4K ROM

A complete list of all the BASIC instructions available from the ZX80 keyboard. Although the ZX80 will accept any of the 22 keywords as both commands and statements, the lists are confined to those which are likely to be useful.

Note: m and n represent integers.
s represents a string.
J represents a loop control variable.
A and B$ represent variables.

Commands Used in Writing and Editing Programs

EDIT — brings a line (indicated by the current line pointer) to the bottom of the screen for editing. Also a useful way of clearing the bottom of the screen of rubbish!

↑ — moves the current line pointer one line up.

↓ — moves the current line pointer one line down.

← — moves the cursor one character to the left.

→ — moves the cursor one character to the right.

RUBOUT — deletes the character to the left of the cursor.

HOME — moves the current line pointer to the position above the beginning of the program.

LIST — displays the first 22 program lines and moves the current line pointer HOME.

LIST n — displays line n with up to 21 adjacent lines, and moves the current line pointer to line n.

transfers a numbered and valid program line from the bottom of the screen into the program.

(2) causes the ZX80 to execute any command typed on the bottom of the screen.

System Commands

Keyword instructions which do not form part of the program, but are keyed in and executed by pressing NEWLINE.

BREAK — interrupts the ZX80 when it is working with a blank screen. Error code Ø/n shows where the program stopped, and any PRINT statements up to line n are executed.

CLEAR — deletes all variables.

CONT — =CONTINUE. Restarts the program after STOP or BREAK. Any screen contents up to the STOP/BREAK line are lost.

GO TO n — starts the program at line n without deleting any variables.

LOAD — loads a new program from cassette recorder into RAM. Some or all of the new program is displayed when loading is complete.

NEW — deletes the existing program.

PRINT — prints on the screen whatever follows the PRINT command.

RUN — deletes all variables and starts the program at the first line.

RUN n — deletes all variables and starts the program from line n.

SAVE — transfers the program and variables from RAM into cassette recorder. When SAVEing is complete, program is displayed on the screen.

Control Statements

These are keywords which form part of the program and determine the way in which the program is run.

CLEAR — clears screen and deletes all variables.

CLS — clears screen only.

FOR J=n TO m — starts a FOR . . . NEXT loop, setting the loop control variable initially at n. If m is equal to or less than n the loop is entered once, otherwise the loop is entered m−n+1 times.

GO SUB n — jumps to a subroutine at line n, and continues from there until RETURN is reached.

GO TO n — jumps to line n of the program and continues from there.

IF . . . THEN . . . conditional statement IF . . . followed by THEN plus any valid keyword, which is executed if the condition is met. If condition not met, program continues at the line following. Most frequently used for conditional jumps, e.g. IF J>25 THEN GO TO 1ØØ.

INPUT stops the program so that the user can input a value to a numerical or a string variable.

NEXT J ends a FOR . . . NEXT loop. Error if no corresponding FOR J = n to m exists.

RET = RETURN. Used in a subroutine and jumps back to the line following the previous GO SUB n. Error if no previous GO SUB n.

STOP stops the program and executes any PRINT statements. Command CONT restarts program with a clear screen.

THEN *see* IF

TO *see* FOR

Other Keyword Statements

These also form part of the program.

DIM A(n) sets up a single dimension numerical array: A(Ø), A(1), A(2), . . . A(n) and sets all values to Ø.

LET assigns a value to a numerical or a string variable. e.g. LET A = 2 LET B = 3*A LET B$ = "XYZ"

POKE n,m puts the value m into the memory address n.

PRINT prints on the screen whatever follows PRINT, which may be one or more of the following: number, numerical variable, expression, literal string, string variable.

RAND = RANDOMISE. Sets a random number as the seed value for future RND(n) expressions.

RAND n sets n as the seed value.

REM indicates a remark, no action to be taken by the computer.

Expressions

These are all integral functions, which have to be typed out in full. Generally used in programs, though they can form part of a command.

Numerical

ABS(n) the absolute value of n.

PEEK(n) the value currently stored at memory address n.

RND(n) a pseudo-random number between 1 and n.

USR(n) calls a machine code structure at address n (machine code programs are outside the scope of this book).

String handling expressions

CHR$ (n) the character which has the code n (null string if n = 1).

CODE (s) the code number for the first character in the string s.

STR$ (n) the number n converted to a string so that the various string handling techniques can be used on it.

TL$ (s) the string s minus its first character. If s is null or has one character, TL$ (s) is the null string.

Logical Operators

The logical operators, NOT, AND, OR, are used with conditional statements.

e.g. IF NOT A = 1ØØ THEN . . .
IF A = 2 AND B$ = "XYZ" THEN . . .
IF A = 1ØØ OR B = 1Ø THEN . . .

Arithmetic Operators

n**m n raised to the power m.
−n the negative value of n.
n*m n multiplied by m.
n / m n divided by m.
n+m n plus m.
n−m n minus m.

Relational Operators

These are used for comparing two numbers, variables or

expressions. = is also used in assigning a value to a variable.

n=m	n equals m
n>m	n is greater than m.
n<m	n is less than m.

Punctuation

;	directs the ZX80 to PRINT the next item on the same line and immediately following the item before the ;
,	directs the ZX80 to move on to the next PRINT zone before printing the next item. Each screen line is divided into four PRINT zones.
"	show the beginning and end of a literal string or a string variable.
: . ?	can be used inside literal strings, but no other significance.

Appendix 2

Glossary of Terms

Address The number which identifies a byte of memory.

Array A set of variables each identified by an array name and a subscript number, e.g. A(Ø), A(1), A(2), . . .

Back-up storage Some method of long term storage of programs and variables, e.g. a cassette recorder.

BASIC Originally designed for beginners, now one of the most widely used high level languages for micro-computers.

Binary digit (Bit) One digit from a binary number; can only be Ø or 1.

Binary number A number in the binary system (base 2), where all the digits are Ø or 1, instead of Ø to 9 as in the decimal (base 1Ø) system.

Bug An error in a program which prevents it from doing what is required of it.

Byte A binary number 8 bits long, the normal storage unit in a microcomputer memory.

Character Any item which can be stored in one byte and printed on the screen, e.g. A 1 ; PRINT are all ZX80 characters.

Character codes The single byte number which identifies each character — these may vary from one computer to another.

Command An instruction which does not form part of the program, but which makes the computer take action of some kind.

Conditional jump Causes a jump to a different part of the program if a given condition is met.

Crash The program stops running because of a program or data error.

Debug To find and remove errors from a program.

Edit To select and alter any chosen line in a program.

Enter To transfer a program line, or a command, or some data from the keyboard to the computer (by pressing NEWLINE on the ZX80).

Error code A signal from the computer showing the nature and the position of an error.

Firmware Sometimes used to denote the interpreter program, and other permanent programs found in ROM.

Flowchart A representation in diagrammatic form of a series of connected operations to be done in a specified sequence.

Hardware The physical parts of a computer and the surrounding equipment, as opposed to programs.

High level language Programming language made up of a set of recognisable English words.

Integer A whole number which may be positive or negative.

Integer BASIC All calculations are carried out with integers, and any decimal parts are chopped off and lost.

K (of memory) A unit of memory containing 1024 bytes.

Keyword A command or statement occupying one byte of memory and entered by a single keystroke.

Literal string A set of characters enclosed by quotation marks and printed literally on the screen by the computer.

Load To transfer a program from back-up storage to the computer.

Loop Part of a program which is carried out repeatedly.

Low level language Programming language which uses machine code.

Machine code Programming code which uses the hexadecimal system to represent binary numbers.

Nested loops Loops within loops, so that the instructions in inner loops are carried out several times for each pass round the outer loop.

Null string A string containing no characters at all.

Numerical variable A variable with some given name, to which can be assigned any desired number value or numerical expression.

Priority The order in which arithmetical or logical operations are carried out.

Program A numbered list of instructions to be carried out by a computer.

Pseudo-random numbers These have an apparently random distribution but each number is in fact calculated by the computer from the previous number, and they are therefore not truly random.

Random access memory (RAM) Computer memory used by the programmer for storage of programs, data, and so on. Each byte of RAM can be read or altered at will.

Random number A number drawn from a given set, where each number in the set is equally likely to be drawn and the draw is not affected by previous events.

Read only memory (ROM) Permanent computer memory generally used to contain BASIC interpreter programs, operating systems and so on. Can be read but not changed.

Relational operators > (greater than), < (less than), = (equals), used to compare numbers, expressions or strings.

Save To transfer a program into back-up storage for future use.

Software Computer programs and manuals, as opposed to hardware.

Statement An instruction to the computer which forms part of the program.

String variable A variable, identified in BASIC by a name ending in the $ sign, to which may be assigned a string of characters of any kind (with minor exceptions).

Subroutine A part of the program to which the computer can be directed from any part of the main program. When the subroutine has been carried out, the computer is directed back to the line following its original departure point.

Appendix 3

Programs for the ZX80

1. Graph Plotter

```
 1Ø   DIM X(16)
 2Ø   DIM Y(16)
 3Ø   PRINT "GRAPH PLOTTER"
 4Ø   PRINT
 45   PRINT
 5Ø   PRINT "PRESS ANY KEY, THEN TYPE IN YOUR EQUATION
      LIKE THIS:
        12Ø  LET Y(J)=2*X(J)+5"
 55   PRINT
 6Ø   PRINT "THEN PRESS NEWLINE, CONT, NEWLINE"
 7Ø   STOP
1ØØ   FOR J=1 TO 16
11Ø   LET X(J)=J
12Ø   LET Y(J)=2*X(J)+5
13Ø   NEXT J
49Ø   PRINT " Y"
5ØØ   FOR J=1 TO 16
52Ø   LET N=17-J
525   IF N<1Ø THEN PRINT " ";
53Ø   PRINT N;"■ ";
54Ø   FOR K=1 TO 16
55Ø   IF NOT Y(K)=N THEN GO TO 58Ø
56Ø   PRINT "X";
57Ø   GO TO 59Ø
58Ø   PRINT " ";
59Ø   NEXT K
6ØØ   PRINT
61Ø   NEXT J
```



```
62Ø   PRINT " ";CHR$(133);"■■■■■■■■■■■■■■■■"
63Ø   PRINT "Ø 2 4 6 8 1 1 1 1 X"
64Ø   PRINT "          Ø 2 4 6"
```

Line 62Ø uses 16 graphic blocks (SHIFT W).

