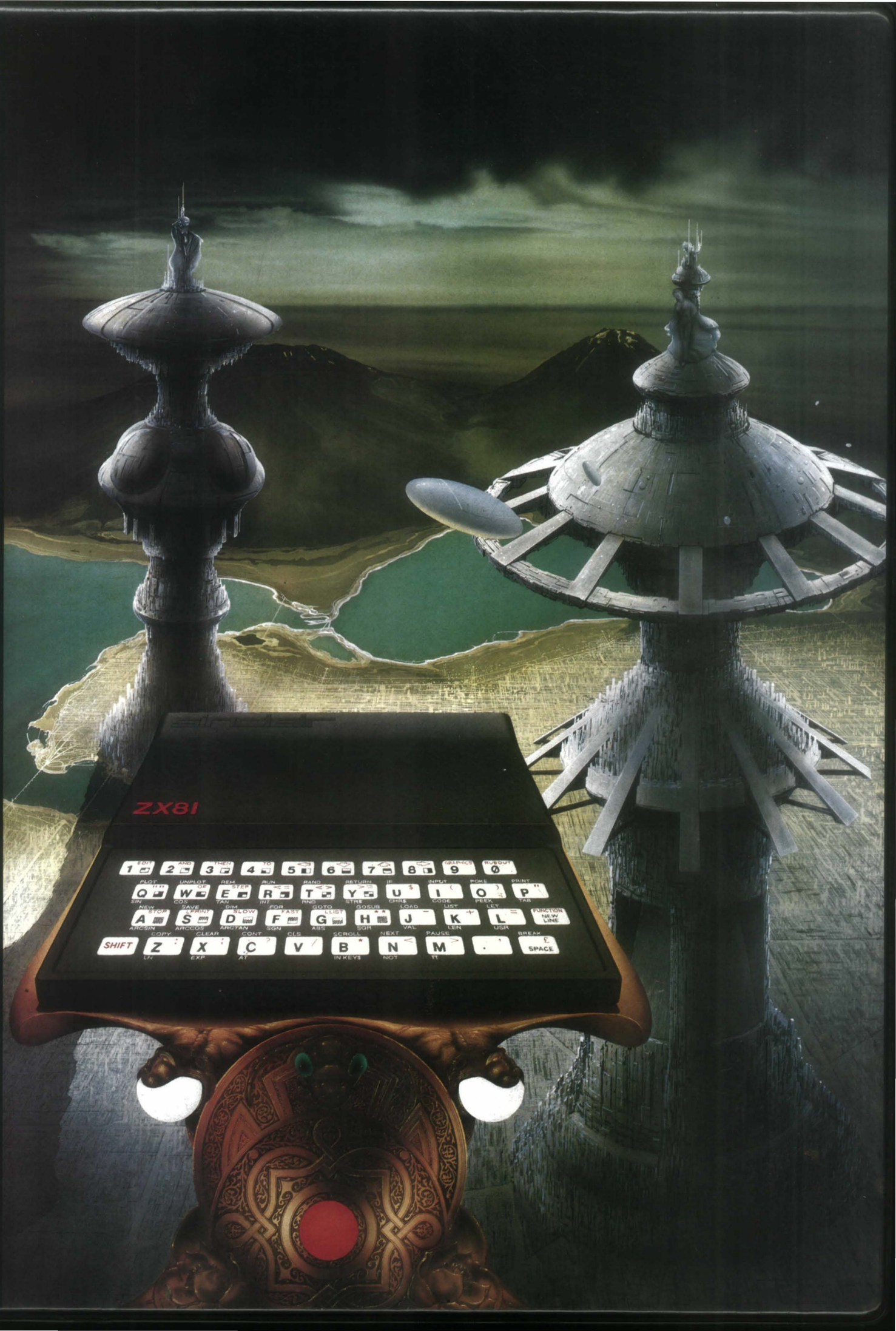
LEARNING LAB

ZX81



ZX81 LEARNING LAB

by Trevor Toms

First Edition

6 King's Parade,
Cambridge CB2 1SN
England

FRONT COVER: Illustration by Jim Burns of Young Artists, specially commissioned by Sinclair Research Limited.

Contents

INTRODUCTION

THE COURSE

This course is intended to teach you how to write programs for your Sinclair ZX81. It will not happen overnight, but you should have a pleasurable time learning.

The most important part of learning to program is *experience*. For this reason, the course is liberally sprinkled with examples of small working programs which you can enter into your ZX81. It also contains many problems for you to solve using your computer, and specimen answers are given for you to check your results against. Tapes are included with the course, containing programs which you can study to learn certain programming techniques.

As you work through the course, you will be asked questions from time to time, the answers will be found on the following page, so cover them over to prevent your eyes gradually wandering across! As you would expect, there is always more than one way of answering the questions, so the answers given are only models showing you one way that you could have gone about solving the problem. Obviously I have tried to direct you towards answering the question in the same way that *I* have, so hopefully you will be able to carry on to another topic quickly. If your answer doesn't match mine, then don't worry, but read my solution carefully and see if it helps you to understand more fully what the text was saying.

Each chapter also includes a set of *exercises*, which do not have solutions provided. You may like to try these exercises in order to practise the topics you have learnt.

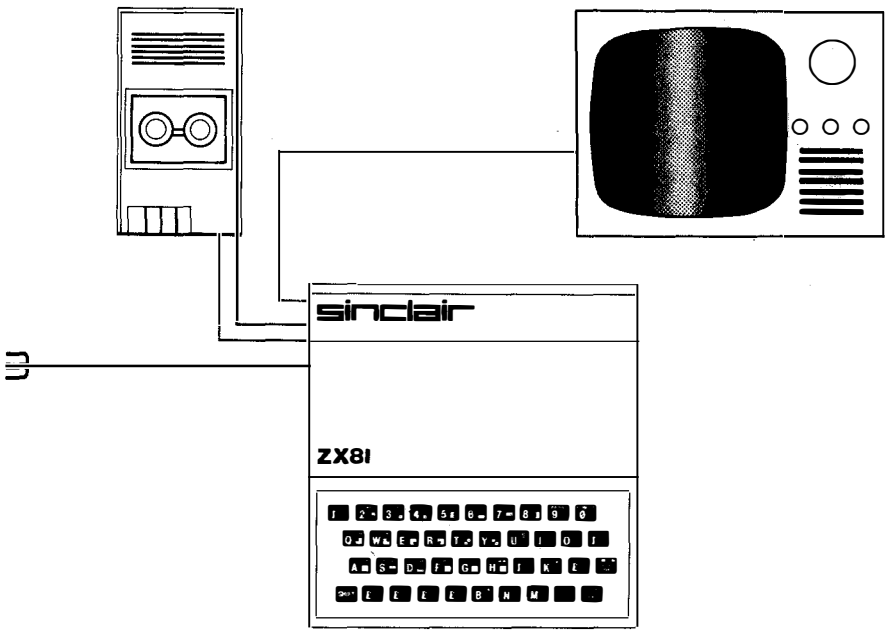
Enough of that – let's try and get your computer going so that you can see what it looks like.

First of all, take a look at the keyboard. You will notice that each key has several different things written on it. All the words *above* the keys are called **KEYWORDS**. Whenever I refer to a keyword in the text it will be printed in **BOLD TYPE**. This means that you do not need to type the whole word out, but just press the key with that word above it. Don't worry – the ZX81 knows what you are trying to do and will spell the whole word for you.

SETTING UP

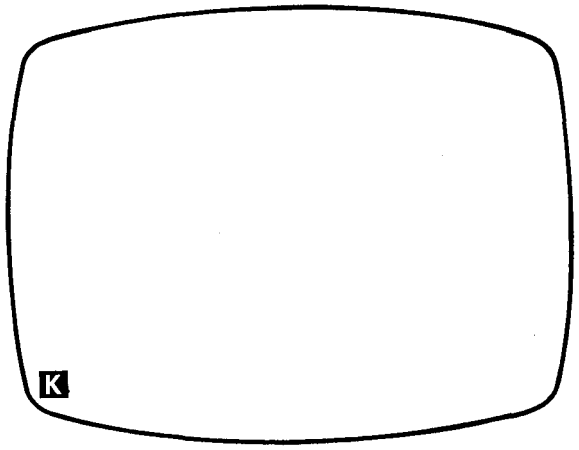
OK, first let's switch everything on. Here's what to do:

1. Connect the ZX81 to the aerial socket of your TV using the single lead supplied. The plugs are different at each end, so you can't get it wrong(!). One end plugs into the large socket at the side of the ZX81.
2. Use the double lead to connect the ZX81 to the EAR and MIC sockets on your cassette recorder – on the side of the ZX81 you will see a guide to tell you which connector goes into which socket.
3. If your recorder has playback tone controls, then set these to maximum treble, and set the volume level to nearly the maximum.
4. Plug the mains adaptor into the mains, and the jack plug into the "9V DC IN" socket on the ZX81 and ensure that the mains is on. Here's a diagram to help you:



Now switch on the TV and turn the sound down. You'll probably find that the screen is full of "snow", which means that you're not tuned in to the correct channel. Choose a spare channel knob on your TV, and tune it to channel 36 (just go from one end of the tuning to the other if your set doesn't have any tuning guides). You should notice at some point that the screen goes a "solid" grey colour (even on a colour set), with a small symbol in the bottom left-hand corner of the picture that looks like **K**. This symbol is actually the letter "K" in a black box. Once you've homed in on this symbol (from here on we'll call this symbol the CURSOR) you can adjust the fine tuning, brightness and contrast to make the picture nice and sharp.

If you have problems trying to get the picture shown below, then refer to Chapter 1 of the Sinclair Handbook (supplied with your ZX81).



It's better to use a spare channel knob/button, as you can leave it set up for the next time you use your computer.

Insert the cassette tape supplied with this course that contains "STARTERS" (look at the index on each tape). Now press the "J" key (which has **LOAD** over it). Instantly, the word **"LOAD"** appears at the foot of the screen! Note also that the cursor symbol **K** has changed to **L**. This tells you that the next key you press will give you a single letter or character just like a standard typewriter.

You should now type:

"STARTERS"

The " character is given by pressing both the SHIFT key and the P key together. You will find it easier to press the SHIFT key first, then WHILE HOLDING THIS KEY DOWN, press the P key. Now type the word STARTERS just like you would on a conventional typewriter. Now we need another quotes character (remember – SHIFT and P).

Before we go on, this has brought up a couple of points that are worth discussing. First, whenever you want to type one of the red symbols in the top right-hand corner of each key (like the " character you've just typed), you need to press the SHIFT key at the same time. These SHIFTed symbols are printed in red on the keyboard. Secondly, and probably more importantly,

WHAT IF YOU MAKE A TYPING MISTAKE?

Don't worry – the ZX81 lets you correct any mistakes you might make (if you're anything like me, then you have to hunt around the keyboard to find each letter, and you do it all with just two fingers . . .). Let's suppose that you typed:

LOAD "STR

and realise that you've made a mistake. Look at the top right-hand key on the keyboard (the 0 key). In red (remember what that means?), you'll see the word RUBOUT. Pressing SHIFT and 0 together will make the ZX81 rub the last character out, and it will disappear from the screen. Try it – you can always type it back again. Press SHIFT/0 again, and the *next* letter will disappear, and so on. So if you'd typed: **LOAD** "STRTERS", you could rub out the last six letters (including the quotes) and type them in again. The RUBOUT key actually rubs out the character to the *left* of the cursor (the **K** or **L** symbol).

Let's carry on. On your screen you should now see:

LOAD "STARTERS"

If you don't, then rub it out (see above), and start again.

We are now about to start the ZX81 off, looking on the cassette tape for something called "STARTERS". If you play the tape, it just sounds like peculiar buzzing noises, but it actually makes sense to the ZX81. Rewind the tape, press the PLAY button on your recorder, and QUICKLY afterwards press the key marked NEWLINE on the ZX81 (it's on the right-hand side). The screen will now change completely, and for a few seconds it will look like the vertical hold needs adjusting (but don't do it!) Then it will go mad for a while (about 30 seconds) and finally will come to rest with the bottom of the screen showing:

0/0

If it goes back to looking as though the vertical hold is out, then the tape has gone too far, and you should start again. Refer to a section at the end of this book titled "Common Problems and Solutions" for more information.

You can now rewind the tape and switch the cassette recorder off, as it has served its purpose for the time being. What you have just done is to load a computer program called STARTERS from cassette tape and you can now type:

RUN

(remember: it's printed in bold type, so you only need to press the key with RUN above it. Which key is that?)

RUN is a command which instructs the ZX81 to start running the program it has loaded. You'll use this command a lot. As always, whenever you want the ZX81 to act upon something that you've just typed, you must press NEWLINE. After a few brief flickers, you will see:

HI THERE
WHAT IS YOUR NAME?

Enter your first name, followed by NEWLINE to tell the ZX81 that the name is complete. Now you can work your way through the questions, entering your answers followed by a NEWLINE when the answer is complete. If you make a mistake, use SHIFT/0 to rub out unwanted letters/characters. If the ZX81 does nothing after you've typed your answer, it probably means that you haven't pressed the NEWLINE key! Try again.

You'll find after a while that a message

9/9999

appears on the bottom of the screen. This means that the program has finished, and the ZX81 is waiting for you to run something else, or even to run the same program again (if you want to, then type **RUN** again). Try giving different answers to the questions. You'll see that your ZX81 is actually checking up on your responses and reacting accordingly. You may get bored answering the same questions all the time. In that case, answering QUIT followed by the NEWLINE key to any of the questions will stop the program from carrying on, and will cause 9/9999 to be shown as mentioned above.

By now, you have had an opportunity to see what your ZX81 looks like when it's running. Not all programs are as elementary as the STARTERS program, and you'll soon find that your computer is capable of doing much more than just asking questions and looking at the answers.

Before we start delving into the ZX81 in more detail, this is an appropriate point to think about exactly what the ZX81 is.

THE ZX81 IN CONTEXT

The ZX81 is a *computer*, like the many others that assist businesses, armed forces, police, gas and electricity boards and even the DVLC at Swansea. So what makes the ZX81 different from all these? Why can't the police use a ZX81 to control their criminal records? There are several reasons why different computers can and cannot be used in certain environments, so let's first see what a computer comprises.

(a) The central processor

The central processor itself can be extremely small – the central processor inside your ZX81 will fit comfortably onto the tip of a finger, but since we humans cannot easily handle something so small, it needs to be packaged in a casing that we *can* manipulate. The ZX81 contains the Z80 central processor chip which is used throughout the world in many different computer systems. The Z80 chip is one of a new breed called microprocessors, they are rapidly finding uses in all walks of life – TV games, washing machines, lift control mechanisms, to name but a few. But not all central processors are so small. Mainframes (so called because of their size) can occupy a room, but they can also perform their "instructions" many times faster than the ZX81 and can perform many independent tasks simultaneously. A typical mainframe computer (central logic portion) can cost from £50,000–£100,000!

(b) Memory

The computer on its own is useless – it needs "instructions" to drive it. This is where "*memory*" comes in – the instructions are stored in a memory which the computer can constantly look at to see what it is to do next.

The ZX81 contains two types of memory – ROM and RAM. The ROM is used to allow you to "talk" to the ZX81 using the touch keys, and for the ZX81 to "talk" to you using the television screen. It also acts as a "translator", converting the words and symbols that make sense to *you* into symbols that the Z80 chip can understand. The RAM is used to hold everything that you ask the ZX81 to do – any calculations performed, and instructions you ask it to obey – all these are put into the ZX81 RAM.

Memory is measured in "K" units (this will be explained later in the course), the ZX81 contains 8K of ROM and 1K of RAM – i.e. it contains eight times more ROM than RAM.

(c) Backing store

Backing store (or storage) is another piece of jargon. Backing store is used to hold instructions and data that is not required in the computer's memory for the time being. The actual *types* of backing store vary considerably. The ZX81 uses a domestic cassette recorder, as you have seen when you asked the ZX81 to **LOAD** the "STARTERS" program.

You can probably appreciate that cassette recorders are slow, and it wouldn't suit the police to keep searching through C90 tapes looking for details of a known criminal! Mainframes use *disk packs* to hold large volumes of information, any piece of which can be obtained by the computer in less than a hundredth of a second. These disk packs can hold up to 200,000K of information – compare that with the 1K RAM inside the ZX81! As you become accustomed to the amount of information you can hold in a ZX81 with or without the 16K RAM pack, you may like to think of these sizes again, and try to consider why the police could not even *start* to use a ZX81. Obviously cost has a great deal to do with the quantity and speed of access. A mainframe disk drive unit can cost in the region of £20,000 and must be kept in an air-conditioned room to avoid dust particles ruining the finely engineered mechanisms.

More recently, *floppy disk* drives have been introduced. As the name implies, the disks are flexible, and are quite suited to the "hostile" environment of a house or office. They are much more in line with microcomputers, as they cost between £200 and £500, can store up to 500K of information on a disk, and are fast enough for small computer systems. These disks can usually retrieve information within half-a-second. As a comparison, the time taken to load the "STARTERS" program from a floppy disk would be around 2–5 seconds instead of 1–2 minutes from cassette tape.

Remember that just as you can change the cassette in your recorder, so can disks be changed in disk drive units to give more storage space. A disk pack for mainframes costs (roughly) £80–£100, while a floppy disk costs about £5–£10. Cassettes of a reasonable quality cost £1 each.

(d) The keyboard and display unit

Now we've got a central processor, memory and backing store. But we *still* need to be able to *operate* the computer, or to be able to direct it some way. It is usually controlled from a *keyboard*, which invariably looks much like a standard typewriter with a few extra keys on it. The ZX81 also uses a keyboard, although it does not have conventional "push keys", but has instead a series of pressure sensitive pads. They are laid out in the same form as a typewriter, however.

You need to see what you're typing – it would be pointless otherwise – and so the ZX81 connects to a standard television set which "echoes" your typing. When you are happy with some typing, you press NEWLINE, and the ZX81 then acts on what you have typed. Any responses that the ZX81 needs to make are also sent to the *display unit* – in this case, the TV – so that *you* may act on them.

Larger computer systems make use of a *visual display unit,* or VDU for short, which is like a combined

typewriter keyboard and TV screen rolled into one unit. You've probably seen similar units appearing in banks, airports, hotels and even some garages. These units are designed for fast typing and legibility – and therefore cost in the region of £500—£1000 per unit.

(e) Languages

Earlier above we saw how the ZX81 ROM acts like a "translator", turning your instructions and commands into a form that the Z80 can understand. The comparison can be stretched a bit further yet. Translators can understand many languages – French, Spanish, American(!) – and turn them into your own language. Similarly, computers can recognise many *computer languages,* each being a different way of giving commands to the computer.

The language used by the ZX81 is called BASIC, which stands for Beginners All-purpose Symbolic Instruction Code. There are many other computer languages – COBOL, FORTRAN, PASCAL, CORAL to name but a few – each making certain aspects of computing easier, but others more complex. BASIC was chosen for the ZX81 because of its initial simplicity. Don't be fooled though, BASIC is quite capable of holding its own against many other languages.

(f) Printers

You will probably find that most of the work you do with the ZX81 will use the display (TV screen) to show results. Sooner or later you may find it necessary to keep a typed or hand-written copy of some of the results. This is where a *printer* comes in. It allows you to put displayed results onto printed paper. The ZX printer can copy the entire contents of the TV screen to paper in roughly 12 seconds.

A mainframe uses printers that can print roughly 2000 lines of printed results *every minute!* The cost is high, and a typical mainframe printer costs £15,000–£20,000. Laser printers are even faster, but I won't frighten you with the cost!

I hope by now that you can see the ZX81 in perspective. The ZX81 has brought computing within the reach of ordinary people, so that we can all see what a computer is capable of doing, and explode some of the myths and fears surrounding them.

The discussion of languages above needs to be examined a bit more closely. Since each different computer system is designed by different manufacturers with different markets in mind, you should be aware that no two versions of the same language are ever completely identical. By "two versions" I do not mean that your ZX81 will differ from your friends, but that the BASIC found in a ZX81 will not quite be the same as that found in other microcomputers. There are many reasons why this is so, and many attempts have been made to produce a Standard BASIC. This is also true for other languages, but no-one has succeeded yet in creating a "machine-independent" language.

If you ever start to use a different type of computer that uses BASIC, then you can rest assured that your ZX81 has given you a good grounding in the language – you will have few problems in learning the new techniques.

Appendix A contains a comparison of ZX81 BASIC with several other personal computer systems. Hopefully this will give you some idea of the differences.

Summary

What have you learnt in this Introduction?

– how to switch on the ZX81 and connect all the various leads to the TV and cassette recorder.
– how to load an example program from tape.
– how to correct typing mistakes.
– how to use the SHIFT key to get the letters/characters marked in red.
– what the inverse character K or L is called (the cursor).
– when to use the NEWLINE key.
– what computer systems are, what they comprise (memory, keyboard, backing store, etc), and where the ZX81 fits into this spectrum.

In the chapters that follow, you'll find out what each of the commands can do, and how you can build up a program by putting lots of commands together in a sequence.

This introduction has not posed you any questions, but all the next chapters will. Please try to answer each question honestly and only look at the answer (on the following page) if you're stuck. There's no point in merely reading the text and looking straight at the answers, as you'll find yourself getting quickly out of your depth. If you get *REALLY* stuck, then skip over the question and carry on with the next section.

From now on, I'll abbreviate some of the ideas as follows:

1. I will refer to the NEWLINE key by **nl**, so if you were to type your name again, you would type, for example,

   JIM **nl**

2. Whenever I want you to load a program from tape, I'll say:

   LOAD "program name"**nl**

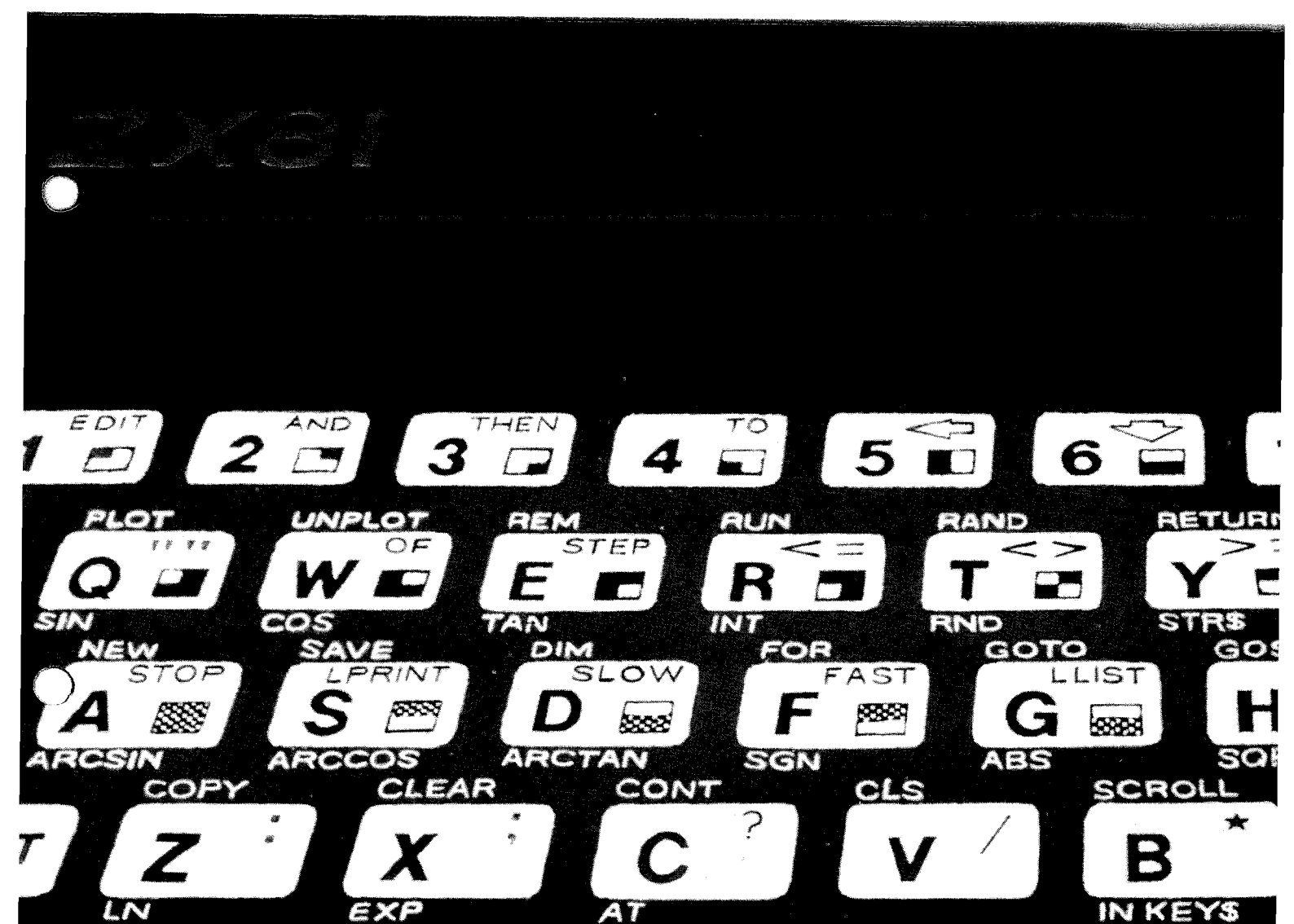
   The name within quotes will be given to you. I'll also assume that you'll start the tape running before you press the NEWLINE key, and that you watch out for the finish symbol . . .

   Ø/Ø

   Note that future programs that you load will pass over several programs to reach the one that you have asked for. This may take up to three or four minutes, so don't panic if nothing happens instantly. Should the ZX81 get all the way to the end of the tape without the "finish" symbol showing, refer to a section at the end of this course entitled "Common Problems and Solutions". Once the program has loaded, you should start it running by typing:

   RUN nl

OK then, now have a go with the ZX81 acting as a clever calculator! This is the starting point in Chapter 1.


EDIT
AND
THEN
TO
PLOT
UNPLOT
REM
RUN
RAND
OF
STEP
SIN
COS
TAN
INT
RND
STR$
NEW
SAVE
DIM
FOR
GOTO
STOP
LPRINT
SLOW
FAST
LLIST
ARCSIN
ARCCOS
ARCTAN
SGN
ABS
COPY
CLEAR
CONT
CLS
SCROLL
LN
EXP
AT
IN KEY$

CHAPTER 1

The Computer as a Calculator

This chapter is divided into four major sections. The first section will show you how you can calculate the area of a rectangle and in doing so you will learn more about the cursor, discover what a "report code" is, and what a "syntax error" means. You will also use a new command, **PRINT**.

Section 2 shows you how to solve more complicated calculations using brackets, and leads on to a discussion on priorities of operators.

Section 3 will show how the computer can store numbers in variables and use them again in numeric expressions. This section introduces the **LET** command and a method of representing numbers called "scientific notation".

Finally, section 4 covers mathematical functions available on the ZX81.

The ZX81 can do much more than even a full scientific calculator, and so this chapter contains some fairly mathematical working. It is important to get a good idea of what is going on even if you never use some of the facilities that the ZX81 can offer.

1.1 INTERACTION WITH THE ZX81

1.1/1 CALCULATING AN AREA

You are probably aware of the formula for the area of a rectangle, but just in case:

area of rectangle = length × width

so that, for example, the area of a rectangle 6in wide by 3in long would be:

3×6=18sq.in

Fairly easy, and you probably wouldn't bother to use a calculator. But what if the dimensions were 1.533 wide and 27.92441 long? Would you dash for pencil and paper, or your calculator? On your calculator, you would work it out by saying:

1.533	(display shows 1.533)
×	(display still shows 1.533)
27.92441	(display shows 27.92441)
=	(display shows . . . what?)

So now let's try this on the ZX81.

1.1/2 ARITHMETIC OPERATORS

First, we need to find the four keys for add, subtract, multiply and divide. We also need to find the equals key.

The four operations are denoted by:

Operation	*Symbol*	*Where on keyboard*
addition	+	SHIFT/K
subtraction	−	SHIFT/J
multiplication	*	SHIFT/B
division	/	SHIFT/C

So if we want to enter 2×3 (as on a normal calculator), we would type:

2*3

and if we wanted 35÷7 we would type:

35/7

1.1/3 USING COMMANDS

But what about the "equals" sign? There is one on the keyboard, but this is not for giving calculated results, as you will see in later sections. What do we do then? This is where the COMMANDS come in.

A command is an instruction to the ZX81 to *do* something. In our case, multiplying two numbers, we want to instruct the ZX81 to tell us the answer, or to **PRINT** the answer on the TV screen. To get our answer, then, we would type:

PRINT 1.533*27.92441 (remember about **BOLD FACE** words? See the Introduction if you've forgotten.)

But nothing happens, try it.

Question
Why doesn't anything happen?

Answer
In the Introduction, you were told that the ZX81 does not act on anything until the NEWLINE key is pressed. If you remembered, then well done. If not, you may like to read that section of the Introduction again.

Now press the NEWLINE key. There's the answer to the calculation – at the top left-hand corner of the screen! If you're in doubt, check it on your calculator.

1.1/4 CHECKING SYNTAX

There's another reason why nothing might have happened – that is a SYNTAX ERROR. It's an awkward word, but means that the ZX81 cannot understand what you have typed. If you've got a syntax error, then you'll notice the symbol **S** somewhere in the middle of the line that you've typed (just like the **K** and **L** symbols we've called the CURSOR). The **S** shows the point at which the ZX81 stops being able to understand you. Rub out the characters following the **S**, and try again.

1.1/5 DIRECT COMMANDS

What we have just done is to give the ZX81 a DIRECT COMMAND to obey. We asked it to **PRINT** the result of a calculation, and that's exactly what it has done. At the bottom of the screen, you can see the symbol

0/0

again. This means that the ZX81 has completed its task with no errors. Just as you can get overflow errors on a calculator, you can get errors on the ZX81. A full list of these can be found at the back in the reference sections, but here we'll give you an idea of what they mean.

The first number (in front of the /) is the report code. A report of 0 means that no error occurred, and so we can continue. The second number need not worry us at the moment, since it will always be 0 for direct commands. We shall come across other values for this later in the course. If the first number is not 0, then you can look the meaning up in the reference section, headed "report codes".

1.1/6 ANOTHER LOOK AT THE CURSOR

Now for a bit more about the cursor.

As you've seen in the Introduction, the cursor can be either **K** or **L**.

The cursor is really a marker that indicates several different things to you:

1. It indicates where the next typed letter will go on the screen.

2. It indicates what the ZX81 is expecting next and how it will interpret it.

In the second case, the letter in the box defines what you should be entering:

K is a Keyword, or command. The ZX81 will give you the keyword *above* the key that is pressed next.

L is any Letter or character.

F is any Function, obtained by first pressing SHIFT/NEWLINE. The next key pressed will give the function underneath the keys (e.g. **COS, SQR** etc). More on this in a later section.

G indicates that the next letter/character will be interpreted as a Graphics character. Again, we'll meet these later on in another chapter.

Whenever you press the NEWLINE key, the ZX81 checks what you have typed and if it finds an error, or something it can't understand, it will not accept the line. Instead, it puts the SYNTAX ERROR marker **S** at the point on the line where it stops understanding what you have entered. Rub out the wrong letters to the right of the error and re-type them. For example if you typed

PRINT 27.3:6 **nl**

meaning 27.3*6, then after pressing NEWLINE, you would see

PRINT 27.3**S**:6**L**

You should use RUBOUT to remove the 2 characters on the right and re-type them. RUBOUT always takes off the character to the *left* of the cursor. Suppose that you had entered a long line and the syntax error was back at the beginning, like this:

PRINT 27.3**S**:6+(44.877−(15/1Ø0))*38.7**L**

Would it be necessary to rub out the whole of the line just to correct that single mistake? The answer is no. Look at the top row of keys on the keyboard. The 5 key and 8 key have red arrows on them pointing left and right respectively. If you press SHIFT/5, the cursor will move to the left one position, and SHIFT/8 will move it to the right. Try typing this:

PRINT 27)6*33.8 **nl**

You will get the **S** marker in front of the) character, while the cursor **L** is at the right-hand end of the line. Press SHIFT/5 several times and the **L** will move back along the line towards the) character. When it is next to it, press SHIFT/Ø (RUBOUT). This removes the wrong). Now press SHIFT/K and you will see the + symbol being put in instead of the). Press NEWLINE and the corrected line will be taken in and the answer will be given. Make good use of these "arrow" keys – they will save you a lot of time correcting mistakes.

Question
Name two of the three COMMANDS that you have used so far.

Answer

You have met **LOAD**, **PRINT** and **RUN**.

If you got the answer wrong, then go back and read the beginning of Section 1.1/3 again.

Now you have done one calculation, you can probably go straight on to try others. Here are a selection of small problems for you to solve. The questions are written without telling you which symbols to use, so if you've forgotten, you'll have to re-read Section 1.1/2. The answers are given for you to check. The first example is given in full.

Question 1

If a shopkeeper has 127 boxes of chocolates in stock, and he orders a further 229, how many will he have in total?

Answer
We need to find: 127+229=? which we would enter into the ZX81 as:

PRINT 127+229 **nl**

Answer: 356. (did you *really* use the ZX81?)

Question 2
A mouse chews through 2.5gm of cheese each night in the kitchen. How much will it get through in 105 days? Give the answer in grammes.

Question 3
A boy has found 37 conkers, but in his rush to get to school on time, 18 fall out of his pocket. How many are left?

Question 4
Seventeen women win a prize of £11,135.00 in a joint competition. How much does each woman win?

Answers

2 262.5gm. Your entry to the ZX81 should have read

PRINT 2.5*105 **nl**

3 19. Your entry should have been

PRINT 37−18 **nl**

4 £655. This time you should have put

PRINT 11135/17 **nl**

or

PRINT 11135.00/17 **nl**

1.2 COMPLICATED CALCULATIONS

1.2/1 SOME COMPARISONS

By now you will have grasped the concept behind the command **PRINT**. It can do quite a bit more than just add a few numbers together and print the answer. In fact it is one of the most powerful commands on the ZX81. You are probably thinking, "This is all very well, but my pocket calculator can *still* do lots more things than that". Some of these are:

- chain calculations together
- store results in a memory and recall them when needed
- calculate squares, square roots, percentages
- use trigonometric functions, like tan, sin, cos

Have no fear, the ZX81 can do all of these, and more besides. Let's look at some of the points raised.

– chain calculations together.
This can be done by putting more items into the **PRINT** command, for example,

PRINT 27.5*33/16.8 **nl**

– store and recall from memory.
We'll come onto this in section 1.3 on Variables

– calculate squares, square roots, etc.
You can use the operator ** on the keyboard to raise any number to the power of another, e.g.

PRINT 55**2 **nl**

would print the value of 55 squared. The ** function is on the H key (using SHIFT). So

PRINT 21**3.5 **nl**

gives the value of 21 raised to the power 3.5

– Use trig. functions.
All of these, plus many more, are available on the ZX81. They are covered in section 1.4.

1.2/2 CHAINED CALCULATIONS

Whenever you perform chained calculations on a calculator, you soon run up against the problem of *priority*. Certain types of operators take priority over others. For example, if we wish to calculate:

2+3*6

on a calculator, the answer would be obtained by imagining (or putting, on some calculators) brackets around the two multiplied numbers,

2+(3*6)

Here, then, the priority of calculation is *higher* on multiplication. Try the example on a calculator and see which answer you get:

2+3*6 (with no brackets) =?

Calculator answer	***Calculated as***
30	(2+3)*6
20	2+(3*6)

Mathematicians choose the second solution as preferable and state:

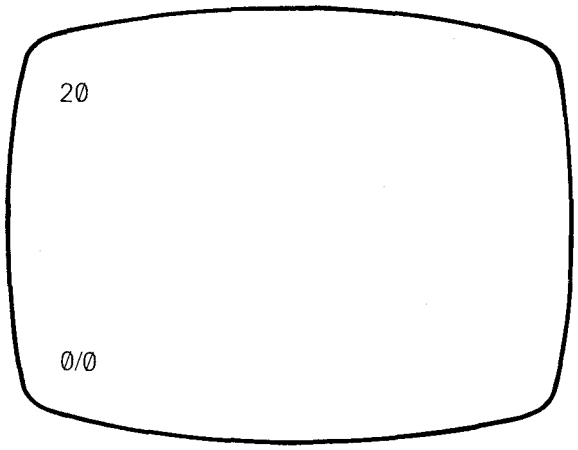
multiply and divide take priority over add and subtract

Question
Now try the same calculation on the ZX81. First write down how you think you should enter the problem, then try to enter it and see what the result is. The "brackets" can be found by pressing SHIFT/I and SHIFT/O.

Answer
You should have written down:

PRINT 2+3*6 **nl**

If you did, you should have seen



If you got any report code other than zero, you've done something wrong. If you didn't get 20 in the top left-hand corner, then you've probably mistyped something. Go back and try again. If you don't understand how we arrived at the solution at all, then read sections 1.2/1 and 1.2/2 over again, paying particular attention to the early stages of the **PRINT** command.

1.2/3 BRACKETED EXPRESSIONS

Inevitably, you'll come up against a formula that cannot be arranged in such a way as to be entered onto one **PRINT** command. This can happen for a variety of reasons, the most likely at this stage is that you cannot get the numbers ordered to make the calculation give the correct result. For example, in the case given above, what would you do if you wanted the answer to have given 30, so that the calculation is interpreted as

(2+3)*6? (look back to section 1.2/2)

Well, the ZX81 is quite happy to be given brackets around the calculations that you want done first. In fact, you can put brackets within brackets (these are called NESTED brackets) to any depth that you require.

Let's try it. Enter the following:

PRINT (2+3)*6 **nl**

What did you get? Unless your typing is awful, you should have seen the answer 30.

OK, so "multiply" and "divide" come first. But what happens if you've got the same type of operator? Suppose we had something with two divisions next to each other? Under these conditions, the ZX81 always works from left to right along the calculation. If you had 4/2/2, then the answer would be 1 rather than 4. The ZX81 has treated it as (4/2)/2 rather than 4/(2/2). We'll work through a complete valid calculation:

PRINT 5*(7+(90/5)/(55−46)) **nl**

Taking this a section at a time, we can break it down as follows:

1. The innermost brackets are calculated first, i.e. (90/5) and (55−46) giving the two results 18 and 9 respectively.

2. The next outer set of brackets are calculated, i.e. (7+(18)/(9)). But here we must take priority into account, and since division is a higher priority than addition (see section 1.2/2) the division will be done first, giving us the result (7+(2)). This addition gives us our next result 9.

3. Finally, the outermost calculation is done, which is 5*9, giving the end result of 45.

********** PHEW! **********

Question
What would you expect the result of the following calculation to be? Don't try it on the ZX81 until *after* you have written an answer down on paper.

PRINT 4*(8+4/2−(9/3))

Answer
You should have written 28. If you got anything else, then read sections 1.2/2 onwards again.

Time for a new topic.

1.3 VARIABLES

1.3/1 THE CONCEPT OF VARIABLES

On a traditional calculator, a memory is normally available with three of four keys to allow you to store and recall the number contained in it, or add to/subtract from it, and to clear it (same as storing zero in it). On the ZX81, we can store a number in a "memory" by using the command **LET**. The big difference between a calculator and your ZX81 is that the ZX81 can have as many of these "memories" as you need!

We can identify a "memory" by a single letter, such as D, or X. Just like algebra, we can put any value into these variables (or memories). In algebra we would say,

Find the value of: 4y+3 when y=2.

The important part here is "when y=2". This could be done on the ZX81 by stating:

LET Y=2 **nl**
PRINT 4*Y+3 **nl**

You should notice the following points –

1. We have done this in two stages The first stage was to put a value to the "memory" we have called Y, and the second was to print the result of an equation using Y.

2. The ZX81 will remember the value of Y from the first stage to the second. It has been put away for later use.

Now try typing those two lines on your ZX81. You'll find the command **LET** on the "L" key. What answer do you get? If you get a report code 2/Ø on the bottom of the screen instead of Ø/Ø, then it means you've mis-typed something. Try again.

We call Y a *variable* rather than a "memory", because it can have any number value. As you've seen in the Introduction, the word "memory" has a slightly different meaning in the computer world, so we will stop using the word "memory" as of now.

Question
How many variables do you think the ZX81 can hold at the moment?

Answer
26 variables, since there are 26 letters of the alphabet, and so far we have said that a variable can only be identified by a single letter. Read on for further information on variable names.

1.3/2 VARIABLE NAMES

We can call a variable not by just a single letter, but by a group of letters of any length we like. So some valid variable names would be

Y
HGTYYC
FRED
TRY2
TEE42
and so on.

The last two examples show that numbers can be used as part of the name of a variable. As long as the name starts with a letter, the rest of the name can be any mixture of letters or numbers. You can even put spaces in if you like, just to make the names easier to read! For example:

DOLLS EYES AND FISH HOOKS
CAPN PUGWASH
TIME 4 BED

are all suitable names, whereas

4TEE2
WYNOT?
FESTER BESTER-TESTER

are *not* suitable since they either start with a number, or contain characters other than letters or numbers (? and -).

Question
By now, a few minutes will have passed since you typed the command "**LET** Y=2" into the ZX81. Let's check to see if it's still there. What would you type to see if Y is still holding the number 2?

Answer

PRINT Y **nl**

If you got the correct answer, then you can skip to section 1.3/3.

The command **PRINT** lets us look at the result of any calculation or equation. So if you were asked in algebra to solve

What is the value of y when y=2?

you should have no trouble at all. That is *exactly* what we have done when we asked you to check if the variable Y was still holding the number 2. You have effectively typed these two commands:

LET Y=2 **nl**
PRINT Y **nl**

Try reading sections 1.3/1 and 1.3/2 over again.

1.3/3 ALTERING VARIABLE VALUES

You could wait all day – even longer – and whenever you tried, printing Y would still give you the answer 2. This can change in a number of ways. One way is to switch the ZX81 off. A bit drastic, but nevertheless, it works. Variable Y would be lost completely, and if you typed **PRINT** Y **nl**, you would see

2/0

at the bottom of the screen. This report (remember that the report code is the first of the two) means that the variable has not been found. You have not given it any value as yet.

Another way is to alter the value of Y. You can do this by another **LET** command, thus:

LET Y=99 **nl**

and now if we print Y, we'll see the number 99.

In the last section, you saw that a variable name could be a group of letters, not just a single letter. This means that you can make your variables have names that give an indication of their meanings, like:

LET MILK PRICE=20 **nl**
or
LET QUANTITY=10.30 **nl**
or
LET BREAD PRICE=42 **nl**

Type those few in, and then **PRINT** them. You'll notice that they don't alter even though you've entered more variables in between. The spaces in the names are cosmetic only – if you miss one out when you type them in, the ZX81 will still know what you mean.

Now you know how to store some numbers inside the ZX81, we'll show you how much more you can do with them. The command **LET** does not just allow you to store numbers in variables, but also allows you to state that one variable is given the value of another. To give you an example of this, we could say –

LET Y=QUANTITY **nl**

and the variable Y would be given the same number that QUANTITY is holding.

Question

What number is variable Y now holding?

Answer
10.30.
You could check this by entering:

LET Y=QUANTITY **nl**
PRINT Y **nl**
PRINT QUANTITY **nl**

Read section 1.3/3 again if you got it wrong.

1.3/4 NUMERICAL EXPRESSIONS

Now comes the *real* fun. You may have noticed that the command **LET** has two parts to it – a variable name followed by an equals sign, which gives the name of the variable that we want to alter in some way, and something following the equals sign which gives the value that we want the first variable to be given. This second part of the **LET** command is called . . .

a numerical expression

You'll meet this phrase a lot, so lets explain what it means by giving you more insight into the **LET** command.
This "second part", or numerical expression, that follows the equals sign can be a mixture of variables and arithmetical functions (add, subtract, multiply, etc), so that the "first part" can be given almost any value we please. For example,

LET Y=MILK PRICE*2 **nl**

is quite acceptable and would set Y to the value 40, since MILK PRICE is holding the number 20, and (20*2) is 40.

Question
Assuming MILK PRICE still holds 20, and BREAD PRICE still holds 42, write down what you think Y would hold after the command:

LET Y=MILK PRICE+BREAD PRICE **nl**

You can set this up on the ZX81 by typing

LET MILK PRICE=20 **nl**
LET BREAD PRICE=42 **nl**
LET Y=MILK PRICE+BREAD PRICE **nl**

Write your answer down first, then enter

PRINT Y **nl**

Answer
Did you write down 62? If so, then well done. Carry on with the next section. If not, then maybe what follows will help a little.

We have already seen how we can use the **PRINT** command to give the results of various problems involving mixtures of numbers and arithmetical functions (add, subtract, etc). In exactly the same way, we can put this result into a variable for later use with the **LET** command. If we try:

PRINT 20+42 **nl**

we get the answer 62. Similarly, if we try:

PRINT MILK PRICE+BREAD PRICE **nl**

then we also get the answer 62, since we have previously allocated MILK PRICE and BREAD PRICE with the commands:

LET MILK PRICE=20 **nl**
LET BREAD PRICE=42 **nl**

What we have *now* done is to go one stage further and put this result into variable Y, by stating:

LET Y=MILK PRICE+BREAD PRICE **nl**

1.3/5 MORE EXPRESSIONS

We can now define "a numerical expression" as any combination of numbers, numeric variables, and arithmetical functions which give a value.

Here are some valid numerical expressions:

```
MILK PRICE*7
4.3+8/2
(27+3)*9
(27+MILK PRICE)*9
(BREAD PRICE+MILK PRICE)*QUANTITY
```

1.3/6 SCIENTIFIC NOTATION

The ZX81 is quite at home using *scientific notation* (or exponential notation) to represent number values. If you're uninterested in this form then you can skip this section, but there are occasions when the ZX81 may give you a result in this form, so be sure you know that it can happen!

This type of notation means that the ZX81 can handle massive numbers and minute numbers with ease, giving you a great deal of flexibility. How does it work? Instead of entering a number as (say) 1000, you could enter it as:

1E3

for example **LET** Y=1E3 **nl**

What does it mean? The number following the letter E (which stands for Exponent) gives the number of decimal places to shift the number in front of the E. If the second number is *positive*, then you shift the first number *left*. If the second number is *negative*, you shift to the *right*. Here are a few examples:

1E3	1000	(shift 1 left 3 times)
2.5E−2	0.025	(shift 2.5 right twice)
5.1223E8	512230000	(shift 5.1223 left 8 times)

Got the idea? You can type in a number using this form *at any time you like*. The ZX81 will know what you mean.

Question
This question has two parts – try to answer both before you look at the next page.

(a) Write down what you think the result of these commands will be:

```
LET RUBY=12 nl
LET RUBY=RUBY+4 nl
PRINT RUBY nl
```

That second line may have got you guessing! You can type it in and try it to check your answer.

(b) Give another way of writing these numbers:

6.4E3
1.4E–3
1Ø55
Ø.075

Answer

(a) The answer is 16.

Your ZX81 is clever enough to realise that you were trying to use the same variable, and has taken the correct value for it. The value on the right-hand side of the equals sign is always worked out before the variable on the left-hand side is altered. In our case, the value of RUBY+4 is worked out, and as we can see, has the value 16, and then this is stored in – oh, the same place – RUBY. It makes no difference to the ZX81 at all.

(b)
```
6400
0.0014
1.055E3
7.5E-2
```

How did you get on with that lot? If part (a) gave you some trouble, then read sections 1.3/3 and 1.3/4 again.

If part (b) was tricky, then try section 1.3/6 once more. You can always type a few examples of your own into the ZX81 just to check that I'm not having you on!

1.3/7 ASSIGNING VALUES

Up until now we have continually talked about "storing a value in" a variable. This is extremely cumbersome to keep saying, and besides, we find cases where we can't really fit this expression in and make good sense. You're now going to meet another piece of jargon. When we "store a value" in a variable, we call it

assigning a value to a variable.

So, for instance, if we want to put the value 3 in a variable called JIM, we would say

Assign the value 3 to the variable JIM

or

Assign 3 to JIM

for short. This would be entered into the ZX81 with the command:

LET JIM=3 **nl**

Question

Write down which of the following two statements is true and which is false:

(a) we can assign 4 to Y
(b) we can assign Y to 4

Answer

(a) is true and (b) is false.

Why?

It makes sense if we write it out in full and say that:

(a) we can put the value 4 in a variable called Y

but it doesn't make any sense to say:

(b) we can put the value Y in a variable called 4

since we have seen that variable names are groups of letters, and 4 is a number!

It is true to say:

we can assign MILK PRICE to Y,

as this is the same as entering

LET Y=MILK PRICE **nl**

on the ZX81.

Once again, it's time for another topic. Now we're going to investigate more maths besides the usual add, subtract, multiply and divide.

1.4 MATHEMATICAL FUNCTIONS

1.4/1 WHEN FUNCTIONS ARE USED

Sooner or later, you'll need to perform some more demanding tasks on your ZX81. If you're involved in engineering or statistics, then you may need additional mathematical functions to help find the solution to a problem.

As with most scientific calculators, the ZX81 is equipped with a full set of "trig" keys – **SIN, COS, TAN** etc. Let's look to see how they are used. We will start by trying to find the diagonal length of a rectangle – for those unfamiliar with the problem, we'll be using Pythagoras' theorem to help us. This formula states that the length of the diagonal is equal to the square root of the sum of the squares of both the sides of the rectangle. In formula, this is written:

$D=SQR\ (L^2+B^2)$ where D is the required length,
L is the length of the rectangle
B is the breadth.

We've already met the "square" operation – this used the symbol ** (SHIFT/H) to raise any number to the power of another. But what about the square root? Look at the words printed *underneath* the keys on the keyboard. These are called *functions*. The function we are after here is the "square root" function, and this can be found under the "H" key, labelled **SQR**. How do we get at it? This requires two actions.

1. Press SHIFT/NEWLINE (the NEWLINE key has the word FUNCTION written in red on it). You'll notice that the cursor changes to **F**, indicating that the *next* key pressed will give the function value of that key, or the word printed underneath.

2. Press the appropriate key for the required function.

But don't do it yet – we need to see how they're used in a calculation first.

A mathematical function key operates on whatever follows it. So the "square root of 16" would be entered as:

SQR 16

using these keys:

SHIFT/NEWLINE	to enter function mode
H	to get **SQR** function
16	as normal.

This value can be printed, or assigned to any variable, since the function itself is a numeric expression.

Here are a few legal examples of the **SQR** function:

PRINT SQR 93.566 **nl**
LET K=**SQR** (MILK PRICE+BREAD PRICE) **nl**
PRINT SQR (Y**2) **nl**

Notice that the second and third examples have got brackets surrounding the items after the **SQR** function. This forces the ZX81 to work out the bracketed expression first, and *then* calculate the square root.

The last example merely prints the value of Y, as it has first calculated Y squared from the expression Y**2, and then printed the square root of it – in other words, Y!

Be careful to use brackets where you want the function to operate on several items. Try these two examples to see why:

PRINT SQR 9+16 **nl**
PRINT SQR (9+16) **nl**

It is extremely important to remember that functions *do not* affect the expression that follows – they merely operate on the value of the expression. So if you wrote:

LET TESTER=25
PRINT SQR TESTER

then the variable TESTER will *still* hold the value 25 after that second command has been obeyed. The **SQR** function has only had a look to see what TESTER holds without affecting it in any way. If we wanted to alter TESTER, then we would write:

LET TESTER=**SQR** TESTER

Let's go back to the original problem which was to find the length of the diagonal of a rectangle.

Question

Write down what you would enter onto the ZX81 to find the length of the diagonal of a rectangle whose whose sides are 4cms and 3cms?

You know how to get the square of a number, and the formula has been given to you above, so all you really need to do is translate that formula into something that the ZX81 will understand, and change all the symbols in the formula to actual numbers.

Answer
The answer could be:

PRINT SQR (3**2+4**2) **nl**
or
PRINT SQR ((3**2)+(4**2)) **nl**
or
PRINT SQR (9+16) **nl** (if you were lazy!)

If you got the answer wrong, try reading section 1.4/1 again.

1.4/2 MORE FUNCTIONS

Now that you've seen how to use the **SQR** function, it only needs to be said that all the other mathematical functions listed below work in a similar way, but just give different solutions.

Please don't think that you have to learn this lot – you'll find it much more useful just to look and see what there is, and later on when you get involved in more programming, use the reference section at the end to give you a complete list of all functions and commands.

Function	*Where on keyboard*	*What it does*
SQR	H	Gives square root
SIN	Q	Sine
COS	W	Cosine
TAN	E	Tangent
INT	R	Supplies the next lower integer. All fractions are removed e.g. **PRINT INT** 27.66 **nl** would print 27. All the decimal places are lost. Note that **INT** −27.66 gives −28 as it is the next *lower* value
ASN	A	Inverse of **SIN**
ACS	S	Inverse of **COS**
ATN	D	Inverse of **TAN**
SGN	F	Gives a result depending on the sign of the argument. −1 if negative, 0 if zero, +1 if positive. E.g. **PRINT SGN** −44.9 **nl** would print −1
ABS	G	Gives the absolute (or mod.) value, thus the result is always positive. E.g. **PRINT ABS** 24.3 **nl** and **PRINT ABS** −24.3 **nl** would both print 24.3
LN	Z	Gives the natural logarithm
EXP	X	Gives natural anti-log, or exp

There is also one mathematical function which has no bracketed expression following it – π (found under the M key), which is displayed as **PI**. This, when seen in a calculation, gives the value 3.1415927. In the strict mathematical sense, **PI** is not a function, but a constant.

As you look down that list, you're probably wondering what on earth some of these functions are for. Who wants to know what the ABS value of a number is? Well, as we delve into programming more and more, you'll gradually learn that the computer is really an idiot that has to be told *everything*. It makes no assumptions. Under certain conditions, it may need to check that the number it is working on is positive (for example, trying to find **SQR** −1 would be rather senseless), and so we have already found a need for the ABS function. You'll find more examples showing most of these functions in use later on.

Question

Let's suppose that we have two variables, called VARA and VARB, which are holding two unknown numbers. What would you enter into the ZX81 in order to see the integer part of their sum? Look down that list of functions again. One of them is the clue to the answer, but you'll need to be careful when you write it out, because we are after the *sum* first.

Answer

PRINT INT (VARA+VARB) **nl**

Try it by entering:

LET VARA=25.4 **nl**
LET VARB=31.769 **nl**

and then the solution as given above. This will print the answer 57, since 25.4+31.769 gives 57.169, and the integer part of this is 57.

If you got the correct answer, then carry on. If not, then you'll need to read this section over again. If you used the wrong function but otherwise got the correct answer, then don't worry, just look at that table once again and check the answer to the problem over to get it clear in your mind.

Summary

We have covered quite a lot of ground in this chapter, and you'll have done extremely well if you've followed it all with no real problems. Some of the more complex points will become clearer in time as you work through more and more examples – and there are a lot more examples to come, both in the text and on tape.

In the next chapter, we start to look at how to build up your own programs, based on some of the ideas that you've met so far.

Here's a list of the topics that we've covered in this chapter:

– how to use the ZX81 as a calculator, and how the four arithmetic operators are used.

– what a SYNTAX ERROR is, how it can be corrected using the cursor keys SHIFT/5 and SHIFT/8.

– what the report codes represent.

– how to use the **PRINT** command to display calculated results of fairly complicated expressions, and how to make use of the different priorities given to operators.

– What a VARIABLE is, how it can be assigned a value using numeric expressions.

– when SCIENTIFIC NOTATION can be used and how to interpret it.

– how we can use mathematical functions to increase the calculating power of the ZX81.

Exercises

1. Calculate the area of a circle with radius 4 inches.

2. Calculate the volume of a sphere with radius 4 inches.

3. Try to find:
   (a) the largest number that the ZX81 can hold using scientific notation,
   (b) the smallest number (positive) that the ZX81 can hold using scientific notation.

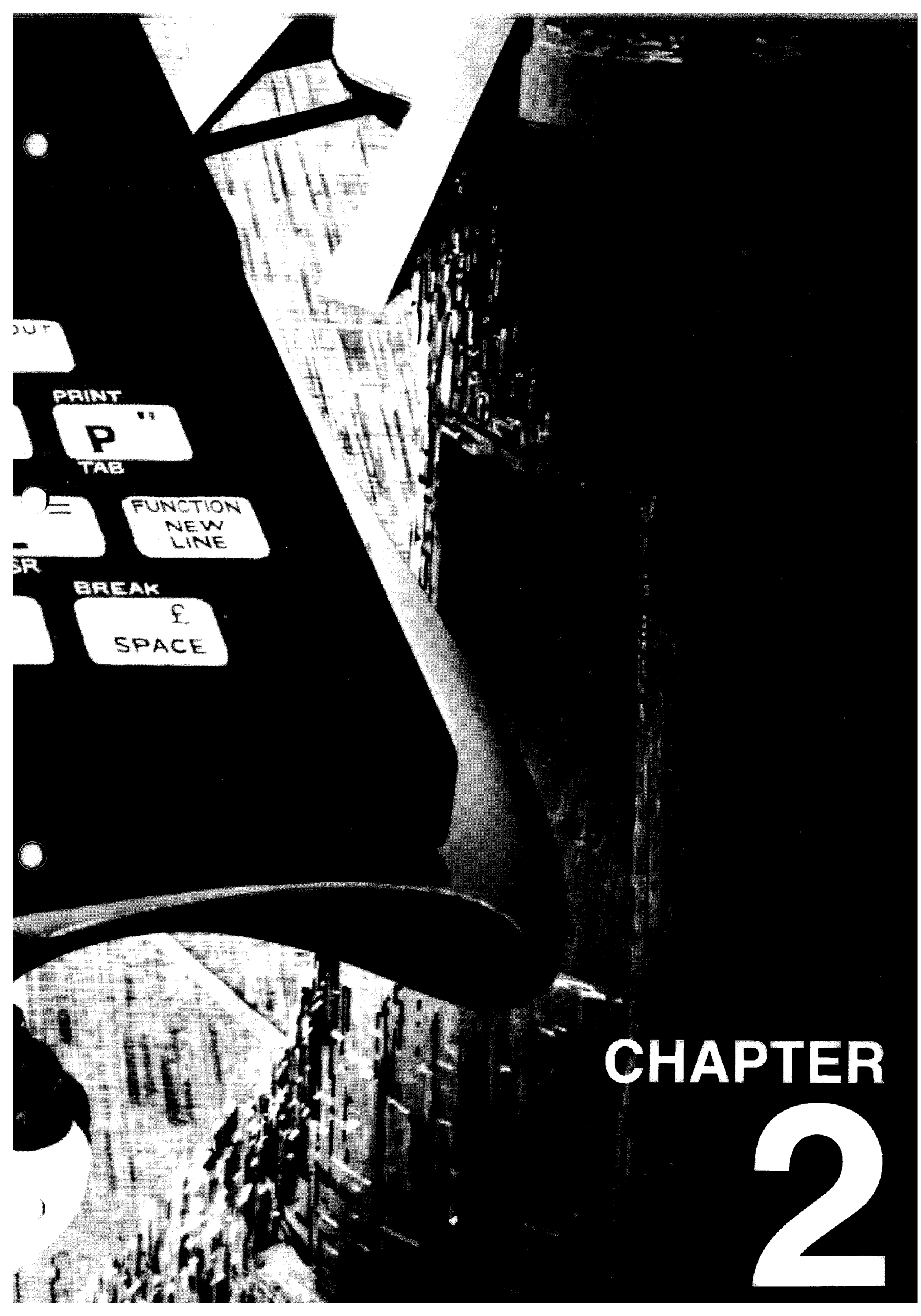
4. Investigate what happens when you try entering numbers with more than nine digits, or when the result of a calculation exceeds nine digits.

5. Use the fact that a 45 rpm single cost 32p in 1960 to discover the increase cost as a percentage (but don't cry when you discover the answer!).

6. A television screen has 625 lines which are sent 50 times per second (UK). How many lines are sent in one hour?

7. If a cheap rate long distance call costs 9p for five minutes (when dialled direct on your own phone . . .), how much will a call lasting 37 minutes cost?

8. Roger Bannister was the first person to run a mile in just under four minutes. Approximately how many yards did he cover each second?



CHAPTER 2

Starting to Program the ZX81

In this Chapter you will see the beginnings of computer programming. The chapter is divided into four major sections.

Section 1 shows how program statements are created, and defines a computer program.

Section 2 deals with making programs re-usable, so that they can be used for repetitive calculations. you will also see how programs can be annotated with comments of your own, and how to set about unravelling an existing program.

In section 3 you will be shown how to edit a program so that any errors can be removed.

The last section gives some initial details of print formatting, so that your displayed results can be given more appeal.

2.1 SIMPLE PROGRAMMING

2.1/1 RESETTING THE ZX81

The first thing we are going to learn is how to clear out the ZX81. It's a bit like pressing the "C" button on a calculator, or switching it off and on (this also works on the ZX81, but it is not recommended unless there is no alternative). The command to use is

NEW nl

and all previous calculation results, all variables that have been stored, in fact *everything* will be cleared out. Whenever you want to start something fresh, use **NEW** first.

2.1/2 HOW TO STORE COMMANDS

Let's go back to our very first problem in Chapter 1, which was to calculate the area of a rectangle. What you did there was to specify a command like:

PRINT LENGTH * BREADTH **nl**

which, if LENGTH and BREADTH had been set up using the **LET** command, would print out the appropriate area. But it becomes a trifle tiresome typing these lines in each time, and as we already know, the ZX81 is quite capable of remembering the value of any variables that we set up, so why can't it remember some commands as well? The answer is that it can. It can store up quite a lot of commands, and then obey them when you are ready, instead of one at a time as we have seen so far.

You will probably say to this: "But whenever I type a command into the ZX81, it is always obeyed when I press NEWLINE. So how can I stop it from doing this?" The solution is to put a *number* in front of the command when it is typed. This identifies the command uniquely, and is called

a LINE NUMBER.

Let's look at our example again. Try typing:

1 **PRINT** LENGTH * BREADTH **nl**

You do not need to use the "space" key when you are typing this, because the ZX81 will automatically space the various items out for you. Do this before you read on.

Something quite unexpected has now happened. Instead of getting some answer to a calculation, your line has been taken in and put at the top of the screen! The ZX81 has filed it away for you to use later. At this stage, don't worry about the symbol that has appeared in between the number 1 and the command **PRINT**. We'll come on to this shortly.

Of course, the command that has just been filed away requires two variables to be set up before it can calculate the area – LENGTH and BREADTH. We could do that by entering two direct commands using **LET** to assign values to LENGTH and BREADTH. Or, as we now know, we could save the lines for later, along with the **PRINT** commands that have already been saved.

Problem

What do we do if we want a command to be saved inside the ZX81 for later?

Answer

The command should have a *line number* in front of it.

If your answer was wrong, then read section 2.1.2. over again, following the examples thoroughly.

2.1/3 COMMAND SEQUENCING

Now we come against a *real* problem. If we save two more commands inside the ZX81 (in this case, two **LET** commands which assign values to LENGTH and BREADTH) then how can the ZX81 know which one is to be done first? And which one second? Well, you may have already guessed. It is the *line number* that tells the ZX81 which command is to be obeyed first. The lowest numbered command will be done first, then the next lowest, and so on.

But in our example, we called our line number 1, and you can't get much lower than that! OK, so we've made a mistake. We *should* have called it line number . . . what? If you thought "line number 3", then you're on the right track. We want to add two extra lines – two **LET** commands which will set up the variables LENGTH and BREADTH before the **PRINT** command is obeyed. So let's do it all again. Type these lines:

1 **LET** LENGTH=12 **nl**
2 **LET** BREADTH=57.33 **nl**
3 **PRINT** LENGTH * BREADTH **nl**

You should have noticed these points:

(a) Line number 1 has been replaced with the new **LET** command that we've just typed in. Remember that for the future.

(b) All the other lines that we have entered have been put underneath each other at the top of the screen.

Question

What should you do if you now realise that line number 2 *should* have read

2 **LET** BREADTH=58

What can you do about it?

Answer

You should type the corrected line in again. It will replace the line of the same number. Read section 2.1.3. again if you got the wrong answer.

2.1/4 DEFINITION OF A PROGRAM

Now you have three commands inside your ZX81 just waiting for you to say "GO" or something. The command that you should use, and you've already used it, is

RUN nl

Try it. After a flash, on the screen you'll see the answer to the calculation, just as if you'd typed those three lines in as direct commands one at a time. The big difference is that those three lines are still there – sitting inside your ZX81 waiting to be run again, and again, and again . . . until you get bored watching the same numbers come up on the screen all the time.

How do we look at our saved commands after we've run them? Here is a new command for you.

LIST nl

Type it in. You'll find the screen shows you your saved commands again. In this instance you could have just pressed NEWLINE to get the same effect. These saved commands have a special name:

a COMPUTER PROGRAM

What you have just done is to enter a program into your ZX81 and run it. And this is what programming is all about, entering commands into a computer, and running them in sequence.

Question

How does the ZX81 know which command in a program is to be obeyed first?

Answer

By looking at the line numbers. The lowest numbered command is the one that is obeyed first, and so on. Read section 2.1.3. over again if you got that one wrong.

2.2 MAKING PROGRAMS RE-USABLE

2.2/1 THE INPUT COMMAND

We now know how to store program commands for later use, and how to run them, but you must admit that our example is rather limited in use. In fact it can only ever give you the same result. It would be nice it we could somehow set up those variables LENGTH and BREADTH each time we run it. But we don't want to enter them in as program commands each time, because we might just as well type the whole thing in as direct commands. The ultimate would be for the ZX81 to stop running the program and wait for you to type in the length and breadth, then carry on to give the solution.

No problem. There is a command called **INPUT** which does just that. It allows us to assign any value to a variable while the program is being run.

Let's change the program. In fact, let's get this program from one of the tapes that you got with the course. Can you remember how to set the cassette recorder up and how to **LOAD** something from tape? If not, look back to the Introduction. There you were talked through the loading of a program called "STARTERS". Read that again to remind yourself.

Set the tape up and type

LOAD "AREA1" **nl**

When it's loaded (you'll see 0/0 at the bottom of the screen), you can switch the cassette off. But don't type **RUN** at this stage, because we want to look at the program, not see how it runs as yet.

Question

There are two parts to this question.

1. How do you look at the stored program commands? What should you now enter?
2. How would you remove *everything* from the ZX81?

Answer
1. You should enter **LIST nl**
2. Type **NEW nl**

Check back over section 2.1. "Simple Programming" if you failed either part of that question.

These last few paragraphs have raised some more points that you should take note of:

(a) the program you are looking at on the screen was typed by me – the author of this course. And yet there it is on your TV screen! It was all kept on that cassette tape. So you'll be able to enter programs once only and keep them almost forever. That'll come in useful.

(b) even though we have stored some commands in a program, we still need to enter direct commands to make the ZX81 do what we want it to – like **LOAD** or **LIST** or **RUN**.

2.2/2 COMMANDS AND STATEMENTS

You will gradually see that there are two distinct types of commands. Those that are really intended for using within a program, and those that are used to control what the ZX81 is to do for us next. We call those commands that are held in a program "**STATEMENTS**", and those that control the ZX81 are called "**COMMANDS**" just like we have been using up until now. Some of the ZX81 keywords, such as **INPUT**, have no real value when used as a direct command while others (like **RUN**) are not really used as statements within a program.

Back to the example program called "AREA1". What do you notice about the line numbers? They go up in tens rather than ones. This is for a *very* good reason. If, at a later date, I wanted to add some new lines into my program, I could type them in as, say, line 12, or line 4, and the ZX81, since it obeys the commands in line number sequence, will obey my new program in the order in which I want it to. If, however, the lines were numbered 1, 2 and 3 – as we had earlier, then how would I manage to add a new command which was obeyed after line 1 but before line 2? Well, I suppose you would say "Use line 1.5", but unfortunately, line numbers are only allowed to be whole numbers, no fractions, no negatives, and no line number zero. This is why I have used line numbers in steps of ten.

Question
Write down what you think the result would be if you typed the following program into the ZX81 and ran it (don't enter it, as we still have our example program in):

```
10 LET X=5
20 LET Y=10
30 LET Z=X + Y
25 LET Y=3
40 PRINT Z
```

Answer

Did you spot that the line numbers are not in order? The answer that would be printed in 8. The program when rearranged into line number sequence is:

```
10 LET X=5
20 LET Y=10
25 LET Y=3
30 LET Z=X + Y
40 PRINT Z
```

and so line 25 effectively does away with line 20, as Y is given the value 10, and then immediately given the value 3. Read section 2.1 "Simple Programming" again.

2.2/3 DATA ENTRY

Notice the two statements using the **INPUT** command. They specify the name of the variable that is to be entered from the keyboard when the program is run.

```
10 INPUT LENGTH
20 INPUT BREADTH
```

Run the program and see what happens. Type **RUN nl**, and the screen goes blank, with just the cursor at the bottom of the screen containing the **L** symbol. This is the **INPUT** command doing its job. The ZX81 is waiting for you to type in something. But we know that the program we are running is expecting a number to put into the variable called LENGTH. So obviously, we must now type in a number. Don't forget to press NEWLINE when you've done it.

Now what? The screen blanks out again, and there is another **L**-cursor at the bottom. This is for the second **INPUT** command, requesting the BREADTH variable. Enter another number, followed by NEWLINE. This time, we get the result on the screen just like we have had before.

Question

How could you rewrite the following program to allow different values to be entered when the program is run?

```
10 LET COST=20.00
20 LET VATRATE=0.15
30 PRINT COST * VATRATE
```

Answer

```
10 INPUT COST
20 INPUT VATRATE
30 PRINT COST * VATRATE
```

In practice, you would probably leave line 20 as it was, since the VAT rate only changes at infrequent intervals, and it would become tiresome to keep entering 0.15 each time.

If your answer was wrong, then read section 2.2/3 again.

2.2/4 PROGRAM ANNOTATION

You can see drawbacks in the example program "AREA1". First, when the screen went blank, you had no idea what you were supposed to be entering. OK, so we had just looked at the program, and could see that it was a variable called LENGTH that was expected. But how would you know if someone else had written the program, and asked you to try it out? Obviously, some sort of indication is required on the screen as the number is entered.

Secondly, in order to understand what the program was doing, we had to look at it line by line. If you bought a program from a shop, you wouldn't be very pleased if you had to unpick it all line by line in order to understand how to use it! This second point is important.

There is a command called . . .

REM

which allows us to put comments (REMarks) into a program which are ignored by the ZX81. Whatever follows the command **REM** is up to us. The ZX81 is just not interested. For example:

```
10 REM THIS LINE DOES ABSOLUTELY NOTHING
20 INPUT LENGTH
...
...
```

We can now put REM statements into our program to let someone know quite a lot about how the program runs, who wrote it and when, what a particular portion of the program does, etc.

2.2/5 PRINTING TEXT

Reset your cassette recorder, and now load another program:

LOAD "AREA2"

When it is loaded, use the **LIST** command to have a look at it. The program does exactly the same as the program "AREA1" that we have studied so far, but this time it has been tidied up to make it much easier to understand both when it is run and also when it is listed.

Let's take each line in turn. Whenever we look at a program like this – line by line – we are effectively "looking into the future". We are trying to understand what the program will do when it is run. Always bear this in mind when we work our way through the examples.

```
10 REM A PROGRAM TO CALCULATE AREAS
```

This line introduces the program. The ZX81 will ignore it because it begins with **REM**. It tells us what the program does. Notice that the line is too long to fit across the screen and so the ZX81 has put the word "AREAS" on a new line.

```
20 REM REQUIRES LENGTH AND BREADTH
```

Another REMark which indicates what the program needs to get it to run properly. This line is also too long. I (the author) have had to add extra spaces between the words so that the word "BREADTH" was not split halfway over two lines on the screen.

```
30 PRINT "ENTER LENGTH"
```

This is something new. The ZX81 treats the letters in quotes as *text*, and will print it on the screen for us. It is not restricted to just printing numbers. If we want any letters/characters printed, we just put quotes (") around them.

```
40 INPUT LENGTH
```

As we had before. The ZX81 will stop and wait for a number to be typed in from the keyboard.

```
50 PRINT "ENTER BREADTH"
```

This is some more text that tells us that we should enter a number representing the breadth of the rectangle.

```
60 INPUT BREADTH
```

The same as the original program.

```
70 PRINT "THE AREA IS"
```

This text introduces the result.

```
80 PRINT LENGTH * BREADTH
```

Now we print the result. What happens when "AREA2" is run? Do it and see for yourself.

Did you find it easier to use? It would certainly be easier for you if you had never seen the program before. Let us mention some points that are worth remembering:

1. When the program is run, each **PRINT** statement (lines 30, 50, 70 and 80) starts a new line on the screen, leaving the others intact.

2. When you enter the LENGTH variable you cannot see it any more, so if you forget what you type, you're in trouble.

2.2/6 · A PROGRAM FOR YOU TO WRITE

Question

Using the example in the previous question, write out a new version of the VAT program which contains some idea about what it does, and gives the operator (i.e. you) prompts for the required variables. The program was:

```
10 INPUT COST
20 LET VATRATE=0.15
30 PRINT COST * VATRATE
```

The answer printed is the amount of VAT to be added to the cost. First enter **NEW** to clear out the old program, then try entering your program to see if it really works.

Answer

Your answer may not match this one exactly, but should at least follow the general outline.

```
10 REM PROGRAM TO CALCULATE VAT
20 REM ASKS FOR COST TO BE ENTERED
30 PRINT "ENTER COST"
40 INPUT COST
50 PRINT "VAT AMOUNT IS"
60 PRINT COST * 0.15
```

You may notice that the line **LET** VATRATE=0.15 has disappeared. It has become mixed in with the new line number 60 as a direct number. Did your program work? If it did, then congratulations. Carry on with section 2.3.

We have covered quite a lot in this chapter so far, and maybe you are rushing ahead a bit too fast. Where did your program go wrong? Below is a list of things that may have caused you problems. Refer back to the sections mentioned.

(a) You didn't understand what **REM** did. See section 2.2/4.

(b) This business of "text" is confusing. Read section 2.2/5 again.

(c) Why use **INPUT**? Reading section 2.2/3 to 2.2/5 will give you some more reasons.

(d) You got a report code at the foot of the screen when the program was run. Look at the reference section titled "Report codes" at the back of this course to see what it means. Try correcting the error by typing the corrected line number in again. It will replace the incorrect line.

(e) You got a syntax error when you were entering a line, and you couldn't get past it. Look back at the answer to see how each line was formed. Check it against yours, and you'll probably see why the ZX81 won't accept your line. Don't forget that the **S** symbol indicates where the ZX81 thinks the error is.

2.3 PROGRAM EDITING

2.3/1 FULL CURSOR CONTROL

Now we want to think about something else. Let's load in that program "AREA2" again

Reset the tape and **LOAD** "AREA2"

What would we do if one of the lines in the program was wrong? Suppose that line number 30 should say

```
30 PRINT "TYPE LENGTH"
```

instead of ENTER LENGTH. This may seem a trivial change, but it highlights a big problem, especially when programs have very long lines in them. We could obviously type the whole line in again, but this seems a bit of a shame – nearly all the line is correct except for the few letters we want to change.

Look at the "1" key (top left). This key has written on it:

EDIT (SHIFT/1)

The edit key allows us to alter any part of a line that we wish, and put it back again inside the ZX81. The rest of the line remains unaffected. Before we can use it, we must obviously let the ZX81 know which line we want to alter (or edit). By now you've probably noticed the symbol **>** alongside the line number of one of the lines in a program. This is called the *program cursor*. It indicates the line number that the ZX81 would alter if the EDIT key were to be pressed now.

If we want to move the program cursor to a different line, we can do it in two or three ways. There are four keys on the top row of the keyboard marked with arrows (SHIFT/5, SHIFT/6, SHIFT/7 and SHIFT/8) – you've already seen how to use two of these. The two with arrows pointing up and down, move the program cursor to the line above or below respectively. At the moment, since we have just loaded "AREA2" again, the program cursor will be resting on line number 10. Try pressing the "down" arrow key (SHIFT/6) and see what happens. Try again. You will notice that each time you press the key, the > marker moves down to the next line. Now try the "up" arrow key. Getting the idea? The program cursor can also be moved *directly* to a line number by typing:

LIST nnnn **nl**

. . . where nnnn is any line number that yuo wish. We've used this command before. The difference here is that we have also included a line number. So if we just say **LIST,** this means "show the program starting with the first line number".

Move the program cursor to point to line number 30. We're about to change it. Press EDIT (SHIFT/1). What has happened? The line that the program cursor was pointing to has been put at the bottom of the display (screen), ready for us to alter. Now you can use the left and right arrow keys (SHIFT/5 and SHIFT/8) to move the cursor along the line to change characters as you've done before. We'll now try to change the word "ENTER" into the word "TYPE".

Use the cursor-right key (SHIFT/8) to move the cursor until it has just gone past the letter "R" in the word "ENTER". Do this while you are following this text. Now press the RUBOUT key five times – this removes the word "ENTER". Type in the new word "TYPE", and there we are! Ready to go back. Just press NEWLINE and back it goes.

You should now be able to change any line in a program that you wish. The steps are:

1. Move the program cursor (>) to the line that needs to be changed.

2. Press EDIT, which puts the line at the foot of the screen.

3. Alter the parts of the line that are incorrect by using the cursor-right and cursor-left keys and RUBOUT. Insert the new pieces. Remember that RUBOUT erases the letter to the *left* of the cursor, and new letters are inserted at the current cursor position.

4. Press NEWLINE to put the altered line back again.

Question

Name two ways of getting the program cursor to move to line number 50.

Answer

1. Using the cursor-up or cursor-down keys (SHIFT/7 and SHIFT/6 respectively)

2. **LIST** 50 **nl**

Read section 2.3/1 again if you were wrong.
The second answer (**LIST** 50) is quicker when the line you want to change is a long way away.

2.3/2 EDITING LARGER PROGRAMS

Big programs bring us onto something else – What happens when a program has more lines than can be held on the screen? We're going to see something quite clever now. One of the example programs on the tape is just big enough to fit onto the screen. It doesn't do anything much, but will serve to show you how the ZX81 deals with larger programs.

Set up the cassette, and type

LOAD "BIGONE" **nl**

When it's done, **LIST** the program. There are more lines to this program than are shown on the screen, but we cannot see them as yet. Again, there is more than one way of seeing them.

1. We can use the "down arrow" key to move the program cursor to the last line. As it is pressed again, the ZX81 moves the whole screen up one line, losing the top line, and gaining the new line on the bottom of the screen.

2. We can **LIST** the last line on the screen. The **LIST** command always puts the listed line at the top of the screen, so all the following lines will be shown as well. The lines above are still inside the ZX81, but they are not shown on the screen, that's all.

Try playing with this "BIGONE" program to get used to the feel of moving around with the cursor keys (the "arrow" keys are called the *cursor* keys).

We can also remove lines from a program, and lines 150 and 160 in the "BIGONE" program can be removed. This is done by merely typing the line number of the unwanted line, followed by NEWLINE. Try deleting lines 150 and 160.

Now let's add a new line and see exactly what the ZX81 does with it. Type:

185 **PRINT** "THIS LINE IS NEW" **nl**

You'll see that is has been put in between lines 180 and 190, in the proper place for its line number! So not only does the ZX81 obey each line in number sequence, but it also keeps the lines stored inside in line number sequence. This makes it much easier to look at a program and follow it through.

Question

What would you type now to remove the line numbered 185 that we have just added?

Answer

185 **nl**

Read section 2.3/2 again if you were troubled by that question.

2.4 PRINT FORMATTING

2.4/1 PRINTING MULTIPLE ITEMS

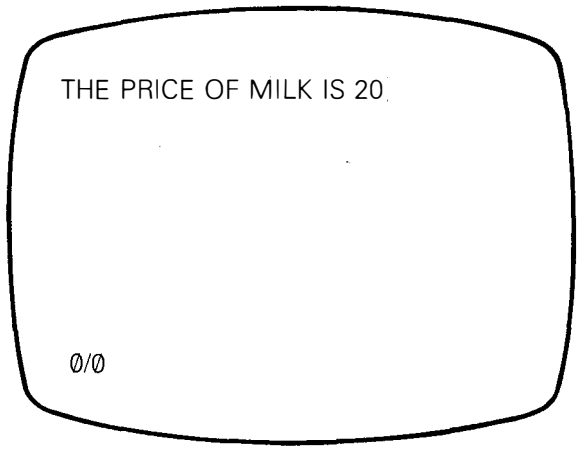
We now come to the final part of this chapter. The **PRINT** command has been mentioned quite a lot, but since it is probably the most powerful of the ZX81 commands, this is not surprising. This time we'll see how the **PRINT** command can be used to print more than one item on the same line of the screen.

If an expression in a **PRINT** command is followed by a semi-colon ";", then the next printed expression will be placed immediately after it with no spaces in between.

Look at these examples:

```
10 LET MILK PRICE=20
20 PRINT "THE PRICE OF MILK IS ";
30 PRINT MILK PRICE
```

Since line number 20 has a semi-colon following it, the *next* printed expression (in this case the number held by MILK PRICE) will be placed immediately after the first. Type them in and run them – don't forget **NEW** if there's anything inside the ZX81 at the moment! See what happens. The screen show this:



If you didn't get a space between "IS" and "20", then it is because you didn't put a space after the word "IS" in line 20. Remember that the next printed field follows the first one *with no spaces between.* If you want them, you must put them in yourself. Edit line 10 and try it again.

We can also put more than one expression on the same line of a **PRINT** statement, like so:

```
10 PRINT "THE PRICE OF MILK IS "; MILK PRICE
```

or

```
80 PRINT "THE AREA IS "; LENGTH*BREADTH
```

The second example is contained in another program on cassette called "AREA3", and we'll be looking at that in a few minutes. The **PRINT** statement can have as many expressions on it as you want – except that if you put too much into it, the ZX81 will split it onto two lines on the screen, just like the **REM** statements did in the program "AREA2". You need to keep track of exactly what you are printing.

2.4/2 PRINT ZONES

Another useful item is to separate expressions with *commas* in a **PRINT** command. This means that the next printed expression will start at the next "print zone" on the screen. What does that mean?

A normal typewriter has a "tab" key, which lets you move the carriage to a predetermined place on the paper, so that letters can appear in regular columns down the page. The ZX81 can do the same – whenever it sees a comma in a **PRINT** command, it moves to the next "print zone". These "print zones" are always set at position 0 and position 16 on each line of the display screen.

In case you are not aware, the display screen is built up as follows:

24 lines down
32 positions across

although the last two lines on the display screen cannot be used by a program – the ZX81 uses these to tell you if an error has occurred, or to ask for some data when it obeys an **INPUT** statement.