List of variables

X(16), Y(16) — coordinates of the points on the graph.

J, K — FOR/NEXT loop counters.

N — where J has the values 1 to 16, N has the values 16 to 1.

Notes

This program is intended to show the forms of graphs of different mathematical expressions for Y as a function of X. On RUN, the program pauses to allow the user to insert his own equation. In the example given, the equation Y = 2X + 5 is typed in as:

```
12Ø  LET Y(J)=2*X(J)+5
```

Any equation which uses the ZX80 mathematical operators can be typed in, but some juggling with constants may be needed to make the graph appear within the scales X = Ø to +16 and Y = Ø to +16.

It is worth noting the use of PRINT CHR$(133) in line 62Ø. This prints a character which is not available on the ZX80 keyboard, in this case to supply the 'corner' where the two axes join.

Line 7Ø	stops to allow the equation to be typed in. STOP must be followed by CONT to make the program continue.
Lines 1ØØ to 13Ø	sets X and Y coordinates for each point on the graph. Line 12Ø shows a sample equation.
Lines 49Ø to 64Ø	draws the graph. Note that to draw the graph the 'right way up', J = 1 TO 16 has to be reversed to give N = 16 to 1.
Line 56Ø	plots the points on the graph.
Lines 53Ø and 62Ø	draws the Y and X axes.

2. Klingon Missile

```
10Ø  LET MX=14
11Ø  LET MY=14
12Ø  LET FX=Ø
13Ø  LET FY=Ø
14Ø  LET Q=Ø
18Ø  CLS
19Ø  LET J=14
2ØØ  PRINT "Y"
21Ø  IF J<1Ø THEN PRINT "  ";
22Ø  PRINT J;
23Ø  IF MY=J OR FY=J THEN GO TO 2ØØØ
24Ø  PRINT "▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
25Ø  LET J=J-1
26Ø  IF J>-1 THEN GO TO 21Ø
27Ø  PRINT "  Ø12345678911111X"
28Ø  PRINT "                 Ø1234"
29Ø  IF Q=1 THEN STOP
3ØØ  PRINT "FIRING AT WHICH SQUARE?"
31Ø  PRINT "X=";
32Ø  INPUT FX
33Ø  PRINT FX,"Y=";
34Ø  INPUT FY
4ØØ  LET MX=ABS(MX-RND(3))
41Ø  LET MY=ABS(MY-RND(3))
5ØØ  IF MX>1 OR MY>1 THEN GO TO 6ØØ
51Ø  CLS
52Ø  PRINT "END OF GAME"
53Ø  PRINT "YOU HAVE JUST BEEN EXTERMINATED"
54Ø  STOP
6ØØ  IF MX=FX AND MY=FY THEN LET Q=1
61Ø  GO TO 18Ø
2ØØØ  FOR K=Ø TO 14
2Ø1Ø  IF Q=1 AND FX=K THEN GO TO 3ØØØ
2Ø2Ø  IF FX=K AND FY=J THEN GO TO 4ØØØ
2Ø3Ø  IF MX=K AND MY=J THEN GO TO 5ØØØ
2Ø5Ø  PRINT "▒";
2Ø6Ø  NEXT K
2Ø9Ø  PRINT
21ØØ  GO TO 25Ø
3ØØØ  PRINT "POW . . . .";
3Ø1Ø  GO TO 2Ø9Ø
4ØØØ  PRINT "X";
4Ø1Ø  GO TO 2Ø6Ø
```



```
5ØØØ  PRINT "M";
5Ø1Ø  GO TO 2Ø6Ø
```

List of variables

MX, MY } coordinates of the Klingon missile.

FX, FY } coordinates of your own defensive shots.

Q 'flag' to show that missile is destroyed.
J number of rows of graphics blocks in grid.
K FOR/NEXT loop counter.

Notes

You are at the bottom left of a 15×15 grid, and an enemy missile is at the top right. The missile makes a series of random moves towards your corner, until eventually it destroys you. You have to predict what the position of the missile will be after each move, and try to eliminate the missile by firing shots at the predicted positions.

Lines 1ØØ to 14Ø initialising the variables.
Lines 18Ø to 28Ø drawing the 15×15 grid, and directing the program to line 2ØØØ to print the missile and the defensive shots.
Line 29Ø stops the program when the missile is destroyed.
Lines 3ØØ to 34Ø inputting the coordinates of the defensive shots.
Lines 4ØØ to 5ØØ randomly sets new coordinates for the missile and prepares to draw a new grid.
Lines 51Ø to 53Ø stops the program when you have been exterminated.
Lines 6ØØ to 61Ø sets the 'flag' to show that the missile has been destroyed, and returns to line 180 to draw a new grid.
Lines 2ØØØ to 5Ø1Ø this routine prints rows in the grid which contain the missile and/or the defensive shots, and also indicates when the missile has been destroyed.

3. Fox and Hounds

```
1Ø   DIM X(4)
2Ø   DIM Y(4)
1ØØ  FOR J=1 TO 4
11Ø  LET X(J)=2*J-1
12Ø  LET Y(J)=Ø
13Ø  NEXT J
14Ø  LET X=4
15Ø  LET Y=7
2ØØ  GO SUB 2ØØØ
3ØØ  PRINT "HOUND NO. ";
31Ø  INPUT A
32Ø  PRINT A;
33Ø  PRINT "TO WHICH SQUARE? X=";
34Ø  INPUT X(A)
35Ø  PRINT X(A); " Y=";
36Ø  INPUT Y(A)
49Ø  CLS
5ØØ  LET YY=Y-1
51Ø  LET Q=RND(2)*2-3
52Ø  FOR M=1 TO 2
53Ø  LET XX=ABS(X+Q)
54Ø  IF XX>7 THEN LET XX=6
55Ø  FOR L=1 TO 4
56Ø  IF XX=X(L) AND YY=Y(L) THEN GO TO 59Ø
57Ø  NEXT L
58Ø  GO TO 72Ø
59Ø  LET Q=-Q
6ØØ  NEXT M
61Ø  IF YY=Y+1 THEN GO TO 7ØØ
62Ø  LET YY=Y+1
63Ø  IF YY>7 THEN LET YY=6
64Ø  GO TO 51Ø
7ØØ  PRINT "NOWHERE TO GO, YOU WIN"
71Ø  GO TO 82Ø
72Ø  IF YY=Ø THEN PRINT "GOT THERE, I WIN"
8ØØ  LET Y=YY
81Ø  LET X=XX
82Ø  GO SUB 2ØØØ
83Ø  IF Y=Ø THEN STOP
84Ø  GO TO 3ØØ
2ØØØ PRINT," 01234567 X"
```

A Fox and Hounds program for the PET was published in *Computing Today*, October 1980.


```
2Ø1Ø FOR J=0 TO 7
2Ø15 PRINT,J;
2Ø2Ø FOR K=0 TO 7
2Ø3Ø FOR L=1 TO 4
2Ø4Ø IF NOT (X(L)=K AND Y(L)=J) THEN GO TO 2Ø7Ø
2Ø5Ø PRINT L;
2Ø6Ø GO TO 213Ø
2Ø7Ø NEXT L
2Ø8Ø IF NOT (X=K AND Y=J) THEN GO TO 211Ø
2Ø9Ø PRINT "F";
21ØØ GO TO 213Ø
211Ø IF NOT ((K+J)/2)*2=K+J THEN GO TO 2125
2115 PRINT "▧";
212Ø GO TO 213Ø
2125 PRINT " ";
213Ø NEXT K
214Ø PRINT
215Ø NEXT J
216Ø PRINT, "Y"
217Ø RETURN
```

List of variables

X(4), Y(4)	coordinates of the four hounds.
X, Y	coordinates of the fox.
A	number of the hound to be moved.
J,K,L,M	FOR/NEXT loop counters.
XX, YY	dummy variables for the coordinates of the fox.
Q	random +1 or −1 to determine whether fox goes left or right.

Notes

On RUN, the ZX80 prints a chess board with four numbered hounds spaced evenly along the top row, and a single fox on the bottom row. The fox and hounds move alternately, the fox trying to get to the top row, and the hounds trying to stop him by boxing him in. You move the hounds (there are no anti-cheat tests, that is up to you!), and the computer moves the fox at random, but always

forwards if possible. You can make the game harder by cutting down the number of hounds (e.g. line 1ØØ FOR J=1 TO 3 and line 2Ø3Ø FOR L=1 TO 3 for three hounds). The fox is not intelligent, that needs more memory than you have got.

Lines 1ØØ to 15Ø sets start coordinates for hounds and fox.
Line 2ØØ GOSUB to print the board with fox and hounds.
Lines 3ØØ to 36Ø picks a hound and moves it to a chosen square.
Lines 5ØØ to 64Ø fox tries all possible legal moves, first the two forward ones, then backwards. He stops on any square not occupied by a hound.
Lines 7ØØ to 71Ø fox has tried all squares without success and so loses the game.
Line 72Ø fox has reached the top row (Y=Ø) and won the game.
Line 84Ø back to Line 3ØØ for another set of moves.
Lines 2ØØØ to 217Ø subroutine to print the chess board with four hounds and a fox.
Lines 2Ø3Ø to 2Ø7Ø hound printing loop.
Lines 2Ø8Ø to 21ØØ printing the fox.
Lines 211Ø to 2125 printing alternate black and white squares.