Effectively, then, the print zones split the screen in two, since position 16 is exactly halfway across. Here are some more examples of **PRINT** statements using commas and semi-colons. You should try entering them all and running the resulting program to see what each one does.

```
10 LET NUM=7
20 PRINT "THERE ARE ";NUM;" DAYS IN A WEEK"
30 PRINT "THE SEVEN TIMES TABLE IS"
40 PRINT 1*NUM,2*NUM
50 PRINT 3*NUM,4*NUM,5*NUM,6*NUM
```

These examples show you that you can mix text, numeric expressions, semi-colons and commas in any way that you wish. You could even have a line like:

```
60 PRINT ,"HELLO"
```

which would print the word "HELLO" starting at position 16 on the line.

Question

Write a small program to print the first six letters of the alphabet down the middle of the screen. Type **NEW** first to remove all the bits and pieces in the ZX81 that we have been playing with, then enter your program to see if it works.

Answer

As with all programs, there are several ways to do the same thing, but broadly, your program should look like this:

```
10 PRINT ,"A"
20 PRINT ,"B"
30 PRINT ,"C"
40 PRINT ,"D"
50 PRINT ,"E"
60 PRINT ,"F"
```

or

```
10 PRINT ,"A",,"B",,"C",,"D",,"E",,"F"
```

Did your solution work? Did it match one of these? If so, then carry on with the next section. If you're interested in the second solution, then the reason it works is as follows:

– the first comma moves the ZX81 to the next "tab" stop, which is halfway across the screen, and then the ZX81 prints the letter A.

– The next comma takes the ZX81 to the next "tab" stop, which is off the end of the line, so it goes onto the beginning of the next line on the screen. A further comma takes it to halfway across this line and prints "B".

– this is repeated for the next 4 letters.

Read sections 2.4/1 and 2.4/2 again to understand it more fully.

2.4/3 UNRAVELLING A PROGRAM

Now we can load that program "AREA3".

There is no need to type **NEW** before loading a program from tape, since the loading process automatically wipes out whatever is there.

```
LOAD "AREA3"
```

LIST the program to see what it contains. It is almost the same as "AREA2", but has a few subtle differences. These are:

– line 30 has a semi-colon following it, which means that whatever is printed next will follow this field immediately.

– line 45 is new. It prints the value of LENGTH that has just been input from the keyboard, and therefore lets us see the value of LENGTH even after we have entered it. This problem was highlighted in the points raised previously.

– lines 50 and 65 do the same as lines 30 and 45, but with the BREADTH variable instead.

– line 70 has been altered to just the statement **PRINT** with nothing following it. This has the effect of printing a blank line. A useful trick which is worth remembering.

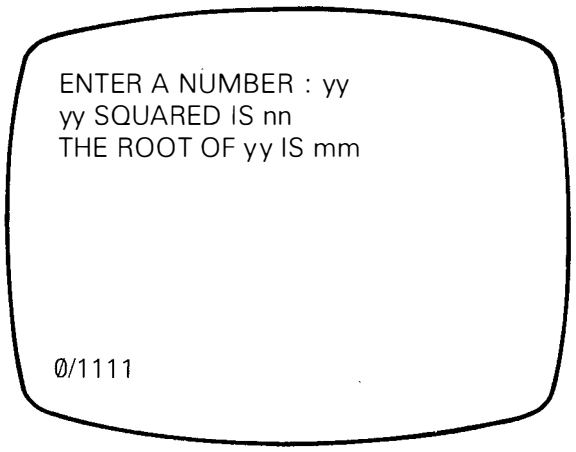
– line 80 now prints the answer on the same line as the text which introduces the answer.

Run this program and see how it differs from "AREA1" and "AREA2".
Here's a final problem for you:

Question

Write a program to do the following:

Ask for a number from the keyboard and print out the square of the number and the square root of the number with explanatory text. The screen results should look like:



(1111 is the line number of the last obeyed command)

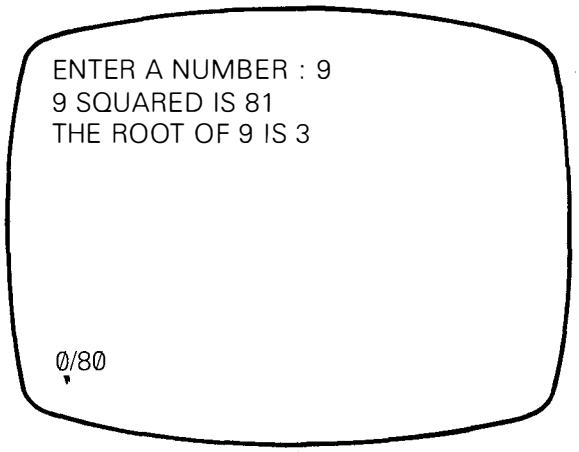
Note that the colon (in ''ENTER A NUMBER :'') is part of the text, not a special symbol of any sort.

Answer

Your program should look something like:

```
10 REM PROGRAM TO CALCULATE SQUARES AND ROOTS
20 PRINT "ENTER A NUMBER :";
30 INPUT NUMBER
40 PRINT NUMBER
50 PRINT
60 PRINT NUMBER;" SQUARED IS ";NUMBER*NUMBER
70 PRINT
80 PRINT "THE ROOT OF ";NUMBER;" IS ";SQR(NUMBER)
```

An example run would then give:



How did you get on? Did your program work? You have done extremely well if it did. Continue with the next page.

You may have got stuck trying to remember how to get a square root. This was covered in Chapter 1, section 1.4 "Mathematical Functions".

When you have checked over your program and found out why it didn't work properly, use the editing keys to alter it and try it again. It is quite important to get it working at this stage, as further chapters depend on your understanding each stage of the course. If you are finding it hard, then it is the right time to take a breather and look over some of the tricky sections again.

Summary

Let's take a last look at some of the topics we've covered in this chapter.

In the next chapter we'll be meeting some commands which allow the ZX81 to perform repetitive tasks, taking some more of the drudgery away from you.

– you've seen how line numbers are used for several reasons; they tell the ZX81 to store them as part of a program, and also indicate the sequence of execution when run.

– how the **LIST** command can be used both to assist editing and to view portions of a program on the screen.

– how the **INPUT** command can make a program re-usable.

– how to use the cursor keys and the EDIT key to alter program statements.

– how to print text strings, and how commas and semi-colons can be used to format printed expressions.

– how you can annotate your programs with **REM** statements to make it clearer for both yourself and others to understand.

Exercises

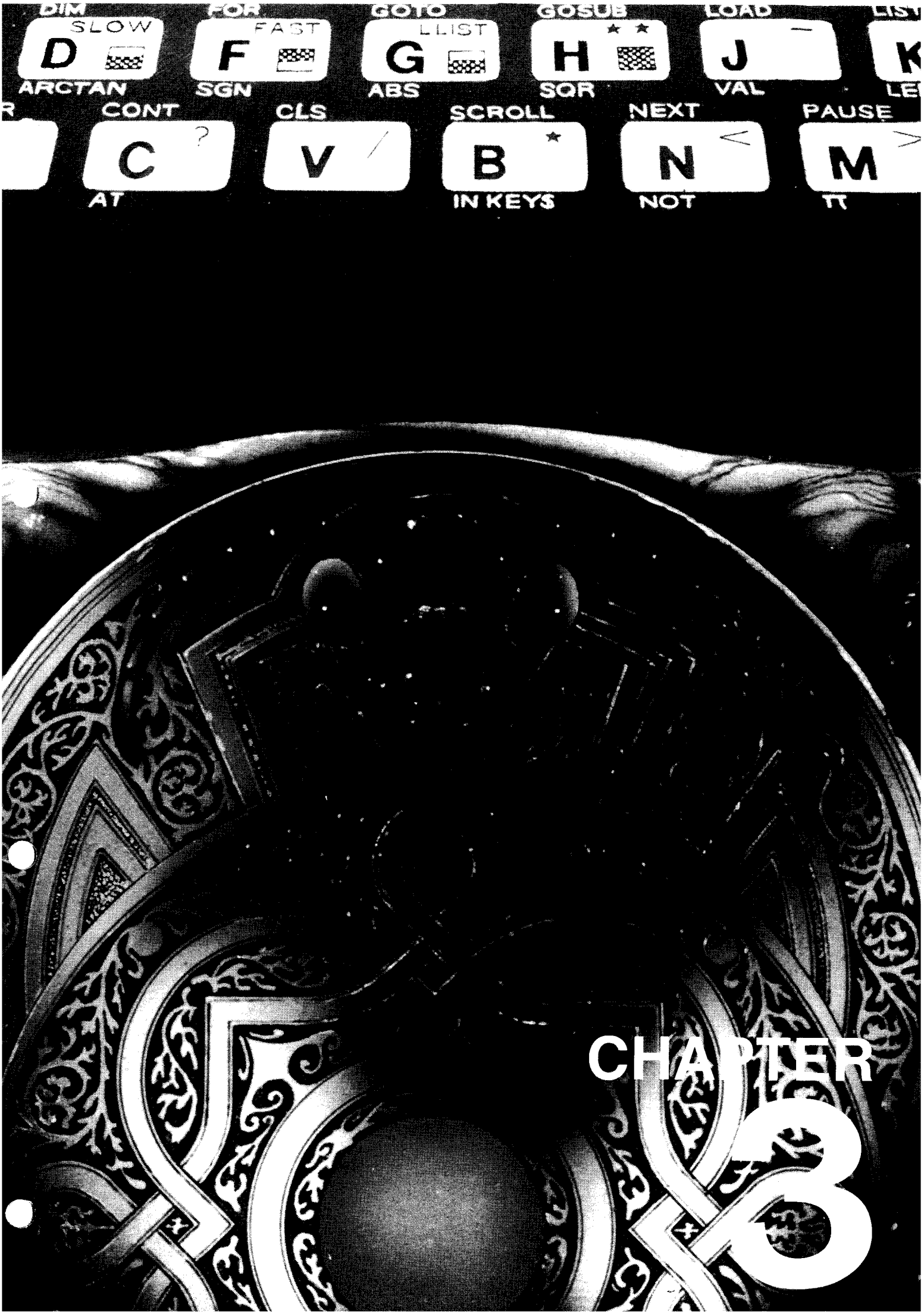
1. Write a program to convert from one currency to another (£1 = $2, £1 = 11fr, £1 = 2100 lira).

2. Calculate the roots of a quadratic equation using the formula:

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3. Calculate the velocity of a falling object after t seconds, where velocity = 32t (32 is the force of gravity, i.e. 32ft per second^2).

4. Using your answer to (3), how fast do you think the apple was travelling when it fell on Newton's head? A tree is roughly 20ft high and you will need to know that distance = $16t^2$ (when measured in feet).


DIM
FOR
GOTO
GOSUB
LOAD
SLOW
FAST
LLIST
D
F
G
H
J
ARCTAN
SGN
ABS
SQR
VAL
CONT
CLS
SCROLL
NEXT
PAUSE
C
V
B
N
M
AT
IN KEYS
NOT
CHAPTER
3

Getting Around (Using the IF and GOTO commands)

(Using the **IF** *and* **GOTO** *commands)*

In this chapter we will be studying two main topics. Section 1 looks at the concept of iteration, showing also how a program can be stopped.

Section 2 deals with conditional expressions, which, when used together with the ideas introduced in section 1, becomes an extemely powerful way of solving problems with a computer.

There is a third section, covering more functions in detail, since these new functions can be invaluable aids to conditional expressions.

3.1 ITERATION (1)

3.1/1 WHAT IS ITERATION?

Iteration is really the crux of computing, since it is where the computer scores over the human brain, pencil and paper.

Your ZX81 can perform literally hundreds and thousands of different instructions each second – far faster than you can. Yet we have only seen it handle one thing at a time so far. Here's where iteration comes in. The ZX81 can handle endless repetitive calculations, and do each one in less than the time it takes you to lick the end of your pencil.

Let's go back to our area programs. They were quite acceptable – certainly "AREA3" was – but they only calculated one area at a time. To calculate another area you type **RUN** again.

Another point that may have crossed your mind – what use are line numbers? They must do something else other than just tell the ZX81 which order to obey the statements in, surely? If you *did* think that, then you were correct. We can tell the ZX81 to perform the statements in a different sequence. The command to use is

GOTO (on the G key)

followed by a line number. The line number tells the ZX81 which line to obey next. How can we use this new command?

Here's a simple example of the "AREA1" program slightly altered:

```
10 INPUT LENGTH
20 INPUT BREADTH
30 PRINT LENGTH*BREADTH
40 GOTO 10
```

Notice that I have stopped telling you to press NEWLINE after each program statement – from now on I'll assume you're going to do this automatically. All we've done is to add a new line – line number 40. This time, when we run the program, the ZX81 will at first do exactly what it has always done – ask for the two variables and print the result. But when it reaches this new line 40, it will immediately carry on back at line number 10 again. It has been told to *go back and continue the program at line number 10.*

A **GOTO** command can specify any line number that it wishes – either forwards or backwards (as we have just used). So the "AREA1" program could calculate as many areas as we want without having to type **RUN** at all!

Question

Don't worry if you can't answer this question, as it is intended only to make you think a bit, not to test you.

What problems can you see with the **GOTO** command? There are several, and the answer will cover each of them, but try to think of a few for yourself.

Answer

We will deal with each of the solutions in later sections, but here are some of the problems you may encounter with **GOTO**.

1. How do we ever get the "AREA1" program to stop? Whenever a result is printed, it always starts back at line number 10 again.

2. What happens after we have printed 22 results? The previous chapter told you that were only 22 lines down the screen that a program can use.

3. What would happen in the "AREA1" program if we had put **GOTO** 574 instead of **GOTO** 10? Line number 574 is not there.

You may have thought of other points, but the ones above will no doubt cover most of them.

3.1/2 STOPPING A PROGRAM

There are many ways of getting a program to stop, although most of the "proper" ways will have to wait until the next chapter. The method to use depends on what exactly the ZX81 is doing.

One method is the SPACE key which has BREAK written over it. Whenever a program is running, pressing the BREAK key will make the ZX81 stop with an error letter D (break key pressed). Break does not work when the ZX81 is waiting for some input from the keyboard, so use the first method in those circumstances.

If the ZX81 is waiting for you to enter a number (i.e. it is obeying an **INPUT** command) then by entering the command.

STOP (SHIFT/A – have care! If you do not press SHIFT properly you'll get A instead!)

the ZX81 will stop running the program giving report code D (see the section at the end of this course for all report codes). **STOP** can also be used as a command in a program to stop it from carrying on. Under these conditions, the ZX81 gives report code 9. We'll see more about this later on.

What happens when we have printed 22 results (or lines on the screen)? The ZX81 will stop running the program and give a report code 5, showing you the results so far. If you then enter the command

CONT (on the C key)

(standing for "CONTINUE"), the ZX81 will clear the old results off the screen and carry on from where it left off. If the screen fills up again, the same events occur. This is a useful way of keeping an eye on what's going on while a program is running.

CONT will work after any report code. It is a useful way to restart a program that has failed, after you have looked to see why it went wrong and corrected the program line(s) that caused the problem.

The last point raised in this problem was that of an "undefined line number", where the ZX81 tried to carry on at a line number that didn't exist. This is quite simple. The ZX81 always continues with the *next greater line number* if the one given does not exist. If this means dropping out of the bottom of the program, then report code 0 is given, meaning that the program has finished. So if the "AREA1" program above had:

40 **GOTO** 5

then it would work exactly the same, since line 5 does not exist, and the next higher line number is 10.

Question

Set up the cassette recorder and load a program "LOOPER":

LOAD "LOOPER"

Start the program by **RUN**, then try to stop it. It is considered cheating to switch the ZX81 off at the mains!

Answer

If you haven't done it, then press the BREAK key.

You should read section 3.1/2 again if that foxed you.

3.1/3 CONDITIONAL GOTO

Now you've been introduced to the **GOTO** command, it's time to look a bit more deeply. The line number that follows the word **GOTO** is actually a numeric expression, so we could write:

GOTO ALPHA

or even

GOTO BETA * 7

Note that ALPHA and BETA must have been previously assigned with **LET** statements. If you have a think about that, you'll quickly realise that there's a great deal of hidden potential in the **GOTO** command.

Load another program from tape called "GOTEST":

LOAD "GOTEST"

Run it to see what happens. You are asked for a number between 1 and 5. Enter a number, press NEWLINE. Run it again, but try a different number.

Each time, the ZX81 gives you a different result. How? Somewhere it must be looking at what you are typing and doing something different depending on this value. Let's have a look. Type **LIST** when you've had enough.

We can quickly skip over line numbers 10 to 50 since these should by now be fairly familiar to you. But line number 60 is the key to the whole program. It takes the number that you have entered, multiplies it by 10, adds 100, then carries on at this newly calculated line number! The **GOTO** forces the ZX81 to *transfer control* to a new line number! Instead of continuing with the next line in sequence, it carries on with – which line? We can't tell until the program is run, since the variable NUMBER could contain almost anything. If you entered the number 2, then it would carry on at line 100+2*10, or 120.

Although you have been asked to enter a number between 1 and 5, there's nothing stopping you from entering *any* number. Run the program again, this time entering negative values (like –13), or 1.5, or 3.6, or 925588, just to see what the ZX81 does when it tries to **GOTO** an invalid line number.

This type of **GOTO** is called *conditional*, because it is conditional upon the value of the following expression.

Question

How would you restart a program that has stopped showing report code 5 at the foot of the screen?

In fact, what does report 5 mean?

You may refer to any of the sections at the end of this course to assist you, but do not refer to the previous few pages.

Answer

Report 5 means that the screen has become full, and no more lines can be printed. Typing **CONT** will restart the program, and allow another screen-full of items to be printed.

3.2 CONDITIONAL EXPRESSIONS

3.2/1 WHAT DOES IT MEAN?

We're about to meet another topic which goes hand-in-hand with the **GOTO** command, and when this has been covered, you'll have a few more meaty problems to get stuck into.

The last section introduced the idea of *conditional* branching, and now this idea is about to be expanded.

You have probably thought that some of the methods given earlier for stopping a program seemed a trifle crude; if you did, then you were right – they were crude. Using the "break" key is a bit drastic, and is normally only used when you suspect that your new program has "got itself lost", or, using the terminology of the computer world, "got stuck in a loop". The LOOPER program was a good example of this.

So how can we avoid this?

IF we could test the value of a variable, **THEN** we could stop the program more tidily.

In fact, **IF** we could test the value of a variable, **THEN** we could do lots more with a program, not just stop it.

The answer has really just been given to you. The command is

IF (on the U key).

We can use the **IF** command to test the value of a variable (or expression) to see if it contains a certain value. For example:

```
IF MILK PRICE=20 THEN GOTO 200
```

You'll find **THEN** by pressing SHIFT/3. What will this do when the ZX81 tries to run it? Well, first it checks to see if MILK PRICE is equal to the value 20. If it is, then whatever statement follows the word **THEN** will be obeyed – in this case **GOTO** 200.

If MILK PRICE does *not* hold the number 20, then the rest of the command is ignored, and the ZX81 continues with the next line number as normal.

Let's try this on the ZX81. Type the following small program in, but don't run it just yet. The word **THEN** can be found by pressing SHIFT/3. It will be good practice for you to type this all in – it's not very big. Notice that the line numbers go up in hundreds. This is deliberate, as you will find out later on.

```
100 PRINT "ENTER A NUMBER (1–5)"
200 INPUT NUMBER
300 IF NUMBER=1 THEN PRINT "ONE"
400 IF NUMBER=2 THEN PRINT "TWO"
500 IF NUMBER=3 THEN PRINT "THREE"
600 IF NUMBER=4 THEN PRINT "FOUR"
700 IF NUMBER=5 THEN PRINT "FIVE"
```

Now let's look to see what's going on. First of all, the program (when run) will ask you to enter a number. Although the message says "1–5", there is nothing to prevent you entering *any* number. Anyway, you will then enter a number and press newline. The next five lines in the program in turn look to see what value has been entered. Line number 300 checks to see whether the number was 1, and if so, prints a word corresponding to this. Line 400 sees if the number was a 2, and if so, then prints the text "TWO". Note that if the number was 1, then line 400 will not do anything, since the number cannot be 1 and 2 at the same time! The same logic applies to lines 500, 600 and 700.

Now run the program and check that it works in the way we've just discussed.

Question

What do you expect the program to do if a number (say) 41 is entered instead of 1 to 5?

Answer

Nothing, since lines 300 to 700 only check for numbers in the range 1 to 5. The program would merely give a code 0/70 at the foot of the screen.

3.2/2 EQUALITY AND INEQUALITY

Wouldn't it be nice if we could get this program to throw out numbers that it wasn't interested in?

We can. The conditions in an **IF** command can be grouped together to form a really complicated statement if you want, althugh it is normally better to restrict them to manageable sizes.

Here we are going to find another list of items to be remembered.

You have already seen that two values can be tested for equality (using the = sign in an **IF** statement); here now is a full list. They are called *relational operators,* meaning that they allow one value to be tested relative to another:

operator	*where on keyboard*	*meaning*
=	SHIFT/L	equal to
>	SHIFT/M	greater than
>=	SHIFT/Y	greater than or equal to
<	SHIFT/N	less than
<=	SHIFT/R	less than or equal to
<>	SHIFT/T	not equal to

So we could write:

IF MILK PRICE>20 **THEN GOTO** 200

if MILK PRICE holds a value greater than 20, then the ZX81 continues with line number 200, otherwise it just carries on with the next line number.

IF NUMBER<=5 **THEN GOTO** 200

if the variable NUMBER contains a value less than or equal to 5, then the ZX81 continues at line 20.

The second example here gives a clue as to how we can alter our program to reject invalid numbers that are entered.

Question

Can you add two new lines to the program given above which will force the program to ask for another number if the one given is less than 1 or greater than 5?

Use line numbers in between the existing ones.

Answer

Here is one solution. Yours may vary slightly, but should follow these guidelines:

```
210 IF NUMBER<1 THEN GOTO 200
220 IF NUMBER>5 THEN GOTO 200
```

Your answer could have returned to line 100, or you could have used <= or >= symbols with different testing values. The lines could also have been placed *after* line 700. There are many solutions that are all equally valid, and you should try entering your lines into the program to check that they have the desired effect.

Did it work? If not (that word *if* gets everywhere!) then read sections 3.2/1 onwards again. You really must grasp this concept.

3.2/3 LOGICAL OPERATORS

Now we come to a second list of things to remember. This time, they allow for more than one condition to be tested at the same time, and are called *logical operators*. The answer given above can be reduced to a single **IF** command using one of these.

operator	*where on keyboard*	*function*
AND	SHIFT/2	two conditions must be true for the whole statement to be true.
OR	SHIFT/W	one of the two conditions must be true for the whole condition to be true.

This looks complicated when written like this, but it's really quite straightforward. We'll be looking at both of these in turn, but first let's see exactly what an **IF** statement is doing for us.

3.2/4 CONDITIONAL EXPRESSION VALUES

Taking the answer to the previous question in section 3.2/2, we can now re-write it as:

```
210 IF NUMBER<1 OR NUMBER>5 THEN GOTO 200
```

The **OR** in between the two expressions means that if either one is true (i.e. NUMBER is less than 1, or NUMBER is greater than 5), then the statement as a whole is true, and the statement following **THEN** is obeyed.

Hang on a minute – can we really call "NUMBER<1" an *expression*? Surely an expression represents a numerical value? (see chapter 1 for the definition of an expression)

The answer to both these questions is *yes*.

Whenever you ask the ZX81 if something is true or not (like **IF** NUMBER<1 . . .), then this is actually given a value. If the comparison results in a "true" answer, the value is 1, and if the comparison is "false", then the value is 0 (zero).

So all the **IF** statement does is to check whether the results of all these tests are "true" – if they are, then the rest of the line is obeyed (whatever follows **THEN**).

Try this simple test. Type the following:

```
NEW        to remove anything inside
10 IF 1 THEN PRINT "THIS HAS WORKED"
```

Now run it. Because the statement is "true" (has a value 1), the words "THIS HAS WORKED" have been printed out for you.

Edit the line to say:

```
10 IF 0 THEN PRINT "THIS HAS WORKED"
```

What happens this time when it's run? Absolutely nothing, because the statement has a "false" (or zero) value.

So when we write:

```
10 IF MILK PRICE=20 THEN PRINT "THIS HAS WORKED"
```

the ZX81 looks to see if the *expression* "MILK PRICE=20" is true or false. If it is true, we will get "THIS HAS WORKED" printed on the screen. If it is false nothing will happen, and the ZX81 will just carry on with the next line number.

We'll look at one more example of this to make the point really clear.

```
210 IF NUMBER<1 OR NUMBER>5 THEN GOTO 200
```

This is broken down by the ZX81 into smaller chunks. First it asks "is NUMBER less than 1?" and remembers the answer to this question. Next it asks "is NUMBER greater than 5?" and also remembers this answer. It is fairly obvious to you and me that if NUMBER is less than 1, than it cannot also be greater than 5, but the ZX81 is not quite that bright – it needs to work it out by checking both conditions. The ZX81 now has three possible results from all this:

NUMBER<1	NUMBER>5	example
true	false	–3
false	true	41
false	false	2

But the original statement said "OR" – so the ZX81 now asks "is either one **OR** the other of these two results true?" If this new expression is true, then the overall question is true, and the ZX81 will **GOTO** 200.

The table earlier gave another type of operator – **AND**. This is fairly similar in the way it works to **OR**, and you can most likely see how to use it straight away. Here is an example of it in use with full explanations:

```
20 IF MILK PRICE=20 AND BREAD PRICE=42 THEN GOTO 50
```

Both conditions must be true before the ZX81 will **GOTO** line 50 – i.e. MILK PRICE must equal 20 *and* BREAD PRICE must equal 42. If either has a different value, then the statement is ignored.

Here's an example of a program using **IF** statements in plenty – type it in and run it:

```
10 PRINT "ENTER YOUR AGE ";
20 INPUT AGE
30 PRINT AGE
40 IF AGE<13 THEN PRINT AGE-1;" YEARS OLDER THAN ME"
50 IF AGE>=13 AND AGE<=19 THEN PRINT "A TEENAGER, HUH?"
60 IF AGE=21 THEN PRINT "I KNOW . . . I KNOW . . ."
70 IF AGE>19 AND AGE<>21 AND AGE<40
   THEN PRINT "YOU ARE AT YOUR BEST"
80 IF AGE>=40 THEN PRINT "HONESTY IS THE BEST POLICY"
```

You may recognise the fact that a similar routine was used in the "STARTERS" program in the Introduction. Run the program giving several different ages to see how the ZX81 reacts each time.

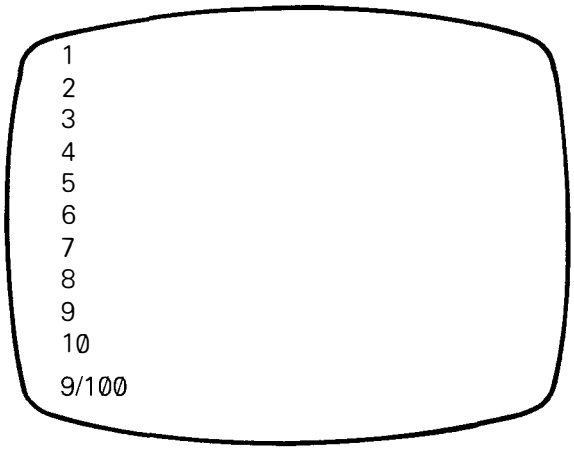
3.2/5 A SERIES OF PROBLEMS

Question

Time to write a small program.

This program must print the numbers 1 to 10 on consecutive lines down the screen. Remember that you use the **STOP** command to get your program to finish tidily.

Here is an example run:


1
2
3
4
5
6
7
8
9
10
9/100

Answer

This solution is one of many:

```
10 LET NUMBER=1
20 PRINT NUMBER
30 LET NUMBER=NUMBER+1
40 IF NUMBER>10 THEN STOP
50 GOTO 20
```

On this occasion, two more solutions are provided to give you some ideas as to the different ways in which the same program can be written. Normally, only one solution will be shown.

```
10 LET X=0
20 LET X=X+1
30 PRINT X
40 IF X=10 THEN STOP
50 GOTO 20
```

Or,

```
10 PRINT 1
20 PRINT 2
30 PRINT 3
40 PRINT 4
50 .....
.........
100 PRINT 10
110 STOP
```

You may find this solution frivolous, but it *is* a valid answer to the question as posed. This is meant to show you that there is never a "correct" way to write a program. Obviously, this last solution is longer than the other two, and would take you longer to enter, so, if for no other reason, the first two solutions are preferable.

If you found that question straightforward (it may not have been simple, but that's not the point) then carry on with the next section.

One common problem when anyone starts programming, is *how to get going*. Perhaps you have found this out by looking at the answers given – don't feel ashamed to cheat from time to time, but always try to answer the questions on your own first, and then only look at the answers if you are truly stuck.

If you found that problem particularly stiff then perhaps you should read this chapter over again, paying good attention to all the small examples. Always try to run them on your ZX81 – it helps put it all in perspective.

Questions

Here is a small set of questions for you to answer. Try to answer them all before you turn the page over to check your answers.

(a) Write down what you think the following small programs will do if run:

(i)
```
10 IF 23>7 THEN PRINT "SEASIDE"
```

(ii)
```
10 LET F=12
20 IF F<>5 THEN PRINT "QUITE RIGHT"
```
(use the table given in Section 3.2/2 to see what the symbol <> means)

(iii)
```
10 LET VIC=5
20 IF 10>VIC+6 THEN PRINT "OH YES IT IS"
```

(iv)
```
10 LET X=100
20 LET Y=40
30 IF (X/2)=Y THEN PRINT "WHY NOT?"
```

(b) In the next group of small programs, two variables are used throughout. These are:

ALPHA which holds the value 10
BETA which holds the value 20

so if we said "What happens in the following program"

```
30 IF ALPHA<BETA THEN PRINT "HELLO"
```

your answer should be "the program would print 'HELLO'."
Got it?

If you want to enter these programs to check the answers, then enter:

```
10 LET ALPHA=10
20 LET BETA= 20
```

before you enter the next line given below.

(i) `30 IF ALPHA=10 OR BETA=93 THEN PRINT "YES"`
(ii) `30 IF ALPHA=10 AND BETA=93 THEN PRINT "YES"`
(iii) `30 IF BETA>=20 THEN PRINT "YOU GUESSED"`
(iv) `30 IF ALPHA*2=BETA AND BETA/2=ALPHA THEN PRINT "OK"`

Answers

In each case, the answer shows what would be printed if the program were run. Where nothing would be printed (because the condition was false), the solution is "nothing".

(a) (i) SEASIDE
(ii) QUITE RIGHT
(iii) nothing
(iv) nothing

(b) (i) YES
(ii) nothing
(iii) YOU GUESSED
(iv) OK

These problems have actually introduced some new features associated with **IF** statements, but you probably worked them out for yourself quite logically.

If you found them impossible, then read section 3.2 over again.

3.2/6 COMPLEX CONDITIONAL EXPRESSIONS

The last series of problems talked about some new features – what are they?

First of all is the fact that **IF** statements can contain numeric expressions, not just variables. So it is quite legal to say:

```
250 IF HOPPER*10+3=((STACKER-7)/(X+8)) THEN GOTO 300
```

since the relational operators (remember them?) test the value of the numeric expressions either side.

Secondly, we come back to a topic raised in Chapter 1 – priorities. The examples given above did not mention the order in which you should work out the various expressions – this was left to you. The ZX81, however, needs to have some sort of rule that tells it what to work out first. Chapter 1 gave a list showing the priority of the various arithmetic operators (e.g. addition, subtraction, division etc). Now we need to extend this list so that we can see exactly where **AND**, **OR**, =, <> and all the others come in the ZX81's list of priorities.

Operator	Priority	Description
()	12	bracketed expressions (innermost first)
any function	11	e.g. **SIN**, **SQR**, etc (excluding **NOT**)
**	10	"to the power of", or exponentiation
−n	9	unary minus (i.e. negative numbers)
*	8	multiplication
/	7	division
+	6	addition
−	6	subtraction
=	5	equals
<>	5	does not equal
>	5	greater than
>=	5	greater than or equal to
<	5	less than
<=	5	less than or equal to
NOT	4	inversion (more on this later)
AND	3	logical
OR	2	logical

So how does this affect *you*?

Let's take some examples, and see how the order of priority works in the ZX81.

```
10 IF ALPHA=10 OR BETA=20 AND ALPHA=97 THEN GOTO 50
```

Assuming that ALPHA holds 10 and BETA holds 20, what do you expect the outcome of that statement to be?

Remember that AND takes priority over OR, so the order of working out is:

(a) "equals" is the highest priority operator in the line, so first of all the ZX81 works out the three "equal" conditions to see if they are true or false. This results in something like:

```
10 IF ALPHA=10 OR BETA=20 AND ALPHA=97 THEN GOTO 50
      (true)      (true)      (false)
```

(b) AND has the next highest priority, so the conditions BETA=20 **AND** ALPHA=97 are considered next, and we get:

```
10 IF ALPHA=10 OR BETA=20 AND ALPHA=97 THEN GOTO 50
      (true)      (true)      (false)
      (true)      (      false      )
```

since AND means that *both* conditions must be true for the whole to be true.

(c) Last of all, the two remaining conditions are connected by OR, which means if one of the two is true, then the whole condition is true. So:

```
10 IF ALPHA=10 OR BETA=20 AND ALPHA=97 THEN GOTO 50
      (true)      (true)      (false)
      (true)      (      false      )
      (            true             )
```

Because the overall statement is true, the ZX81 will continue at line number 50.

It would have had a different outcome if the priority of AND and OR were reversed, and you may like to work that out for yourself using a similar method.

Of course, it is perfectly legal for you to write brackets around the items you want the ZX81 to work out first – just like you were shown in Chapter 1. In most cases this is not necessary, but if you want to write an expression like those above, it is always worthwhile putting brackets around the various items purely to make them clearer to yourself (let alone the ZX81!). At least the statement will work in the way you intend it to, and not in some peculiar fashion that you can't quite grasp.

One more point before we finish with **IF** – you have seen above that whatever follows **THEN** can be *any valid statement*. Since **IF** is itself a valid statement, one **IF** can follow another, like this:

```
IF MILK PRICE=20 THEN IF BREAD PRICE=42 THEN PRINT "IT IS"
```

In this particular case, you could have achieved the same thing by using **AND**:

```
IF MILK PRICE=20 AND BREAD PRICE=42 THEN PRINT "IT IS"
```

although there are one or two cases where the first method is the only possible way of writing the statement without using two conditions. You'll meet this in another chapter later on – for now, just remember that it is quite legal to use two or more **IF** statements together.

Well, time for a bit of revision.

Question

(a) Give two examples of *relational operators*.
(b) Give two examples of *logical operators*.
(c) What is the value of a "true" expression?

Answer

(a) You can have any two of = <> < <= > >=

(b) There are only two logical operators, and they are AND and OR.

(c) 1

Section 3.2/2 covered relational operators, and section 3.2/3 covered logical operators. You should read these sections again if you have confused the two types.

If your answer to part (c) was wrong, then read section 3.2/4 again.

3.3 MORE FUNCTIONS

3.3/1 DEFINITIONS

Now that you can handle **IF** and **GOTO**, it's time to introduce you to a few more functions.

The first is the **INT** function.

INT function. This function gives the *integer* value of the expression following (which may be enclosed in brackets), thus losing any decimal places which the value may have held. For example:

PRINT INT 12.789

would print 12. All the decimal places are lost. We can write a small routine to separate the "whole numbers" from the "fraction" of any number as follows:

```
100 LET WHOLE=INT NUMBER
110 LET FRACT=NUMBER-WHOLE
```

. . . assuming that NUMBER contains the desired value.

If NUMBER held 12.789, then after running that small routine, WHOLE would contain 12, and FRACT would contain 0.789.

Don't forget that if you put brackets around an expression, then the **INT** function operates *after* the expression has been fully evaluated, so that

PRINT INT (2.6 * 2)

. . . would print 5, whereas . . .

PRINT INT 2.6 * 2 (no brackets)

. . . would print 4.

The **INT** function always rounds down, so negative numbers take the next lower integer value – e.g. **INT** −2.3 gives −3.

SGN function. This function returns the *sign* of the following expression – if it is negative, then the value −1 is returned. If positive, then +1 is returned. If, however, the expression is zero, then **SGN** returns zero. Here are a couple of examples:

SGN −27	would give −1
SGN 45.887	would give 1
SGN (6*2−12)	would give 0.

The **SGN** function is quite often used in computer games which involve boards or square grids, so that the direction of one object relative to another can be determined.

Here's a full example for you to try so that you can get a better picture of what's going on.

```
10 LET V=-23
20 LET X=57
30 LET X=X * SGN V
40 PRINT X,V
```

ABS function. The **ABS** function always returns the positive value of the expression that follows. So if the expression was, for example, −37.5, then

PRINT ABS −37.5 would give 37.5 (Try it!)

Again, in board games, **ABS** is used to check that a distance entered in using an **INPUT** command is within the correct range (say, +10 and −10). This would be written in a program as:

```
10 INPUT MOVE
20 IF ABS MOVE>10 THEN GOTO 10
. . . .
. . . .
```

therefore any number outside the range −10 to +10 would not be accepted.

NOT function. **NOT** causes the result of the following expression to be inverted – i.e. true becomes false and false becomes true. So, for example,

IF NOT (ALPHA=12) **THEN PRINT** "IT IS 10"

if ALPHA holds 10, then this would print "IT IS 10", since ALPHA=12 is false and the **NOT** alters this to true. Don't forget that ALPHA=12 is actually a numeric expression with a value of 0 or 1 according to whether it is false or true.

NOT might cause you a few unexpected problems, since it has a lower priority than "equals", but a higher priority than **AND** and **OR**, so an expression such as

10 **IF NOT** ALPHA=97 **AND** BETA=33 **THEN GOTO** 200

does not quite do what you might expect.

For now, it is much better that you use <> to represent "does not equal" rather than worry about **NOT**. When you have gained more confidence in your abilities, you may like to refer back to this section to try and work out exactly why that last example would not "goto 200".

Don't be too concerned for now if you're having trouble remembering the functions we've covered.

Of the functions mentioned above, by far the most useful is the **INT** function. Since the ZX81 always works in "floating" decimal places (just like a calculator), there are occasions when the result of a division (or some other function) leaves some unwanted fractional parts. The **INT** function can then be used to get rid of these – try to ensure that your programs are "exact", and don't rely on "rounding" to sort things out for you.

3.3/2 THE FUNCTIONS IN USE

To round off this chapter, here are some small programs for you to write.

Question

Both parts of this question ask you to write fairly lengthy programs (roughly 15 lines each). Don't be afraid to spend quite some time working on them – it's all good practice at this stage. Refer back to previous examples and sections on functions to help you work out what you want to do. Type the programs in to check that they work.

(a) Write a program that asks for a number from the keyboard. It then prints out the sign of the number and the absolute value of the number on the same line. It should keep asking for more numbers until either ten numbers have been entered, or a value of zero is entered. An example run is:

```
ENTER A NUMBER : -46.5
SIGN IS -1 AND ABSOLUTE IS 46.5
ENTER A NUMBER : 2
SIGN IS 1 AND ABSOLUTE IS 2
ENTER A NUMBER : 0
THAT IS ALL FOR NOW

9/9999
```

(b) Write a program that takes in a number from the keyboard and prints the nearest whole number (i.e. round the number to the nearest integer). If the number has been rounded up, then print the text "ROUNDED UP" alongside, and print "ROUNDED DOWN" if the number was rounded down. Fractions of 0.5 and above are to be rounded up. Don't forget that when rounding negative numbers, that -45.5 would round UP to -45 and -45.6 would round DOWN to -46. Stop the program when either 10 numbers have been entered or a number of zero is entered (as in the previous example). A sample run:

```
ENTER A NUMBER : 5.433
5 ROUNDED DOWN
ENTER A NUMBER : 72.50001
73 ROUNDED UP
ENTER A NUMBER : -32.4
-32 ROUNDED UP
ENTER A NUMBER : 0.0000
THATS ENOUGH

9/9999
```

Answers

Once again, there are many, many ways of writing these programs. Here are two solutions:

(a)

```
10 LET COUNT=0
20 PRINT "ENTER A NUMBER : ";
30 INPUT NUMBER
40 PRINT NUMBER
50 IF COUNT=10 OR NUMBER=0 THEN GOTO 100
60 PRINT "SIGN IS ";SGN NUMBER;" AND
   ABSOLUTE IS ";ABS NUMBER
70 LET COUNT=COUNT+1
80 GOTO 20
100 PRINT "THAT IS ALL FOR NOW"
110 STOP
```

(b)

```
10 LET COUNT=0
20 PRINT "ENTER A NUMBER : ";
30 INPUT NUMBER
40 PRINT NUMBER
50 LET COUNT=COUNT+1
60 IF NUMBER=0 OR COUNT>10 THEN GOTO 200
70 LET WHOLE=INT (NUMBER+0.5)
                                    (why do you think this works?)
80 PRINT WHOLE;
90 IF SGN (NUMBER-WHOLE)=-1 THEN GOTO 120
100 PRINT " ROUNDED DOWN"
110 GOTO 20
120 PRINT " ROUNDED UP"
130 GOTO 20
200 PRINT "THATS ENOUGH"
210 STOP
```

Well, although those programs had some similarities, they were not all that easy! Probably the second program gave you more difficulties than the first, and the *real* test comes when you run the programs. Do they work? When a zero value is entered, do they stop? It obviously doesn't matter if your programs don't print a nice little message when they've finished; the important point is that they don't just keep on going forever.