4. Fruit Machine With Optional Nudge

```
 1Ø  DIM W(3)
 2Ø  RANDOMISE
1ØØ  FOR J=1 TO 3
11Ø  LET W(J)=RND(6)
12Ø  GO SUB 5ØØØ
125  PRINT
13Ø  PRINT "HIT KEY C TO CHANGE LAST WINDOW OR
     NEWLINE TO GO ON TO NEXT"
14Ø  INPUT A$
15Ø  IF A$="" " THEN GO TO 3ØØ
16Ø  IF A$="C" THEN LET W(J)=W(J)+RND(4)
17Ø  IF W(J)>6 THEN LET W(J)=W(J)-6
18Ø  CLS
19Ø  GO SUB 5ØØØ
195  PRINT
2ØØ  PRINT "HIT NEWLINE TO OPEN NEXT WINDOW"
21Ø  INPUT A$
22Ø  IF A$="" " THEN GO TO 3ØØ
3ØØ  CLS
31Ø  NEXT J
39Ø  PRINT
```



```
4ØØ  PRINT "ANOTHER GO? PRESS NEWLINE"
41Ø  INPUT A$
42Ø  IF A$="" " THEN GO TO 44Ø
43Ø  STOP
44Ø  CLS
45Ø  CLEAR
46Ø  GO TO 1Ø
5ØØØ  PRINT "00000000  00000000  00000000 "
5Ø1Ø  PRINT "0      0  0      0  0      0 "
5Ø2Ø  FOR K=1 TO 3
5Ø3Ø  GO SUB (55ØØ+1Ø*W(K))
5Ø4Ø  NEXT K
5Ø5Ø  PRINT
5Ø6Ø  PRINT "0      0  0      0  0      0 "
5Ø7Ø  PRINT "00000000  00000000  00000000 "
5Ø8Ø  RETURN
55ØØ  PRINT "0          0 ";
5595  RETURN
551Ø  PRINT "0   BELL   0 ";
5515  RETURN
552Ø  PRINT "0   APPLE  0 ";
5525  RETURN
553Ø  PRINT "0  CHERRY  0 ";
5535  RETURN
554Ø  PRINT "0   PEAR   0 ";
5545  RETURN
555Ø  PRINT "0   STAR   0 ";
5555  RETURN
556Ø  PRINT "0 ANCHOR 0 ";
5565  RETURN
```

List of variables

W(1 to 3) three random numbers to select 'fruits' for each window.

J, K } FOR/NEXT loop variables.

A$ INPUT string variable to allow the player to choose branches and to make the program continue after a pause.

Notes

On RUN the program prints three windows, with a 'fruit' in the first one. The player has the option of changing that fruit with a 'nudge',

or going on to open the second window. Again, when the second fruit is displayed, the player can change it if he wishes, and likewise the fruit in the third window. The object is of course to get three fruits the same.

Lines 100 to 310 — a FOR/NEXT loop — the program goes round the loop three times to deal with the three windows.
Line 110 — generates a random number from 1 to 6.
Line 120 — sends the program to subroutine 5000 to print windows and fruits.
Lines 130 to 150 — gives player the option of nudging the last window to change the fruit.
Lines 160 to 190 — routine to change the last fruit.
Lines 200 to 220 — pause to let player see the windows and then move on to the next.
Lines 390 to 460 — offers the player another go. Note line 450 CLEAR to empty the windows before returning to line 10 for the next try.
Lines 5000 to 5565 — subroutine to print windows and fruits, controlled by the three random numbers W(J). Note that the window frames are made up of 0s, but any graphic blocks may be used.

5. Tables Test

```
 10  RANDOMISE
 90  LET TR=0
100  FOR J=1 TO 10
110  GO SUB 5000
120  PRINT
130  PRINT "HIT NEWLINE FOR NEXT QUESTION"
140  INPUT A$
150  NEXT J
160  CLS
170  PRINT
180  PRINT "YOU GOT ";TR;" RIGHT OUT OF 10"
190  PRINT
200  PRINT "HIT N/L FOR 10 OR S TO STOP"
210  INPUT A$
220  IF A$="S" THEN STOP
230  GO TO 90
5000 LET X=RND(12)
5010 LET Y=RND(13)-1
5020 LET Z=X*Y
```



```
5030 LET B=RND(4)
5040 CLS
5050 PRINT
5060 GO TO 5100+20*B
5120 PRINT X;" TIMES ";Y;" EQUALS ?"
5130 GO TO 5300
5140 PRINT X;" X ";Y;" = ?"
5150 GO TO 5300
5160 PRINT Z;" DIVIDED BY ";X;" EQUALS ?"
5170 GO TO 5400
5180 PRINT X;" INTO ";Z;" GOES ?"
5190 GO TO 5400
5300 PRINT
5310 FOR K=1 TO 2
5320 INPUT ZZ
5330 PRINT ZZ
5340 PRINT
5350 IF ZZ=Z THEN GO TO 5500
5360 PRINT "SORRY — WRONG"
5370 NEXT K
5375 PRINT
5380 PRINT X;" X ";Y;" = ";Z
5390 RETURN
5400 PRINT
5410 FOR K=1 TO 2
5420 INPUT YY
5430 PRINT YY
5440 PRINT
5450 IF YY=Y THEN GO TO 5500
5460 PRINT "WRONG THAT TIME"
5470 NEXT K
5475 PRINT
5480 PRINT X;" INTO ";Z;" GOES ";Y
5490 RETURN
5500 IF K=1 THEN LET TR=TR+1
5510 PRINT "CORRECT"
5520 RETURN
```

List of variables

TR — total of right answers.
J, K — FOR/NEXT loop counters.
A$ — INPUT null string variable to continue after a pause.

X, Y — the two random numbers which are the basis of each question.
Z — the product of X and Y.
B — a random number from 1 to 4, used to select the form of the question.
ZZ — INPUT answers to the multiplication questions.
YY — INPUT answers to the division questions.

Notes

On RUN, the program prints a multiplication or division sum based on the multiplication tables up to twelve, and the user has to type in the answer. If wrong, he is given one chance to correct it, after which the correct answer is shown. After nine more questions, the total of first time correct answers out of ten is shown.

Lines 10Ø to 15Ø — FOR/NEXT loop sends the program ten times to subroutine 5ØØØ for a question and answer.

Lines 16Ø to 22Ø — display the number of correct answers out of ten, and offer the chance of another ten questions.

Lines 5ØØØ to 502Ø — set up the multiplication sum on which each question is based. Note that X can not be Ø, this avoids questions in which a number is divided by zero.

Lines 503Ø to 506Ø — controlled by a random number, send the program to one of the four possible forms of question.

Lines 512Ø to 519Ø — contain the four different forms of question, two multiplication and two division.

Lines 531Ø to 539Ø — deals with answers to multiplication sums. Stops to let the user insert his answer, tests it, allows another try if wrong, and then prints correct answer and returns.

Lines 54ØØ to 549Ø — do the same function as above, in this case for division sums.

Lines 55ØØ to 552Ø — signal correct answers and count them.

6. Throwing a Single Dice

```
 1Ø  RANDOMISE
 2Ø  CLEAR
 3Ø  DIM Y(6)
```



```
 4Ø  CLS
11Ø  PRINT "DICE THROWING"
12Ø  PRINT
13Ø  PRINT
15Ø  PRINT "HOW MANY DICE THROWS?"
16Ø  INPUT A
17Ø  CLS
18Ø  PRINT A;" DICE THROWS"
19Ø  PRINT
2ØØ  GO SUB 1ØØØ
22Ø  PRINT
23Ø  PRINT
24Ø  PRINT "DO YOU THINK THE DICE IS FAIR?"
25Ø  PRINT
26Ø  PRINT "PRESS NEWLINE TO PUT THE RESULTS ON A BAR
     CHART"
27Ø  INPUT A$
28Ø  GO SUB 2ØØØ
3ØØ  PRINT "PRESS NEWLINE FOR ANOTHER GO"
31Ø  INPUT A$
32Ø  IF A$="" THEN GO TO 2Ø
9ØØ  STOP
1ØØØ FOR J=1 TO A
1Ø1Ø LET X=RND(6)
1Ø2Ø LET Y(X)=Y(X)+1
1Ø3Ø PRINT X;" ";
1Ø5Ø NEXT J
1Ø6Ø RETURN
2ØØØ CLS
2Ø1Ø FOR J=1 TO 18
2Ø2Ø LET N=19-J
2Ø3Ø IF N<1Ø THEN PRINT " ";
2Ø4Ø PRINT N;" ";
2Ø5Ø FOR K=1 TO 6
2Ø6Ø IF NOT Y(K)>N-1 THEN GO TO 2Ø9Ø
2Ø7Ø PRINT "□";
2Ø8Ø GO TO 21ØØ
2Ø9Ø PRINT " ";
21ØØ PRINT " ";
211Ø NEXT K
212Ø PRINT
213Ø NEXT J
214Ø PRINT "  1 2 3 4 5 6"
215Ø PRINT "  DICE THROW"
216Ø RETURN
```

List of variables

Y(6)	The number of times that each dice throw result (1 or 2 or 3 . . . etc) has turned up.
A	number of dice throws INPUT by the player.
A$	null string variable to move the program on after a pause.
J, K	FOR/NEXT loop counters.
X	the number turned up when one dice is thrown.
N	N is needed to turn the bar chart the right way up. When J=1 TO 18, N is 18 to 1.

Notes

The user decides how many times he wants to throw a single dice. The results are shown as a collection of random numbers from 1 to 6, and these are then sorted, totalled and displayed in the form of a bar chart. Of course we would expect a perfect dice to give an equal number of ones, twos, threes, and so on. Perfection is not to be obtained in this world, and all dice are subject to the laws of probability. What the program will show is that as the number of dice throws is increased, the result begins to approach the 'expected' result.

Lines 15Ø to 18Ø determine the number of dice throws (this is limited to about 60 by the size of screen).

Lines 1ØØØ to 1Ø6Ø throw the dice the required number of times, sort out and total the results, and print the results as a series of numbers from 1 to 6.

Lines 2ØØØ to 216Ø prints the results in the form of a bar chart, showing how many ones, how many twos, and so on.

7. Throwing a Pair of Dice

```
 1Ø  RANDOMISE
 2Ø  DIM Y(12)
1ØØ  CLS
11Ø  PRINT "THROWING A PAIR OF DICE"
12Ø  PRINT
13Ø  PRINT "HOW MANY TIMES?"
14Ø  INPUT A
28Ø  CLS
29Ø  PRINT "THE TOTALS ARE"
```



```
3ØØ  FOR J=1 TO A
31Ø  LET X=RND(6)+RND(6)
32Ø  LET Y(X)=Y(X)+1
33Ø  PRINT X;" ";
34Ø  NEXT J
35Ø  PRINT
4ØØ  PRINT
41Ø  PRINT "PRESS NEWLINE FOR A BAR CHART OF THE
     RESULTS"
42Ø  INPUT A$
5ØØ  CLS
51Ø  FOR J=1 TO 15
52Ø  LET N=(16-J)*2
53Ø  IF N<11 THEN PRINT " ";
54Ø  PRINT N-1;" ";
55Ø  FOR K=2 TO 12
555  IF NOT Y(K)>N-1 THEN GO TO 57Ø
56Ø  PRINT "▧";
565  GO TO 595
57Ø  IF NOT Y(K)>N-2 THEN GO TO 59Ø
58Ø  PRINT "▨";
585  GO TO 595
59Ø  PRINT " ";
595  PRINT " ";
61Ø  NEXT K
62Ø  PRINT
63Ø  NEXT J
64Ø  PRINT "  2 3 4 5 6 7 8 91Ø1112"
65Ø  PRINT "  TOTAL FOR TWO DICE"
```

List of variables

Y(12)	Array of cumulative frequencies for scores 2 to 12.
J, K	Loop control variables.
A$	INPUT string variable to continue after a pause.
N	Values for vertical scale of bar chart.

Notes

On RUN you are asked to input the number of times to throw the dice — up to 1ØØ will ensure that you stay clear of 'variable space full' errors. You then see a display of the actual scores for each throw, and then press NEWLINE for a bar chart of the results.

Lines 300 to 340	loop from 1 to the number of throws chosen, which throws dice and prints each score.
Line 310	throws two dice and adds up the two throws.
Line 320	accumulates the frequency for each score from 2 up to 12.
Lines 510 to 630	outer J loop prints the vertical scale of the bar chart. Inner loop prints bars and spaces. A half block (SHIFT T) represents 1 on the vertical scale, and a whole block (SHIFT A) represents 2.

8. Pontoon

```
20    DIM X(5)
30    LET M=100
90    CLS
100   PRINT "FIRST CARD: ";
110   LET J=0
120   GO SUB 2000
130   PRINT "YOUR BET?";
140   INPUT B
150   LET M=M-B
170   CLS
180   PRINT "YOUR CARDS ";
190   GO SUB 2000
200   LET PT=T
210   PRINT "YOUR BET:£";B
240   IF T>21 THEN GO TO 900
250   PRINT
260   PRINT "NOW WHAT? T=TWIST S=STICK"
270   INPUT A$
280   IF A$="S" THEN GO TO 1000
290   IF A$="T" THEN GO TO 170
300   GO TO 270
900   PRINT "YOU HAVE BUST"
910   GO TO 1230
1000  LET J=0
1010  CLS
1020  PRINT "YOU HAVE ";PT;" POINTS"
1030  PRINT "BANKS TURN ";
1040  GO SUB 2000
1050  IF T<14+RND(3) THEN GO TO 1010
1060  IF T>21 THEN GO TO 1200
1070  IF PT>T THEN GO TO 1210
```



```
1080  PRINT "YOU LOSE",
1090  GO TO 1230
1200  PRINT "BANK BUST"
1210  LET M=M+2*B
1220  PRINT "YOU WIN",,
1230  PRINT "YOU HAVE £";M
1240  INPUT A$
1250  GO TO 90
2000  LET J=J+1
2010  LET T=0
2020  LET X(J)=RND(13)
2030  LET XX=0
2040  FOR K=1 TO J
2050  LET X=X(K)
2060  IF X>1 AND X<11 THEN PRINT X,
2070  IF X=11 THEN PRINT "J",
2080  IF X=12 THEN PRINT "Q",
2090  IF X=13 THEN PRINT "K",
2100  IF X>10 THEN LET X=10
2110  IF X>1 THEN GO TO 2150
2120  PRINT "A",
2130  LET XX=1
2140  LET T=T+10
2150  LET T=T+X
2160  IF NOT(XX=1 AND T>21) THEN GO TO 2190
2170  LET T=T-10
2180  LET XX=0
2190  NEXT K
2200  PRINT "TOTAL POINTS =";T
2210  RETURN
```

List of variables

X(5)	up to 5 cards dealt.
M	amount of money left.
J	number of cards to deal.
B	players bet on first card.
T	total points in hand.
PT	total points in player's hand after 'stick'.
A$	input string variable, "T" or "S" or " ".
XX	marker to show presence of an ace with value 11.
X	dummy variable for current card.
K	FOR/NEXT loop counter.

Notes

The player starts with £100. On RUN he is dealt a card and bets on it (make your own rules about betting limits). He then gets his second card and is given the usual option of twisting (T) or sticking (S). This is repeated until he finally decides to stick (at up to 21 points) or is bust (over 21). Assuming that he sticks, the bank now deals himself cards until their points total is 15 to 17 (random number) or greater, and bank pays out on scores higher than his own.

Most of the basic rules of pontoon are obeyed. Aces are scored properly as 1 or 11, but a pair of aces gets the scoring into a muddle. Pontoons and five card tricks are not recognised as having any value above their points score — the player will have to give himself credit for these now and again. The option of buying a card face down is not included.

Lines 1ØØ to 15Ø deal the first card and accept a bet on it.

Lines 17Ø to 91Ø deal a second card, and then further cards as requested by T, display cards and points score each time. Test for 'bust', and on the command S pass the program on to the bank's turn.

Lines 1ØØØ to 125Ø the bank deals himself cards, deciding whether or not to stick according to whether his points score is below the random number 15 to 17. At the end of his deal, the banker may be bust or have a lower points score than the player — in these cases the player wins. If the bank's score is equal to or above the player's score, the bank wins.

Lines 2ØØØ to 221Ø each time this subroutine is entered, it deals one more card, displays all the cards dealt so far together with the points score.

Lines 212Ø to 218Ø deal with the special case of an ace, which can be scored as 1 or 11.

9. Hog

```
 1Ø  RANDOMISE
 2Ø  DIM T(2)
 3Ø  LET J=1
1ØØ  PRINT "THE GAME OF HOG"
11Ø  PRINT
12Ø  PRINT "TRY TO REACH 1ØØ BEFORE I DO"
13Ø  PRINT
```



```
14Ø  PRINT "PRESS N/L TO THROW OR S TO STOP"
15Ø  PRINT "PRESS N/L FOR NEXT TURN"
16Ø  INPUT A$
17Ø  IF A$="99" THEN STOP
18Ø  CLS
2ØØ  PRINT "YOUR TOTAL=";T(1),"MY TOTAL=";T(2)
21Ø  GO SUB 1ØØØ
3ØØ  LET T(J)=T(J)+R
31Ø  PRINT
32Ø  PRINT
33Ø  PRINT R;" ON THAT TURN"
34Ø  PRINT
35Ø  IF T(1)>99 THEN PRINT "YOU WIN"
36Ø  IF T(2)>99 THEN PRINT "I WIN"
37Ø  IF T(J)>99 THEN STOP
38Ø  LET J=3-J
39Ø  GO TO 15Ø
1ØØØ  PRINT
1Ø1Ø  LET R=Ø
1Ø2Ø  IF J=2 THEN GO TO 14ØØ
1Ø3Ø  PRINT "YOUR TURN"
1Ø4Ø  PRINT
1Ø6Ø  INPUT A$
1Ø7Ø  IF A$="S" THEN RETURN
1Ø8Ø  LET Z=RND(6)
1Ø9Ø  PRINT Z,
11ØØ  IF Z=6 THEN GO TO 13ØØ
111Ø  LET R=R+Z
112Ø  GO TO 1Ø6Ø
13ØØ  PRINT "TOO BAD . . ."
131Ø  LET R=Ø
132Ø  RETURN
14ØØ  PRINT "MY TURN NOW"
141Ø  PRINT
142Ø  LET Z=RND(6)
1425  PRINT Z,
143Ø  IF Z=6 THEN GO TO 161Ø
144Ø  LET R=R+Z
145Ø  LET D=99-T(2)
146Ø  LET E=(1Ø8-T(1))/8
147Ø  LET C=(1Ø8-T(2))/E
148Ø  IF C<15 THEN LET C=14
149Ø  IF C>D THEN LET C=D
15ØØ  IF R>C THEN RETURN
151Ø  GO TO 142Ø
```

```
161Ø  PRINT "BOTHER. . . .";
162Ø  LET R=Ø
163Ø  RETURN
```

List of variables

T(2)	total scores for player and computer.
J	marker to show whose turn is in progress, 1=player, 2=computer.
A$	input string variable to make game continue.
R	total score for each separate turn.
Z	one individual dice throw.
C,D,E	variables calculated to allow the computer to decide whether to throw the dice again or stop.