Testing a program to make sure it works properly normally takes as long (if not longer) as it takes to write the program in the first place! Don't forget to EDIT any incorrect lines and try again. It's not quite good enough to just say "Oh, I can see what's wrong – there's no point in trying again just for *that*!". It may be that there is something else wrong with the program that was hidden by the first error.

If you had problems, let's try to refer you back to the appropriate section.

– you could not grasp the "functions". Read Chapter 1, section 1.4 "Mathematical Functions" again, and then come back to read section 3.3 again.

– the "IF" statement is causing you difficulties. Try reading section 3.2 "Conditional Expressions" again.

– this "GOTO" business is confusing. Read section 3.1 "Iteration (1)" once more. Now you've seen them in use, you may find the early sections a bit easier.

– the PRINT statements in the answers were confusing. Although the PRINT statement was largely covered in the earlier chapters, this is the first time that we've used them to any real degree. You may like to read Chapter 2, section 2.4 "Print Formatting" over again.

Summary

Chapter 4 looks at more ways of "getting around" in a program, and starts to look at ways of making a program easier to write. But for now, let's just recap all the things that have come up in Chapter 3.

– how the **GOTO** command can be used to allow repetitive calculations to be made, and how the flow of the program can be controlled using this command.

– that the **STOP** command can be used in an input data string to force a program to stop.

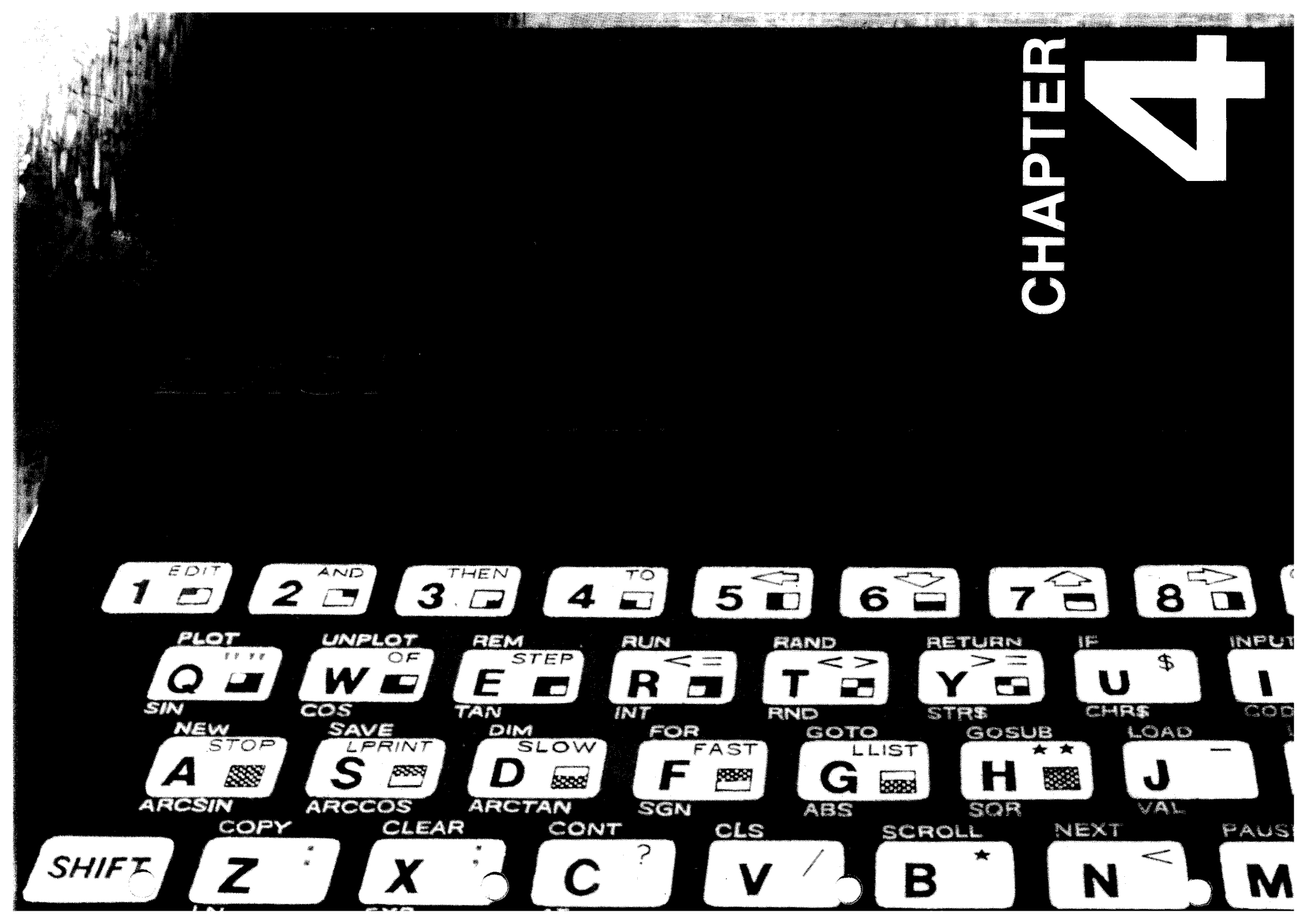
– how conditions can be evaluated using the **IF** commands to enable a program to cater for unexpected (or expected) events.

– that certain functions can assist the evaluation of expressions without necessarily having a "mathematical" purpose.

Exercises

1. Write a program to calculate the first 5 rows of Pascal's triangle.

2. The exercises in Chapter 2 asked you to write a currency conversion program (exercise 1). Now see if you can produce a program more worthy of constant use. The program should allow several currencies to be converted by entering an initial selection, then repeating conversions until a value of zero is entered.


CHAPTER 4
1 EDIT
2 AND
3 THEN
4 TO
5
6
7
8
PLOT
Q
SIN
UNPLOT
W OF
COS
REM
E STEP
TAN
RUN
R
INT
RAND
T
RND
RETURN
Y
STR$
IF
U $
CHR$
INPUT
I
NEW
A STOP
ARCSIN
SAVE
S LPRINT
ARCCOS
DIM
D SLOW
ARCTAN
FOR
F FAST
SGN
GOTO
G LLIST
ABS
GOSUB
H
SQR
LOAD
J
VAL
SHIFT
COPY
Z
CLEAR
X
CONT
C
CLS
V
SCROLL
B
NEXT
N
M

Tidying Up (Using Flowcharts, FOR and NEXT)

This chapter deals with two main topics – section 1, which deals with methods of making programs concise and efficient, and section 2 which shows a new form of iteration that doesn't involve using line numbers.

A final section is included in which we study iteration at work, seeing it operate in a program.

Throughout the chapter we will be concerned with several programs, although one particular program is used to tie together the various points raised in the sections.

4.1 PROGRAM DESIGN

4.1/1 SIMPLE STEPS

First of all we must consider *program design*. What's that?

Whenever you sit down to write a program, there's no point in starting until you have got an objective in mind. Programs (and ideas for programs) do not just flow out of your pen!

To begin with, we're going to look in detail at how to write a program to calculate prime numbers. This involves some commands that we haven't yet introduced, so there's quite a bit in store for you.

In case you're not aware, a prime number is one that cannot be divided by any number (other than itself and 1) leaving whole number answers.

The program shall have the following definition:

Take two numbers from the keyboard. The first is the starting point and the second is the last number to be checked, so that we would give (say) a starting point of 23 and a finishing point of 122, and the program would print out all prime numbers between these two points. On completion, the program will ask for another set to be entered. If *both* numbers that are entered are zero, then the program will stop.

We must initially write down in clear steps exactly what the program is to do.

1. Display a message and input a number from the keyboard. This is the starting point.

2. Display a message and input a second number from the keyboard, this time the finishing point.

3. Check to see if both numbers are zero, in which case stop the program.

4. Test that the finishing point is larger than the starting point (had you thought about that?), and if not, reject the finishing point and ask for another.

5. Check to see if the start number is prime, and if so, then print it.

6. Add one to the starting point number, and test to see if it has gone past the finish point. If it has, then start back at step number 1 again. Otherwise, continue at step number 5.

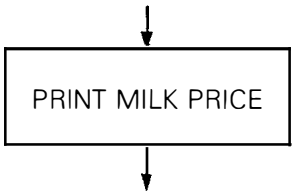
Already, some of these steps may be suggesting some commands or statements to you, and the outline of a program taking shape in your mind.

4.1/2 FLOWCHARTING CONCEPTS

Now that we've defined in some detail what we want to happen, we must begin to look at those 6 steps a bit more closely. This stage is called *flowcharting*, and involves you in writing a sequence of boxes each of which contains a single action. From this *flowchart*, you can directly write a program. Although flowcharts can be really glamorous at times, here we'll restrict it to only two types of boxes, an "action" box, which instructs the computer to do something, and a "decision" box, which asks the computer to test something and **GOTO** the appropriate place depending on the test being true or false.

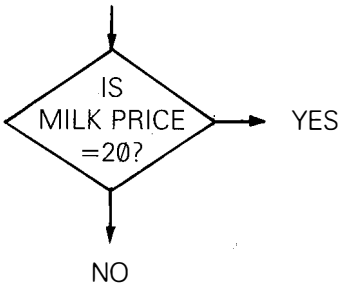
Here is an example of each type of box:

(a) ''action'' box



The ''arrowed lines'' leading into and out of the box indicate the flow of the program, so the next action in the program would be connected to this box by the arrowed line leading out from the base. You'll see a full flowchart of our example program soon.

(b) ''decision'' box



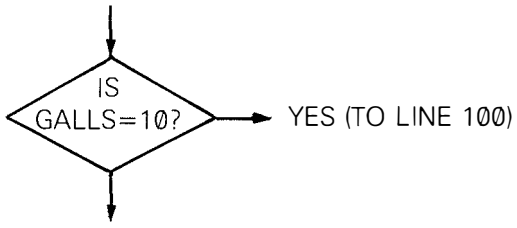
In a ''decision'' box, there are two ways out since any decision (or condition) can result in two answers – true (or YES) and false (or NO). Again, these arrows would connect to the next action or decision box according to the given result.

Question
Write down how you think the following would be written in a flowchart:

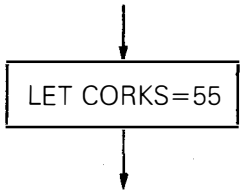
(i) 50 **IF** GALLS=10 **THEN GOTO** 100
(ii) 430 **LET** CORKS=55

Answer

(i)



(ii)



Read Section 4.1/2 again if you were wrong.

4.1/3 STUDYING THE PROBLEM

Before we can get really stuck into our flowchart we need to know a bit more about prime numbers – step 5 above was a bit hazy, wasn't it? Let's have a bit more of a think about them first.

What do we know? Well, there should be a remainder after the number has been divided by any other number. That's easy enough – one of the examples in Chapter 3 asked you to separate an integer from fractions and print the two separately, so we can manage *that* fairly easily. What else?

Any number can be divided by 1, so we obviously don't want to include 1 in our search.

Quite clearly, we must divide the number by other numbers, starting with 2, checking to see if there is any remainder after each division. If we find no remainder, then the number cannot be prime, as we will have found *at least one* number that will divide exactly into our suspected prime number.

At what point do we stop? We can't go on dividing by numbers indefinitely. This part is a bit cunning. We could go on dividing by numbers, starting from 2, until we got to the number itself – but as the example below shows, this involves some duplicate work:

To test if the number 9 is prime.

9/2=4.5 Answer has fractions, so carry on.

9/3=3.0 Answer does not have fractions so 9 is not a prime number.

To test if the number 11 is prime.

11/2=5.5	Fractions, so carry on.
11/3=3.666	More fractions.
11/4=2.75	Fractions again.
11/5=2.2	And again.
11/6=1.83	And again.
11/7=1.57	Fractions.
11/8=1.375	Still fractions.
11/9=1.22	Yet more.
11/10=1.1	Last of all, fractions.

11 is therefore shown to be prime. But do we really need to check all those divisions?

The answer is no – we only need to check up to the *square root* of the number (in this case 11), because after that point we are basically checking the same things over again. This is due to the fact that

7*3=3*7 i.e. the order of two items multiplied together is irrelevant . . .

and once we exceed the square root, we are merely reversing the order of the test (in effect).

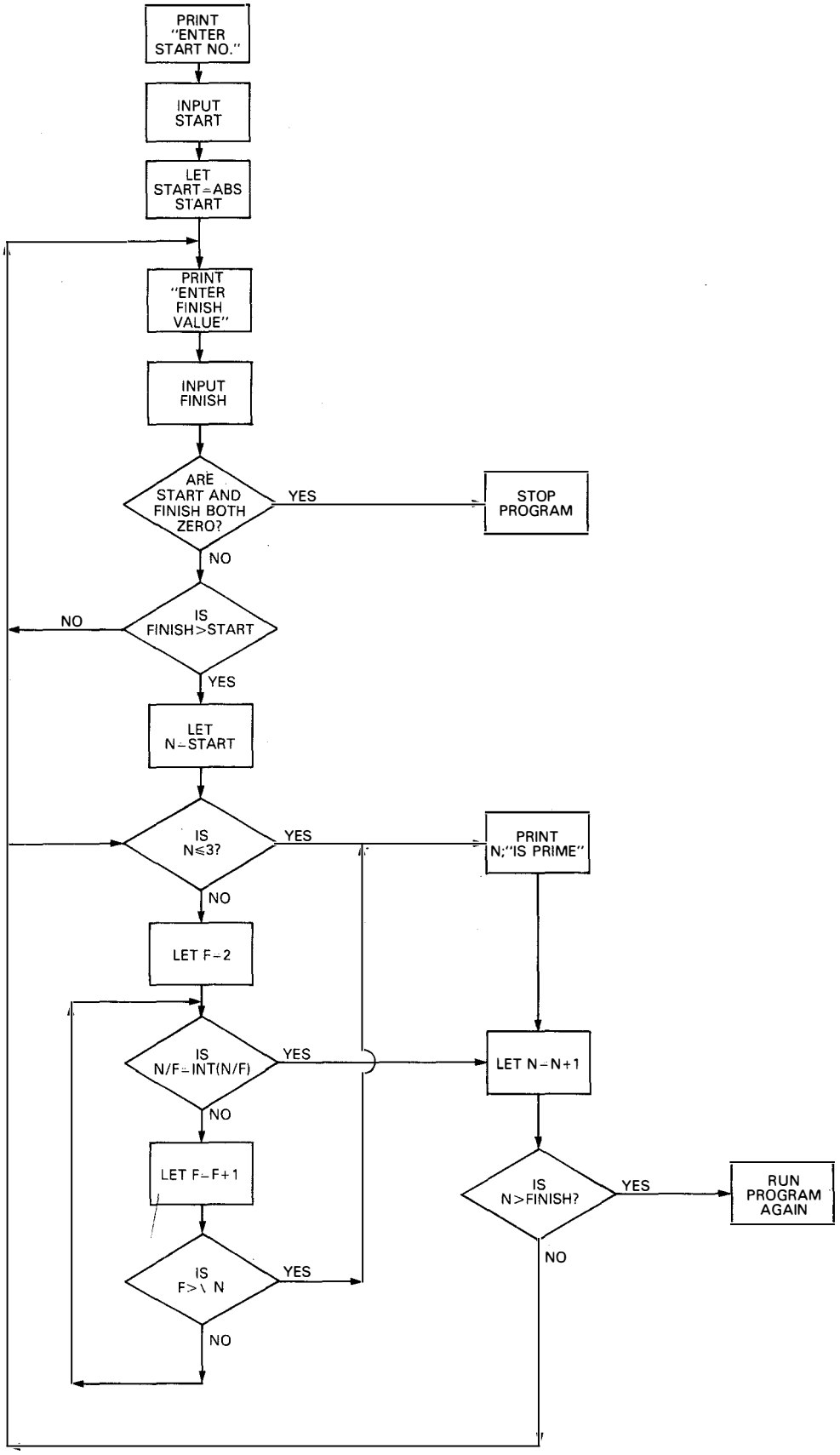
We now know all there is to be known about prime numbers, at least as far as this program is concerned.

This all may seem a bit pointless, but what we are trying to achieve is the need to check out your ideas properly before you start writing a program. If your "image" of a program's workings is not properly thought out, then the program will most likely not work at all, or at best work very inefficiently. Under these conditions, you'll probably spend twice as long struggling to get the thing going and end up having to write it all again anyway.

Always think it out first.

4.1/4 THE FULL FLOWCHART

Back to our program. Now we can write our full flowchart of the prime numbers program. It should look something like:



4.2 ITERATION (2)

4.2/1 FOR AND NEXT

Although it would be perfectly possible for you to write that program with the knowledge you have gained so far, we can tidy things up quite a bit, and avoid a few unnecessary **GOTO** commands. Where possible, it is better to avoid **GOTO** for several very good reasons. At the moment, one of the most important reasons is the problem of line numbers.

Let's put the prime numbers program to one side for now and study this new problem for a while.

We have seen that it is better to separate line numbers in gaps of ten, so that new lines can be added into a program. Sooner or later, though, there will come a time when you'll want to fit a whole load of new program lines in and there just will not be room. The program starts to get cluttered up with odd lines tucked in here and there and gets extremely cumbersome. At this point, you'll want to go through and *renumber* all the program line numbers, so that they start again in nice gaps of ten.

But here's a big problem. If all the line numbers are changed, what happens to all those lovely **GOTO** commands that are referring to the original line numbers?

Quite simply, all the **GOTO** commands must be altered as well so that they use the new line numbers. And that is not an easy task.

Whenever we can avoid using line numbers in a program, it is better to do so.

Here's one way, and we shall be using this in our prime numbers program:

FOR (on the F key).

This command can replace quite a few other commands. Let's side-step for a bit and consider a small example. You have already written a program (in Chapter 3) to print the numbers 1 to 10 down the screen. My solution was:

```
10 LET NUMBER=1
20 PRINT NUMBER
30 LET NUMBER=NUMBER+1
40 IF NUMBER>10 THEN STOP
50 GOTO 20
```

By using the **FOR** statement, this can be reduced to three lines! Look at this:

```
10 FOR N=1 TO 10
20 PRINT N
30 NEXT N
```

And that's it. OK, so it doesn't make much sense just yet, so let's look at it in detail.

4.2/2 HOW FOR AND NEXT WORK TOGETHER

The **FOR** statement comes in two parts – **FOR** and **NEXT**, which is shown on line 30 above.

FOR allows you to specify the *starting* and *finishing* value of what is called a *control variable*; in this case the variable is N (look at line 10 again – **FOR** N=....). A small disadvantage of using this method is that a control variable *can only be a single letter*, and not a nice name like NUMBER. Oh well, it's a small price to pay, and even a single letter can be quite meaningful at times.

Line 10 therefore says:

Set variable N to the value 1, and continue all the way through to the value 10.

Continue what?

The control variable holds its value all the way through the following lines of program until it comes to the **NEXT** statement. At this point, the variable is updated to its next value (so if N was 2, it will become 3, and so on), and the ZX81 starts back at the line number *following the* **FOR** *statement*.

If the variable has reached its finishing value (given on the **FOR** line) when it gets to the **NEXT** statement, the ZX81 will just drop through the **NEXT** and carry on with whatever follows.

Time to unpick that last example.

The first time that the ZX81 comes across line 10, it sets up a new variable N to the value 1, since the line says **FOR** N=1...., and therefore 1 is the starting value.

The next line (20) says **PRINT** N, and so the number 1 will be printed on the screen.

Now the ZX81 sees line 30 – **NEXT** N, so it adds 1 to N, giving 2, checks to see if 2 is greater than the finishing value on the **FOR** line (which is the value 10 – **FOR** N=1 **TO** 10), and since it isn't (2 is less than 10!), carries on with the line number *following* the **FOR** statement. In this case line number 20.

Once again, line number 20 says **PRINT** N, so we get the number 2 printed out.

This carries on until N reaches the value 10, at which point line number 30 will add 1 (**NEXT** N), giving 11 and since 11 is greater than 10, the ZX81 will drop through to the next line.

In this case there is no other line, so the ZX81 will stop with the message 0/30 at the foot of the screen.

Type those three lines into your ZX81 and run them. See what happens.

Question

(a) Which of the following three statements are valid?

(i) 10 **FOR** X=5 **TO** 23

(ii) 10 **FOR** XTRA=9 **TO** 18

(iii) 10 **FOR** P=0 **TO** 10

(b) Write a program to print the values 10, 20, 30, 40 up to 100 down the screen. Use **FOR** and **NEXT**.

Answers

(a) (i) Valid
(ii) Not valid – control variable names can only be a single letter
(iii) Valid

(b)
```
10 FOR N=1 TO 10
20 PRINT N*10
30 NEXT N
```

How did you get on? If your answers were wrong, then read sections 4.2/1 and 4.2/2 over again.

4.2/3 CONTROL VARIABLE LIMITS

Before we carry on with our original prime numbers program, let's clear up a couple of further points associated with **FOR** and **NEXT**.

1. The starting and finishing values on the **FOR** statement can be any numeric expressions – this means you can write things like:

```
100 FOR N=(FREDDY+3)*10 TO JIMMY-3
```

2. If the finishing value is *less* than the starting value, then no statements between the **FOR** and **NEXT** will be executed. Here is an example. Type it in and run it to check what it does:

```
10 FOR N=500 TO 499
20 PRINT "CAN YOU SEE THIS?"
30 NEXT N
40 PRINT "MAYBE YOU NEED SPECS"
```

3. We can alter the "step" value of the **FOR/NEXT** loop. The examples above all assumed that the control variable would be updated by 1 each time. If we write:

```
10 FOR N=1 TO 10 STEP 2
```

then each time the **NEXT** statement is met, the control variable will have 2 added to it rather then 1. The "step" value can also be any numeric expression.

These points mean that you could count *backwards* if you wanted, by writing the following program:

```
10 FOR N=10 TO 1 STEP -1
20 PRINT N
30 NEXT N
```

"Ah!" your are saying. "But the finishing value is *less* than the starting value, and so nothing would happen!"

OK. Time to own up. The actual definition of when a **FOR/NEXT** loop is terminated is:

– when the **FOR** is met, an initial check is made to see if start is greater than finish (if step is greater than or equal to zero), or start is less than finish (if step is less than zero). If this is true, then all following statements are ignored until a corresponding **NEXT** is found.

– when the **NEXT** is met, the same test is made. If the test is true (i.e. the loop has finished) then the ZX81 "drops through" to the next line number. If not, the ZX81 returns to the line number *following* the **FOR** statement.

Whenever the **STEP** is missing from a **FOR** statement, a step value of 1 is assumed by the ZX81.

4.2/4 NESTED LOOPS

It is perfectly possible to put **FOR/NEXT** loops inside each other, like this:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 3
30 FOR C=1 TO 9
40 PRINT C;
50 NEXT C
60 NEXT B
70 PRINT
80 NEXT A
```

What will happen? First of all, this little program is on a tape for you to load and look at.

```
LOAD "FORTEST"
```

Compare it with the listing above.

Initially, variable A is set to the value 1 (line 10). Line 20 then sets B to the value 1, and line 30 also sets C to the value 1.

Line 40 prints C – i.e. it prints 1 – but is followed by a semi-colon, so more is to follow on this line.

Line 50 updates C to its next value (2) and since this doesn't exceed its finishing value, the ZX81 will go back to line 40 again.

This continues until the ZX81 has printed 123456789, at which point C goes beyond its finish point, and so we drop through to line 60, which takes B to its next value (2), and returns to line 30 again. But line 30 forces the ZX81 to print 123456789 (like it did above). So now we get three lots of "123456789" printed on a line. When B has reached 3 (its finishing value), the ZX81 drops through to line 70 which just says **PRINT**. This will make the ZX81 start a new line next time round.

All of this is repeated 10 times while variable A goes from 1 to 10.

Watch it while it runs – you'll see 123456789 printed (that's variable C doing that) three times (variable B) on each line, (variable A) ten times over.

Question

In the last question, you were asked to write a program that prints 10, 20, 30, 40, up to 100 down the screen. The solution given was:

```
10 FOR N=1 TO 10
20 PRINT N*10
30 NEXT N
```

Write a similar program but with one important difference – the program is to print the results *backwards* (i.e. print 100, 90, 80, 70 . . . stopping at 10).

Answer

```
10 FOR N=100 TO 10 STEP -10
20 PRINT N
30 NEXT N
```

How did you get on? If you messed it up or couldn't do it, then read section 4.2 "Iteration (2)" once more.

4.3 ITERATION AT WORK

4.3/1 WATCHING A PROGRAM RUN

Back to prime numbers.

The program is on a tape for you – we'll look at it on the ZX81 and pick out all the interesting bits for you in this text.

LOAD "PRIMES"

Try running it before you look at the program. For example, attempt the following sample run, just to see what happens:

```
RUN
ENTER START NUMBER 10
ENTER FINISH NUMBER 50
. . . .
. . . .
ENTER START NUMBER 0
ENTER FINISH NUMBER 0
9/80
```

Well, not exactly the most exciting program in the world, but 14th century mathematicians would have given blood for it!

Time to look at the program itself. Type **LIST**.

One thing to notice straight away is that the last line shown ends half-way through, and there's a new error number at the foot of the screen – report 4! (If you've bought a 16K RAM pack then this won't have happened to you.)

This means that the ZX81 has got no more room inside it to show any more of the program on the screen. It doesn't affect you all that much at this stage, but one or two strange things can start to happen if you are not aware of it. Whenever you see report 4 while listing or running a program you are writing, check the section on "Common Problems and Solutions" at the end of this course. It will save you some awful heartaches.

The "PRIMES" program almost fills the ZX81 up (without the 16K RAM pack, anyway), and so report 4 is given. There are many ways of making more room available, and you'll be introduced to them later on in the course. For now, though, we'll be content to carry on with only a little space left.

Lines 1 to 100 should be fairly easy stuff for you to follow by now, with the one possible exception of line 30.

Question

(This is memory-jogging time!)

What does line 30 really do?

Answer

It makes sure that any value given for START *must be positive*. The **ABS** function converts any following expression to positive.

This was introduced in Chapter 3, section 3.3 "More Functions". Without spending too much time, check back through that section again if your answer was wrong.

Notice how lines 90 to 110 check for any incorrect FINISH value and give you a nice message if anything is wrong (like FINISH value is less than START value). The beauty of this is that you are always aware if something is not liked by the program. It rejects the input and tells you so. Unless you have good reasons, *always* make sure your programs let someone know what's going on. It's not too bad when you're the only person that ever runs them, but when you give them to a friend (or enemy?!!), they'll certainly not thank you when they can't work out what the program is doing all the time!

Lines 200 to 400 are a big **FOR/NEXT** loop. Each number from START to FINISH is taken one at a time ("step" value of 1 is assumed), and checked to see if it is a prime number. Because numbers 0 and 2 are prime, line 210 makes sure that numbers less than three (i.e. 0, 1 and 2) are treated as prime straight away, by sending them to line number 300 (which prints them out as prime).

Lines 220 to 250 is where all the real work is done. Earlier on, you were shown that it was only necessary to test for prime up to the square root of the number. Since any number is divisible by 1, line 220 forces a loop using variable F (which stands for FACTOR here), checking to see if F divides exactly into the number N, starting at 2, and going up to the next whole number after the square root of N – **INT** (**SQR** N)+1. Remember that the **INT** function removes all the decimal fractions from the following expression – in this case **SQR** N, or the square root of N.

Line 230 is also quite complex. The variable WHOLE is set to the lowest whole number obtained by first dividing N by F, and discarding any left-over decimal places – **INT** (N/F). When this is multiplied back by F, if this new value is unequal to the original number (N), then the number F did not divide exactly into N. Let's make that a bit more clear by looking at a real example:

Assume that N is 11, and that we are checking to see if the number 2 (F) will divide exactly into 11. Taking N/F, this gives 11/2, or 5.5.

Taking **INT** (N/F) or **INT** (11/2), we get answer 5.

When this is multiplied back by F (5*2) we get 10, and this is not equal to 11. This is because the 0.5 has been dropped by the **INT** function.

Line 240 compares this result with the original number. If they are equal, then F divided exactly into N, and therefore N is not a prime number, so the program goes to line 400, which gets the next N value. Otherwise a message N IS PRIME (line 300) is printed out.

After all the N values from START to FINISH have been tried, line 410 will start the whole thing off again.

4.3/2 PROBLEMS FOR YOU

Question

In section 4.1, you were given 6 points which showed what our prime numbers program was to do.

Write down, against each step, the line numbers in the program that correspond to that step.

Answer

Step 1	lines 10 to 40
Step 2	lines 50 to 70
Step 3	line 80
Step 4	lines 90 to 110
step 5	lines 200 to 300
Step 6	lines 400 and 410

If your answer was wrong, then you should check carefully the answer given. Try to understand how the original list of points has been translated into a program.

Also, whether your answer was right or wrong, check the program listing back to the flowchart. Look to see how closely connected the two are.

There are many occasions, especially with smaller programs, when it seems a bit unnecessary to write a full flowchart. This is not a golden rule, and I would be wrong to tell you to flowchart everything you write, but you will nearly always find your programs are more compact and error-free when you have first written a flowchart.

Question

Now you've had a good opportunity to look at a flowchart, it's time to write one for yourself. You are required in this question to give three results:

(a) a step-by-step guide to a program
(b) the flowchart
(c) the actual program

The outline of the program is:

This program calculates the metric equivalent (in centimetres) of inches. The conversion rate is one inch = 2.54 centimetres. the program should accept a number in, which is assumed to be in inches, and the program should print out the value in centimetres. It then goes on to print the next 11 inches converted as well, so if 5 inches were entered, the program would print the results from 5 to 16 inches down the screen. A value of zero inches means that the program is to stop.

No example run is given this time – you are left to create your own display (but remember how the primes program worked).

Answer

Here is one possible solution:

(a) 1. Display a message and input a number from the keyboard.

2. Check that if the number is zero, then stop the program.

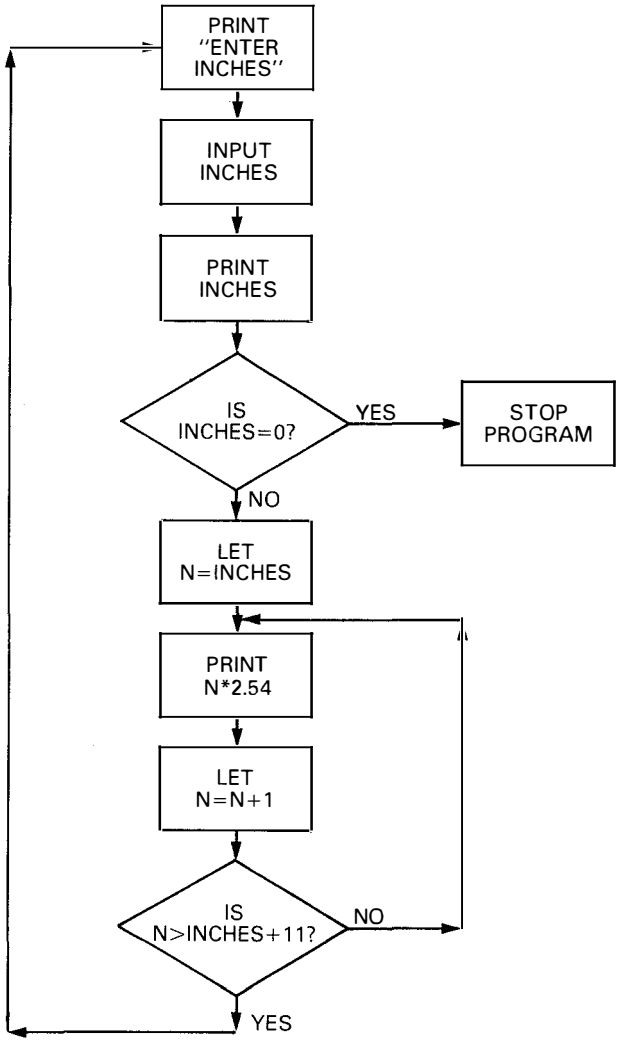
3. If the number is negative, then convert the number to positive.

4. Convert the number to centimetres and print the result.

5. Add 1 to the inches value. Repeat step 4 for 11 more conversions.

6. Start at step 1 again.

(b) A flowchart of this would look like:



(c) The whole program could then be:

```
  1 REM METRIC CONVERSION
 10 PRINT "ENTER INCHES";
 20 INPUT INCHES
 30 LET INCHES=ABS INCHES
 40 PRINT INCHES
 50 IF INCHES=0 THEN STOP
100 FOR N=INCHES TO INCHES+11
110 PRINT N;" INS=";N*2.54;" CMS"
120 NEXT N
130 GOTO 10
```

Perhaps you found it a bit too long-winded and so skipped over a lot of it. Well, it doesn't matter too much, as long as you feel confident about the topics we've covered.

Program design is largely a personal affair – my style very probably differs from a lot of other people's, and it is important to remember that practice will develop you own skills more than anything else at all. The computer needs to be told absolutely everything, and the reason for flowcharting is to remind you that you may be assuming things that the ZX81 will not take for granted.

Looking at the way other people have written a program is an extremely good way of learning. If you know someone else who is using or has used this course, try to get hold of their solutions to some of these problems. You may find that you prefer their method to the ones given here, or even think that your own is the best. Either way, it helps you to see how your own style can develop, and gradually become a quick and efficient programmer.

Summary

In the next chapter, we'll be looking at how to use the display to more advantage, and how you can run the ZX81 in one of two modes.

Here are the topics we've covered in this chapter:

– we've seen how program design can make programming much easier, how a step-by-step guide and a flowchart can make the program strategy much clearer.

– that it is far better to avoid using line numbers wherever possible (i.e. avoiding the **GOTO** command where **FOR/NEXT** can be used).

– how to use control variables to allow an iteration to be undertaken any number of times, also allowing the **STEP** value of the variable to be altered.

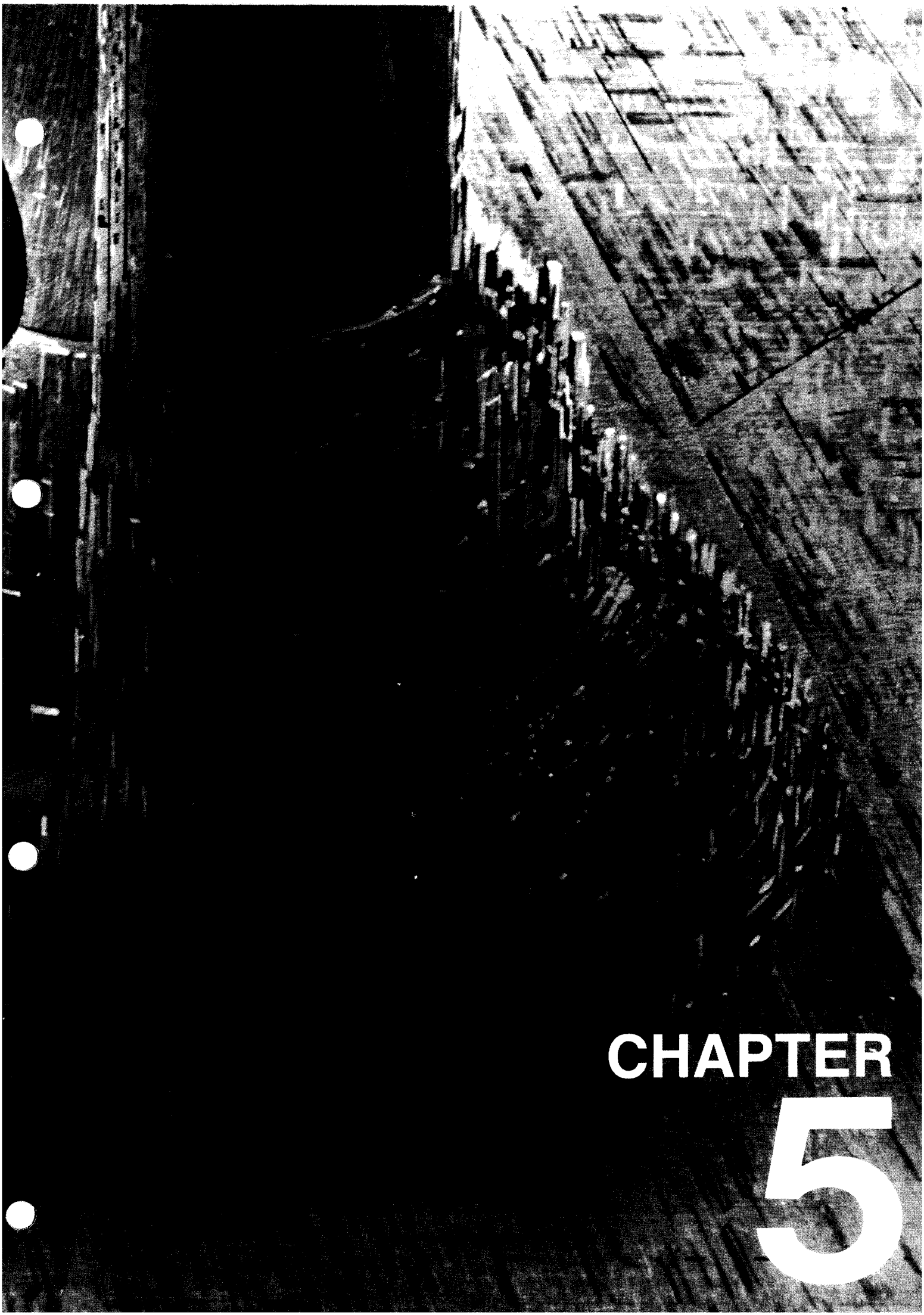
– that **FOR** loops can be nested within each other.

– that seeing a technique in action can make that theory easier to understand.

Exercises

1. Given that 1st January 1900 was a Saturday, write a program that will tell you the day of the week for any date that you enter (up to 31st December 1999). You will probably need to flowchart this. So, on which day of the week were you born?

2. Extending exercise (1) a little further, you could allow the program to calculate the number of days that have occurred between any two dates entered (dates during the 20th century).


CHAPTER
5

Speeding Up and Looking Nice (Using FAST and SLOW)

We are about to discover two ways of running your ZX81 which allow you to run different types of programs under optimum conditions. This is discussed in section 2, while in section 3 we go into more detail on how to make use of the display, so that report 5 can be avoided, and you can create a more "presentable" screen in your programs. The final section is for those who want to use the ZX printer.

First, though, we'll have another look at expressions.

5.1 USING CONDITIONAL EXPRESSION VALUES

To begin with, let's study another program from cassette:

LOAD "SPEEDY"

This program will serve two purposes for us, so first of all **LIST** the program on the screen.

The use of *true/false expressions* has already been shown to you, mainly in Chapter 3, where they were used in conjunction with **IF** statements. There is no reason why they cannot be used in **LET** statements, though, because they are still only numeric expressions giving a value of 0 or 1. This can be an extremely powerful tool when writing some programs. Here, the idea is used to allow a black and white square to be alternately printed.

Line 20 sets variable C to an initial value of zero – remember: this is also the equivalent of "false".

As we have seen, the screen is split up into 22 (usable) lines down, and each line contains 32 positions across. Program lines 30 and 40 start two **FOR/NEXT** loops which run for 32 times (line 40) for every one of 16 times (line 30).

Now look at line 50. This line, or a very similar one, was shown to you in Chapter 3. The ZX81 says "**IF** the expression is *not* true, then obey the following action, which is to **PRINT** a black square." Since variable C has been set to "false", or "not true", the ZX81 will print a black square. Notice that the black square is put in quotes (" ") so that it is actually treated as text. You can put almost anything you like in between quotes – it is printed exactly as you put it.

The print statement on line 50 is followed by a semi-colon, so a new line is not yet started.

Line 60 is the exact opposite of line 50 – if C is "true", or has a value 1, then a blank space (or white square) is printed. As yet, variable C only has a "false" value.

This is where line 70 comes in. This line says:

70 **LET** C=(C=0)

What on earth does that mean? We'll need to take it bit by bit to understand it properly. First of all, the ZX81 looks at the expression in brackets – these are always worked out first. The expression C=0 is given one of two values – it can be either true or false.

Another way of looking at this is to read the statement above as:

"Let the variable C be assigned the value of the result of the expression 'Is C equal to zero?' ".

So if the expression is *true*, variable C is assigned the value of true – i.e. 1 – and if the expression is *false*, then C is assigned the value of false – zero.

At the moment, C contains zero, so the expression C=0 is true. So the statement can therefore be regarded as **LET** C=1. After line 70 has been run, variable C will contain the value 1, or "true". The ZX81 has swapped the value of C from 0 to 1.

In case you are worried, there is nothing to stop you using the same variable name twice on the same **LET** statement – the ZX81 is clever enough to understand that you want to use the *old* value of the variable on the right-hand side of the line, and set the *new* value in only after everything else has been done. This is why a statement like **LET** X=X+1 works.

Line 80 now takes the ZX81 back to line 50 with the next value of variable X – a step value of 1 is assumed, so X now contains the value 2.

Line 50 says **IF NOT** C . . . but this time around C is *true*, so this line will be ignored.

Line 60 says **IF** C . . ., and so a white square will be printed.

Back to line 70 again.

Question
What do you expect line 70 to do this time? Work it out in the same way that we did above, by considering the line bit by bit.

Answer

Variable C will contain the value 0, or false, after line 70 has been run.

In case you didn't understand fully, we'll look at it once more. The expression (C=0) is worked out first, because it is in brackets. C currently holds 1, or ''true'', so C=0 is therefore false, since C does *not* equal 0. So the line can be regarded as **LET** C=0, since (C=0) gave the value zero, or ''false''.

Perhaps the full significance of line 70 is starting to hit you – every time line 70 is run, the variable C alternates its value between 0 and 1. Whatever value C contains (0 or 1) when the program reaches line 70, its value is swapped over.

As an example, if you were writing a games program for two players, this line 70 (or something similar) could be used to indicate whose turn it is – player 0 or player 1.

Extremely useful.

Just before we leave this topic, here's a final problem for you.

Question

How would you write a single line that alternates variable P between the values 0 and 50?

Answer