Notes

Hog is a simple but compulsive dice throwing game. The player starts, and is allowed to throw as often as he likes to build up a high score, but if he throws a six his score for that turn is reduced to 0, and the turn passes to the ZX80 which plays in the same way. Play continues with alternate turns, and the winner is the first to reach a total of 100.

Lines 18Ø to 39Ø print the totals at the start of each turn, go to subroutine 1ØØØ for the required number of dice throws, print the score for that turn and add it to the total, test the new total for a win and then returns to give the other player a turn. Line 38Ø gives J the values 1 and 2 alternately.

Lines 1Ø3Ø to 132Ø the player's turn — he has a free choice after each throw of throwing again or stopping. Each dice throw is printed, and the score for that turn is reduced to 0 when a six is thrown.

Lines 14ØØ to 163Ø the ZX80's turn — he throws in the same way, and shows some 'intelligence' (lines 145Ø to 151Ø) in basing his decision as to whether to throw or stop on his own and the player's score.

10. Submarine Hunt

```
 1Ø  DIM P(4)
 2Ø  DIM Q(4)
 3Ø  DIM X(15)
```



```
 4Ø  DIM Y(15)
 5Ø  RANDOMISE
 6Ø  LET S=4
1ØØ  FOR K=1 TO 4
11Ø  LET P(K)=RND(8)
12Ø  LET Q(K)=RND(8)
13Ø  NEXT K
2ØØ  FOR J=1 TO 15
22Ø  PRINT S;" SUBS TO SINK"
23Ø  PRINT "FIRING AT WHICH SQUARE? X=";
24Ø  INPUT X(J)
25Ø  PRINT X(J);" Y="
26Ø  INPUT Y(J)
3ØØ  CLS
31Ø  PRINT "LAST SHOT X=";X(J);" Y=";Y(J)
4ØØ  FOR K=1 TO 8
41Ø  FOR L=1 TO 8
42Ø  FOR M=1 TO J
43Ø  IF NOT (X(M)=L AND Y(M)=K) THEN GO TO 46Ø
44Ø  PRINT "X";
45Ø  GO TO 48Ø
46Ø  NEXT M
47Ø  PRINT "▨";
48Ø  NEXT L
49Ø  PRINT
5ØØ  NEXT K
59Ø  LET T=Ø
6ØØ  FOR K=1 TO 4
61Ø  LET Z=(X(J)-P(K))**2+(Y(J)-Q(K))**2
62Ø  IF Z<3 THEN LET T=1
63Ø  IF NOT Z=Ø THEN GO TO 7ØØ
64Ø  PRINT "DIRECT HIT"
65Ø  LET S=S-1
66Ø  IF S=Ø THEN GO TO 1ØØØ
67Ø  LET P(K)=19
68Ø  GO TO 72Ø
7ØØ  IF Z<3 THEN PRINT "1 SQUARE OFF";
71Ø  IF Z=2 THEN PRINT "DIAGONALLY"
72Ø  NEXT K
73Ø  IF T=Ø THEN PRINT "MISSED"
74Ø  PRINT
75Ø  NEXT J
8ØØ  PRINT "END OF GAME. ";S;" SUBS LEFT"
81Ø  STOP
1ØØØ  PRINT "ALL SUBS SUNK IN ";J;" SHOTS"
```

List of variables

P(4), Q(4)	coordinates of the four submarines.
X(15), Y(15)	coordinates of the player's 15 shots.
S	number of submarines left.
J,K,L,M	FOR/NEXT loop counters.
T	indicator to show whether or not player's shot has missed.
Z	variable calculated to show whether the shot was a hit, or within one square of a submarine.

Notes

Four submarines are hidden at random, each occupying one square in an 8 by 8 grid. It is possible, but rare, for more than one submarine to be on the same square. On RUN, you are invited to fire at one square on the grid by entering X (squares along) and Y (squares down) coordinates. The ZX80 then prints the coordinates of this shot, draws the grid showing all shots so far, and tells you whether you have made a direct hit, or are within one square of a submarine. You have a total of 15 shots to sink all four submarines.

Lines 10 to 130	sets the coordinates of the four submarines.
Lines 230 to 310	inputs the coordinates of the present shot and prints them.
Lines 400 to 500	prints the 8 by 8 grid, showing the position of all shots fired so far. This is done by a triple nested loop, which accounts for the time delay in carrying out this part of the program.
Lines 600 to 740	compares the coordinates of each shot with the coordinates of each submarine, and announces direct hits, near misses or complete misses.

11. Bulls and Cows

```
 10  DIM G(4)
 20  DIM N(6)
 50  LET X=0
 60  RANDOMISE
100  FOR J=1 TO 6
110  LET Q=RND (6)
120  IF NOT N(Q)=0 THEN GO TO 110
130  LET N(Q)=J
140  NEXT J
150  LET N=N(1)*1000+N(2)*100+N(3)*10+N(4)
160  PRINT "* BULLS AND COWS *"
170  LET X=X+1
180  PRINT
190  IF X>10 THEN GO TO 900
200  PRINT "GUESS ";X;"?",
210  INPUT G
220  LET G(1)=G/1000
230  LET G(2)=G/100-G(1)*10
240  LET G(3)=G/10-(G/100)*10
250  LET G(4)=G-(G/10)*10
260  PRINT G,
300  FOR J=1 TO 4
310  IF G(J)=N(J) THEN GO SUB 1000
320  NEXT J
330  FOR J=1 TO 4
340  FOR K=1 TO 4
350  IF G(K)=N(J) THEN GO SUB 1100
360  NEXT K
370  LET N(J)=ABS(N(J))
380  NEXT J
700  IF NOT G=N THEN GO TO 170
780  PRINT
790  PRINT
800  PRINT "GOT IT IN ";X;" GOES"
810  STOP
900  PRINT
910  PRINT "SORRY, IT WAS ";N
920  STOP
1000 PRINT "▬ ";
1010 LET G(J)=0
1020 LET N(J)=-N(J)
1030 RETURN
1100 PRINT "▭ ";
1110 LET G(K)=0
1120 LET N(J)=-N(J)
1130 RETURN
```

List of variables

G(4)	four digits of the player's guess.
N(6)	a random array of the numbers from 1 to 6.
X	counts the number of guesses.

J, K	loop control variables.
Q	random number from 1 to 6.
N	the hidden 4-digit number.
G	the player's guess.

Notes

On RUN the ZX80 generates a 4-digit number which has each digit between 1 and 6, and different. The player makes a guess, and the result on the scoreboard is shown as black 'bulls' (right digits in the right place) and grey 'cows' (right digits in the wrong place). Up to ten guesses are shown on the scoreboard, and then the number is revealed.

I prefer this easier version, but if you want to play it the hard way, where the digits in the hidden number may be the same, you need to make these changes:

Delete lines 10Ø to 13Ø

```
12Ø  FOR J=1 TO 4
13Ø  LET N(J)=RND (6)
```

Bulls and Cows presents some interesting programming problems.

Lines 1ØØ to 14Ø	Sets the numbers 1 to 6 into a random array.
Lines 16Ø to 19Ø	counts the guesses.
Lines 2ØØ to 21Ø	inputs the next guess.
Lines 22Ø to 25Ø	splits the guess into four digits.
Lines 3ØØ to 32Ø	checks each digit for a black 'bull'.
Lines 33Ø to 38Ø	checks each digit for a grey 'cow'.
Line 37Ø	restores each hidden number digit after it has been checked.
Line 7ØØ	loops back for another guess.
Lines 1ØØØ to 1Ø3Ø	subroutine for 'bulls'. Prints a black 'bull' on the scoreboard, erases the current guess digit so that it can give no further score.
Lines 11ØØ to 113Ø	subroutine for 'cows', as for 'bulls' subroutine above.

12. In the Caves

```
1Ø   RANDOMISE
2Ø   LET P=29
3Ø   LET L=1
4Ø   DIM Q(2)
5Ø   DIM R(2)
6Ø   LET D=RND(9)
7Ø   LET F=RND(22)
8Ø   LET E=Ø
9Ø   LET K=RND(9)
2ØØ  LET M=RND(23)
21Ø  FOR J=M+1 TO 24
22Ø  LET N=J
23Ø  LET X=M*N-1
24Ø  IF (X/P)*P=X THEN GO TO 5ØØ
25Ø  NEXT J
26Ø  GO TO 2ØØ
5ØØ  PRINT "HIT N/L FOR NEXT TURN"
51Ø  INPUT A$
52Ø  CLS
53Ø  PRINT "PLAYER ";L;" IN CAVE ";Q(L)
54Ø  LET R(L)=Q(L)
81Ø  PRINT "WHICH WAY? F/B/S"
82Ø  INPUT A$
84Ø  IF A$="B" THEN LET A=Q(L)*M
85Ø  IF A$="F" THEN LET A=Q(L)*N
86Ø  IF A$="S" THEN LET A=P+2-Q(L)
87Ø  LET A=A-(A/P)*P
88Ø  IF A<24 THEN LET Q(L)=A
9ØØ  CLS
91Ø  PRINT A$;" WENT FROM CAVE ";R(L);
94Ø  IF Q(L)=R(L) THEN PRINT " TO A DEAD END"
95Ø  IF NOT Q(L)=R(L) THEN PRINT " TO CAVE ";Q(L)
1ØØØ IF NOT Q(L)=23 THEN GO TO 11ØØ
1Ø1Ø PRINT "KEY 1 IS ";D
1Ø2Ø PRINT "GO TO CAVE ";F;" TO FIND KEY 2"
1Ø3Ø LET E=K
11ØØ IF NOT (Q(L)=F AND E>Ø) THEN GO TO 12ØØ
1111 PRINT "KEY 2 IS ";E
1121 PRINT "TAKE KEYS TO CAVE Ø"
12ØØ IF NOT (Q(L)=Ø AND E>0) THEN GO TO 14ØØ
121Ø PRINT "KEY 1+KEY 2=?"
122Ø INPUT H
123Ø IF NOT H=D+E THEN GO TO 13ØØ
124Ø PRINT H;" CORRECT. PLAYER ";L;" WINS"
125Ø STOP
13ØØ PRINT "WRONG TOTAL. KEEP LOOKING"
14ØØ LET L=3-L
141Ø GO TO 5ØØ
```

List of variables

P	a prime number, 29 in the above program.
L	number of the player, 1 or 2.
Q(2)	the number of the cave in which each player now is.
R(2)	the number of the cave from which each player has just come.
D	this is Key 1, a random number from 1 to 9.
F	the number of the cave in which Key 2 is to be found.
E	this is Key 2, set at Ø until Key 1 has been discovered.
K	the value which will be put into Key 2.
M	a random number from 1 to 23 (23 sets the number of caves).
J	loop control variable.
N	a number between M+1 and 24, such that X=M*N−1 and X is a multiple of the prime number P.
A$	input string variable to restart after a pause, and to input the instruction F, B, or S.
H	input variable, the answer to the sum of Key 1 + Key 2.

Notes

This program, the ZX80's answer to Dungeons and Dragons, generates one out of a number of different networks of 23 caves. From each cave you can move forwards (F), backwards (B), or sideways (S) at will, and each move may take you into another cave or a dead end. The game is for two players (if you want to practise on your own, delete line 14ØØ). Both players start at Ø, from which they move sideways into cave number 2. After that anything may happen, and each player is on his own, he knows where the other one is, but not what moves he has made. If you are going to get out alive, you will need to draw a map, and a sample one is shown on the following page.

The first object is to get to cave 23, where you are given two bits of information:

Key 1 (a number from 1 to 9).
Which cave Key 2 is in.

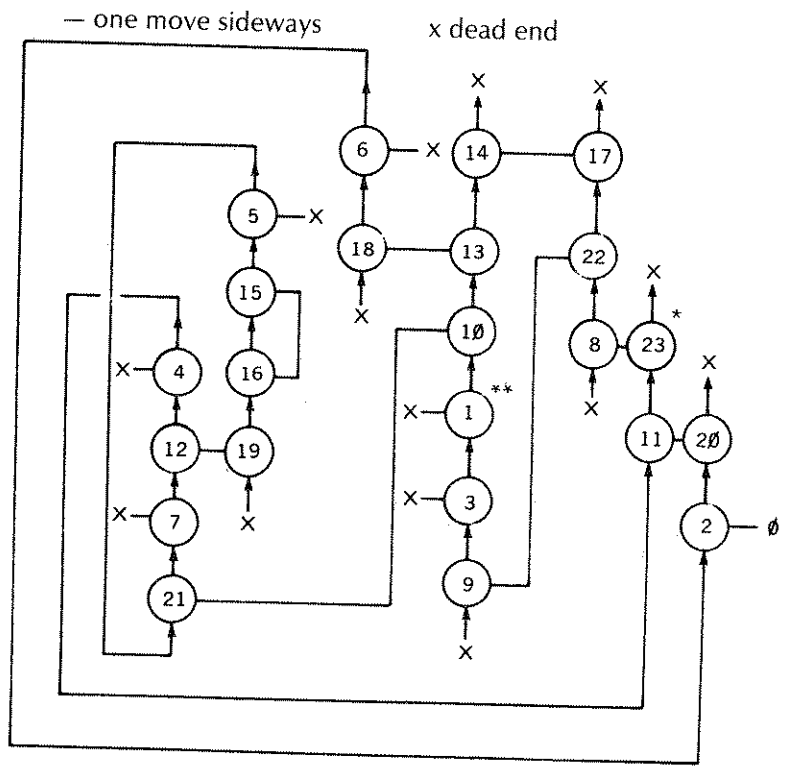
You have to make your way to this cave and find Key 2. The first player to get home to Ø, and produce the correct sum of Key 1 + Key 2 is the winner.

Lines 2ØØ to 26Ø	sets up the network of caves.
Line 53Ø	prints the present cave number.

Line 54Ø	puts the present cave number into a dummy variable.
Lines 81Ø to 88Ø	inputs F, B, or S and moves to a new cave, or stays in the same one if a dead end.
Lines 91Ø to 95Ø	prints where your move has taken you from and to where.

In the caves — a sample layout of caves

↑ one move forwards (backwards in reverse direction to arrow)



*Key 1 is always in Cave 23, which may be anywhere in the network.
**Key 2 was in Cave 1 on this occasion, but may be in any cave other than 23.
Ø Start and finish here.

Now come a number of tests:

Lines 1000 to 1030 Is the player in cave 23? If so he is given the values in variables D and F, and Key 2 is given its value K.
Lines 1100 to 1120 Is the player in the cave where Key 2 is hidden, and has Key 1 been found yet? If so then he is given the value of Key 2.
Lines 1200 to 1210 Is the player back home at 0 and has Key 1 been found? If so then he is asked the total of Key 1 + Key 2.
Lines 1220 to 1300 Is the total of Key 1 + Key 2 correct? If so the player wins.
Line 1400 changes from one player to the other.
Line 1410 loops back to line 500 for next move.

13. Multiples

```
10  PRINT
20  PRINT
30  PRINT "CHOOSE A NUMBER BETWEEN 0 AND 99";
40  INPUT NN
50  CLS
60  PRINT "MULTIPLES OF ";NN;
70  IF NN=0 THEN PRINT " (ONLY ONE)";
80  IF NN=1 THEN PRINT " (EVERY ONE)";
90  IF NN=2 THEN PRINT " (EVEN NUMBERS)";
100 IF NN=5 THEN PRINT " (IN 2 COLUMNS)";
110 IF NN=10 THEN PRINT "(ALL IN 1 COLUMN)";
120 PRINT
130 PRINT
150 FOR J=0 TO 9
200 FOR K=0 TO 9
210 LET N=10*J+K
215 IF NN=0 AND N=0 THEN GO TO 2000
220 IF NN=0 THEN GO TO 230
225 IF (N/NN)*NN=N THEN GO TO 1000
230 PRINT N;
240 IF N<10 THEN PRINT "  ";
250 PRINT "  ";
260 NEXT K
270 PRINT
280 PRINT
290 NEXT J
```



```
300 PRINT "ANOTHER NUMBER?";
310 GO TO 40
990 STOP
1000 LET A=CODE(STR$(N))
1010 LET B=CODE(TL$(STR$(N)))
1030 IF B=1 THEN LET B=0
1040 PRINT CHR$(A+128);CHR$(B+128);
1050 GO TO 250
2000 PRINT CHR$(156);CHR$(128);
2010 GO TO 250
```

List of variables

NN	chosen number for multiples.
J, K	loop control variables.
N	current number in the number square.
A, B	codes of the two digits of the number to be printed in inverse.

Notes

You are asked to input a number between 0 and 99. The program then prints out a 0 to 99 number square in which all the multiples of your chosen number are printed in inverse. A few relevant comments are thrown in for good measure.

Note that 0 times any number equals 0, so that 0 has to be shown as a multiple of every number in the square.

Lines 150 to 290 print the number square.
Lines 215 to 220 make special provision for 0.
Line 225 tests each number in the square for divisibility by the chosen number.
Lines 1000 to 1050 routine for inverse printing of one- or two-digit numbers.
Lines 2000 to 2010 inverse printing of 0.

14. Number Base Changing — Base 2 / 10

```
10 LET P=-1
20 LET D=0
30 DIM B(8)
40 LET X=0
```

```
 8Ø  PRINT
 9Ø  PRINT
1ØØ  PRINT
11Ø  PRINT
12Ø  PRINT "WHATS THE NEXT NUMBER BASE"
13Ø  PRINT ,"2 OR 1Ø?"
14Ø  INPUT B
15Ø  CLS
16Ø  IF B=2 THEN GO TO 5ØØ
17Ø  IF NOT B=1Ø THEN GO TO 13Ø
2ØØ  PRINT "ENTER DECIMAL NUMBER, UP TO 255"
21Ø  INPUT N
22Ø  CLS
23Ø  PRINT
24Ø  PRINT "DECIMAL NUMBER=";N
27Ø  PRINT
28Ø  PRINT "BINARY NUMBER=";
3ØØ  LET F=128
31Ø  IF N/F=Ø THEN GO TO 37Ø
315  LET X=1
32Ø  LET N=N-F
33Ø  PRINT 1;
34Ø  GO TO 38Ø
37Ø  IF X=1 THEN PRINT Ø;
38Ø  LET F=F/2
39Ø  IF F=Ø THEN GO TO 1Ø
4ØØ  GO TO 31Ø
5ØØ  PRINT
51Ø  PRINT "ENTER BINARY NUMBER,UP TO 8 BITS"
52Ø  INPUT A$
6ØØ  CLS
61Ø  PRINT "BINARY NUMBER=";A$
65Ø  FOR J=1 TO 8
66Ø  LET B(J)=CODE(A$)
67Ø  LET A$=TL$(A$)
68Ø  NEXT J
7ØØ  FOR J=Ø TO 7
71Ø  LET N=8-J
72Ø  IF B(N)=1 THEN GO TO 75Ø
73Ø  IF B(N)=28 OR B(N)=29 THEN LET P=P+1
74Ø  IF B(N)=29 THEN LET D=D+2**P
75Ø  NEXT J
8ØØ  PRINT
81Ø  PRINT "DECIMAL NUMBER=";D
82Ø  GO TO 1Ø
```


List of variables

P	power variable for calculating decimal numbers.
D	decimal number calculated from a binary number.
B(8)	array of up to 8 binary digits, used to calculate D.
B	choice of base (2 or 10) for the input number.
N	input decimal number, to be changed to binary.
F	power of 2, used to calculate binary number from a decimal.
A$	input binary number, up to 8 bits, to be changed to decimal.
J	loop control variable.
X	'flag' to suppress the printing of Øs at the start of a binary number.