```
10 LET P=(P=0)*50
```

The logic is identical to that given above, except that 0*50 equals 0, and 1*50 equals 50. So P will alternate between 0 and 50.

You will also notice line 90 in the SPEEDY program is the same as line 70. Whenever a complete line of squares has been printed (variable X goes from 1 to 32), line 90 swaps the value of C around so that the **NEXT** line printed has the squares in the opposite order.

5.2 RUNNING MODES

5.2/1 TIMING THE TWO MODES

The ZX81 can run programs in one of two ways – **FAST** and **SLOW** (these commands are found by SHIFT/F and SHIFT/D respectively). Both methods have advantages and disadvantages, so the mode you select for a program can make use of the advantages without any disadvantages being noticed.

Let's look to see what difference there is when just running a simple program.

Run "SPEEDY". You should do two things while it runs. First, time it to see how long it takes before the message 9/9999 appears at the foot of the screen. Secondly watch to see those squares alternate between black and white. This is all as a result of that line 70 doing its stuff.

How long did it take to run? I'm not going to tell you – it's up to you to record it on a piece of paper for later.

Now we're going to alter the ZX81 to its other mode – **FAST**. Press SHIFT/F for **FAST**, and newline. Run the program again, and also time it to see how long it takes before the message 9/9999 appears at the foot of the screen. Also watch to see what happens.

Question

Well, how long did it take? How much faster (approximately) do you think the **FAST** mode is than the normal mode?

Answer

In fact, **FAST** is roughly 4 times faster than the normal mode.

But you probably saw the disadvantage of **FAST** – that the screen goes blank while the program runs. You were not able to watch the black and white squares being drawn. We'll deal with this in a minute, but first let's examine what else happens when the ZX81 is running in **FAST** mode.

LIST the program again, and try using the cursor keys to move the program cursor up and down the lines of the program.

You may notice that the TV picture jumps around a bit, but don't worry – this is quite normal when the ZX81 is in fast mode. However, you can see that moving the program cursor is a lot faster when you're running in fast mode – the ZX81 does not list the program out again each time (in fact, it does – it's just a bit too fast for you to see it!).

Try using the EDIT key to alter one of the lines of the program. Try inserting a new program line. All in all, you'll find the ZX81 a lot quicker in fast mode, and whenever you type in a new program, or edit an existing program, you'll probably find it better to enter **FAST** before you start.

So why use normal mode?

Well, if you write a program that relies on your being able to see what happens while the program is running, then fast mode is not going to be all that useful!

Just to show you these effects in practice, let's load another program:

LOAD "TRACE"

This program lets you draw a picture on the screen using the cursor keys (you don't need to hold SHIFT this time, just press the 6 key to draw a line down, 8 to draw across to the right, etc). As you hold the key down, a continuous black line starting from the top left-hand corner is drawn. If you take your finger off the key, the line stops as well, so you can draw nice pictures in tracing fashion on the screen.

If you press the C key, the screen is cleared down again, but the line will still carry on from where it left off. If you press the X key, you stop the program.

Note that if you have a basic ZX81 (without the 16K RAM expansion pack, then your "drawing board" will only fit about two-thirds across the screen, and two-thirds down the screen.

Whenever the line reaches the edge of the "board", it stops and waits for you to change direction.

Now try running the program in fast mode. Although the picture is being drawn at four times the speed, you can't see it! The only way of seeing the picture you've drawn is by pressing the BREAK key, or pressing the X key (which is interpreted as "stop" by the program).

Obviously, as you write any program, you must decide which mode the ZX81 is to run in, and enter a line into the early part of the program to select the mode, for example:

```
10 REM CALCULATE PRIME NUMBERS
20 FAST
30 PRINT "ENTER START NUMBER"
....
....
```

It is worth mentioning that **FAST** and **SLOW** are frequently used as commands for immediate mode execution as well as statements within programs.

5.2/2 WHEN THE DISPLAY IS SHOWN

When running in fast mode, the screen will be displayed only under the following circumstances:

(a) when an **INPUT** command is met

(b) when a **PAUSE** command is met (we'll see more on this in another section)

(c) when a **STOP** command is met

(d) whenever an error occurs. This also includes "end of program" condition error 0

So there are many ways of seeing the screen in fast mode, but the program *must* be written with this in mind.

Whenever you switch the ZX81 on, it automatically comes up in normal (or **SLOW**) mode, which is also called "compute-and-display" mode.

For now, do not pay too much attention to the listing of the program "TRACE". It makes use of several features that we will be meeting over the course of the next few chapters, and you will have an opportunity to look at this program in a lot more detail later on.

Question

(a) In which mode does the screen go blank?

(b) When would you use **SLOW**, or "compute-and-display" mode?

Answer

(a) Fast

(b) When you want the results of a program to be displayed while the program is running.

If your answers were wrong, read the whole of section 5.2 "Running Modes" over again.

5.3 MAKING USE OF THE DISPLAY FACILITIES

5.3/1 CLEARING THE SCREEN

Now let's look at the screen display in a bit more detail.

One problem that we had in the "PRIMES" program was that of report 5. Whenever 22 lines of the program were displayed on the screen, report 5 cropped up, and we could only start the program again by using **CONT** to continue running. It would be nice if in some way, the program could keep tabs on how many lines have been displayed, and stop the program from running or, better still, whenever a new set of numbers is entered, the screen would be cleared before the prime numbers are printed. That way, although report 5 may still appear when the screen is full, at least the maximum number of primes will have already been shown.

The answer is a command **CLS**.

This command, when met in a program, clears the screen back to blank again. It does not affect the value of any variables that may have been created, it purely acts on the display.

A classic use of **CLS** is when checking values that are being input. Look at this program:

```
10 PRINT "ENTER A NUMBER (1–5)";
20 INPUT VALUE
30 PRINT VALUE
40 IF VALUE>0 AND VALUE<6 THEN GOTO 100
50 PRINT "I ASKED FOR 1 TO 5. TRY AGAIN."
60 GOTO 10
. . . .
. . . .
```

Question

Can you spot a problem with this program? (hint: try running it on your ZX81 and enter invalid numbers.)

Answer

If 11 invalid numbers are entered in a row, the program stops with error 5 – the screen has become full of nasty little messages.

Obviously, it would be much better if the screen wasn't cluttered up with our attempts to get a correct value in. We still want to see what we've entered when a proper value is input, but it doesn't matter too much if we lose the screen after a wrong value has been entered.

Question

What line would you add into the program above to stop the screen filling up when a stream of invalid numbers are entered? What line number would you use? Think carefully about the line number – try to see why by running the program.

Answer
A command **CLS** is required. Where?
If your solution was:

```
45 CLS
```

then well done. In fact, any line number between lines 40 and 50 will do.
If you had a different line number, then let's see why it will not work properly.
There are many places that the line could go, and here are one or two with the reason why it will not work properly.

15 **CLS**	the message which lets us know what is to be entered is lost.
55 **CLS**	the message informing us of the error is lost.
35 **CLS**	if a correct value is entered, it is not shown on the screen.

Try each of these to see why they are wrong.
Perhaps a different way of writing the program in the first place would help? That's probably true, and here is one way –

```
10 PRINT "ENTER A NUMBER (1–5)";
20 INPUT VALUE
30 PRINT VALUE
40 IF VALUE>0 AND VALUE<6 THEN GOTO 100
50 CLS
60 PRINT VALUE;"? WHAT IS THAT?"
70 GOTO 10
```

This way, we lose one line of the display if an incorrect value is entered, but should further incorrect values be entered, then still only one line is lost, since the **CLS** command will keep the nasty messages down to a single line. Also, the incorrect value is printed in the error line. This is always a good idea as it reminds you of what you have entered – it's always nice to be able to say "Oh, I know what I've done!".
CLS can be used in immediate mode, in which case it moves the print position to the top left-hand corner of the screen.

5.3/2 PRINTING AT A SPECIFIED POSITION

Wouldn't it be helpful if we could actually format our screen? By this is meant that we would be able to print items at any position on the screen. Well, we can.
What do we know so far about the **PRINT** command? We know that we can print any number of *expressions* that are separated by either commas (print items in "zones") or by semi-colons (next expression is to follow this one with no spaces between).
We can also include text (or "strings") in the print.
So, one way of putting items on the screen is to use a succession of **PRINT** commands to print something on (say) the fourth line down on the screen.
Here's a small program that prints the word "HELLO" in the middle of the screen.

```
  10 FOR Y=1 TO 12
  20 PRINT
  30 NEXT Y
  40 PRINT "                HELLO"
9999 STOP
```

Lines 10 to 30 print 12 blank lines to space down the screen, and line 40 prints the word "HELLO" in the middle of the next line.

This is all very well, but it does seem a bit wasteful, doesn't it?

We can alter the next print position at any time we like by putting

AT y,x

in the **PRINT** command as an item to be printed. **AT** is a function, so it requires the usual two keystrokes to get it – first, SHIFT/NEWLINE to tell the ZX81 that you want a function, then the "C" key on the bottom row. This will give **AT**.

AT doesn't actually print anything at all – it just moves the print position to line number y, character position x.

Here's an example:

PRINT AT 12,13;"HELLO"

Try it.

That single line replaces the program given above. It also prints "HELLO" in the middle of the screen.

Notice the semi-colon after **AT** 12,13; – if anything else followed it (like a comma, or nothing at all) then the print position would be moved on again, which would rather defeat the purpose of using **AT** in the first place!

Any number of **AT** functions can appear in a **PRINT** statement, but remember that any comma that follows will force the next field across to the next "zone" on the screen, and any **PRINT** command that is terminated without a comma or semi-colon at the end will cause an extra line to be added.

Also, you should remember that the *first* line on the screen is represented by line number 0, and the first column on the screen is column number 0.

So the top left-hand corner of the screen would be given by:

PRINT AT 0,0;"X"

The bottom right-hand corner is:

PRINT AT 21,31;"X"

If you always remember that "**AT** y,x" is treated just like an expression, except that nothing is printed, then all the other rules of the **PRINT** command apply as normal.

Question

Write a *single* line that prints the letter A in the top right-hand corner of the screen, and also prints the letter Z at the bottom left-hand corner of the screen.

Answer

PRINT AT 0,31;"A";**AT** 21,0;"Z"

If your answer was wrong, then read section 5.3/2 again.

5.3/3 TABULATING PRINT

By now you're probably catching on that wherever you see a number in a command or function, like the **AT** function, then it can be entered as any numeric expression.

The line number and column number in the **AT** command can be any numeric expression at all, so valid statements would be:

```
PRINT AT LUCKY/3,LUCKY+4;"HERE WE ARE"
PRINT "FIRST";AT (LINE-2)/2,COL+4;"SECOND"
```

AT is only valid as part of a **PRINT** command. If you try to use it for anything else, like **LET** X=**AT** 9,5 then you'll get a syntax error. It doesn't make sense in other commands anyway.

Here's an example of how to use **AT** to print the numbers from 1 to 9 backwards. Type it in and run it:

```
10 FOR N=1 TO 9
20 PRINT AT 10,9-N;N;      (notice all the semi-colons!)
30 NEXT N
```

Before we leave the formatting power of the **AT** function, there is one other function that allows us to present display lines in an ordered fashion. This time, however, it is only concerned with the current line. The function is

TAB n

where n represents the column number that you want the *next* item printed in. Obviously, this function should be followed by a semi-colon – as is **AT** – otherwise there is no real point in moving the print position across to a specified column number! Here's an example of **TAB** in use:

```
10 PRINT "HI";TAB 24;"THERE"
```

If you run this single line program, you'll see something like:

```
HI                      THERE
```

and the word THERE would start in column 24 on the display. The value of n can be any numeric expression – also the same as **AT**, but the real difference is that **TAB** does not need to be told which line number to move to – only which column number.

If the column number in a **TAB** function has already been passed by other printed items, then the ZX81 moves down to print the next item in the correct position on the *next line down*, so be careful that you know where you are when you use **TAB**. If the line above was changed to:

```
10 PRINT "HI";TAB 1;"THERE"
```

the ZX81 would print this out as:

```
HI
 THERE
```

since the column 1 in the **TAB** function has already been passed.

The value that follows **TAB** should be in the range 0 to 31 (inclusive). If any higher values are used, the ZX81 takes the remainder after dividing by 32 as the **TAB** value. As an example, **TAB** 67 would tab over to column 3, since 67 divided by 32 leaves remainder 3. (In mathematical terms, this is called *modulus* 32).

Strictly speaking, **AT** and **TAB** are not functions at all since they do not give any value as a result. Because they appear on the *underneath* of the C and P keys, they are obtained in the same way as functions and this is how they will be referred to later on.

Question
Write a small program that takes in a number from the keyboard, checks that the number is between 0 and 20, then prints the word "HELLO" in that column on the first line of the screen.

Answer
You could use either the **AT** or **TAB** functions to handle this – here's one solution:

```
 10 PRINT "ENTER A NUMBER;
 20 INPUT NUMBER
 30 IF NUMBER>=0 AND NUMBER<=20 THEN GOTO 100
 40 CLS
 50 PRINT NUMBER;"? YUK. TRY AGAIN."
 60 GOTO 20
100 CLS
110 PRINT TAB NUMBER;"HELLO"
9999 STOP
```

You could replace line 110 by:

```
110 PRINT AT 0,NUMBER;"HELLO"
```

Did your solution work? You're getting on quite well if it did. If not, then try reading that last section again – section 5.3/3.

5.3/4 SCROLLING THE DISPLAY

Another command which makes good use of the display is the command

SCROLL (on the B key)

Before video-type displays were used on computers, a device called a Teletype (brand name) was used, which was basically similar to a typewriter, although it could be computer controlled, and the computer could type messages to the operator and vice versa.

Because of the popularity and flexibility of these devices, most video-based computer systems still use this method of printing information – a line at a time, and each new line pushes the previous ones up, with the new line appearing at the foot of the screen. It is called *scrolling*.

The problem with video systems is that a screen only has a limited number of lines that can be seen at any one time – on the ZX81 it is 24 (although two are reserved for the ZX81 itself), whereas the printed paper on a Teletype could be any length at all! So a video screen has to lose lines once they reach the top of the screen.

Let's see an example of this.

LOAD "SCROLLER"

This small program demonstrates how the **SCROLL** command can be used to make the ZX81 work just like the old Teletypes.

Whenever the ZX81 sees a **SCROLL** command, it moves every line on the display up one line, leaves the bottom line blank, and moves the print position to the start of this blank line.

Run the "SCROLLER" program to see how it works. You'll see each line shifting up one each time. Watch what happens when the lines get to the top.

The only way to stop the program is by . . .

Question
What?

Answer
If you said (or did!) "Pull out the plug", then no marks. Your answer should have been:
"Press the BREAK key". This was covered in Chapter 3.

LIST the program. I expect you can work it all out for yourself, but the important lines are lines 100 and 110.

Line 100 scrolls the whole display up one line, leaving a blank line at the bottom, and losing one off the top.

Line 110 prints a new line at the foot of the (usable) display – line number 21.

So if you want to write a program like the "PRIMES" program, you could use something like this to accomplish it. Before you print anything, include a **SCROLL** command. The following **PRINT** statement will automatically appear at the foot of the screen.

As mentioned, the "PRIMES" program would have been a good candidate for this type of printing, because the report 5 would never arise.

5.3/5 A SECOND EXAMPLE

For those with a mathematical mind, here's a pretty program that draws a continuous sine wave down the screen. It combines several of the points we've covered in this chapter so far, and it's worth typing in and running purely to see them all in use.

```
10 LET A=0
20 SCROLL
30 PRINT TAB (INT (SIN A*15+16.5));"X"
40 LET A=A+(PI/2)/5      (PI is given as a function on the M key)
50 GOTO 20
```

We're not going to unravel this particular program – it's up to you if you feel you would like to. Although it may appear quite complex, if you understand the nature of trigonometry, sines and radians, then it'll pose you no real problem!

Question

(a) If you saw the statement **PRINT AT** 5,10;"HELLO", on which line of the screen would you expect the word to appear?

(b) And in which column?

(c) What is the number of the first display line on the screen?

(d) Name three ways in which the screen can be seen when the ZX81 is running in **FAST** mode.

Answers

(a) Line number 5

(b) Column number 10

(c) Line number 0

(d) Any three of these four:
- an **INPUT** statement
- a **STOP** statement
- a **PAUSE** statement
- when any report code is given

If your answer to (a), (b) or (c) was wrong, then you should read back over sections 5.3/2 and 5.3/3. If you got (d) wrong then read section 5.2 "Running Modes" again.

5.3/6 DELIBERATE PROGRAM DELAYS

There are obviously going to be occasions when a program running in fast mode wants to show some results that have been calculated, then continue without any sort of intervention.

For example, the sine-wave drawing program given above will not work in fast mode as it stands, because the screen goes blank.

Fortunately, a command has been set up in the ZX81 that allows you temporarily to stop a program that is running in **FAST** mode and show the display. The command is

PAUSE n

where n represents a numeric expression that tells the ZX81 how long you want the program to pause for. If n is 50, then the ZX81 will pause for one second. You can make the ZX81 stop for any length of time that you wish – up to a maximum of roughly 11 minutes, since the top limit of n is 32767. If n is larger than this, the ZX81 will "pause" forever.

Whenever the **PAUSE** statement is used in a program running in **FAST** mode, it must *always* be followed by the statement:

POKE 16437,255 (**POKE** is on the O key)

otherwise the program will probably fail when it is run. For the time being, you will have to accept this – a later chapter will explain this new command in more detail.

By adding in two new line numbers, 45 and 46, to the sine-wave program, we can make it work in fast mode:

```
45 PAUSE 25          (a half-second pause)
46 POKE 16437,255    (to stop PAUSE failing)
```

The program will still work in slow mode, but to a large extent, the **PAUSE** command is wasted, since you can already see the results!

One use of **PAUSE** in slow mode is when you want to put a small delay in the program to slow the whole thing down – like a game program where you only want to allow your player a certain length of time to see something on the screen.

A side effect of the **PAUSE** command which can be useful is that the command is finished when either the time (specified by n above) has elapsed, or *a key is pressed*. This can be put to good use in some programs.

Try this example to see how **PAUSE** can be used in **FAST** mode:

```
 5 FAST
10 FOR X=0 TO 15
20 PRINT AT 10,X;">"
30 PAUSE 25
40 POKE 16437,255      (must follow PAUSE)
50 PRINT AT 10,31-X;"<"
60 PAUSE 25
70 POKE 16437,255
80 NEXT X
90 PRINT AT 10,14;"BANG"
```

The program can be altered to run in **SLOW** mode by changing line 5 (change it to **SLOW**), and removing lines 30, 40, 60 and 70. Now try it again.

Here's a similar type of program that will give you a laugh:

```
 5 SLOW
10 FOR X=0 TO 26
20 PRINT AT 10,X;".TRAIN"      (don't forget the ".")
30 PRINT AT 10,26;"TUNNEL"
40 IF X<>0 THEN GOTO 70
50 PRINT AT 0,0;"PRESS NEWLINE WHEN READY"
60 PAUSE 40000      (wait forever till key is pressed)
70 NEXT X
```

The **POKE** was not needed in this example as the ZX81 is used in **SLOW** mode.

Question

Write a program that drops a man from an aeroplane down to the ground. The man should jump when a key is pressed. You can use the letter "Y" to denote the man coming down, and a row of hyphens (SHIFT/J) to denote the ground. If you can manage it, then try to make the program wipe out the "trail" that the man leaves as he comes down.

Answer
My solution is (try it if you like!):

```
10 PRINT AT 21,14;"————————"   (draw ground)
20 FOR Y=0 TO 21               (for each line on screen)
30 PRINT AT Y,16;"Y"           (draw man)
40 IF Y<>0 THEN GOTO 80        (skip after 1st time)
50 PRINT AT 0,14;"PLANE"       (draw plane in sky)
60 PAUSE 40000                 (wait for a key)
70 GOTO 90                     (continue)
80 PRINT AT Y-1,16;" "         (remove "trail")
90 NEXT Y                      (next line)
100 PRINT AT 21,14;"SPLAT"     (better luck next time)
```

You should read section 5.3/2 if you had trouble using **AT** (or you could have used **TAB**, in which case read section 5.3/3), and section 5.3/6 if you could not get the **PAUSE** command to work properly.

5.3/7 USING GRAPHIC CHARACTERS

This section will only serve as an introduction to graphics. You'll be able to make much more use of graphics when you've covered a few more chapters, but for now, it's enough to see how graphics can be used as part of **PRINT** commands.

As a small demonstration, load this program:

LOAD "PICTURE"

and run it. Isn't that pretty? When you've got bored with it, press the BREAK key and look at the program listing.

As we've said before, whenever you print text, or as it's properly called, a *string*, you can print anything you want between the quotes characters. So you can print numbers, letters, or these funny symbols called *graphic characters*.

How do you get at them?

Simple. By pressing SHIFT/9 (the 9 key has the word "GRAPHICS" in red), you'll first of all notice that the cursor changes to a G.

Let's try it. Type:

PRINT " and then press the GRAPHICS key (SHIFT/9). At this point, the cursor alters as described. You should see on your screen:

PRINT "G

Now, whenever the G symbol is showing, any letter or number key that you press will be shown as white on a black background, instead of the usual black on white. Some of the keys, the keys whose SHIFT codes give whole words or multiple symbols (like SHIFT/S which normally gives LPRINT), will now give you special graphics characters as shown on the key in the lower right-hand corner of the key.

The keys that do *not* have these special graphics codes will always give the "inverse" (that's white on black) of the key, or of the SHIFTed key if SHIFT is used.

Try typing a few in to see what you get. Try SHIFT/S, and SHIFT/C, and just the ordinary keys without SHIFT.

When you've typed a few, you'll need to get out of graphics mode so that you can type the " character needed to finish the text string off. But how? Whenever you type the quotes character, you just get an inverse quotes! The answer is to press SHIFT/9 again. This cancels the graphics mode, and returns you to the usual "letters" mode that you're used to.

Now you can put in the " that is needed, and then NEWLINE will cause the ZX81 to accept the line.

This is how the PICTURE program was created – the black boxes are created by GRAPHIC/SPACE, the other symbols were all created from the special characters given on the A, S and Q keys.

Question
Write down the keys you need to press to print the following symbols:

(a) an inverse P

(b) inverse ''

(c) ¾ black box

(d) (no answer to this part!) Try to think of the appropriate characters to use in order to print the picture of a spaceship.

Answers

(a) SHIFT/9 (to get into graphics)
P

(b) SHIFT/9 (to get into graphics)
" (SHIFT/P)

(c) SHIFT/9 (to get into graphics)
SHIFT/Q (to get the box)

If you couldn't work them out, then read section 5.3/7 again, and try to get used to the graphics mode a bit more by experimenting.

5.4 THE ZX PRINTER

If you're lucky enough to have afforded a printer to go with your ZX81 then here's the section for you! The section is really information only, since all of these features have to a large extent been covered in other ways.

There are three commands that are relevant – mostly they are similar to other commands.

The first is:

LLIST (no it's not Welsh!)

You'll find it by pressing SHIFT/G.

This command works in *exactly* the same way as **LIST** – only it lists the program onto the printer! You can list the whole program or just from a particular line by stating **LLIST** nnnn (where nnnn is the particular line number you're interested in).

This command is a good way of keeping a "hard copy" of your programs.

Another useful printer command is:

COPY (on the Z key)

This command allows you to put a copy of whatever's on the screen onto the printer. So if you've just run a program and you're particularly keen to keep a copy of the results, then press **COPY**.

The printer even gives you all the nice graphic characters too!

The last command for the printer is:

LPRINT (SHIFT/S)

As you might already expect, this command is equivalent to the normal **PRINT** command except, of course, the printing is sent to the printer instead of the TV.

There are a couple of small points to watch out for with this command, however.

The **AT** facility of the usual **PRINT** command will still work, although the line number field has no effect. You should give this a value of zero. Here's an example:

```
10 LPRINT AT 0,13;"HELLO"
```

If the line number is *not* zero, then it will mostly be ignored, but you may get an error if it's greater than 21, so play safe and keep it at zero.

The printer can be a very valuable tool and if you've got one, the best way of finding out how to use it is to try writing some of the future programs with **LPRINT** commands instead of **PRINT** commands.

All the printer commands can be used as immediate mode commands as well as program statements – in fact, the **COPY** command will probably be used most frequently in this way.

Summary

This chapter has largely been concerned with making the most of your display under the two modes of running: normal (or **SLOW**) and **FAST**.

The need to design your programs around the features of **SLOW/FAST** was highlighted.

In the next chapter, we'll find out how to use the cassette recorder to save programs, and how to make the most of your own program libraries.

Here's a summary of this chapter:

– you've seen how conditional expressions are not just confined to the **IF** command, but can be used in a **LET** command to assign a conditional value to a variable.

– how the ZX81 can run in two modes – **SLOW** and **FAST** – and how you can make the most out of them.

– that there are many ways of creating an attractive display, using the **AT** and **TAB** options, or using graphic characters embedded in text strings.

– how the ZX printer can be controlled.

Exercises

1. Print successive values of n! (factorial n) until overflow (error report 6) occurs. This will direct you towards the largest value that the ZX81 can hold (see exercise (3) in Chapter 1).
   n! is calculated as:

   n*(n−1)*(n−2)* 4*3*2*1

   such that 3!=6 (i.e. 3*2*1) and 4!=24 (i.e. 4*3!). You will probably need to use **FAST** mode to handle this program, and display results using . . . . what?

2. Write a program that asks for a number between 1 and 100 to be entered. A second person must now try to guess this number while the program replies "TOO HIGH" or "TOO LOW" in reply to the various guesses. Make use of the print formatting features to give a good presentation of the program's responses.


CHAPTER
6

Using the Cassette

By now, it has started to become important for you to be able to save your programs on cassette tape. Most of the programs you have written so far have only been a few lines long, and have not really been worth saving anyway.

This chapter is of a totally different nature from the others; there are few questions asked, and mostly the text consists of advice, hints and tips to make sure that your programs will be kept with the minimum number of troubles.

Before we start writing programs of much larger size, we'll learn how to use the cassette properly – this way it's up to you to decide whether or not you want to save a program that you've just written.

6.1 WHY SAVE PROGRAMS?

There is a very good reason for saving programs on tape.

Security. Not that any burglar would be interested, but so that you don't have to type the whole thing in again!

What happens if the dog trips over the mains lead just after you've spent an hour typing in your brand new program? I can assure you it will happen – and the time that it happens will be the time when you have thought "Oh, I won't bother for now".

Also, what happens if the first time you run the program, it goes wrong, and you can't get back into it? The only thing to do is load it again – but if you haven't saved it beforehand, then you're in trouble.

ALWAYS **SAVE** A NEW PROGRAM **BEFORE** YOU RUN IT.

6.2 SETTING UP THE CASSETTE

Let's consider the connections we need.

Supplied with your new ZX81 were two leads – the double lead is the one that connects to your cassette recorder, and you probably have one of the leads plugged into the "ear" socket already.

Connect the other half of the double lead into the "mic" socket of the recorder, and into the socket marked "MIC" on the ZX81.

Now find yourself a blank cassette.

Wind it on so that the start of the proper tape material is showing (just past the leader tape), and insert it into the cassette recorder.

Type in this small program:

```
10 FOR Y=0 TO 21
20 PRINT AT Y,10;"WELL DONE"
30 NEXT Y
```

What we're about to do is to put this program onto your tape and try to get it back in again. The reason for using a small program like this is so that you don't have a lot of typing to do if it fails!

The command to get the ZX81 to put the program out is

SAVE "program name" (don't do it yet!)

where "program name" can be anything you like to call the program by.

When you come to **LOAD** the program again, this is the name that you will have to use.

We're going to call our program "TESTER" for now. Type the following, but **DON'T** type the NEWLINE just yet . . .

SAVE "TESTER"

Now start the cassette off as if you were recording something through the microphone. Once the cassette has started, press NEWLINE.

Then sit and watch. You should see something like this:

The screen initially goes blank for five seconds, and then it goes berserk for a period of time. This period depends on how big your program is. If you are not using a 16K RAM pack, then it will not be any longer than 30 seconds. With a 16K RAM pack attached, it can be anything up to 7 or 8 minutes, but this would only be for an extremely large program! Our TESTER program will only take about ten seconds.

Eventually, the screen shows our old faithful message:

Ø/Ø

which means that the program is on the tape. Leave the recorder running for a further 5 to 10 seconds before you switch it off. There is a very good reason for this, which we'll see later on.

Time for the fun. At this point, we have our only copy of the program inside the ZX81, and we don't yet know if the tape is recorded properly.

Question

How can we check that the tape is OK without losing the only copy of the program that we have got – the one inside the ZX81?

Answer

We can't.

This might sound a bit harsh, but the only foolproof way of checking the tape is to try to load the program in the way that we've always done so far. The only trouble is that loading a program from tape destroys whatever is inside the ZX81, so if the tape hasn't been recorded properly, you lose the program completely.

Now you can see why the program we've experimented on is only a few lines long!

6.3 GUIDELINES

So how can we minimise the chances?

We can reduce the liklihood of losing a program to almost nil . . .

if you follow these guidelines.

1. Always, *always,* check that the leads are connected to the correct sockets before you start. The ZX81 cannot tell you that you've pushed them into the wrong holes, so you've got to check them. If the leads are not connected, you won't be able to tell by looking at the screen – the ZX81 carries on quite happily.

2. Make sure that you're using decent tapes. It's no good using these "cheapies" – they aren't man enough (person enough?) to take the signal required by the ZX81. You can buy special computer tapes from computer shops (or mail order – see the monthly computer magazines) for not much more than ordinary tapes, although ordinary low-noise tapes will surfice. In practice, they work out cheaper, since you only buy C12 tapes instead of C60. We'll see why this is better in another section.

3. Make sure the tape is positioned correctly. If you're starting a new tape, the oxide tape should be in view – you might start recording over the leader tape, and that means the ZX81 won't be able to recognise the program when you come to load it again.

4. Leave the cassette running for 5 to 10 seconds after the ZX81 has finished.

5. If this is the first time that you're saving a program, try these two tips:
   (i) save the program twice. If your tape has some oxide drop-out in the early few inches, then at least the second copy should be OK.
   (ii) when you've saved it twice, listen to the tape through the loudspeaker (don't forget to adjust the volume before and after). You should hear five seconds of silence (as you pressed the NEWLINE key), followed by a sound which has earned the nickname "a supercharged bumblebee" (as mentioned in the original ZX80 manual). This is an excellent description of the noise you should hear. When this subsides, you'll hear 5 to 10 seconds of a light "buzzing" noise, which is where the recorder was left running for a bit. If you hear this, then you're probably going to be all right. If not, then check back over the leads and try again.

6.4 CHECKING THE TAPE

Adjust the volume and tone controls ready for loading, rewind the tape back to the beginning, and try to load your program in again – just like we have always done:

LOAD "TESTER"

Do you get the 0/0 at the foot of the screen? Don't be too impatient – remember how long it took to save the program, as it will take just as long to load it back in again.

If you're careful, you can watch the screen while the program is loading, and recognise the periods of 5 seconds silence, and the time while the "supercharged bumblebee" is being played. The actual screen varies from one TV to another, so I can't help you much more than to say that if you watch, you'll learn to recognise your own TV's picture while loading programs.

If nothing happens after a couple of minutes, then something has obviously gone wrong. Press BREAK to stop the ZX81. Check all the leads, then try the whole thing over again. If you *still* have trouble

(and this would be fairly rare by now) then consult the section at the end of this book entitled "Common Problems and Solutions".

Run the program to check that it works – it should print "WELL DONE" down the middle of the screen.

6.5 LOADING AN "UNKNOWN" PROGRAM

So what happens if you can't remember the name of a particular program?

All is not lost – the ZX81 allows you to load a program *without* specifying a name, and when you do this, the ZX81 loads the *first* program that it comes to on the tape.

For example, you can type:

LOAD ""

to load the program that you saved earlier.

If the program is in the middle of a program library tape (you'll see what this is in the next section), then it's not quite so easy. First you must get to the beginning of the program, and the best way of doing this is to listen to the tape through the loudspeaker.

Each program has a gap of five seconds silence in front of it (section 6.3 covered the sound of a tape), and so if you listen to the tape, counting each gap as it comes along, you can fairly easily reach the beginning of any program on the tape.

Stop the recorder once the gap has started before the program you wish to load, then type

LOAD ""

and carry on loading the program as normal.

6.6 PROGRAM LIBRARIES

So what else can we do with the cassette?

First of all you can create a *program library*. This is a collection of tapes, each tape containing several programs – just like the set of tapes supplied with this course.

How do several programs come to be on the same tape, then?

Here's how to do it.

Whenever you are about to write a new program, always use a fresh, blank tape (or keep one tape aside specially for temporary storage of new programs).

Make sure every tape has a list of the programs on each side *in the correct order*.

Save your new program on the temporary tape and test it to make sure it works properly. Don't put it onto the end of a long tape if you haven't checked it out thoroughly!

Now take the tape that you wish to keep the program on, and load the *last* program on the tape.

Stop the recorder when 5 to 10 seconds have passed after the last program has loaded and given 0/0 at the foot of the screen.

Rewind the tape slightly so that the "light buzzing" noise can be heard. Now put this tape to one side.

Load the new program from the temporary tape, and save it again onto the end of the master tape.

Rewind the master tape and it's all done.

Don't forget to write down either on the tape, or the index card, what you have called the program. If you forget the name, then refer to the details given in section 6.2.

When you keep several programs on one tape, try to limit yourself to a maximum of five programs per side – it can be extremely frustrating to wait for the ZX81 to hunt all the way down a C90 tape! For the same reason, it is better to use short tapes (like special C20 tapes) so that the time to wind from one end to the other doesn't seem to drag quite so much.

6.7 SAVING VARIABLES

It is perfectly possible to save a program with all the variables intact.
Let's take an example.

LOAD "EXAMPLE"

List the program. Lines 100 to 140 set some variables to certain values, and lines 200 to 240 print them for you to check.
Now run the program. Write the results down on a piece of paper so that you can check them later on.
List the program again, and now delete lines 100,110,120,130 and 140.

Question
How do you delete program lines?

A = 0
B = 27
C = 32775
D = 5.88
E = 204.124

Answer

By typing the line number followed by NEWLINE.

This was covered in Chapter 2, section 2.3 "Program Editing".

Set your recorder up with a spare cassette, and **SAVE** the program (you can choose any name you like).

The variables have all been saved with the program, but just to make sure I have nothing up my sleeve . . . switch the ZX81 off and on. This removes everything.

Load your program back in again, BUT DO NOT RUN IT JUST YET.

One effect of the **RUN** command is that all variables are cleared out before the program starts.

Since we want to keep the variables, we obviously cannot use **RUN**.

We must use **GOTO** instead. This will allow us to start the program running, but we know from previous programs that **GOTO** doesn't remove all the variables!

Type:

GOTO 1

Since line number 1 is the lowest line number possible, this will always have the effect of starting the program from the beginning.

What do you get? All the variables *still* print out their original values, even though there are no **LET** statements in the program!

So if you write a program that needs to set up a large number of variables (especially later on when we deal with "arrays" in more detail), you can remove the **LET** statements when the program has set them up (by running), and save the program with preset variables.

By saving a program with variables preset, it means that you can now add extra lines in the program to replace the **LET** statements that have been deleted – this means a larger program than you might otherwise have been able to write.

There is one other command to watch out for under these conditions:

CLEAR (on the X key)

This command clears any variables that exist at the moment, and is often used in a program which allows you to start again (the "PRIMES" program could have used **CLEAR**, although it wasn't really necessary).

6.8 "LOAD-AND-GO PROGRAMS

One final feature before we leave the cassette – you can make a program start running automatically when it is loaded from tape. If you include a **SAVE** statement within the program itself, when the ZX81 meets this, it saves the program (in the usual way) then carries on normally with the next line number. When this program is re-loaded, however, it will also carry on at this point. Look at this:

```
10 PRINT "A LINE OF TEXT"
20 STOP
30 SAVE "SOMETHING"
40 PRINT "HERE IS";
50 GOTO 10
```

Start the program by entering **RUN** 30 (note the line number after the command **RUN** – it tells the ZX81 to start at line 30 instead of at the beginning of the program). Before you do, though, start the cassette recorder off for saving a program. After the **SAVE** command on line 30 has finished, the program carries on as normal.

Now load the program back in again by entering

LOAD "SOMETHING"

As soon as the program is loaded, it starts in exactly the same way! This is extremely useful when you want to save a program with some preset variables – it prevents you from typing **RUN** by mistake!

Summary

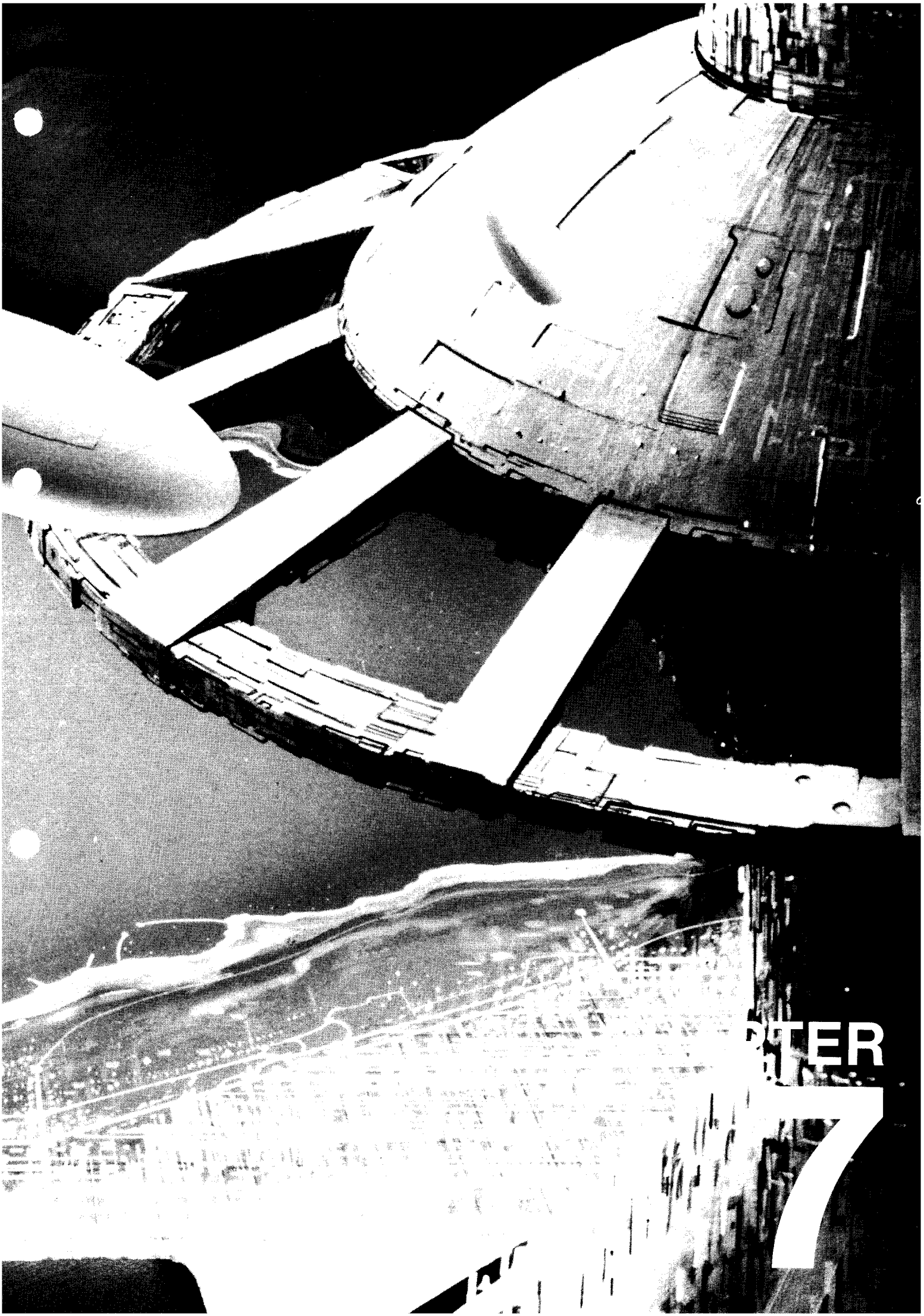
In the next chapter we'll meet *subroutines,* a more professional way of writing your programs. But here's what we've covered in this chapter:

– how to prepare the recorder for saving programs
– why and when you should save programs
– how you can reduce the likelihood of trouble to a minimum
– how to create a program library
– how variables can be saved on tape along with a program
– that programs can be made to start automatically when they are loaded.

There are no exercises for this chapter.

RUN", CLEAR VARIABLES (CLEAR ALSO.)

GOTO 1 RUNS PROGRAM BUT DOES NOT CLEAR VARIABLES.


PTER
7

Subroutines

What a peculiar title for a chapter! Subroutines?!

Nearly all of this chapter is devoted to this topic, although two smaller extra sections are included towards the end. As with chapter 4, one program is followed throughout the chapter, gradually building it up into a useful visual aid for representing graphs. Section 2 shows how the ZX81 can be used to draw graphs, while section 3 gives you some useful information about the **INPUT** command.

7.1 MAKING USE OF SUBROUTINES

7.1/1 DEFINITION OF A SUBROUTINE

We'll kick off this chapter with a program that plots a small graph, using X and Y coordinates entered from the keyboard. The program is quite large – and would be almost impossible to write for a basic ZX81 without the use of *subroutines*.

The program draws horizontal and vertical axes, both 20 units in length. It then asks for an X and Y coordinate to be entered. These are checked to make sure that they are between 0 and 19, and do not have any fractions (or decimal places). This point is then plotted on the graph. This continues until the word QUIT is typed.

So what exactly is a subroutine? Here's the definition:

It is a section of a program that can be used many times from different parts of the program.

That sounds confusing. Let's go a bit deeper.

7.1/2 LOOKING AT THE PROBLEM

To start with, we'll have a look at this graph-drawing program.

LOAD "GRAPH"

but don't look at the program just yet. Run it first.

You'll see this:

The program draws the two axes, and then invites you to enter an X coordinate. The number you enter can be any number from 0 to 20, but without decimal fractions (e.g. 10.33 would not be allowed). After pressing NEWLINE, the program asks for a Y coordinate – again, you can enter any number between 0 and 20. This point will then be plotted, and the program will ask for more coordinates.

A couple of points to watch out for:

(i) Any points plotted with a coordinate of zero will not be visible, because they lie on the axis.

(ii) If you enter an invalid number, the program will ask you to enter the number again.

(iii) When you've finished, type QUIT, and you'll be given a report code which stops the program.

You're going to write this program bit-by-bit, and we'll explain all of the concepts used in it as we go along. If you're stuck, then you can look at *my* solution – just list the program! Don't do that until you've had a good try.

Ready?

7.1/3 A FIRST ATTEMPT

Let's start with entering the coordinates.

Question

Write a small program that asks for two numbers from the keyboard, and checks them to be between 0 and 19 (inclusive) and that they have no decimal fractions.

Don't worry about what line numbers you use at the moment, because this program probably won't last very long!

The first number is called "X" and the second number is called "Y", so a sample run would be:

```
X? 5
Y? 27
27?
Y? 3

9/9999
```

Answer

You should have had no real problems writing this, as you've done a very similar program before! My solution is:

```
10 PRINT "X? ";
20 INPUT X
30 PRINT X
40 IF X>=0 AND X<=19 AND INT X=X THEN GOTO 100
50 CLS
60 PRINT X;"?"
70 GOTO 10
100 PRINT "Y? ";
110 INPUT Y
120 PRINT Y
130 IF Y>=0 AND Y<=19 AND INT Y=Y THEN GOTO 200
140 CLS
150 PRINT Y;"?"
160 GOTO 100
200 STOP
```

What do you notice about the solution? For now, I'll use my solution to show you what's going on.

To a certain extent, the program does the same thing twice – it prints a message, gets a number, checks it to be valid and if it is *not* valid, then prints another message and asks for another number.

This is done twice – once for the "X" number, and once for the "Y" number.

We could obviously cut the size of our program down if the routine only needed to be written once – the capacity of the ZX81 is not infinite!

Question

Can you write another version of the same program that only duplicates two or three lines, instead of 6?

The trick obviously involves some sort of devious **GOTO**, but what?

Don't spend more than 10 minutes trying. Give up and look at the answer if it's baffling you.

Answer

You very probably didn't succeed at that one! There is a method that would work using the idea of holding a line number in a variable, then saying **GOTO** V or something. But it's messy.

7.1/4 GOSUB AND RETURN

The answer is a new command

GOSUB n (on the H key)

which works in a similar way to **GOTO**, but has one very important difference.

GOSUB needs a line number that tells it where the SUBroutine is located so, for example, we could say:

```
20 GOSUB 560
```

or

```
60 GOSUB LUCKY*7
```

and the ZX81 would immediately start running program lines at the new line number – just like a **GOTO** which we met in Chapter 3.

Here's the difference. Whenever the ZX81 sees a **GOSUB** command, it *remembers* the line number that it is on now, so if it saw

```
20 GOSUB 560
```

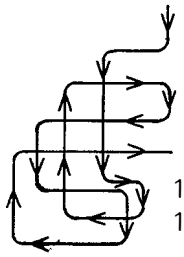
it would tuck the line number 20 away for safe keeping. Then it jumps to line 560 and carries on in the usual way.

So what does it want to save line number 20 for? This is the beauty – as soon as the ZX81 sees a line containing the command . . .

RETURN (on the Y key)

it recalls the line number that it had put aside, and then *automatically* goes back to the line number immediately following, and carries on with whatever it was doing there.

Let's take a very small example of this, and look at it in close detail.