Notes

You choose to start with a decimal or binary number, and the program converts this into the corresponding number in the other base. As written, this program is limited to 8 bits maximum in binary, or 255 maximum in decimal.

Lines 3ØØ to 34Ø	if the decimal number is divisible by the present power of 2, subtracts that power of 2 from the number and prints a binary digit 1.
Line 37Ø	if the binary number has been started, and if the decimal number was not divisible by the present power of 2, prints a binary digit Ø.
Line 38Ø	reduces to next lower power of 2. returns to start when units digit has been printed.
Lines 51Ø to 68Ø	inputs a string representing a binary number of up to 8 bits, and splits this into an array of 8 character codes.
Lines 7ØØ to 75Ø	examines the 8 characters in reverse order.
Line 72Ø	ensures no action if character was a null string, that is, no binary digit was present.
Line 73Ø	increases to next power of 2 if a binary digit (Ø or 1) was present.
Line 74Ø	increases the calculated decimal number by the present power of 2 if the binary digit was 1.

Appendix 4

Sample Answers to Exercises

Note that these are sample answers only — your own answers may be different but equally correct.

Exercise 5.1

```
1Ø  PRINT "THREE LINES GONE, ONE LEFT"
```

Exercise 5.2

The keyword is NEW

```
1ØØØ  PRINT "SINCLAIR ZX80 MICROCOMPUTER"
2ØØØ  PRINT
3ØØØ  PRINT "  MADE BY SCIENCE OF CAMBRIDGE"
```

Exercise 6.1

"A" is a literal string (see Chapter 4), and so the ZX80 will print the letter A instead of the value of the variable A.

Exercise 6.2. Exchange rates

```
1Ø  LET L=275
2Ø  LET B=69*L
3Ø  PRINT B
4Ø  PRINT "BELGIAN FR. FOR £"
5Ø  PRINT L
1ØØ  PRINT
11Ø  LET B1=5382
12Ø  LET L1=B1/69
13Ø  PRINT L1
14Ø  PRINT "£ NEEDED FOR "
15Ø  PRINT B1
16Ø  PRINT " BELGIAN FR."
```

Exercise 7.1. Miles per gallon

```
1Ø  LET M=258
2Ø  LET G=8
3Ø  LET MPG=M/G
4Ø  PRINT "PETROL MILEAGE=";MPG;" M.P.G."
```

Result is 32 M.P.G., or 32.25 M.P.G. with a calculator.

Exercise 7.2. Family transport

```
1Ø  PRINT "NAME","COLOUR","MAKE","TYPE"
2Ø  PRINT
3Ø  PRINT "DAD","WHITE","AUSTIN","CAR"
4Ø  PRINT "JOHN","GREEN","TRIUMPH","BIKE"
5Ø  PRINT "MUM","BROWN","K","SHOES"
```

Comma spacing can give troubles. Try changing line 3Ø to give DAD a VOLKSWAGEN.

Exercise 8.1

PRINT 36*9Ø/54 gives the correct answer (6Ø).
PRINT (73Ø/25)*45 gives the answer 13Ø5. The correct answer is 1314, the small error being due to integer BASIC.

Exercise 8.2. Temperature conversion

```
1Ø  LET F=77
2Ø  LET C=(F-32)*5/9
3Ø  PRINT F; " DEG. F=";C;" DEGREES C"
```

(Answer: 25 degrees C)

Exercise 8.3. Volume and weight of cuboid

```
1Ø  LET L=14
2Ø  LET B=9
3Ø  LET D=6
4Ø  LET V=L*B*D
5Ø  PRINT "VOL. OF LEAD BRICK=";V;" CUBIC CM"
6Ø  LET W=11*V
7Ø  PRINT
8Ø  PRINT "WEIGHT=";W;" GRAMS"
```

(Answer: 756 cubic cm, 8316 grams)

Exercise 9.1. Inflation

```
1Ø  LET P=8Ø
2Ø  LET Y=Ø
3Ø  LET Y=Y+1                    (next year)
4Ø  LET P=P*12/1Ø                (pay increased by 2Ø%)
5Ø  IF P<1ØØØ THEN GO TO 3Ø      (ends loop when P is 1ØØØ or
                                  more)
6Ø  PRINT "PAY IS £";P;" PER WEEK AFTER"
7Ø  PRINT Y;" YEARS"
```

(Answer: £1011 per week after 14 years)

Exercise 9.2. Chess prize

```
1Ø  LET S=1
2Ø  LET C=1
3Ø  LET TC=1
4Ø  LET S=S+1
5Ø  LET C=C*2
6Ø  LET TC=TC+C
7Ø  IF S<15 THEN GO TO 4Ø
8Ø  PRINT "TOTAL=";TC;" COINS ON ";S;" SQUARES"
```

The answer is 32767 coins on 15 squares, so one more square would cause an arithmetic overflow.

Exercise 11.1. Running average

```
1Ø  LET N=Ø
2Ø  LET T=Ø
3Ø  PRINT
4Ø  PRINT "INPUT NEXT ITEM NOW"
5Ø  INPUT X
6Ø  LET T=T+X
7Ø  LET N=N+1
8Ø  LET A=T/N
9Ø  CLS
1ØØ  PRINT N;" ITEMS SO FAR, AVERAGE=";A
11Ø  GO TO 3Ø
```

Note. Keep this program to practise editing in Chapter 12.

Exercise 12.1. Editing

Your EDITed program should give an output like this:

```
5 NUMBERS SO FAR
AVERAGE=4
INPUT THE NEXT NUMBER NOW
L S
```

Exercise 14.1. Savings

```
1Ø  LET TS=Ø
2Ø  FOR M=1 TO 12
3Ø  PRINT
4Ø  PRINT "HOW MUCH THIS MONTH?"
5Ø  INPUT S
6Ø  CLS
7Ø  LET TS=TS+S
8Ø  PRINT "£";TS;" SAVED IN ";M;" MONTHS"
9Ø  PRINT
1ØØ  IF TS>99 THEN PRINT "SEND FOR YOUR ZX80 NOW"
11Ø  NEXT M
```

Exercise 14.2. Compound interest

```
1Ø  LET C=1ØØ
2Ø  PRINT "£1ØØ AT 9 PER CENT INTEREST"
3Ø  PRINT
4Ø  PRINT "YEAR",,"CAPITAL"
5Ø  PRINT
6Ø  FOR Y=1 TO 7
7Ø  LET C=C*1Ø9/1ØØ
8Ø  PRINT Y,,C
9Ø  NEXT Y
```

Exercise 15.1. Number square

```
100 FOR J=0 TO 9
110 FOR K=0 TO 9
120 IF J=0 THEN PRINT "  ";
130 PRINT 10*J+K;
140 PRINT "  ";
150 NEXT K
160 PRINT
170 PRINT
180 NEXT J
```

Exercise 15.2. Chessboard

```
 10 FOR J=1 TO 4
 20 PRINT
 30 PRINT
 40 FOR K=1 TO 4
 50 PRINT "W B ";
 60 NEXT K
 70 PRINT
 75 PRINT
 80 FOR K=1 TO 4
 90 PRINT "B W ";
100 NEXT K
110 NEXT J
```

Exercise 16.1. Songwriter

```
100 LET A$=" MEN WENT TO MOW"
110 LET B$="WENT TO MOW A MEADOW"
120 LET C$="1 MAN AND HIS DOG"
130 PRINT "HOW MANY MEN NOW (1 TO 12) ?"
140 INPUT M
200 CLS
210 PRINT M;A$
220 PRINT B$
230 PRINT
240 FOR J=0 TO M-2
250 IF M=1 THEN GO TO 280
260 PRINT ,M-J;" MEN"
270 NEXT J
280 PRINT ,C$
290 PRINT
300 PRINT B$
310 PRINT
320 PRINT
330 PRINT
340 PRINT
350 GO TO 130
```


Exercise 16.2. Form filling

```
100 PRINT "PLEASE TYPE IN YOUR SURNAME NOW"
110 INPUT S$
120 PRINT "NOW YOUR CHRISTIAN NAME"
130 INPUT C$
140 PRINT "YOUR AGE IN YEARS ?"
150 INPUT A
160 PRINT "WHERE DO YOU LIVE ?"
170 INPUT W$
180 CLS
190 PRINT
200 PRINT C$;"  ";S$;" IS ";A;" YEARS OLD"
210 PRINT "AND LIVES AT ";W$
220 PRINT
230 PRINT "THANKS ";C$
```

Exercise 18.1. Penny tossing

```
 10 RANDOMISE
 20 LET S2=0
 30 LET S3=0
 40 LET S4=0
100 FOR J=1 TO 100
110 LET S=RND(2)+RND(2)
120 IF S=2 THEN LET S2=S2+1
130 IF S=3 THEN LET S3=S3+1
140 IF S=4 THEN LET S4=S4+1
150 NEXT J
170 PRINT "TOSSING TWO PENNIES"
180 PRINT
190 PRINT
200 PRINT "TABLE OF RESULTS"
210 PRINT
```

```
220  PRINT
230  PRINT "RESULT",,,"HOW MANY"
240  PRINT ,,,"TIMES"
250  PRINT
260  PRINT "TT (SCORE 2)",,S2
270  PRINT "HH (SCORE 4)",,S4
280  PRINT "TH OR HT (SCORE 3)",S3
```