```
10 LET NUMBER=10
20 GOSUB 100
30 LET NUMBER=33.4
40 GOSUB 100
50 STOP
100 PRINT "ANSWER IS ";NUMBER*27.5
110 RETURN
```

The arrows indicate where the ZX81 jumps to each time it comes to a **GOSUB** and a **RETURN**.

Line number 10 merely assigns the value 10 to variable NUMBER.

Now. Line 20 causes the ZX81 to do two things. First, the ZX81 puts the number 20 to one side and secondly, it goes to line 100.

Line 100 prints "ANSWER IS 275", and then the ZX81 sees line 110.

"Ah!" says the ZX81, "here's a RETURN command. I'll fetch that number that I put away – now where is it? – Oh, here it is! It was line number 20. I'll go back to that line again."

So the ZX81 comes back to line 20, and carries on with the next line number, which is line number 30.

Line 30 sets the variable NUMBER to the value 33.4, and line 40 once again says **GOSUB** 100.

By a similar process, the ZX81 puts number 40 to one side, and jumps to obey the statement at line 100, which is to print another answer using the variable NUMBER.

Line 110, again, recalls the number that was put aside, but in *this* case the number is 40, and so the ZX81 comes back to line 40, drops through (since it has already run line 40), and comes to a **STOP** command on line 50.

7.1/5 WHY USE SUBROUTINES?

What has this program achieved?

It's saved writing line 100 twice which, apart from the obvious benefit of avoiding duplicate lines of program, has other real advantages.

This program was really too small to gather the full implication of **GOSUB/RETURN,** but larger programs really gain a lot from the use of them.

Here are some reasons why the use of subroutines is recommended:

1. Smaller programs, because duplicate sections of programs can be made common in a subroutine (those lines 100 and 110 in the last example are called a subroutine).

2. Easier to test, since the subroutine only needs to be tested once, and then it works wherever it is used.

3. Easier to design a program in the first place, as chunks of program which are related can be put into a subroutine (both mentally and physically), and referred to by just a line number.

4. Easier to understand when you try to look at the program at a later date. It doesn't take long for you to forget your train of thought, and using subroutines to break your programs up makes it easier to get back into the swing again when you want to make some alterations.

Question

Re-write the following program to use a subroutine.

```
 10 PRINT "ENTER A NUMBER";
 20 INPUT N
 30 PRINT N
 40 LET S=N
 50 PRINT "ENTER A NUMBER";
 60 INPUT N
 70 PRINT N
 80 IF S=0 AND N=0 THEN STOP
 90 PRINT S;" TIMES ";N;" IS ";S*N
100 GOTO 10
```

Answer

```
 10 GOSUB 200
 40 LET S=N
 50 GOSUB 200
 80 IF S=0 AND N=0 THEN STOP
 90 PRINT S;" TIMES ";N;" IS ";S*N
100 GOTO 10
200 PRINT "ENTER A NUMBER";
210 INPUT N
220 PRINT N
230 RETURN
```

If you had trouble, then read sections 7.1/3 and 7.1/4 over again, and study carefully through the examples. Also try running some of these small programs just to see what they do.

7.1/6 SUBROUTINE REPLIES

Now you can probably write the original program you were asked to write (take two numbers, etc see section 7.1/3) in a much more efficient manner, although there is a slight difference.

We wanted a different message printed out to indicate whether the "X" or "Y" number was to be entered. What happens if the number entered is invalid? Presumably, all this would be inside the subroutine since, if you look back to your solution, the lines of program that check the value of "X" and "Y" are duplicate lines.

The answer is quite simple.

The subroutine needs to set a variable that can be tested *after* the subroutine has been finished. This variable would tell if the number that was entered from the keyboard was valid or not.

7.1/7 NESTED SUBROUTINES

An important feature of subroutines is that one subroutine can call another – so our subroutine above could include another **GOSUB** to a different (or even the same) subroutine. The ZX81 keeps track of the line number, just the same, and will always come back afterwards. The limit of this is (almost) endless – I'm sure you'll run out of steam before the ZX81 does! Here's an example of *nested subroutines* (this is what they're called):

```
 10 LET P=1                    set a marker for first time
 20 GOSUB 100                  print first half of message
 30 LET P=2                    now second time through
 40 GOSUB 100                  print second part of message
 50 STOP                       . . . that's all

100 PRINT "TO BE";             guess what?
110 IF P=1 THEN GOSUB 200      if first time, print "OR NOT"
120 RETURN                     end of subroutine

200 PRINT "OR NOT ";           middle section
210 RETURN                     end of second subroutine
```

Type this in and run it – you'll get a very good insight into what happens when one subroutine calls another, and how they never get lost.

Question

Try writing the first question again (from section 7.1/2). Start your program at line number 200, because future questions will need to add some more lines into this program.

Refer back to section 7.1/2 to see what was required.

Answer

My model answer looks like:

```
200 PRINT "X? ";
210 GOSUB 1000                          get X from keyboard
220 IF NOT V THEN GOTO 200              see if it was valid, and if
                                        not, then go back again
230 PRINT "Y? ";
240 GOSUB 1000                          now get Y
250 IF NOT V THEN GOTO 230              if invalid, go back again

1000 INPUT N                            ask for a number
1010 PRINT N                            print it back
1020 LET V=1                            this sets V "true"
1030 IF N<0 OR N>19 OR INT N<>N THEN LET V=0
                                        this sets V "false if any
                                        error is found in N
1040 IF V THEN RETURN                   go back if all OK
1050 PRINT N;"?"                        print invalid number
1060 RETURN                             go back again
```

In the subroutine at line 1000, I use a variable called V to indicate if the number from the keyboard is valid or not.

Line 1020 initially sets V to "true", and line 1030 sets V to false if the number entered is invalid.

When the ZX81 has finished the subroutine (i.e. it comes to a **RETURN** command), the variable V is tested (lines 220 and 250) to see if another number needs to be asked for. If V is false, then another number must be entered.

I don't expect your answer looked anything like mine! But the important point is, did your answer work? If it did, then carry on with the next section.

Let's try to find out what went wrong. First of all, work your way through my solution to see what it is that you cannot understand. Then look down the list of references that follows, and read the appropriate section(s) again.

– Can't understand **GOSUB** and **RETURN**. Try reading this section once more.

– You can't understand these complicated **IF** statements. This was largely covered in Chapter 3, section 3.2 "Conditional Expressions", although there are further references in section 5.1 "Conditional Expression Values" which are well worth following up.

– You don't understand this business of "true" and "false". Again, this is really Chapter 3, section 3.2 (see above), but section 5.1 will be even more relevant to you, because this deals with conditional expression values in great detail. Try again.

7.1/8 SOME SLIGHT IMPROVEMENTS

By now, you're becoming quite a master of programming! For those that are interested, there is a way of cutting down the size of the previous answer even more. Section 5.1 covered the use of conditional expressions in **LET** statements, and if we make use of this, we can make the subroutine even smaller.

I am not going to explain the logic behind this in detail; if you are interested, then it should be quite straightforward, especially if you look back to section 5.1 – "Conditional Expression Values". I will only write the subroutine again:

```
1000 INPUT N
1010 PRINT N
1020 LET V=(N>=0 AND N<=19 AND INT N=N)
1030 IF NOT V THEN PRINT N;"?"
1040 RETURN
```

That's another two lines shorter, and does exactly the same thing!

7.2 GRAPH PLOTTING

7.2/1 HOW TO START

Now what? We've seen how subroutines can help make our programs small, neat and tidy, so we can go on to create the rest of the original graph-drawing program.

Before we can do this, we need to learn another pair of new commands. These commands allow us to turn the display into a sort of graph paper, and plot individual squares on the graph paper. The two commands are:

PLOT (on the Q key)

and

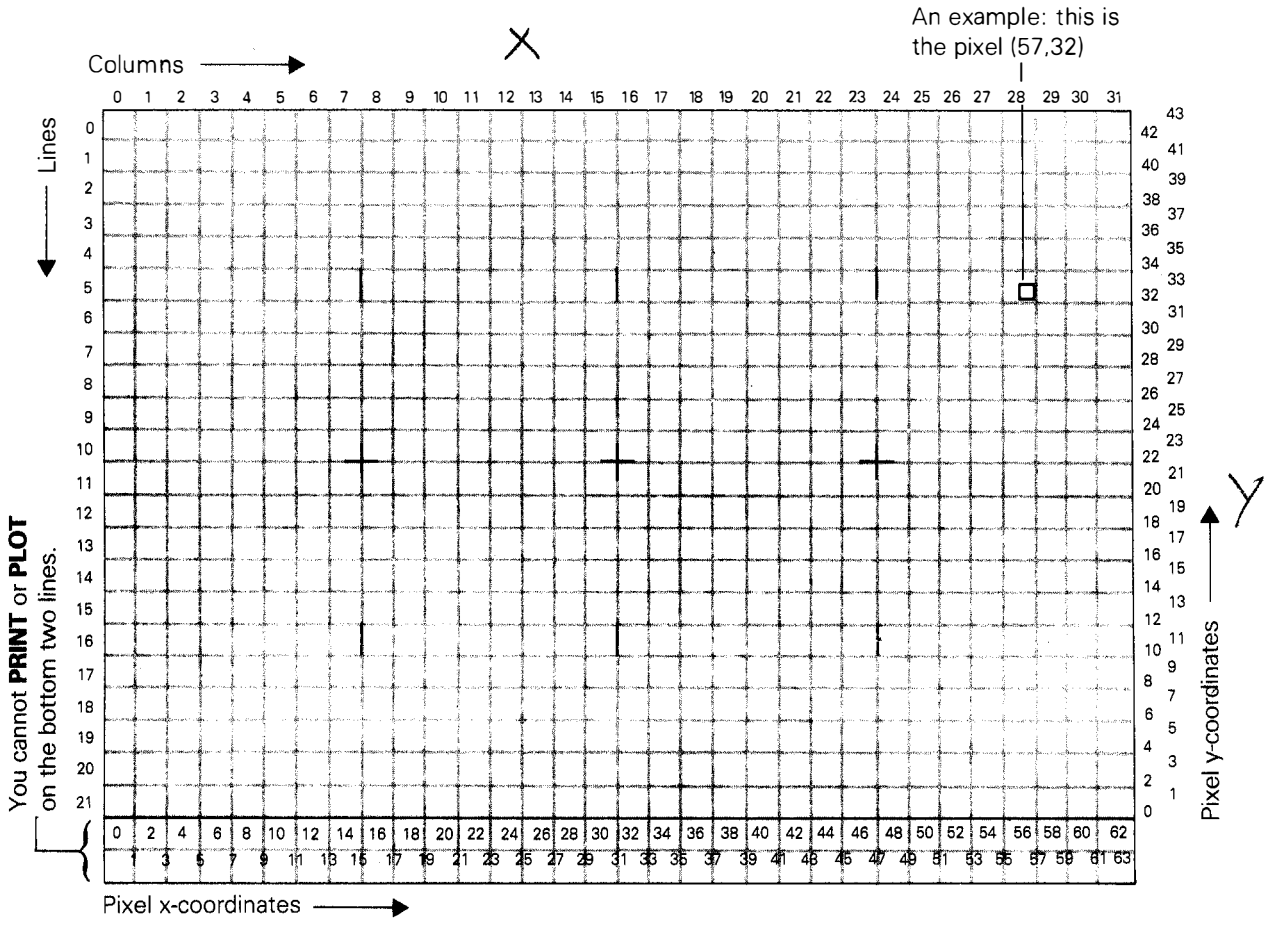
UNPLOT (on the W key)

Both commands need two extra parameters to get them to work. They need to be given an X-coordinate, and a Y-coordinate, like this:

PLOT x,y **UNPLOT** x,y

where x and y can be any numeric expression.

Let's picture the screen as graph paper like this:



The imaginary sheet has 64 positions across (in the X or horizontal direction) and 44 positions up (in the Y or vertical direction).

These positions are numbered from 0 to 63 in the X direction (the X axis), and 0 to 43 in the Y direction (the Y axis). Look at the diagram carefully.

Each of these squares is called a *pixel*.

Notice the difference between **PLOT** and **PRINT AT**. In **PLOT**, the column is given first, and the line numbers start at 43 for the top of the screen coming down to 0 at the bottom. In **PRINT AT**, the line number is first and the top line of the screen is line number 0 coming down to line 21.

We can black out any of these squares by using **PLOT**, giving the X and Y coordinate of the square. Let's try a couple. Type in this small program:

```
10 INPUT X
20 INPUT Y
30 PLOT X,Y
40 GOTO 10
```

Run the program. You won't get any nice prompt messages, because you're about to use the whole screen anyway.

Type in your X-coordinate (a number between 0 and 63 inclusive), followed by NEWLINE.

Then enter your Y-coordinate, a number between 0 and 43 inclusive. As soon as you press **NEWLINE**, that square will be "blacked out", and the program will wait for you to type in another pair of coordinates.

Don't get too eager making up pretty pictures, because the program does not cater for you making any mistakes! Try entering a number *outside* the range given above. You'll get an error B/30 after the Y value has been entered. This means that you have attempted to **PLOT** something that the ZX81 cannot manage.

UNPLOT works in exactly the same way, except that it changes the squares back to white again, so you can rub out a point in a program when it is no longer needed, but keep all the rest of the picture as it was.

Question

Which of these commands are valid, and which are invalid?

(a) **PLOT** 10,20

(b) **UNPLOT** 10*20,20

(c) **PLOT** 43,63

(d) **UNPLOT** 0,0

(e) **PLOT** 100/2,27+3

Answer

(a) Valid

(b) Invalid – 200 is outside the limit of X-coordinates

(c) Invalid – the X-coordinate comes first, Y-coordinate second, and this example would only work if the two numbers were reversed.

(d) Valid

(e) Valid

How did you get on? If your answers were wrong, try entering those values into the small demonstration program that was given to you to try. If you get a report code B, then you know that the values were wrong. Answers (b) and (c) will give you a report code B.

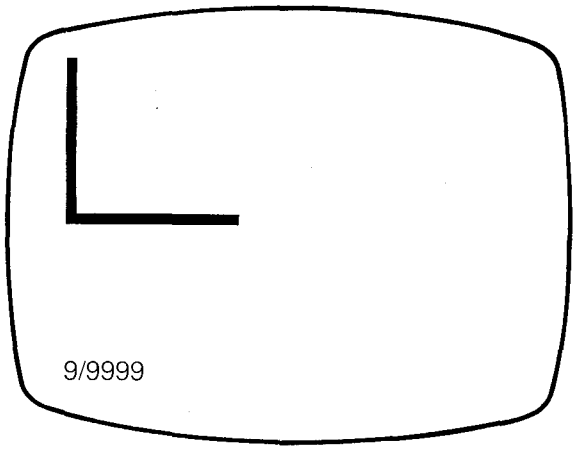
7.2/1 PLOTTING IN PRACTICE

So now you should be able to write a small program that draws two lines on the screen – one down, and one across. Have a go at this:

Question

Write a program that draws two lines on the screen. The vertical line should be on the left-hand edge of the screen, starting at the top of the screen, and should be 20 squares long. The horizontal line should start at the foot of the vertical line and stretch across the screen for 20 squares.

Your screen should look something like this when you run the program:



(hint: you'll need to think **FOR** a minute before you turn to the **NEXT** page!)

Answer

Sorry about that rather un-subtle clue! Here's my solution:

```
10 FOR Y=24 TO 43  }
20 PLOT 0,Y        } draw vertical line
30 NEXT Y          }
40 FOR X=0 TO 19   }
50 PLOT X,24       } draw horizontal line
60 NEXT X          }
```

Well? That wasn't too bad, was it? If you thought it *was* bad, then you should read over Chapter 4, section 4.2 "Iteration (2)" again – it covered the **FOR** and **NEXT** commands.

Perhaps you had forgotten that **FOR** loops can start with a number like 24 – you can start with any numeric expression whatsoever – refer to Chapter 4, section 4.2 if you need reminding.

It's also worth remembering that **FOR** loops are *inclusive,* that means that the loop is run for all values including the start and finish values of the control variable.

If you didn't quite grasp the **PLOT** and **UNPLOT** commands, then hopefully that solution has helped you – even so, read section 7.2/1 again.

7.2/3 THE FULL PROGRAM

We've managed to create a routine that draws a pair of axes on the screen (length 20 by 20), and to write a routine that takes in numbers from the keyboard and checks that they are in the range 0 to 19. Not much further to go then.

We have finally to put these two routines together, add a few bits and pieces just to smarten up the presentation of the screen, and that'll complete the program.

If you've overwritten the "GRAPH" program, then load it again and look at the listing.

Lines 10 to 60 are taken straight from the example above.

Lines 100 to 140 are new – these lines clear out three lines on the screen below our graph axes, as these are going to be used for asking for input, and printing any error messages.

Lines 200–260 are almost the same as an earlier solution – the only exception to this is the use of **PRINT AT** to print the messages. The reason for this change is because we want the graph to remain on the screen all the time, so we cannot use **CLS,** or we would lose the graph so far. So I have decided to keep all my messages on lines 13 to 15, and these three lines are cleared out (lines 100–140) after each pair of coordinates is entered.

Line 300 merely **PLOTS** the appropriate point on the graph – variable S holds the X coordinate, and the Y coordinate has 24 added to it since the graph is at the top of the screen.

7.3 INPUT EXPRESSIONS

But there is one last point that was mentioned in the original definition of the program (section 7.1). If we want to stop the program, then type QUIT.

Surely this can't work? Try it to see.

It *does* work, it gives rise to error report 2, which means that a variable name is not found.

So which variable is not found? It must be a variable called QUIT.

And indeed it is.

You've probably got quite sick of me saying the same things over again, but this time it's quite important – whenever you enter something into an **INPUT** statement, then you may enter

any valid numeric expression.

That might take a while to sink in fully, so let's try an experiment to see what it can do. Type this program in:

```
10 LET A=1
20 LET B=200
30 LET C=5
40 SCROLL
50 PRINT "ENTER AN EXPRESSION"
60 INPUT E
70 SCROLL
80 PRINT "YOU ENTERED ";E
90 GOTO 40
```

Now run it. When it asks you to enter an expression, try entering these:

(a)	25	ordinary number
(b)	A	variable name A equals 1
(c)	9*B	should show 1800
(d)	B/C	gives 40
(e)	A=C	true/false expression
(f)	A<>C	another true/false expression
(g)	QUIT	variable name

Can you see why QUIT gives error report 2?

Think of the possibilities – whenever a program asks you to enter a number that involves calculation, you don't even need to do it! Just type the whole expression in, and let the ZX81 do the rest!

Question

In Chapter 3, section 3.1 "Iteration (1)", you were introduced to several ways of getting the ZX81 to stop running a program. These were:

(a) pressing the BREAK key (but not if the ZX81 is waiting for input) which gives report code D.

(b) typing the command **STOP** if the ZX81 is waiting for input. This also gives report code D.

Can you now add another method to the list? Note that this method is not as "clean" as the others.

Answer

Typing in an undefined variable name. This gives error report 2.

Most programs will never use a variable name like XQYTGGZX, so you would be fairly safe (I'll bet *someone* does, just to spite me!) if you entered a name like that into an **INPUT** command.

If your answer was wrong, or you couldn't answer the question, try reading section 7.3 again.

Before you finish this chapter, here's another program for you to write:

Question

The program is to plot a 20-by-20 letter "X" in the centre of the screen, then immediately "unplot" it. The program should keep running (plotting and unplotting the letter "X") until the BREAK key is pressed. You will need to first decide the coordinates of the position of the letter.

Try to make use of subroutines and **FOR NEXT** loops.

Answer

One of many solutions is:

```
10 REM DRAW X CONTINUOUSLY          describe program
20 CLS                              ready for running
30 LET DRAW=1                       tells us we're plotting
40 GOSUB 100                        subroutine to plot X
50 LET DRAW=0                       unplot marker
60 GOSUB 100                        now unplot the X
70 GOTO 20                          start again
100 REM DRAW OR UNDRAW X
110 LET DIRECTION=1                 horizontal direction +
120 LET X=22                        start position on screen
130 GOSUB 200                       draw one line
140 LET DIRECTION=-1                reverse horiz. direction
150 LET X=41                        start position
160 GOSUB 200                       draw other line
170 RETURN                          end of subroutine
200 REM DRAW OR UNDRAW 1 LINE
210 FOR Y=31 TO 12 STEP -1          for 20 lines down screen
220 IF DRAW THEN PLOT X,Y           plot if required . . .
230 IF NOT DRAW THEN UNPLOT X,Y     . . . otherwise unplot
240 LET X=X+DIRECTION               next horizontal position
250 NEXT Y                          next vertical position
260 RETURN                          end of subroutine
```

I hope that has given you a good example of many of the topics we've covered up until now – the use of subroutines (nested), **FOR NEXT** loops, expressions, graphics – you name it!

A program like this is much easier to write when you've thought it out properly in advance. There is a big temptation to sit down and compose directly into the ZX81 – resist this temptation.

Summary

This chapter has been a fairly mixed bag of items – we've seen how subroutines make programs small and efficient, how to plot graphic-type pictures, and also learnt something new about the **INPUT** command.

The basis for the whole chapter was to let you gradually build up the best part of a complete program, learning all these ideas on the way.

The next chapter takes us onto something that we've skimmed over up until now – the use of strings (or text) in programs.

Meanwhile, have a browse over this list to make sure you're happy with what this chapter has told you.

– what a subroutine is, and how it can be used.

– the benefits to be gained by using subroutines – this is called *modular programming*.

– how subroutines can set variables to give a reply to the main body of the program.

– that subroutines can be nested to maximise their efficiency.

– how to use the graph-plotting facilities of the ZX81.

– how any valid expression can be entered in as input data.

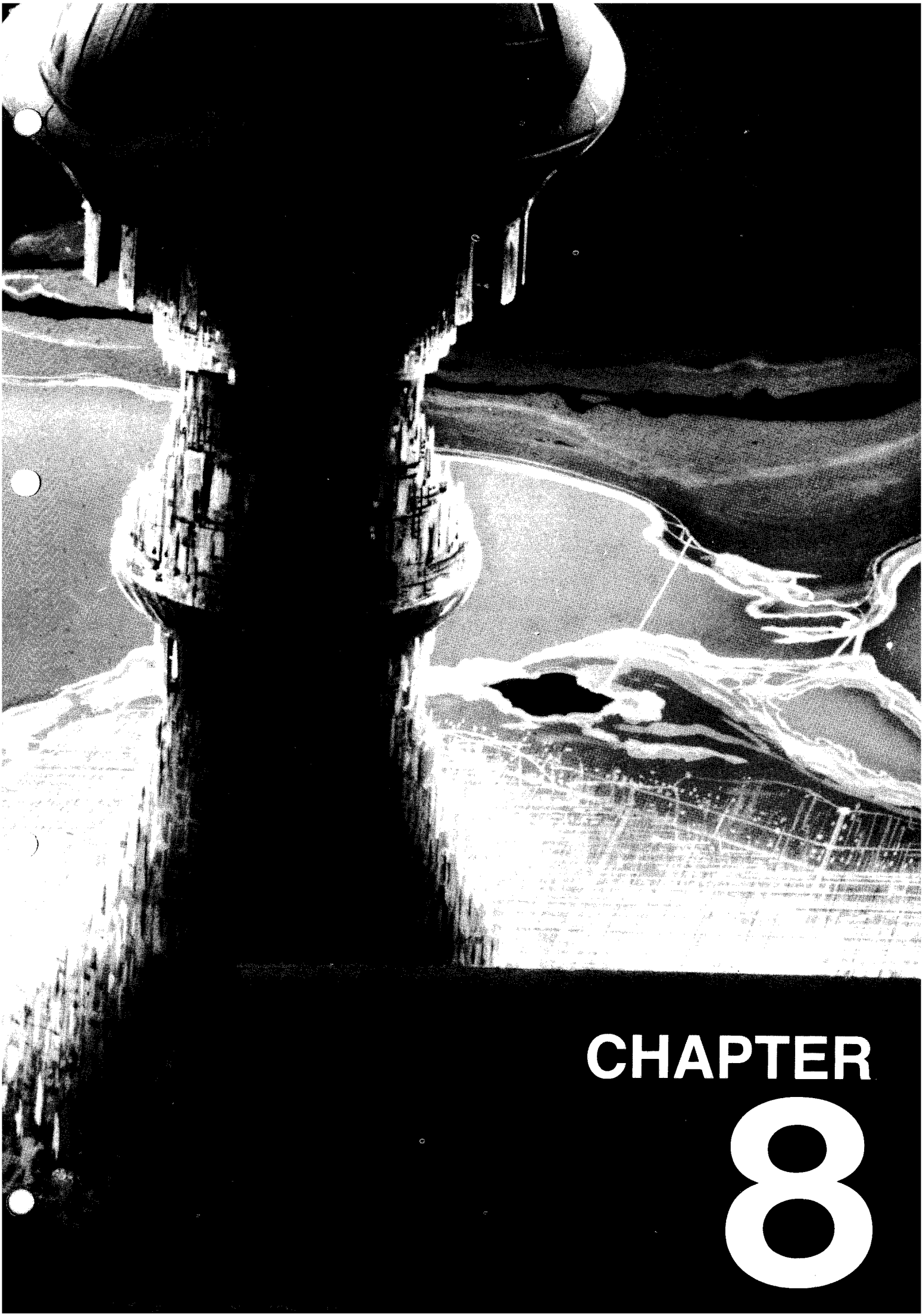
Exercises

1. Write a program that calculates the scale of the coordinates of a graph. The program should accept the data points, calculate the difference between maximum and minimum points and from this, deduce the limit of one pixel.

   If you have a 16K RAM pack, you may like additionally to plot the coordinates with appropriate scales alongside.

2. Take any paragraph from this course, and count the number of times each letter occurs. Plot a bar chart of the frequency of occurrence of each letter,

   If you are feeling up to it, try the same program using paragraphs taken from different languages. Compare the charts – you should discover that each language has a distinct pattern.



CHAPTER 8

Handling Text Strings

Up until now, we've concentrated almost entirely on handling numbers and numeric expressions, although we've seen how some fairly interesting things can be done!

This chapter deals entirely with *strings*. In section 1 we look at how strings can be manipulated, while section 2 studies the way strings are represented within the ZX81.

8.1 STRING MANIPULATION

8.1/1 STRING VARIABLES

The only real use of strings so far has been as part of **PRINT** commands, for example,

```
20 PRINT "HELLO THERE"
```

where the string expression "HELLO THERE" enclosed in quotes is printed directly onto the screen.

Just as the ZX81 can store values in a numeric variable, it can also store strings in *string variables*.

A string variable name can only be a maximum of a single letter (like control variables in **FOR/NEXT** loops), but they are followed by a dollar-sign $ to tell the ZX81 that this variable is for holding text or strings, not numbers.

This sounds quite confusing at first, but take a look at these examples:

```
LET A$="HELLO THERE"
```

Because the variable name A is followed by $, the ZX81 instantly knows that this is a *string variable*. used for holding strings rather than numbers. So it puts the words HELLO THERE into this variable for you to use later on.

```
PRINT A$
```

Now the ZX81 sees that you are trying to print some text, because the variable name is followed by a $ symbol. The text held by variable A$ will be printed on the screen – in this case HELLO THERE.

Of course, variable A$ can hold literally anything, since as we saw in Chapter 5, even those graphic characters and inverse video characters can go in between quotes. Whatever is in quotes is held by A$, and printed out when we say **PRINT** A$.

Type those two lines in to convince yourself that they really work.

We can also use the **INPUT** command to ask for a string from the keyboard just as we have done with numbers up until now. Try this:

```
10 SCROLL
20 PRINT "ENTER SOMETHING"
30 INPUT A$
40 SCROLL
50 PRINT "YOU TYPED:";A$
60 GOTO 10
```

The **SCROLL** commands were put in so that you can keep going for as long as you like without error 5.

The important lines are line 30 – which asks you to input variable A$ – and line 50 which prints it straight back out at you! If you get error 5, then it's because the line you are trying to print out (in line 50)

is too long to fit on the bottom line – **SCROLL** only leaves enough room on the bottom of the screen for one line, so if you type in (say) 50 characters, then the ZX81 can't fit 50 characters on one line!

You'll notice that whenever you use **INPUT** in a program to input a string, that when the program is run, two quotes symbols appear at the bottom of the screen with the L cursor in between. This is actually the ZX81's way of telling you that it is expecting a string rather than a number.

Question

You've seen that a string (as used in a **PRINT** statement between quotes) can contain *any* characters at all on the keyboard. There is one exception to this – can you find it?

Answer
The quotes character (SHIFT/P) cannot be used within a string because it means "the end of the string". If you tried to type a quotes character into that last program, you'll have had a syntax error **S** given to you.

8.1/2 THE QUOTE IMAGE

How can we get round this? Obviously, the quotes character can be quite a useful one. The answer is the funny double-quotes character SHIFT/Q (this is called the *quote image*). If the ZX81 sees one of these inside a string, it automatically prints it out as a single quotes. Try this:

```
PRINT "I AM A "" "ZX81"" " COMPUTER"
```

8.1/3 CONDITIONAL STRING EXPRESSIONS

Now we can store strings, print them and input them. Unless we can actually manipulate them in some way, however, there isn't much point in this.

The ZX81 is equipped with some really useful features for manipulating strings. The rest of this chapter will deal with all of these in detail – even down to understanding how the computer "sees" letters – it's not really the same as you and me.

First, let's study a bit about **IF**.

Just as we can test for two numbers being the same or not, like

```
IF CECIL=7 THEN PRINT "THEY ARE EQUAL"
```

or

```
IF MILK PRICE<>20 THEN PRINT "NO THEY ARE NOT"
```

we can test to see if strings are equal or unequal. Look at this:

```
50 IF A$="NO MORE" THEN GOTO 200
60 IF Q$<>"YES" THEN STOP
```

And so on.

You need to be a bit careful when you're doing this, because the ZX81 checks for an *exact* match. Let's give it a try with another small program:

```
10 INPUT T$
20 IF T$="I GOT IT RIGHT" THEN GOTO 50
30 GOTO 10
50 PRINT "AT LAST"
```

If you try to enter things like

```
I GOT IT RITE
I GOT IT RIGHTER
```

you'll soon see that it's only when you type it *exactly* the same (including the number of spaces between words) that the ZX81 stops.

Question
Write a small program that asks if you would like another game – assuming you had just finished playing one! The question should only accept YES or NO, and any other answer is rejected. Assume, for the purposes of this question, that by writing **GOTO** 1000, the game will start again.

Answer
I expect your answer is quite different from mine, but here goes:

```
10 PRINT "WOULD YOU LIKE ANOTHER GO?"      (nice message)
20 INPUT Q$                                (get your answer)
30 IF Q$="YES" THEN GOTO 1000              (start new game)
40 IF Q$="NO" THEN STOP                    (stop if no)
50 PRINT "ANSWER YES OR NO"                (tick you off)
60 GOTO 10                                 (try again)
```

How did you do?

If you forgot to put a $ symbol after your variable names (which must be a single letter for string variables), then read section 8.1/1 again.

Perhaps you had trouble writing those **IF** statements. Try reading section 8.1/3 again.

8.1/4 CONCATENATION

Another useful trick we can try is to join two strings together making one big string. This is done by using the "+" (addition) symbol. Like so:

```
10 LET G$="ABC"
20 LET H$=G$+"DEF"
30 PRINT H$
```

and line 30 will give the print ABCDEF. Nice, huh?

You can't multiply, divide or subtract, but you can add. It doesn't mean quite the same as you're used to, but when you have a think about what the ZX81 is doing, it *is* consistent. The proper name for this is *concatenation*.

We'll leave this for a minute to consider a bit more about the way the ZX81 is working things out.

8.1/5 STRING EXPRESSIONS

In Chapter 1, you were introduced to the idea of numeric expressions and how they represent a value.

In just the same way, we must now understand the meaning of a *string expression*.

A string expression represents a string, and if printed, will print a string. Here are some examples of string expressions:

```
A$
"ROMEO, ROMEO"
G$+"DEF"
```

In fact, if there is a $ symbol around, or something in quotes, then it must be a string expression. Just as LUCK*3 is a numeric expression, so H$+"DEF" is a string expression, that in our earlier example stood for the string ABCDEF.

Got the idea?

Question
Which of the following are numeric expressions and which are string expressions, and which are neither? Two are also invalid – can you spot them?

(i) G$
(ii) 47/13.33
(iii) 32+"STRING"
(iv) "THEATRE/7"
(v) THEATRE/7
(vi) "THEATRE"/"7"
(vii) **STOP**
(viii) "JOHN"+"WAYNE"
(ix) **THEN**

Answer

(i) string
(ii) numeric
(iii) invalid
(iv) string
(v) numeric (in this case THEATRE is a variable name!)
(vi) invalid
(vii) neither – it's a command
(viii) string
(ix) neither – it's part of the **IF** command

I hope that wasn't too taxing. If your answers were wrong, then you should read Chapter 1 section 1.3 "Variables" and section 8.1/5 over again. Try to get it clear in your mind what these various "expressions" are representing.

8.1/6 STRING FUNCTIONS

Now that you've grasped (hopefully) the idea of a string expression, we can introduce

string functions.

These funny things operate on numeric expressions to produce string results.

In a similar way, the numeric functions operate on numbers to produce a numeric result – for example the **INT** function operates on the following numeric expression to give the **INT**eger value (i.e. all decimal fractions have been removed).

Let's take the

STR$ function.

Notice how the *name* of the function is followed by a $ symbol. This immediately tells you that the result of this function is a string and can only be assigned to a string variable.

The **STR$** function converts the following numeric expression into its equivalent string form.

So

LET N$=**STR$** 45

is the same as writing

LET N$="45"

I know what you're saying – "Why bother using the **STR$** function, when I can just write it the other way?"

Well, just try writing

LET N$=**STR$** (LUCKY*2000.972)

the other way. You'll soon see why **STR$** is worthwhile.

Until we've covered some more ground with strings, the full benefits of the **STR$** function will not be very clear to you. However, you'll at least be aware of what a string function is when you see one.

Question

Although you don't understand what other string functions can do just yet, you should be able to recognise them when you see them.

Which of the following items are string functions, which are numeric functions, and which are neither? You may look at the keyboard if you wish.

(i) **STR$**
(ii) **TAN**
(iii) **INKEY$**
(iv) H$+"DEF"
(v) **VAL**
(vi) **CHR$**

Answer

(i) string function
(ii) numeric function
(iii) string function
(iv) neither – it's a string *expression*
(v) numeric function
(vi) string function

If your answers were wrong, then perhaps you can now see why, although if you had trouble, then try reading sections 8.1/4 to 8.1/6 over again.

8.1/7 SLICING

Now we come to one of the more interesting features of strings.

The ZX81 is such a clever beast that it can look at *individual* letters or characters inside a string. Here's how we go about it.

Whenever we want to look at a certain letter inside a string, we put a *number in brackets* after the variable name (or after the string expression, to be precise). So we can look at the *first* letter in a string by writing:

```
G$(1)
```

or we can look at the fifth letter by writing:

```
H$(5)
```

The variable names are irrelevant – the point of it is the number in brackets that follows the variable name.

Type this in:

```
10 PRINT "ENTER A STRING";
20 INPUT N$
30 PRINT N$
40 PRINT "THE FIRST LETTER IS";N$(1)
50 GOTO 10
```

Now run it and convince yourself that it actually works!

There is only one small point to watch out for. If you press the NEWLINE key without typing anything in between the quotes, you have effectively not got a first letter. The ZX81 realises that there is no first letter, and so gives an error 3. If you look up error 3 at the end of the book, it gives a really complicated description of something. Don't worry for now, you'll understand a bit more after the next chapter.

What makes this so nice is that the number in brackets can be – you've guessed it – any numeric expression. This means we can pick out *any letter we like*. Think about it for a moment. There is only one disadvantage – if we get error 3 whenever we try to get a letter that's not there, and we don't know how long the string is, how can we ever make good use of this feature?

8.1/8 THE **LEN** FUNCTION

There's a special function in the ZX81 that gets us out of this problem. It's called

LEN (under the K key)

and it requires a string expression after it. The **LEN** function tells us *how long the string is*. It gives a numeric result. Look at this:

```
10 PRINT "ENTER A STRING";
20 INPUT N$
30 PRINT N$
40 PRINT "YOUR STRING HAD";LEN N$;"LETTERS"
50 GOTO 10
```

If you're clever, you don't need to type the whole program in again – the previous example had the same first three lines and last line, so all you need to do is enter the new line 40.

So now we can see how many letters there are in a string and make sure that we don't look beyond the end of it. This effectively stops us from getting error 3 when we're using brackets to take a letter from a string.

This single letter we've taken is referred to as a *slice*, because it is like a slice out of a cake.

Question

Taking the following program, can you add some extra lines to make it print each letter of the variable W$ on a different line of the display? Here's a sample run so that you can see what I mean:

```
THE WORD IS CONSTABLE
C
O
N
S
T
A
B
L
E

9/9999
```

Here's the program:

```
10 LET W$="CONSTABLE"
20 PRINT "THE WORD IS";W$
```

Answer
Your best approach is to use a **FOR/NEXT** loop, although to use **GOTO** is not really much different.