Exercise 19.1. Areas of rectangles

```
100  LET L=RND(15)
110  LET B=RND(10)
200  FOR J=1 TO B
210  FOR K=1 TO L
220  PRINT "▨";
230  NEXT K
240  PRINT
250  NEXT J
260  PRINT
300  PRINT "LENGTH=";L;" CM BREADTH=";B;" CM"
410  PRINT
420  PRINT "AREA IN SQUARE CM=?"
430  PRINT "JUST TYPE THE NUMBER"
440  INPUT A
450  PRINT
500  IF A=L*B THEN GO TO 600
510  PRINT "SORRY, ";A;" IS WRONG"
520  PRINT
530  PRINT "AREA OF A RECTANGLE"
540  PRINT ,"=LENGTH×BREADTH"
550  PRINT ,"=";L;" X ";B
560  PRINT ,"= ";L*B;" SQ CM"
570  GO TO 610
600  PRINT A;" SQ CM IS CORRECT"
610  PRINT
620  PRINT "ANOTHER RECTANGLE ? Y/N";
630  INPUT A$
640  CLS
650  IF A$="Y" THEN GO TO 100
```

Exercise 21.1. Moving average

```
110  LET N=6
120  DIM X(7)
```



```
130  LET J=1
200  PRINT "ENTER AN ITEM"
210  INPUT X(J)
220  LET J=J+1
230  CLS
240  IF J<N+1 THEN GO TO 200
290  PRINT "LAST ";N;" ITEMS ";
300  LET S=0
310  FOR J=1 TO N
320  LET S=S+X(J)
330  PRINT X(J);"   ";
340  LET X(J)=X(J+1)
350  NEXT J
400  PRINT
410  PRINT
420  PRINT "AVERAGE OF LAST ";N;" ITEMS=";S/N
430  PRINT
440  PRINT
450  PRINT "ENTER NEXT ITEM"
460  INPUT X(N)
470  CLS
480  GO TO 300
```

For the general case of any number of items at a time, type in these lines:

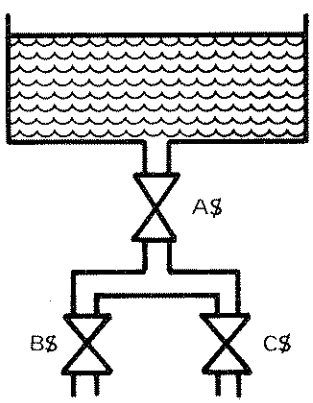
```
100  PRINT "HOW MANY ITEMS AT A TIME ?"
110  INPUT N
120  DIM X(N+1)
```

Exercise 21.2. Simple bulls and cows

```
10   LET B=0
20   DIM C(4)
30   DIM G(4)
100  FOR J=1 TO 4
110  LET C(J)=RND(6)
120  NEXT J
200  PRINT "MAKE A GUESS AT MY NUMBER"
210  PRINT
220  PRINT "4 DIGITS ALL BETWEEN 1 AND 6"
230  INPUT G$
240  LET H$=G$
250  FOR J=1 TO 4
```

```
26Ø  LET G(J)=CODE(G$)
27Ø  LET G$=TL$(G$)
28Ø  NEXT J
285  CLS
29Ø  PRINT "MY NUMBER WAS ";
3ØØ  FOR J=1 TO 4
31Ø  IF CHR$(G(J))=STR$(C(J)) THEN LET B=B+1
32Ø  PRINT C(J);
33Ø  NEXT J
39Ø  PRINT
4ØØ  PRINT "YOUR GUESS WAS ";H$
41Ø  PRINT
42Ø  PRINT "YOU SCORED ";B;" BULLS"
```

Exercise 23.1. The water tank

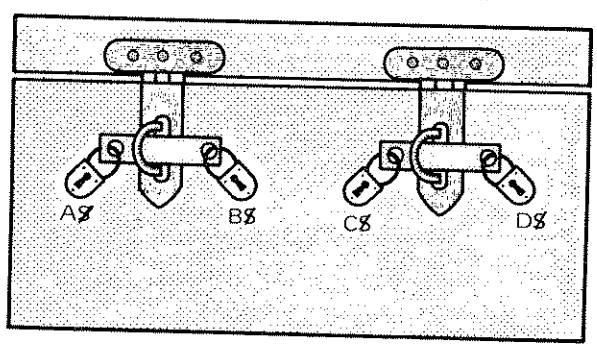


For water to run away, tap A$ must be open AND so must either tap B$ OR tap C$.

Exercise 23.2. The cashbox

```
1Ø  PRINT "ROLLCALL"
2Ø  PRINT,"P=PRESENT A=ABSENT"
3Ø  PRINT
4Ø  PRINT "BOB JONES ? ";
5Ø  INPUT A$
6Ø  PRINT A$
7Ø  PRINT "TOM JONES ? ";
8Ø  INPUT B$
9Ø  PRINT B$
1ØØ  PRINT "BILL BROWN ? ";
11Ø  INPUT C$
12Ø  PRINT C$
13Ø  PRINT "DICK BROWN ? ";
14Ø  INPUT D$
15Ø  PRINT D$
2ØØ  IF (A$="P" OR B$="P") AND (C$="P" OR D$="P")
     THEN GO TO 1ØØØ
5ØØ  PRINT
51Ø  PRINT "CANT OPEN THE BOX, LETS GO HOME"
6ØØ  PRINT
61Ø  PRINT "PRESS N/L FOR MORE OR S TO STOP"
62Ø  INPUT N$
63Ø  IF N$="S" THEN STOP
64Ø  CLS
65Ø  GO TO 1Ø
1ØØØ  PRINT
1Ø1Ø  PRINT "OPEN UP THE BOX"
1Ø2Ø  PRINT "TIME FOR A DIVIDEND"
1Ø3Ø  GO TO 6ØØ
```

Here is one design for the town cashbox:



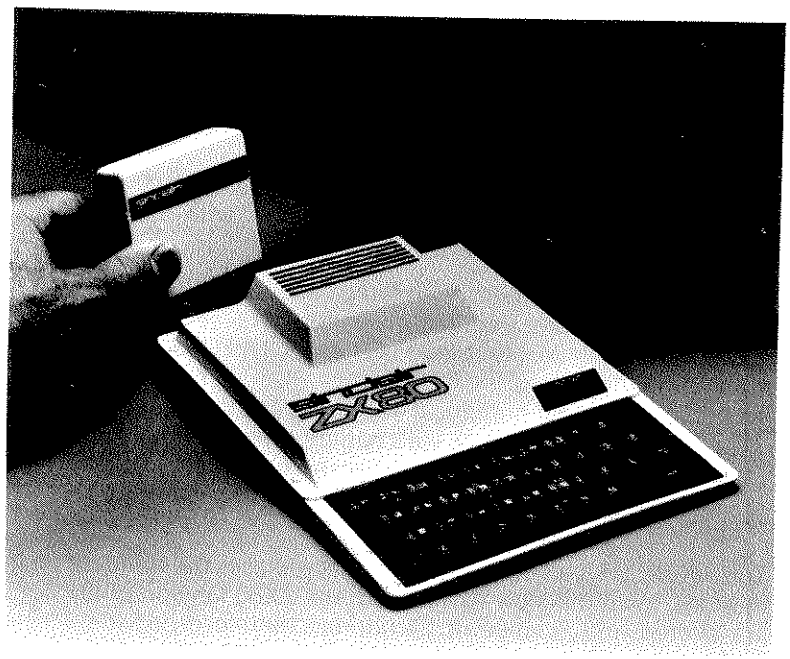
Appendix 5

The Sinclair ZX80 16K RAM Pack

You will have discovered by now that when you start writing complicated programs or handling a lot of data the 1K RAM in your basic ZX80 is soon used up! Science of Cambridge have recently made it possible for you to expand your ZX80 memory simply and economically. For about half the cost of an assembled ZX80, you receive a small box which plugs into the edge connector at the back left of your ZX80 (see photo). This gives you an extra 16K of RAM, and it represents very good value by present day standards. The expansion box gets its power from the ZX80, and the running of your existing programs is unaffected.

Your extra 16K of RAM can be used for writing more complex programs — it will of course give you a full size screen display until you have used up most of the RAM (and that will take an awful lot of programming!).

Another way of using the extra RAM is to use it to store a large amount of data. Remember that your data is saved on tape with the program, and will not be lost when you load it again, provided that you do not press RUN or CLEAR (use GO TO line number instead of RUN).



The ZX80 with the 16K RAM pack being inserted

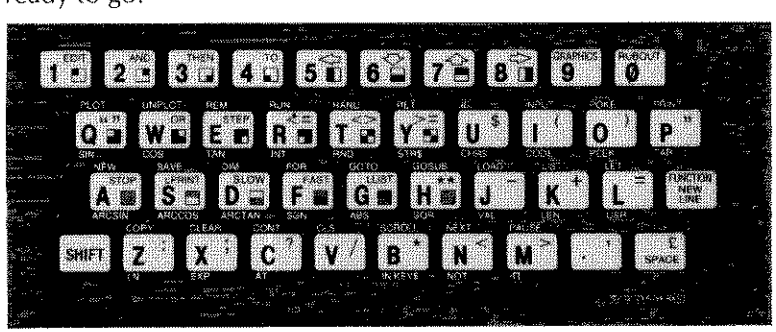
Appendix 6

The Sinclair ZX80 8K BASIC ROM

An 8K BASIC ROM is available for the Sinclair ZX80. This ROM offers many useful features not available with the standard machine:

- Floating point arithmetic with 9-digit accuracy
- Logs, trig functions and their inverses
- Graph drawing facility using PLOT and UNPLOT
- Animated displays using PAUSE
- Multi-dimensional numerical and string arrays
- Cassette LOAD and SAVE with named programs
- All characters, their video inverses and graphics may be entered directly from the keyboard
- Ability to drive Sinclair printer

The new ROM is a 24-pin DIL pack that replaces the standard plug-in ROM. With this new ROM fitted, the ZX80 keyboard functions are altered and Science of Cambridge therefore provide a new keyboard overlay (see below) with the extra functions printed on it. Simply fix the overlay on top of the original keyboard and you are ready to go.



Index