```
 10 LET W$="CONSTABLE"          (as given)
 20 PRINT "THE WORD IS";W$      (as given)
100 FOR S=1 TO LEN W$           (set up loop starting at 1, going
                                through to the last letter in W$, since
                                LEN W$ indicates the last letter)
110 PRINT W$(S)                 (print the next letter)
120 NEXT S                      (next letter)
9999 STOP                       (all done)
```

Don't be too alarmed at this stage if your program didn't work exactly as intended.
If you had problems, then I suggest you type that solution in and run it. Watch what happens. Then read sections 8.1/7 and 8.1/8 once more to make it all a bit clearer.

8.1/9 SUBSTRINGS

You've seen how to get an individual letter out of a string, but what about getting a group of letters?
One way of doing this is to use a small **FOR** loop to take each letter (like the solution above) and join them all together using the + symbol, like so:

Assuming the string variable V$ contains the string "HORACE", we want to get the letters ORA into a string variable P$. That is, we want to take out the second letter up to the fourth letter and put it into P$:

```
  5 LET V$="HORACE"
 10 LET P$=""                   (set up P$ as "empty" – no letters)
                                (more on this later)
 20 FOR L=2 TO 4                (for each letter from no. 2 to 4)
 30 LET P$=P$+V$(L)             (add next letter onto P$)
 40 NEXT L                      (next letter)
100 PRINT P$
```

Does that all make sense? Line 10 is particularly important. If we left line 10 out, then when the program is run, the ZX81 would give error 2 at line 30. Why?
Because the statement **LET** P$=P$+V$(L) relies on P$ existing before it can work. The statement means "take the variable P$, add the Lth letter from variable V$ to the end of it, and put the answer back into P$".
But the ZX81 can only do this if variable P$ exists already. So line 10 is necessary in order for the program to work. What exactly does line 10 do?
It sets up string variable P$ as *the empty string*.
You must remember that there is a big difference between a variable that exists but contains no characters, and a variable that does not exist.

Question
See how you get on with this:

(a) What would you expect the **LEN** function to give you if it were used on an empty string? An example:

```
10 LET P$=""
20 PRINT LEN P$
```

(b) How do you think that you could test for an empty string?

Answer
If you spotted it, then very good. The second part of that question was really another way of asking the first part.

(a) An empty string would give **LEN** equal to zero.

(b) There are two ways:

```
IF LEN X$=0 THEN ....
```
and
```
IF X$="" THEN ....
```

but watch out! – the "" used in all these examples is *not* the character SHIFT/Q – it is SHIFT/P *pressed twice*.

The SHIFT/Q double-quotes is only ever used if you want to print out a quotes character *within a string*. Look back at section 8.1/2 if you're unsure.

Those questions hopefully made you think a bit about the nature of strings inside the ZX81. If you found them impossible, then read section 8.1/9 over again.

What that example was leading to was to show a better way of getting groups of letters out of a string. We can write something like:

```
10 LET V$="HORACE"
20 PRINT V$(2 TO 4)
```

and those two lines replace the whole program shown on the last page. Line 20 says "take the second to fourth characters from the string expression V$ and print them".

But there is no reason why we can't say

```
20 LET P$=V$(2 TO 4)
```
or even
```
20 LET P$=V$(JIMMY TO FREDDY)
```

provided that JIMMY and FREDDY are both previously defined numeric variables, and that the *values* of JIMMY and FREDDY are characters that exist in the string V$, otherwise you'll get that rotten error 3 again.

The figures in brackets only restrict the size of the string expression, so that V$(2 **TO** 4) is *still* a string expression. This means we can write something like:

```
10 LET V$="HORACE"
20 LET V$(2 TO 4)="XYZ"
30 PRINT V$
```

and the ZX81 will print H*XYZ*CE. This feature can be extremely useful.

Question
Write down what you would expect the following small examples to give:

(i) **PRINT** "ALPHABET" (6 **TO** 8)

(ii)
```
LET X=2
LET Y=6
LET F$="HORTICULTURE"
PRINT F$(X TO Y)
```

(iii) Again, using F$="HORTICULTURE" from question (ii),
PRINT F$(Y **TO** 13)

(iv) Still using F$,
LET F$(1 **TO** 2)="PA"

Answer

(i) BET
(ii) ORTIC
(iii) error 3, since 13 is beyond the length of HORTICULTURE
(iv) PARTICULTURE

If your answers were wrong (or even one of them), read sections 8.1/7 to 8.1/9 again.

8.2 STRING REPRESENTATION

8.2/1 HOW THE ZX81 SEES STRINGS

When I started writing this course, I vowed that it would not get too technical, because computers are such complicated animals to fully understand. As far as you are concerned, whatever the ZX81 does inside is completely its own business!

At this point, however, we need to think a little bit about how the ZX81 "sees" things like numbers and letters, because this is one area where strings can be made full use of.

If you can't understand this section, or don't want to, then you'll certainly have enough understanding of strings in order to write some extremely useful programs. But don't let me put you off, since what follows is probably one of the most interesting sides to computers.

The ZX81, in common with all other computers, does not really understand English at all. It always works in numbers, and merely reserves special numbers (called *code numbers*) to represent letters and characters. So while you are entering letters and characters, the ZX81 is translating them into the code numbers that it can work on.

How can we see what the ZX81 thinks each character is?

I've included a small program that you can run:

LOAD "CODES"

When you run it, it shows you two things on each line.

First is a normal number, which starts at zero, and goes up by one on each new line as the program runs. (Run it now so that you can see what I mean.)

Second is the *character* which the ZX81 connects with this code number. So when you see the line:

38 REPRESENTS A

this tells you that the ZX81 connects the code number 38 with the character "A". Note that I use the word *character*, because as far as the ZX81 is concerned, what we call "the letter A" is merely another character – as is the symbol : or * or any other of the items on the keyboard.

OK, OK, I know I said it would be tricky, but if you haven't followed that bit too well, don't panic.

The program "CODES" has a couple of features you may like. If you see something interesting on the screen, then just press any key (except BREAK). The screen will stop for about 10 seconds, then carry on again. If you press another key while it's pausing, then it will carry on even though the 10 seconds aren't up. Have a go.

That "stopping" and "starting" feature makes use of strings (believe it or not!), and you'll soon see how it's done.

Watch the screen as all the various characters go up. You'll see all the items on the keyboard (including the keywords like **STOP**, **RUN**, etc and the functions like **ABS**, **LEN**) gradually appear. The alphabet is there, and all the numbers from Ø to 9. Notice how these numbers (Ø to 9) are represented by a different value. This is because the ZX81 sees "5" as a *character*, and not a value.

There are a lot of numbers that show "REPRESENTS ?". These are where the ZX81 does not have any equivalent character to correspond to the particular number.

So now we've seen that the *code value* of the character A is 38.

How can we make good use of this? There are many ways, but the most important of these is . . .

8.2/2 STRING COMPARISON

Two characters can be compared so that we can see if one character has a larger or smaller value than another character, and therefore we can put characters into a sort of sequence.

Whenever you ask the ZX81 a question like

IF H$="ABCDEF" **THEN GOTO** 60

the ZX81 looks to see if the code value of each letter in the strings are identical. Although it amounts to the same thing to us, it does have more flexibility when it comes to writing things like:

IF H$>"ABCDEF" **THEN GOTO** 60

because we can see what the ZX81 will do.

Supplied with your ZX81 was a handbook giving full details of all that the ZX81 can and can't do. If you turn to the end of this handbook – page 181 – you'll see Appendix A lists each of the ZX81 characters and the code value that goes with each character (ignore the other columns – they won't concern us in this course).

Try your hand at these:

Question

Say whether you think the following conditions are true or false. As an example, I will answer the first one for you.

(i) Is "A" < "Ø" (inverse Ø)?
(do you remember what <,>, <>, >= and <= mean? Look back to section 3.2/3 if you can't)
Answer: Yes. Since we know that the ZX81 compares the code value of each character in a string one-by-one (see above), all we need to do to answer this question is to see if the code value for A (which is 38 – look it up in Appendix A of the ZX81 Handbook) is less than the code value for an inverse Ø, which is 156. Since 38 is less than 156, the answer is
Yes, "A" is less than "Ø" (inverse Ø).

(ii) Is "Q" > "+"?

(iii) Is "AA" >= "AB"?

(iv) Is "6" < "£"?

Answers

(i) we've already answered this one.

(ii) Yes – code of "Q" is 54, and code of "+" is 21

(iii) No. This may have caused you a problem. Remember that the ZX81 compares one-at-a-time, so first it says 'Is code "A" greater than or equal to code "A"?'. The answer is that they are equal, so it carries on to the *next* pair of characters. Now it asks 'Is code "A" greater than or equal to code "B"?'. This time the answer is No, since code "A" is 38 and code "B" is 39.

(iv) No. Code value of "6" is 34 and the code value of "£" is 12. Therefore the code value of "6" is *greater* than the code value of "£".

If your answers to parts (iii) and (iv) were correct then you deserve a gold medal!

It is very easy to make a mistake in answering questions like those, and you are not expected to learn all the different code values for each character. If you didn't understand them at all, try reading that section over again in the light of experience. Look at the "CODES" program running, and check that the results agree with Appendix A in the Handbook.

I expect that by now, you are probably itching to see how you can use all this information.

8.2/3 THE **CODE** AND **CHR$** FUNCTIONS

There are two functions that let you make full use of the way the ZX81 handles its characters inside. These are:

CODE s

where s represents a string expression.

CODE gives a numeric result which is the ZX81's code number corresponding to the *first* character in the string expression that follows. Hang on a minute and you'll see some examples.

The other function is

CHR$ n

where n represents any numeric expression. What do you know about a function that has a $ symbol?

The answer is that it is a *string* function. The **CHR$** function gives you the character that corresponds to the following numeric expression.

Here are a couple of examples to show you what they both mean:

CODE "A"	gives the value 38.
CODE "ABC"	also gives the value 38 since the **CODE** function only looks at the first character.
CODE "£"	gives the value 12.
CHR$ 38	gives the character "A" as a single letter string.
CHR$ 128	gives a black square – look in Appendix A to see what character corresponds to the code value 128.

Since the ZX81 only has characters with codes in between 0 and 255, if you try to write

CHR$ 256

or anything greater than 256, you'll get an error B. This means that the ZX81 cannot give you any such character.

Question

What would you expect the two following examples to give?

(i) **CODE** (**CHR$** 38)

(ii) **CHR$** (**CODE** "A")

Answer
(i) 38
(ii) "A"

The reason is simple. Part (i) asks you to give the **CODE** of the character whose code is 38 (**CHR$** 38). This must be 38!

By the same logic, but with the functions reversed, part (ii) asks you to give the character (**CHR$**) whose code is the code of the character "A" (**CODE** "A") – and that must be "A"!

Sorry if you thought that was a bit of a trick question, but it was meant to show you that **CHR$** and **CODE** are really just the opposite of each other.

One gives you the code value of a character, the other gives you the character that corresponds to the given code value.

If your answer was wrong, or you just couldn't answer, try reading section 8.2/3 again.

8.2/4 THE **VAL** FUNCTION

Here's another new function which is the opposite of **STR$**:

VAL s

again, s represents any string expression. **VAL** gives you the numeric equivalent of the following string. Have care, because the numeric equivalent is *not* the same as the code value. Look:

PRINT VAL "123456"

would print 123456 on the screen.

You can't multiply strings, but you *can* multiply the **VAL** equivalent of a string. for example:

```
LET N$="200"
PRINT VAL N$*3
```

would print 600. The **VAL** function has converted the string into its numeric equivalent.

One point to watch with **VAL** is that you'll get an error C if the string is not a number! So

PRINT VAL "4Z5E"*30

gives an error C. (Who wants to multiply "4Z5E" by 30 anyway?)

A good use of the **VAL** function is to allow you in a program to accept a string variable from the keyboard and convert it to a number. This allows you to enter mixed text and numbers into a program and sort them out to see what was typed.

8.2/5 A FULL EXAMPLE

One last program for you. You are going to write the same program, but it's supplied on tape for you to look at, should you get stuck.

Question

LOAD "SWOPPER"

There is no need to type **RUN** for this program – it will automatically start as soon as it loads. (This is

so that you can't have a sneaky look at the program before you write it!)

The program invites you to enter a string. It then prints the same string back at you, but with all the letters (or characters) reversed.

When you've had enough, just press NEWLINE. The program realises that you haven't typed anything and so it stops.

Have a go to see what it does, then *you* write it.

Answer

Look at the program on the screen to see how it's written. You can get at the listing by stopping the program (refer back to the question).

This program can be written in hundreds of different ways – my own personal preference is to use **FOR/NEXT** loops wherever possible. My reasons?

Firstly it helps to reduce the size of the program – which on the basic ZX81 is something that you should always watch out for. It does not take a very big program to fill up the ZX81's memory.

Secondly, **FOR** loops help to keep all the code in one neat "parcel" that can be mentally put to one side. The line **FOR** introduces quite tidily the following lines. If **GOTO**'s are used, however, you have to read through the lines of program before you find out what the loop control is. Here's just a tiny example of what I mean:

Program (a)

```
10 LET X=1
20 PRINT X
30 LET X=X+1
40 IF X<20 THEN GOTO 20
```

Program (b)

```
10 FOR X=1 TO 20
20 PRINT X
30 NEXT X
```

Quite apart from the fact that (b) is shorter, it also tells you in the *very first line* that the following lines are to be repeated for each value of X from 1 to 20. Program (a) does not have the same ease of reading – it's not until line 40 that you realise that the previous lines are repeated for all values of X from 1 to 20.

Finally, **FOR** loops avoid using direct line numbers. Although they cannot be totally avoided in a program, to keep the quantity of line numbers to a minimum is a good practice as the changes required to a program become more manageable when you don't have to keep checking on each **GOTO** constantly.

Lecture time is over now. How did your version of "SWOPPER" work? If it did the job, then that's what really matters for now. The more programs you write and the more you study other programmers' works the better your own will become.

If you had problems writing "SWOPPER" then you should spend a bit more time understanding the nature of strings. Read this chapter again.

Perhaps you would like to study **FOR** loops again. These were covered in Chapter 4, section 4.2 "Iteration (2)".

Summary

Once again, this has been quite an intense chapter for you and if you've followed it all then you are certainly well on the way to writing your own full programs. The computer version of the game "Mastermind" is now within your capabilities, but don't try to write it just yet! Following chapters will add extra depth to your skills – especially the next one, which is concerned with arrays (all will be revealed!).

But just take a last lingering look at the content of this chapter. If you're unhappy with any part of it, then now is the time to read that section over again.

We've seen:

– that strings can be used in a similar way to numeric expressions; they can be held in string variables, tested, manipulated and printed.

– how two (or more) strings can be concatenated.

– what a string expression means, and how to recognise a string function.

– how individual characters or groups of characters (called "slices") can be extracted from a string or entered into a string without disturbing any other part of the string.

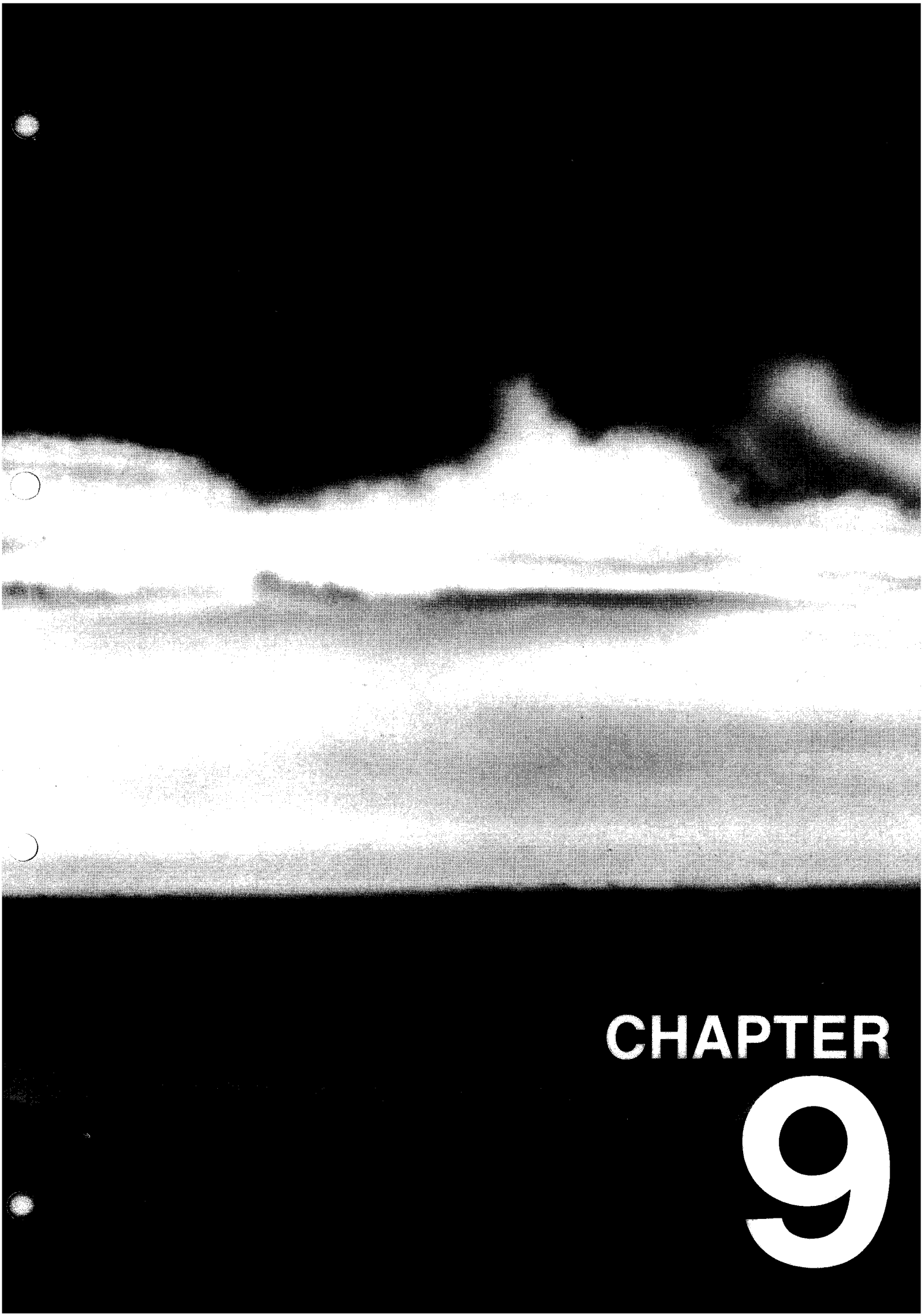
– how the ZX81 "sees" a string, and how this can be used to good advantage when we want the ZX81 to compare two strings.

Exercises

1. Write a program that asks for a string to be entered, then extracts all the vowels from the string and prints it.

2. The ZX81 normally shows letters as ''black-on-white'', but you can reverse this by adding 128 to the code of the character (see Sinclair Handbook, Appendix A). Write a program that asks for a string to be entered, then prints the string in ''white-on-black'' characters (usually called *inverse video*).

3. Create a program which allows you to ''draw'' pictures. The program should input two numbers (which corresponds to a position on the screen) and a string which is to be printed at this point. As more and more strings are entered and printed, so the screen can be gradually made into a picture.



CHAPTER 9

Arrays

This chapter introduces the last major topic in the course – all subsequent chapters merely strengthen some of the ideas that have been introduced as we've gone along. The first section in the chapter deals with string arrays, using a program to demonstrate how much easier they can make certain types of programs.

Section 2 deals with numeric arrays, again using programs from cassette to show how they can be utilised.

9.1 STRING ARRAYS

9.1/1 AN EXAMPLE PROGRAM

Let's start now with trying to identify what an array is and why it should be used.

From experience, the two topics that most beginners find hard are *subroutines* and *arrays* – the former because it is always possible to write a program such that subroutines are not needed (although the program will most likely suffer!) and the latter (arrays) because it seems hard to know exactly when arrays can or should be used.

We're going to work our way gradually into a position where it is more or less impossible to write a program without using an array. This will hopefully give you a clear picture of what an array really is and what it does for a program.

The program is concerned with people living in a certain street, Subscript Street in Ramstown.

This street only has 8 houses, since it connects two other main roads together but still, the postman always has great difficulty in remembering which house has which family living in it. Worse still, each house has no number – they are all called by a name.

Being a rather smart postman, he decided to chalk numbers onto each of the front gates (so that the occupants wouldn't notice, of course) and he ended up with something like this:

House number	Name of house	Family name
1	Dunroamin	Jones
2	Lane End	Smith
3	Hillview	Black
4	Hadenuff	Brown
5	Nextdoor	Walters
6	Alcazar	Evans
7	Whynot	Taylor
8	Clifftops	McPhee

This was all very well, but still it didn't tell him who lived where, since no letters ever had his special "house number" on them.

So he bought himself a ZX81 (surprise, surprise) and decided to write a small program which would help him to remember which house had which family in.

He decided that each night he would run this program until he could reel the names of the houses and families off the top of his head (OK – I know 8 isn't a lot to learn, but just remember that this limit is only set by me to make it all manageable. There could equally be 200 houses).

This is where he got stuck. It would be nice if he could type in either a number, house name or family name and get the rest of the information out. But a program like that would go on for ages – imagine trying to write it!

Question

Don't panic! I'm not going to ask you to write the whole program – just a few lines.

Write down just the *outline* of a program which would show the house name and family name given any house number. Here's a sample run just to show what I mean:

```
ENTER HOUSE NUMBER :3
HOUSE : HILLVIEW
FAMILY : BLACK

9/9999
```

Answer

Your program probably looked something like:

```
10 PRINT "ENTER HOUSE NUMBER :";
20 INPUT N
30 PRINT N
40 IF N=1 THEN PRINT "HOUSE:DUNROAMIN",,"FAMILY:JONES"
50 IF N=2 THEN PRINT "HOUSE:LANE END",,"FAMILY:SMITH"
. . . .
. . . .
110 IF N=8 THEN PRINT "HOUSE:CLIFFTOPS",,"FAMILY:MCPHEE"
```

or

```
10 PRINT "ENTER HOUSE NUMBER :";
20 INPUT N
30 PRINT N
40 PRINT "HOUSE:";
50 IF N=1 THEN PRINT "DUNROAMIN"
60 IF N=2 THEN PRINT "LANE END"
. . . .
. . . .
110 IF N=8 THEN PRINT "CLIFFTOPS"
120 PRINT "FAMILY:";
130 IF N=1 THEN PRINT "JONES"
140 IF N=2 THEN PRINT "SMITH"
. . . .
. . . .
210 IF N=8 THEN PRINT "MCPHEE"
```

Well. You have to admit that they're both long-winded. How did your answer shape up? I expect you got bored writing that lot out, so you gave up and looked at the answer, didn't you?

Sorry to disappoint you, but so far, you know of no other way of writing such a program (if your memory is really good, then you might have tried using a *conditional* **GOTO** – refer to Chapter 3).

But there's no getting away from it. It's a boring drudgery to write a program like that.

9.1/2 INTRODUCING THE ARRAY

This is where *arrays* come in.

Wouldn't it be nice to say to the ZX81 – "just print me out the details of house number 2"?

We can. Here's a small comparison of how it's done.

In the last chapter, you saw how an individual character can be taken out of a string by using a *slice,* for example:

```
PRINT V$ (2)
```

. . . which prints the *second* character from V$.

With a string array (which is what we are referring to in our "postman" problem) we can set up an array consisting of a quantity of independent strings and then take out any one of them. The command that sets up the array is:

DIM (on the D key)

. . . and this tells the ZX81 how many items you want to store in your array. We would write:

```
DIM H$ (8,9)
```

This says "Reserve eight string variables which each hold nine characters".

By then writing:

PRINT H$ (2)

. . . the ZX81 will print out not just one character, but nine – the whole of the second string (as indicated by the number 2). If we wanted to print the fourth character from this second string, we could write:

PRINT H$ (2,4)

All the **DIM** command has done is to extend our slices from characters to whole strings.

9.1/3 LOOKING AT THE PROGRAM

Let's help the postman out and look at a program that lets him enter all his house names and family names into the ZX81.

LOAD "NAMES"

When the program is loaded, it automatically starts running. This is because I have already entered all the names given above and I don't want you to wipe them out straight away! Don't forget – whenever you type **RUN**, the ZX81 clears out all the variables that exist. This was covered in Chapter 6.

If you make a mistake it doesn't matter as you can always load the program back in again.

First of all the program lists the names of the houses and families for you to see, then it stops (error 9). Press NEWLINE to look at the program listing.

The program (with all the names) only just fits into the basic ZX81, so unless you've got the RAM expansion pack, you may find it a bit slow moving around the program to see what's going on. It will help if you type **FAST** before you go too far.

Lines 30 and 40 set up two string arrays H$ and F$. String arrays can only have a single letter variable name followed by a $ symbol of course).

Array H$ is used to hold the house names. Line 30 says "Create an array called H$ which contains 8 strings each with 9 characters". Our original notes above said that there were only 8 houses in the street, so there would be no point in creating an array of any more than 8 strings. The longest house name is nine characters – look back and check for yourself.

Question

How many strings does line 40 set up and how long is each string?

Answer

DIM F$ (8,7) sets up 8 strings, each 7 characters long.

If your answer was wrong, you should read section 9.1/2 again.

Let's carry on with the program. Line 50 clears the screen and lines 100–150 take a number from the keyboard and check that it is in the range 1 to 8. If not, then the number is printed out followed by a question mark and the program tries to get another number instead. This variable N is quite important. It holds the house number that we wish to affect in some way. Lines 200–220 ask for a house name to be entered – but what's this? Line 210 says **INPUT** H$(N)! What does *that* mean? Have a think. We know that N contains the house number – as an example, assume we've entered number 2. Line 210 says "Input a string from the keyboard (since the variable name has a $) and put it into the Nth string in array H$ – since N is 2, put it in H$ (2)".

So the result of this is that the *second* string in array H$ has been entered from the keyboard. Got it? If not, don't worry too much as you'll soon have a go at putting some in yourself to see what happens. But hang on for just a bit longer.

Lines 300–320 do a similar job on the array F$ – this time it's a family name that's being entered.

Lines 500–530 ask if you want to enter any more. If you type "N" then the program will stop. Any other answer is taken to mean that you wish to continue. Line 520 is quite complicated – it says:

IF LEN Y$<>0 **THEN IF** Y$(1)="N" **THEN STOP**

This will probably stump you at first. The meaning of this line is as follows:

1. Is the length of Y$ <> 0? No – then drop through to the next line number (in this case 530) and ignore the rest of these steps.

2. Since the ZX81 has reached this far, the length of Y$ must be non-zero – i.e. something has been typed in.

3. Is the first character of Y$ equal to the character "N"? If the answer is no, then ignore any more of these steps – drop through to line 530.

4. Since the ZX81 has reached this far, the length of Y$ must be non-zero *and* the first character is "N" – in which case **STOP**.

Why not just write . . .

IF LEN Y$<>0 **AND** Y$(1)="N" **THEN STOP**

Just think what would happen if nothing was entered into Y$ (i.e. it's an empty string). LEN Y$ would equal zero – that's OK, but Y$(1) would then give error 3 because the string has not got a first character in it! Catch 22!

You could write the same thing using a series of **IF** statements and **GOTO** statements, but it would take up a lot more room. The program's big enough already!

On with our program.

Lines 600–630 are a self-contained program that list each of the items in arrays H$, and F$ on the screen for you to see. Each line shows the house number X, the house name H$(X) and the family name F$(X).

Line 700 is a "load-and-go" routine described in Chapter 6.

We've now unpicked the program. Have a go at changing a few names to get the feel of what the arrays H$ and F$ are doing.

Type:

GOTO 600

This will list out all the names of houses and families for you. Try it now.
If you want to change any names, then type:

GOTO 50

***DO NOT TYPE* RUN** – it will clear out all the names that are in the program.
The program will ask you to enter the house number that you wish to alter. Enter a number between 1 and 8, then the program will ask you to enter the house name and the family name. It then asks if you want to change any more. If you type *anything beginning with "N",* the program will stop. Otherwise it asks you to enter another house number.
Change some of the names then type **GOTO** 600 again to list them out. See how they've changed.

Question
Our postman wanted a way of just typing a house number and the ZX81 would tell him the name of the house and the family living there. The program you are playing with ("NAMES") can do this if only two lines of the program are removed.
Which two?
Here's an example run of what he wants:

```
HOUSE NUMBER? 3
HOUSE NAME:     HILLVIEW
FAMILY NAME:    BLACK
MORE? N

9/520
```

210 &
310

NOTE – STRINGS ARE RESTORED AFTER LOADING A PROGRAM STRING WILL NOT CHANGE IN DIMENSION BY YOUR EDITING THEM (YOU CAN EVEN REMOVE THEM FROM PROGRAM & IT WILL STILL WORK) TO SEE CHANGES YOU MUST SAVE PROGRAM & THEN RELOAD IT.

Answer

By removing lines 210 and 310 the program will work as the postman wanted.

If you don't believe me – try it! All you've done is to stop the ZX81 from asking for a new name to be entered. It still prints the old one.

Was your answer wrong? If so, then look closely at the program to see what it's doing with those two lines removed.

9.1/4 SUBSCRIPTS

Now you've seen how arrays can save you program time and space (think back to the very first question in this chapter), it's time to move on.

String arrays have been introduced first because the last chapter had already given you some clues on arrays when "slices" were covered.

Before we get going on numeric arrays, you need to understand another bit of jargon.

Whenever we write something like:

H$(X) or F$(A,B)

. . . the expressions inside the brackets are called

subscripts.

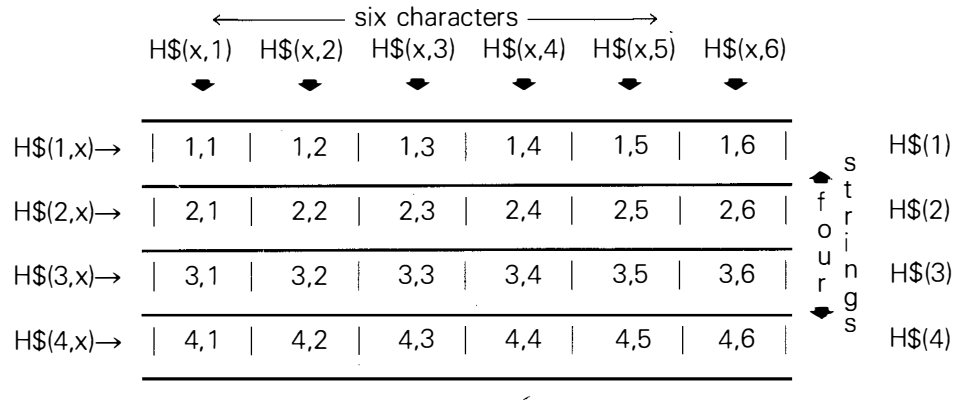
In these examples X,A and B are all subscripts. A common way of reading an expression like H$(X) is "variable H$ subscripted by X".

Notice how important it is to get the subscripts the correct way round when more than one is involved. Look at this:

DIM H$ (4,6)
PRINT H$ (2,3) and **PRINT** H$ (3,2)

. . . are *entirely different*. H$(2,3) prints the third character from the second string while H$(3,2) prints the second character from the third string. Pictorially, this is:

DIM H$(4,6)



So you can now see quite easily why H$(2,3) and H$(3,2) are so different.

Question

Look at this program:

```
10 DIM S$(3,6)
20 LET S$(1)="FREDDY"
30 LET S$(2)="JEREMY"
40 LET S$(3)="JAGUAR"
```

Write down what you think the following statements would do:

(i) `50 PRINT S$(2,3)` R

(ii) `50 PRINT S$(3,2)` A

(iii) `50 PRINT S$(3)` JAGUAR

(iv)
```
50 FOR X=1 TO 6
60 PRINT S$(3,X);
70 NEXT X
80 PRINT
```
JAGUAR

Answer
(i) R
(ii) A
(iii) JAGUAR
(iv) JAGUAR

How well did you do? If all your answers were correct, then you've done well. If some of them were wrong, read section 9.1/4 again. If they were *all* wrong, or you just can't get to grips with these arrays, read the whole of section 9.1 again.

9.2 NUMERIC ARRAYS

9.2/1 ANOTHER EXAMPLE

Once again, I shall give you a parallel example to look at. This time, the program comes in two parts. It's quite adventurous and involves some interesting techniques, so you'll be learning a lot from this example.

The program is for our postman once more – he's really having some problems! This time, he's using his ZX81 to work out first and second class letter rates.

If you've ever tried to work out how much it costs to send a letter, then you'll understand why he needs a computer to do it! It all depends on the weight of the letter. Using the postage rates set up on 26th January 1981, a first class stamp for a letter not heavier than 50 gms costs 14 pence. The same weight of letter, but sent second class costs 11.5 pence.

As the weight of the letter goes up, so does the cost.

Let's look at the *first* part of the program.

LOAD "CREATE"

The reason that this program is in two parts is simply because it's too large to fit into the basic ZX81. You'll soon see how it's done.

List the program.

You'll see lines 100 and 110 use the **DIM** command to set up two arrays – but here's something new – line 100 does not set up a string array. There's no $ symbol. It is setting up a numeric array. What's that?

Just as you have seen how a string array of 4 strings each containing 6 characters can be defined by **DIM** S$(4,6), so you can define a numeric array consisting of four sets of six numeric variables by stating **DIM** L(4,6).

Instead of holding characters, the array is holding numbers. We can use any of the 6 * 4 (=24) elements just like they were numeric variables.

Let's see how this works in practice.

The GPO letter rates have 12 different weight limits like this:

Letter weight (gms)	1st class (pence)	2nd class (pence)
50	14	11.5
100	20	15.5
150	26	19
200	32	24
250	38	30
300	44	36
350	51	42
400	58	48
450	65	53
500	72	58
750	108	87
1000	144	87

If your letter weighs 149gms, a first class stamp will cost 26 pence and second class 19 pence.
Second class maximum is 750gms, and although letters heavier than 1000gms can be sent, this particular program does not cater for them.

So for each of twelve categories, we want to hold three values – the weight limit, first class price and second class price. Now look back to line 100 in the "CREATE" program. What does it say?

DIM L(12,3)

This corresponds to our 12 categories with three values. The numeric array can then be built up so that it holds all the table shown above. And the program "CREATE" lets you do this. Carry on for a bit longer.
Line 110 introduces a string array of three strings each containing 9 characters.
Lines 120 to 140 set up the three strings in the string array.
Line 200 introduces a **FOR** loop that cycles 12 times – once for each category. The control variable W will be used to enter the three values into the appropriate category of the numeric array.
The crux of the program is in lines 220 to 260. These lines are a **FOR** loop that ask for the three values weight, 1st class rate and 2nd class rate. Let's unravel this more closely.

Line 210 initially sets control variable I to 1.

Line 220 then prints the first string (since I=1) from array M$ – this is the string "WEIGHT" – followed by a semi-colon (more to be printed on this line).

Line 240 asks for array element L(W,I) to be entered.

Variable W is the category number (1 to 12 according to the list above), and I is currently 1. So the array element L(1,1) will contain the first weight.

Line 250 takes the next value of I (which is 2).

Back to line 230 again, This time the string "1ST CLASS" is printed, and line 240 asks for array element L(1,2) to be entered since W is still 1, but I is now 2. The first class letter rate will therefore be in L(1,2).

Similarly, the second class letter rate will be placed in element L(1,3).

This is repeated for each of the 12 categories.

Run the program and enter the table of values given above. At the end of the twelfth category, the program stops. Array L now looks like this:

L(12,3)

	L(x,1)	L(x,2)	L(x,3)
L(1,x)	50	14	11.5
L(2,x)	100	20	15.5
L(3,x)	150	26	19
L(4,x)	200	32	24
L(5,x)	250	38	30
L(6,x)	300	44	36
L(7,x)	350	51	42
L(8,x)	400	58	48
L(9,x)	450	65	53
L(10,x)	500	72	58
L(11,x)	750	108	87
L(12,x)	1000	144	87

Question

What values do the following elements contain?

(i) L(8,3)
(ii) L(2,2)
(iii) L(1,3)
(iv) L(12,1)

Answer

(i)	48 pence
(ii)	20 pence
(iii)	11.5 pence
(iv)	1000 gms

You can check any of these by **PRINT**ing them – e.g. try this one: type

PRINT L(8,3)

If you've set things up properly when you entered the values, it will print 0.48.

How did you manage with those? If you had problems, then read section 9.2/1 over again, paying careful attention to the workings of the "CREATE" program.

9.2/2 MAKING USE OF THE ARRAY

Now on to the second part of the postage rates program. The first part has allowed us to create two arrays – L, containing the weight, 1st and 2nd class rates for each of twelve categories, and M$ which contains three strings "WEIGHT", "1ST CLASS" and "2ND CLASS".

Since the first program is not really needed any more (how often do we need to set up the rates?) all these program lines can be deleted. When this is done, you must be *very careful* not to use the commands **RUN** and **CLEAR**. These commands will remove any variables, so destroying all the hard work so far!

It doesn't matter that there is no program inside the ZX81 – all the variables are still there! Prove it for yourself if you want to – delete all the lines in the program and then type:

PRINT M$(1)

. . . and the ZX81 will print WEIGHT. Remember back to Chapter 1 where we used **LET** as a direct command to store a variable.

We can now type in another program, and those two arrays will still be there for the new program to use as it wishes!

We can even save them on tape – in fact, although they can't directly be seen, we can use them in any way at all.

There is only one other command to watch out for. If the program meets another **DIM** statement for the same array that it has already got, then the old array is cleared out. All the strings (in a string array) are set to spaces, and all the values (in a numeric array) are set to zero.

There is no need to keep your array created in the program "CREATE", as I have already done this for you. The second part of the program is also on tape:

LOAD "RATES"

Once again, the program automatically starts. This helps to stop you from typing **RUN**.

The program invites you to enter a weight, then it prints the corresponding first and second class stamp values.

Not bad, eh?

Take a look at the program listing. Notice that there is no **DIM** statement – see above. Yet the program still refers to arrays L and M$ quite happily.

I'm not going to unpick this program for you – it's quite easy and should not cause you any problems.

It is beyond the scope of this course to show you how to get the most out of the basic ZX81 (in terms of the size of program you can write), but what we have done with these two programs "CREATE" and "RATES" is one very good way of getting a larger program to fit inside the ZX81 than would normally be possible. If you already have the RAM expansion pack, then these two programs would not really have been split into two, but this course is intended to work for *any* ZX81.

Question

Name three ways in which an array can be removed from the ZX81.

Answer

1. The **RUN** command.
2. The **CLEAR** command.
3. The **DIM** command (only when the same array name is used).

If your answer was wrong, then read section 9.2/2 again.

9.2/3 AN ALTERNATIVE VIEW

One final program for you on arrays. This program doesn't do much at all – it's designed to give you a different view of arrays and hopefully to give you something useful for certain types of games that you may write.

Way back in Chapter 5, section 5.1, you were shown a method of making a numeric variable alternate its value between Ø and 1. The text even told you that this would come in handy!

The method was:

```
50 LET P=Ø          initial value
. . . .
. . . .
200 LET P=(P=Ø)     Ø becomes 1, 1 becomes Ø
```

Here's a good way of using arrays combined with this feature.

Suppose you want to write a game that involves two players. The players could be you and the ZX81, or you plus a friend – it doesn't matter.

Most games involve keeping a score of some sort – your score and your opponent's score.

This is where a numeric array can be handy. Instead of setting up two variables, one for your score and one for your opponent's, create a two element array using **DIM**. Have a look at what I mean:

```
LOAD "STRATEGY"
```

Line 1Ø sets up an array to hold the scores, and line 2Ø initially chooses player number 1.

Lines 11Ø–13Ø print out the scores of both players – S(1) holds the score of player 1, and S(2) holds the score of player 2.

Line 14Ø tells the next player (which could be 1 or 2) to make his/her/its move.

Line 2ØØ is merely a **REM**ark to tell you that your main game program would go here.

Line 3ØØ adds the score (which is whatever you entered for "move") into the appropriate player's score. Look carefully at this line.

Line 31Ø swaps the player from number 1 to number 2 or from number 2 to number 1. It is slightly different from the example above, but the same logic is at work.

Line 32Ø jumps back for the next player's move.

Try running it. Enter some numbers in at each move, and you'll see the scores go up according to whose turn it was.

The variable names in this program are all quite long just to make it all clear for you, but there's no reason why you can't use just the letter "P" for your version.

Question

Just to give you a comparison, try writing the *same* program without using arrays. Use two variables to hold the scores – P1 for 1, and P2 for player 2.

Answer

Here's one solution:

```
 20 LET PLAYER=1                          "initial player
 30 LET P1=0                              clear scores
 40 LET P2=0
100 CLS
110 PRINT AT 10,0;"PLAYER 1",P1           player 1 score
120 PRINT AT 11,0;"PLAYER 2",P2           player 2 score
140 PRINT AT 21,0;"PLAYER "PLAYER;" TO MOVE"
150 INPUT M
300 IF PLAYER=1 THEN LET P1=P1+M          add to player 1 score
305 IF PLAYER=2 THEN LET P2=P2+M          add to player 2 score
310 LET PLAYER=(PLAYER=1)+1               swap player
320 GOTO 100                              next move
```

How did you get on? Once you've started to use arrays, it becomes quite hard to see any other way of writing programs.

If your answer wasn't complete, or you couldn't answer, then you should read some of the earlier chapters once more. Obviously, I don't expect your answer to match mine, or even for you to write it all down as fully as I have. If you're in any doubt about your own programs then you should type them in and try to run them. Keep changing them until they work!

If you're happy with your version, then that's good enough.

Apart from the fact that the version of this program on tape has some extra **REM** statements, it is much more compact than the version given above.

They are both fairly easy to read and follow, and which version you prefer is just a matter of taste. But consider the same programs written for a four-player game. Which version would you prefer then?

9.2/4 MULTI-DIMENSIONAL ARRAYS

You've seen how arrays can take on one or two dimensions – the array used in "RATES" is a two-dimensional array – so it's really only an extension of this to say that arrays can have as many dimensions as you wish! There is a limit of course, but the limit is that of the room inside your ZX81 to be able to hold the array.

It gets extremely hard to imagine a multi-dimensional array, so perhaps I can paint a small scene in order to help you.

Imagine a small boy who finds a handful of conkers and puts them in his pocket. Then he goes back to his home where he meets with his brothers and sisters. If we now wish to see how many conkers one of these children holds, we obviously need to consider *which* child we are referring to – any one of them might have a pocketful of conkers! Assume our particular boy is the second oldest:

```
DIM CONKERS (family size)   }this won't work, it's an
LET CONKERS (2)=17          }example only!
```

This second line would set up the number of conkers that the second child in the family holds.

But think of all the houses in his town! Each house holds a family, so we must identify the actual house he lives in – perhaps it's the tenth one as you drive into town:

```
DIM CONKERS (houses, family sizes)
PRINT CONKERS (10,2)
```

. . . and this would show how many conkers the second boy in the tenth house holds.

But what about all the towns in the country? It could be any one of them. Looking through the alphabetic list in the AA Handbook, we see that this town is the forty-third:

```
DIM CONKERS (towns, houses, family size)
PRINT CONKERS (43,10,2)
```

Then consider all the different countries in the world – again, we must be sure to say which country we are referring to:

```
DIM CONKERS (countries, towns, houses, family size)
PRINT CONKERS (107,43,10,2)
```

I should hope that you're getting the idea by now. The point is, we can look to see how many conkers (or array element) are held by any child in any house in any town in any country in the world! Just use the **DIM** statement to set up the required number of dimensions, then refer to the particular element by giving successively more detailed subscripts.

Arrays have many uses in computer programming and I hope that the three different examples you've seen in this chapter will give you a good grasp of their workings.

Summary

This chapter has only dealt with arrays (with one or two small digressions), and as mentioned at the beginning, arrays are the last full topic to cover in the course.

Next we look at some complete programs of different types which will give you an opportunity to see some ideas put into practice. There are still more functions to be studied, so don't close the book just yet!

What have we covered?

– string arrays, and how they are used to hold a quantity of independent strings, each individually accessible.

– the concept of a subscript, which qualifies the array element that we wish to obtain.

– numeric arrays, how they are used in practice and how they can reduce the amount of programming required to achieve certain results.

– how multi-dimensional arrays can be pictured.

Exercises

1. Write a program that asks for ten four-letter words to be entered. The program then prints the words out in alphabetical order. You will need to be aware of the ideas covered in Chapter 8, section 8.2.

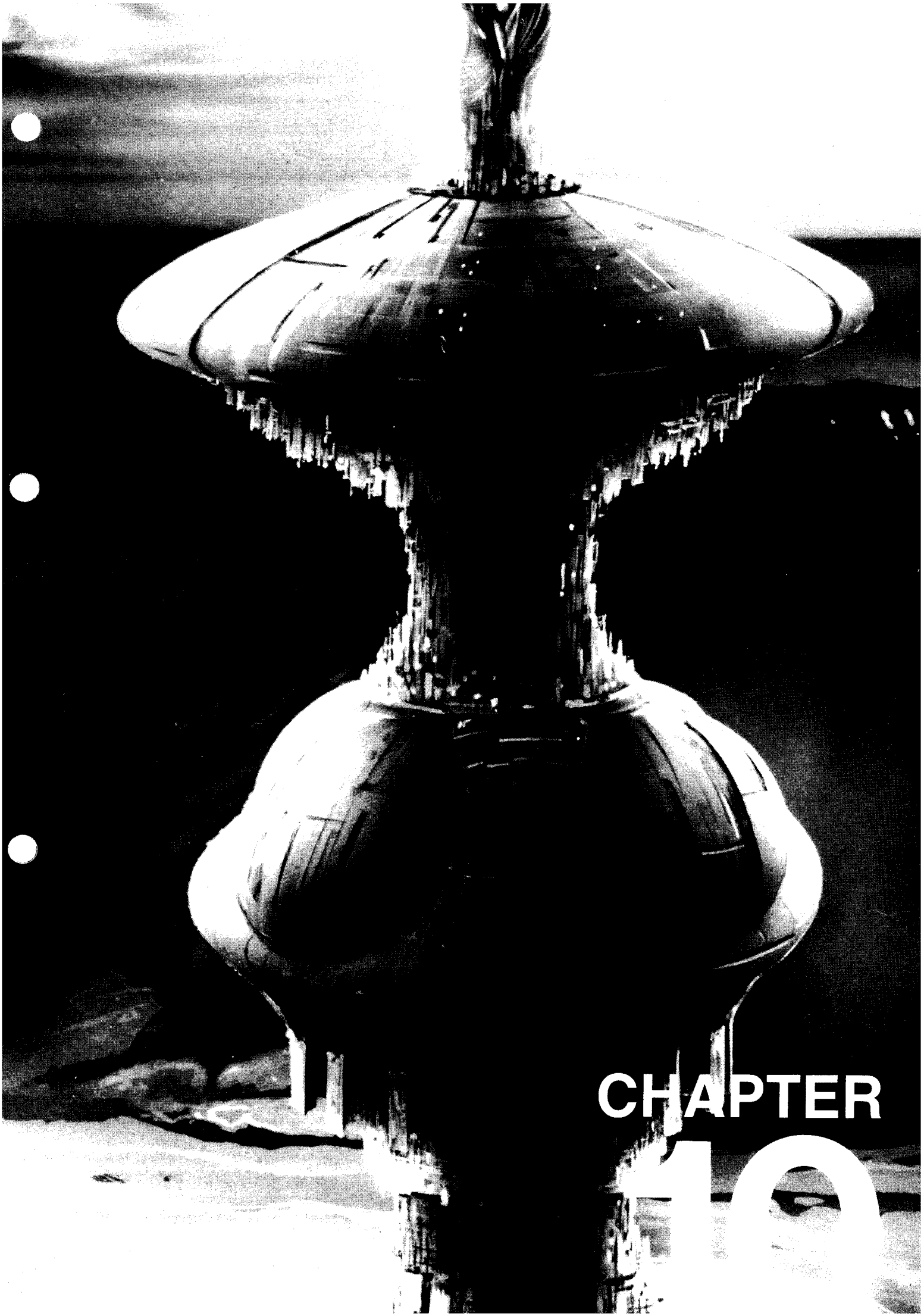
2. This program is a useful aid to code-breakers! Ask for a string to be entered, then analyse the frequency of the letters in the string (beware of spaces in the string!) You should allow the program to count the number of occurrences of each letter, then print either a list, or (if you have enough memory) a bar chart, showing the distribution. This should then assist you in breaking a code, as the letter "E" will normally be the most common letter (assuming the string is English).

3. For those who love yacht racing, try to write a program which records the times of each yacht as it arrives. The program will need to convert hours, minutes and seconds into seconds, then divide by the Portsmouth Yardstick (a preset handicap for each boat), and finally express this back into hours, minutes and seconds. Your program should print the boat numbers and corrected times.

4. Continuing from exercise 3, rank the results obtained into "time" sequence, where the lowest time is the first result. Print the top ten results obtained in this way.

5. Write a routine which searches for one string inside another. For example, if string A$ contains "CAT DOG", and the routine is asked to search for "DOG", then it will give a reply of "I FOUND IT".

   You may like to incorporate this routine (or a similar one) in some of your future programs, as it is extremely useful in a program which relies on textual replies to questions, such as a quiz game, or a tutorial program. The routine can "scan" an answer to see if the appropriate "Keyword" was entered, ignoring all the other words.



CHAPTER 10

From Theory Into Practice

This chapter covers complete examples of programs in use, both games and programs of a more serious nature.

The chapter is divided into four sections. In the first section, you will find ways of making the computer unpredictable, and see how this can turn the ZX81 into a tireless opponent.

Each of the other sections looks at a full program contained on cassette, showing you what the program sets out to achieve, and how I have set about writing it. You will be gradually led through the program, investigating the more important aspects of its workings. Section 3 in particular also covers the idea of an interactive program, introducing an important new function.

10.1 RANDOM NUMBERS

10.1/1 HOW THEY ARE OBTAINED

One stumbling block that prevents you from wading in and writing your own games is that the computer is *predictable*. Any program that you have written up until now has consisted of predictable results from your own input. As an example, the early area programs like "AREA1" always produce the same results from the same input (the area of 20*10 must always be 200).

If we want to play against the computer, then it must become *unpredictable* – it isn't much fun playing games with someone when you know what the next move will be!

The reasons for winning a game are many – sometimes it's pure luck, sometimes skill, and sometimes because the other player made a stupid mistake.

Once again, this chapter concentrates mainly on complete programs so that you can see something larger than just a few lines strung together. It is important at this stage that you can work your way through a program to find out how it works and then be in a position to make some changes to it so that it works the way *you* want it to.

First let's consider how to make the computer unpredictable. This is where *random numbers* come in.

What *is* a random number? Whether or not you have any premium bonds, you are probably aware of ERNIE, the machine that selects winning premium bond numbers at random. It does this by generating a totally random number, which is then converted into a bond number. The ZX81 can also give a number at random. You can, therefore, use a random number to start a game off – the outcome is then unpredictable since you have no idea what number the ZX81 is going to pick.

The function

RND (under the T key)

gives a random number somewhere between 0 and 1 (there is no expression following the function name).

Every time **RND** appears in a program a different number will be given. Try this example program:

```
10 SCROLL
20 PRINT RND
30 GOTO 10
```

When you run it, you'll see loads of different numbers appearing – yet the program remains the same! Notice how each number is smaller than 1, and none of the numbers are negative. You could sit all day watching them, but you wouldn't spot any pattern.

But surely the ZX81 can't make things up on its own? If it could, then the conclusion to reach is that someone has built a computer that can think for itself!

Well, don't worry – they're not going to take over the world just yet! The ZX81 gets its random number from a big list of numbers that it can work out using some complicated mathematical formula. This list is extremely long, which is why you can't see any pattern when the program above is running.

This series means that you can get the same set of events twice – whenever you load a program after switching the ZX81 on, the **RND** function will start giving the same set of numbers. Fortunately, there is another way of making **RND** give different results even under the same conditions. There is a command called

RAND (on the T key)

and it allows the ZX81 to start its random numbers at a different point in the series. This way, the starting point is determined from the TV – the ZX81 counts how many screen frames have been sent out since it was switched on. And that's *bound* to be random!

You can also use the **RAND** command to make the ZX81 give the same set of random numbers each time – if you put any number following the command **RAND**, it tells the ZX81 to start at that number in the series. Try this:

```
 5 RAND 1
10 FOR X=1 TO 20
20 PRINT RND
30 NEXT X
```

You can run this program over and over again, and it will always print the same numbers out.

So why have random numbers and then make them print the same set over again? It doesn't make sense, does it?

Yes – in certain programs, you want to be able to do the same things twice – like starting a game again with the same board positions so that you can see where you went wrong. Yet you would still want the same "random moves" to take place.

Question

The **RND** function always gives a number between 0 and 1, although it cannot take the value 1. Can you write a program that will give a random number between 0 and 20 in a variable named R?

Answer

```
10 RAND
20 LET R=RND*20
```

Line 10 is not strictly necessary, but I've put it in for now.

The program works because **RND** is always less than 1, and 1*20 is 20 – so the highest value that R could take is 19.9999.

How did you do? If your answer was wrong, don't worry – it was merely an exercise in thinking.

10.1/2 USING RANDOM NUMBERS IN GAMES

Let's just study what games programming is all about.

Basically, there are three types of games programs, and I'll give you an example of each type so that you get an idea of what I mean:

strategy games	Chess is a prime example of this type of game. It involves a high degree of skill to write good games in this category, and apart from one or two small games like "Nim" or "Noughts and Crosses", most will be beyond the limits of the ZX81 without the use of the 16K RAM pack, as these programs tend to be large.
"hidden code" games	"Mastermind" and "Hangman" are typical of this category, in fact any game which involves you in discovering something that the ZX81 has previously hidden.
interactive games	The classic interactive game is "Space Invaders", where the game is constantly moving and the player can control the game with various "movement" keys.

This chapter deals with two games of the "hidden code" and interactive category, since strategy games are heavily dependent on the particular game, with general rules hard to identify. If you find yourself becoming interested in strategy games, there are many good books written on the subject, mostly available from mail order firms.

10.1/3 A WORD OF ADVICE

You should avoid, if possible, falling into the trap of using your ZX81 only to run games and programs that you find in magazines and books. Try to create your own. They're so much more fun when they are your own handiwork, and they'll have taught you much more than just simply how to copy from a typed page. Of course, looking at other people's programs is an essential part of learning how to write your own, but it is very easy to forget that you have bought something a long way removed from a TV game!

10.2 A FULL PROGRAM (1)

The first program we're going to study is a "hidden code" game. One of these – the most widely run on computers – is "Mastermind", and I have put a version of this into the course.

Since the game is quite large, the game will give report 4 after 10 attempts at breaking the code. If you are unaware of the rules of Mastermind, then read on. If you know the rules, carry on with the next section.

10.2/1 THE RULES

This version of the rules is fairly condensed, and you are advised to buy the game (it's only a couple of pounds) if you want the full set.

The computer chooses four coloured pegs at random. Your job is to guess what the four coloured pegs are. The pegs can be coloured Red, Orange, Blue, Green, Yellow or White.

You make a guess at the hidden code, and the ZX81 tells you whether your guess is correct or not. However, the answer also gives you clues as to whether or not you are on the right track.

If your guess has a peg of the same colour in the same position as the ZX81 code, then you get a black peg in return. If your peg is of the correct colour but in the wrong position, then you are given a white peg in return, When you are given four black pegs, you have won (since you have got all the right colours and the right order). As an example, let's suppose the ZX81 has hidden the pegs "Blue, Red, Green, Yellow". The order is important.

Hidden code: Blue Red Green Yellow

Your guess	Reply	Reason
B W Y O	1B and 1W	Your guess of blue corresponds to the ZX81 hidden code. It is also in the correct position, so you get a black peg (1B). The White is wrong, as is the Orange. Yellow is a correct colour, but in the wrong position, so you get a white peg in return (1W).
B G R Y	2B and 2W	Blue and Yellow are both in the correct position so you get two black pegs (2B), but although Green and Red are the correct colours, they are the wrong way round, so you get two white pegs for these (2W).
B R G Y	4B and ØW	You have won. All the pegs are the correct colour and correct matching position.

Not all the games are as quick and easy as that example!

10.2/2 THE PROGRAM

Load the game by:

LOAD "MASTERMIND"

Run the program. The ZX81 hides a four colour code. The available colours are: Red, Green, Blue, Yellow, Orange, White.

You only enter the first letter. The ZX81 asks you for your guess, and you type in a four-letter string consisting of the initials of the four colours you wish to guess. The ZX81 tells you how many black and white pegs you would receive for that guess.

As an example:

GUESS: **RGBY** 1B+ØW
GUESS: **OYBW** 1B+2W

and so on. Enter NEWLINE after each guess. If you use more than 1Ø guesses, you'll get report 4, but you can continue by typing **CONT**.

When you've played it to death, have a look at the listing of the program. There is nothing new at all in this game – everything in it has been covered in previous chapters.

Try to understand some of the various routines in the program so that you are able to use them yourself in other programs.

Question

These questions are to make you think about the workings of the MASTERMIND program. There is no point in just playing the game – it is also there for you to study and learn from. Anyone can copy games and play them, but the *real* fun is in writing your own. The purpose of including a couple of games is to give you a bit of fun *while you learn*.

(i) Take a look at lines 50 to 70 in the MASTERMIND program. Can you say what they are doing?

(ii) What is the purpose of line 610?

(iii) What about line 140 – why is it there?

Answer

(i) The **FOR** loop builds up the hidden code in string variable C$. It selects a random letter from string Z$ (which contains the available colours) and adds it to the end of the C$ string so far. The **INT** function makes sure that a whole number is given after multiplying **RND** by the length of the Z$ string. Adding 1 to it makes the answer a number between 1 and **LEN** Z$. Otherwise you would get a report 3 on occasions where the program would try to take a letter out of Z$ which didn't exist.

(ii) This line checks to see if the last guess gave four black pegs as a reply, and if so, then it stops the program. Four black pegs means that your guess was correct. You have therefore won.

(iii) Line 140 makes sure that your guess contains the correct number of letters – i.e. four. If you only type three colours in a guess, or five, then this line will catch you out and ask you to enter another guess. Notice that the program does *not* check whether the letters in your guess are valid colours.

How did you fare with those? They were deliberately searching so that you would spend a bit of time understanding how the program works. If you were struggling, then perhaps the solutions have helped you a bit.

If you had trouble following what was going on, then perhaps you should refer back to one of the chapters that introduced the topic causing you a problem. Chapter 3 covered **GOTO** and **IF**, while Chapter 4 introduced **FOR**. Strings were mainly dealt with in Chapter 8.

It may have taken quite some time to understand the program – a couple of hours even, but understanding is the key to getting on.

10.3 A FULL PROGRAM (2)

10.3/1 INTERACTIVE PROGRAMS

Now for a look at an interactive game. This category is currently the most popular since the introduction of "Space Invaders", but in the long term, these games can become rather boring once the method has been broken. They normally involve quite nice graphics (look back to Chapter 5 for these) and can give you some "instant" fun. Since this type of game is being continuously played while looking at the screen, it must be written to run in a certain mode.

Question
Which mode must the ZX81 run in to play interactive games?

Answer
SLOW mode. This is not strictly true, as some games can be written which try to get round the problem of the display going blank in **FAST** mode, but it's never quite the same! Reserve **FAST** mode for games and programs which do not need to be constantly watched – and also for editing programs.

10.3/2 HOW TO INTERACT WITH A PROGRAM

There is one other problem associated with interactive games – as they are being played, you are able to press certain keys to alter the movement in some way. How can we do this? If an **INPUT** command is used, the whole program will grind to a halt!

The answer is a function called . . .

INKEY$ (under the B key)

This function is extremely valuable for interactive games. Notice that it has a $ symbol in its name, so it must be a string function. What does it do?

The **INKEY$** function has a look to see if any key is being pressed, and if it is, then it gives the character corresponding to that key. If there is no key being pressed, then it gives an empty string – length zero.

Try this:

```
10 PRINT INKEY$;      (don't forget the semi-colon)
20 GOTO 10
```

It will print whatever key you now press. Notice several things about **INKEY$**. If you press more than one key at a time, then the ZX81 will not let this through. The program stops just as if you were not pressing any key at all. Secondly, that it keeps going for as long as you keep your finger on the key. This is because the **INKEY$** function has a look *every time it is obeyed*. This gives you an idea of how fast the ZX81 is running in **SLOW** mode!

Question
Write a program that sits with a blank screen. Whenever a key is pressed, the program prints "GET OFF" in the centre of the screen. As soon as the key is no longer pressed, the screen goes blank again.

Hint: look at page 129 of the Sinclair Handbook.

Answer
Here's one answer for you to study:

```
10 PRINT AT11,13;"       "      (print 7 blank spaces in the centre of the
                                screen)
20 IF INKEY$<>" " THEN PRINT AT 11,13;"GET OFF"
30 IF INKEY$<>" " THEN GOTO 20
40 GOTO 10
```

A slightly more elegant solution is:

```
10 PRINT AT11,13;"       "      (7 spaces again)
20 IF INKEY$=" " THEN GOTO 10   (if no key pressed, wait)
30 PRINT AT 11,13;"GET OFF"     (give fair warning)
40 GOTO 20                      (have a look to see if it's still
                                there . . .)
```

How did your answer work? If it went well, then you've obviously got the message with **INKEY$**.

If you couldn't get it to work, or you just didn't understand, then read section 10.3/2 once more.

The first solution has a slight problem in it. Lines 20 and 30 both use the **INKEY$** function, but as you have already been told, this function gives the key character being pressed *at that time*. So it is quite possible for a different key to be pressed between lines 20 and 30. It doesn't make any difference to this particular program as it isn't concerned with *which* key is pressed, only that a key *is being* pressed.

Another program might not work if the two **INKEY$** functions gave different results, so whenever the program is checking to see which key is being pressed, first store the **INKEY$** value in a string variable and test that instead. The first solution would then be written as:

```
10 PRINT AT 11,13;"       "
20 LET K$=INKEY$
30 IF K$<>" " THEN PRINT AT 11,13;"GET OFF"
40 IF K$<>" " THEN GOTO 20
50 GOTO 10
```

Personally, I still prefer the second solution . . . somehow it seems less clumsy, and although both programs work, part of the appeal of programming is one of taste. When you've become more familiar with programming, you'll find that some programs seem awkward and disorganised, while others have a sort of beauty and simplicity that catches your eye. Yet both probably work just as well.

10.3/3 THE PROGRAM

Now to another complete program. This is an interactive game for one player.

```
LOAD "ARCHERY"
```

The object of the game is to shoot an arrow into the target. You are positioned on the left-hand side of the screen (the blinking blob – that's not an insult!) while the target is on the right-hand side.

You may use the "6" key and "7" key to move up or down. When you think you are opposite the target, press the "0" key to release your arrow.

You can have another go by pressing any key – except BREAK, of course. Try the game out before you look to see how it's written. That way you'll have an idea of what's going on. Before you list the program (use BREAK to stop the program when you're fed up), type . . .

```
CLEAR
```

This way, all the variables used by the program are removed, and you'll get more of the program listing on the screen.

The program combines several techniques that we've covered throughout the course – here's a list of them and the corresponding line numbers:

feature	line numbers
functions	10, 20, 100, 400
random numbers	5, 10, 20
graphics/plotting	50, 60, 160, 180, 200–220
conditional statements	120–150, 310–400
FOR loops	200–220
interaction	100, 400
print formatting	300–320

Not bad for a fairly simple program?

Just to help you understand the workings a bit more, the target position is calculated in lines 10 and 20. Variable TY holds the vertical position, and TX the horizontal. The only reason for moving the target horizontally is so that you can't get used to "lining" the sights up on the edge of the screen all the time. It makes the program a bit more variable.

You could change the program so that the target moves one position up or down at random every 10 times round the main loop of the program. That means that you would also have to time the shooting of your arrow more precisely.

Another alteration is to make it so that the arrow will not go far enough if you wait too long before firing. This would give the game a more exciting pace.

The list is quite endless – each change will make the game slightly different in the way you play against it, some better, some worse, but it's a fun way of learning.

10.3/4 TYING UP A LOOSE END

There was one other program that was introduced earlier in the course – "TRACE". This program was not studied at the time as it contained several features that hadn't been shown to you. Now that you are almost in control of the ZX81 it is the right time to go back and take a closer look at this in a different light.

LOAD "TRACE"

If you can't remember what the program does, then I'm not going to tell you! It *still* contains a function that you haven't met and you won't meet this command until the next chapter. To put your mind at rest, lines 30 to 60 determine how big the "drawing board" can be on the screen. If you have a basic ZX81, then XL and YL will be set to 43 and 16 respectively. If you've bought the 16K RAM expansion pack, then XL and YL are set to 63 and 0. The reason for doing this is that the basic ZX81 would give error 4 if the drawing board took up the whole screen, so I have had to restrict it slightly. XL and YL are the maximum limit of the X and Y coordinates of the screen.

Question

This question is to try and make you think of how to use "moving" graphics in a program.

Write a program that makes a two-by-two black blob move across the screen from left to right. The blob can be either plotted or drawn using **PRINT** commands.

Answer
This answer uses **PRINT AT** to do the job:

```
10 CLS                                  (reset screen)
20 FOR X=0 TO 30                        (all the way across . . .)
30 PRINT AT 11,X;"■■"                   (print top row of blob)
40 PRINT AT 12,X;"■■"                   (print second row)
50 IF X=0 THEN GOTO 100                 (is it first time around?)
60 PRINT AT 11,X-1;"  "                 (if not, blank out last half of blob)
70 PRINT AT 12,X-1;"  "                 (on both rows)
100 NEXT X                              (all the way across)
110 IF INKEY$="" THEN GOTO 110          (wait for a key . . .)
120 GOTO 10                             (start again)
```

If your solution used **PLOT** and **UNPLOT** your blob will be only one quarter the size of mine and move half as slowly.

Did you forget to "wipe out" the trail as the blob moves across the screen? I added the extra lines at the end of my program so that I could keep running it just by pressing any key at all. This makes it slightly easier to use and saves hunting for **RUN** each time.

10.4 A FULL PROGRAM (3)

The third (and last) complete program I'm going to go through with you is certainly not a game. Unfortunately, the program will only work when a 16K RAM pack is fitted to the ZX81, so if you haven't got one, then I'm afraid you'll need to skip this section.

The program is for handling home finances, allowing you to run a budget account on your ZX81 instead of paying £20 per year for the privilege of the same thing from a major bank.

Each month, after you receive your bank statement, total the amount of all cheques drawn on your account.

Enter this figure, along with any additional income (the program automatically credits your basic income) and you will be given a complete balance sheet of your finances, including any standing orders paid, spread payments (budgets) and details of your income. After each run, the program is saved back on tape again with all the latest details of your account, so that the *next* monthly run can carry on from where it left off.

Initially, though, you need to enter the details of your current bank balance (as given at the end of your *previous* statement), details of all income, standing orders, and any budgeted items that you wish to spread the cost of over a period of months.

```
LOAD "FINANCE"
```

Run the program. There are many questions asked of you as you go through the program, and at each stage you should answer "YES" or "NO" (the first letter only is all that's necessary) according to whether you wish to alter some details or proceed to the next stage.

First, enter your current balance. Next, you will be asked if you wish to alter your income details. You should answer "YES" at this stage.

Each item has some text to describe it (this is printed on the balance sheet) and a value corresponding to it. You may alter the text, the value or both. Watch the questions on line 21 – they ask you if you wish to alter *text* or *value*. You do not need to answer yes or no, merely type the text or value and press NEWLINE.

If you wish to alter these details at a later date, you can leave the item unchanged by just pressing NEWLINE and proceed to the next item.

Try entering some dummy text and values initially, just to get the hang of it.

The next stage is to enter details of all the standing orders in your account. This would include items such as mortgage, rates (if paid monthly), HP agreements, insurance – in fact anything that is paid

monthly directly from your account. You need to enter the *number* of items, since the program sets up an array at this stage for the appropriate number.

Enter text and values in the same way that you entered your income details.

Last of all, you can (optionally) enter details of items that you wish to budget over a period of months. Items in this category would be gas, electricity, telephone, holidays, subscriptions, rates (if paid half-yearly) – anything with a non-monthly payment period.

Work out how much the item would cost if paid monthly, and enter this amount as the budget value (budget accounts in major banks adopt the same principle).

Once this has been done, you do not need to enter these details again – they are saved with the program each time. This means you can just change one item (e.g. your income) without affecting all the other details you've carefully entered.

Now the program gives you a "menu" of options. Choose one of these – you can:

(1) FINISH
(2) ENTER MONTHLY VARIABLES
(3) ALTER PRESENT DETAILS
(4) PRINT BALANCE SHEET

Normally, you will run option (2) to enter the date, any additional income this month, and the total amount of cheques drawn this month.

Run option (4) to print a balance sheet. Notice that the program keeps two totals – your actual bank balance (brought forward from the last run) and a "budget balance". This represents the amount of cash you would have if all your budget items were paid monthly – it's a bit like putting the money aside until the bill arrives.

All items are listed with a single code letter in the first column of the screen that indicates the type of item:

I income items
S standing orders
C cheques total
B budget items

The totals at the end show your new actual and budget balances.

When a bill arrives for an item that is budgeted, the money has effectively been taken out already (this is reflected in the budget totals), and so your *actual* balance will be reduced towards the budget balance.

This budget balance gives you a much more realistic idea as to the amount of cash you have to hand. The difference between the actual balance and budget balance is the total of pending bills – don't be tempted to spend this!

There will come a time when you realise that your estimates of budgeted bills were slightly inaccurate, and so you have more or less cash available than you thought. When you run option (2) to enter your monthly variables, you will also be asked if you wish to adjust the brought forward budget balance. Only alter it if you feel it is drifting wildly away after several bills have arrived.

Whenever you print a balance sheet, the program asks if you wish to update the totals. If the sheet looks OK, then answer "YES". Prepare a tape (not the one you've just loaded from – what happens if it fails to "save" properly?) and the program will automatically save itself.

At this stage, it also sets your cheque total to zero, and clears the date. If you now run a new balance sheet, you are seeing an estimate of next month's expenditure – a forecast of cash available. Don't save it again, just take note of the information it gives you. If you have a printer, then this program is ideally made for it!

I'm not going to unravel this program in the text – I will leave it entirely in your hands to see how I have approached a more serious use of your ZX81.

The only portion of it that is intriguing is the balance sheet. Notice that all the amounts are printed in neat columns with two places of pence (you should know by now that the ZX81 never prints decimal

places unless they exist), and the items are all aligned at the right-hand side, like this:

```
123.45
  2.00
 60.50
```

If these items were just **PRINT**ed, they would appear as:

```
123.45
2
60.5
```

Have a think about how you would achieve the same effect, then look to see how I have done it.

Summary

This chapter has been fairly easy in comparison with some earlier ones, but it has given you the chance to see three complete programs at work, and maybe you've had a laugh on the way!

Next you'll be learning a bit more about the way your ZX81 works and learn a small amount about two functions and one command that intrigue nearly every newcomer.

But here are the various points you should have learned:

– that the computer can be made unpredictable and that this can have benefits in creating certain types of programs.

– that a program can be written which relies on interaction to control the program flow.

– how the ZX81 (or, indeed, any computer) can be used to solve not just arithmetic problems, but also more general types of problems that occur in everyday life.

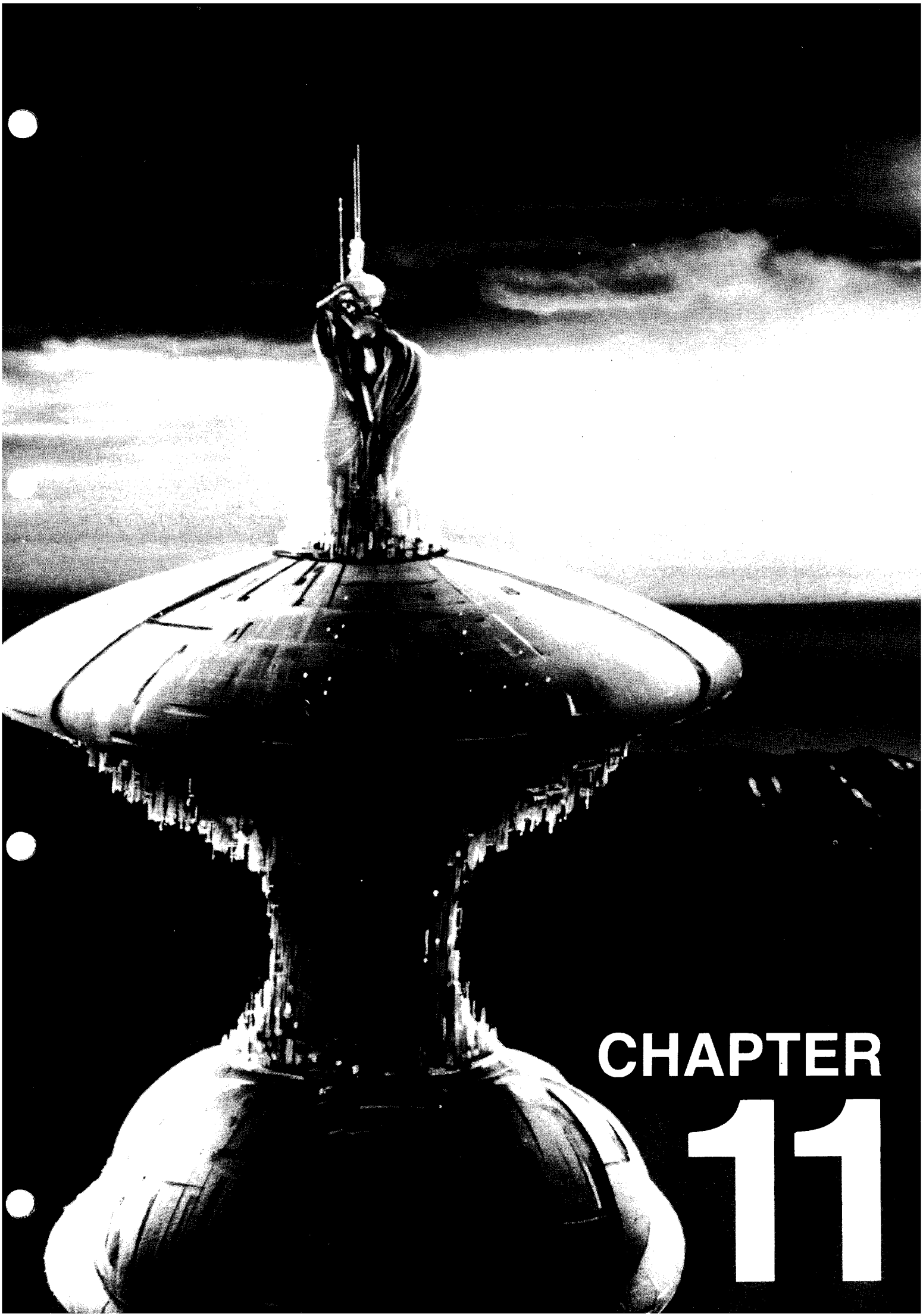
Exercises

1. Write a program to handle full metric conversion. The program should be capable of converting inches to centimetres (and vice versa), pounds to kilos, pints to litres. It would initially ask which type of conversion is required, then ask for a value. The converted result would be given.

2. On the games front, the Hangman game is quite straightforward. The program should contain a list of words hidden by one person. The program then gives a clue as to the number of letters in one of the words (perhaps by printing several question marks like this – ?????). Each letter is guessed and after nine attempts at guessing the letters, the program "hangs" the player. If you've got enough room, you may like to try to draw a picture of a man gradually hanging after each wrong guess!

3. If you're into statistics, then maybe a program to calculate the mean, variance and standard deviation of a list of data points will be up your street!

4. If you are a collector of anything (beer bottles, comics, etc), you may like to create your own indexing program, where the program holds details of the number of items held under various headings (e.g. brewery name, or printer's name).



CHAPTER 11

A Glimpse into Another World

This chapter is not really essential for you to read, as it does not contain anything that will further your programming capabilities. There are two functions and one command that we haven't yet covered, and so this chapter is intended to introduce you to them.

To be honest, at this stage of your "career", these new facilities are probably not going to be at all useful, but so that you don't feel left out, we're going to see what they do.

When you have gained confidence and wish to know more about computers in general – the ZX81 is only one of many different types – then I have included a list of further reading in this chapter, so that what you read now will be put into its proper perspective.

I'm not going to set any real questions in this chapter, but there are some exercises dotted along the way to make you think about what you are doing.

11.1 INSIDE THE ZX81

As you have seen in the Introduction, all the programs you have written so far have been written in a computer language called Basic. A computer cannot understand English.

In order to let you write your programs in Basic, someone has had to write a whole program that converts your Basic commands into the sort of commands that the ZX81 can understand. This program is called the "Basic Interpreter", and I'm not going to define it any more than that. It is held in the ROM memory inside the ZX81.

A computer has its own memory – used to hold your programs and variables – and this memory has a limit. It is broken up into tiny sections that can hold numbers, each of which can be between 0 and 255. Not a very large number. Each of these tiny sections that can hold these small numbers is called

a *byte*

The basic ZX81 is supplied with 1024 of these bytes (this corresponds to the 1K RAM that you met in the Introduction) and if you buy the memory expansion pack, you'll have 16384 of them (hence, the 16K RAM pack, since 1024*16=16384).

You can store numeric values in these "bytes", but you need to be careful, as the ZX81 is already using a lot of them to hold your Basic program and variables in. If you know what you're doing (at the moment you don't!) you can find a spare bit of space that you know the ZX81 isn't using and use it for yourself.

The command to store a value in one of these "bytes" is

POKE (on the O key)

You need to tell **POKE** two things. First, which "byte" it is that you want to put the value in and second, the value that you want put into that byte. But remember that the values you store can only be between 0 and 255. Let's try a small example.

Type this command:

POKE 16507,100

The value 16507 is the number of the particular byte we are addressing, while the value 100 is the value we want to store in this byte.

As an aside, although the basic ZX81 only has 1024 (or 1K) bytes, the "address" of the first one is not number 1. It is 16384. So the range on a basic ZX81 is from 16384 to 16384+1024 (equals 17408).

That command above has the effect of "Take the number 100 and store it in the byte at memory address number 16507".

It may be useful to store numbers somewhere, but unless we can get them out again it seems a bit

pointless. We can take the value back out by using

PEEK (under the O key)

The numeric expression that follows this function is the address of the byte we wish to examine – in our case, 16507.

Since it's a numeric function (no $ symbol) we can assign its value to a variable or even print it.

Try this:

PRINT PEEK 16507

Have you got 100 again? This shows that the 100 you put into 16507 with **POKE** can be taken out again with **PEEK**.

Try to think of these bytes as a massive array where the first element is not number 1 but number 16384. It's sort of like writing:

DIM ZX81(1024)

except that this is not a valid command (so don't bother typing it in!). It might help you to picture what we're talking about.

Exercises

Try these:

(i) **POKE** 16437,100
PRINT PEEK 16437

This tells you that you must know where you're **POKE**ing things, otherwise they'll change unexpectedly!

(ii) What would you expect **POKE** 16507,300 to do? Try it and see if you're right.

(iii) **POKE** 16397,1

This tells you that **POKE**ing things without due care and attention can cause the ZX81 to pack up! Switch if off and on again.

11.2 MACHINE CODE PROGRAMMING

So how can these two items be used? The answer is firstly that you must know what you're doing and at this stage you don't, and secondly that they are mainly used to allow people to enter programs that have been written in not Basic, but something called "Machine Code".

Since this is beyond the scope of this course, I can only refer you to the list of further reading. If you become interested in this type of work, then these books will help you to understand how machine code works and how you can write your own.

There is one more function that hasn't been covered. Since **POKE** is a way of getting a machine code program into the ZX81, how can it be run? The **RUN** command always runs Basic programs.

The answer is to use the function:

USR (under the L key)

The expression that follows is the address of the first byte containing the machine code program.

So you could write something in machine code (instead of Basic), then **POKE** it into some parts of memory that aren't being used and run it by stating:

LET X=**USR** (address of first byte)

You are forced to write "**LET** X=" since **USR** is a function and cannot stand alone in a statement like a command.

11.3 THE ZX81 SYSTEM VARIABLES

One other main use of the **PEEK** and **POKE** is that of altering the ZX81 *system variables* in some way.

These variables are used by the ZX81 to help it look after your Basic programs. Some of them contain information which a program can make use of – one example of this is the program "TRACE" that we studied in Chapter 10. These variables do not have names in the way that you are accustomed to – they are held in bytes in memory. The only way to get at them is to **PEEK** them.

Chapter 28 of the ZX81 Handbook gives an excellent description of these variables, what they contain and how you might use them. It also tells you whether it is safe to **POKE** anything into them.

In the program "TRACE", address 16389 was looked at to see how much memory the ZX81 contains. If this number is greater than 7Ø, then the 16K RAM pack must be attached.

For now, we will not get too involved in how these can be used, as a deeper understanding of the way a computer works is needed to gain full benefit from all this.

Summary

This chapter has probably left you hanging in the air a little, but for now, that's the way that it has to be. To take you into the inner realms of the ZX81 would require a book larger than this! All this has attempted to do is give you some idea how these particular commands and functions are used.

If you wish to become more involved in the way the ZX81 (or, more particularly, the Z80 microprocessor) works, then you will need to become familiar with the following topics:

1. How a microcomputer works (at a high level). This is dealt with in a very readable manner in the books:

   Introduction to Microcomputers Volumes Ø and 1 (Adam Osbourne, Osbourne Associates)

   These books show you how the various "building blocks" of computer systems fit together. They also show you how a typical microcomputer is programmed at machine code level.

   You may find portions of these books rather technical, and if you have no previous knowledge of electronics or its terminology, you should study the beginners' articles contained in the monthly computer magazines before tackling them.

   You should be able to use binary numbers (addition and subtraction), and also understand what is meant (even if you can't use it!) by Boolean logic.

2. The instruction set of the Z80 microcomputer. This can be found in:

   Z80 Assembly Language Programming (Lance Leventhal, Osbourne Associates)

   which will show you how each of the machine code instructions works, and how it can be used within larger machine code programs. It is not necessary to read the books mentioned above first, but you may find a few references in this book to general topics covered in those above.

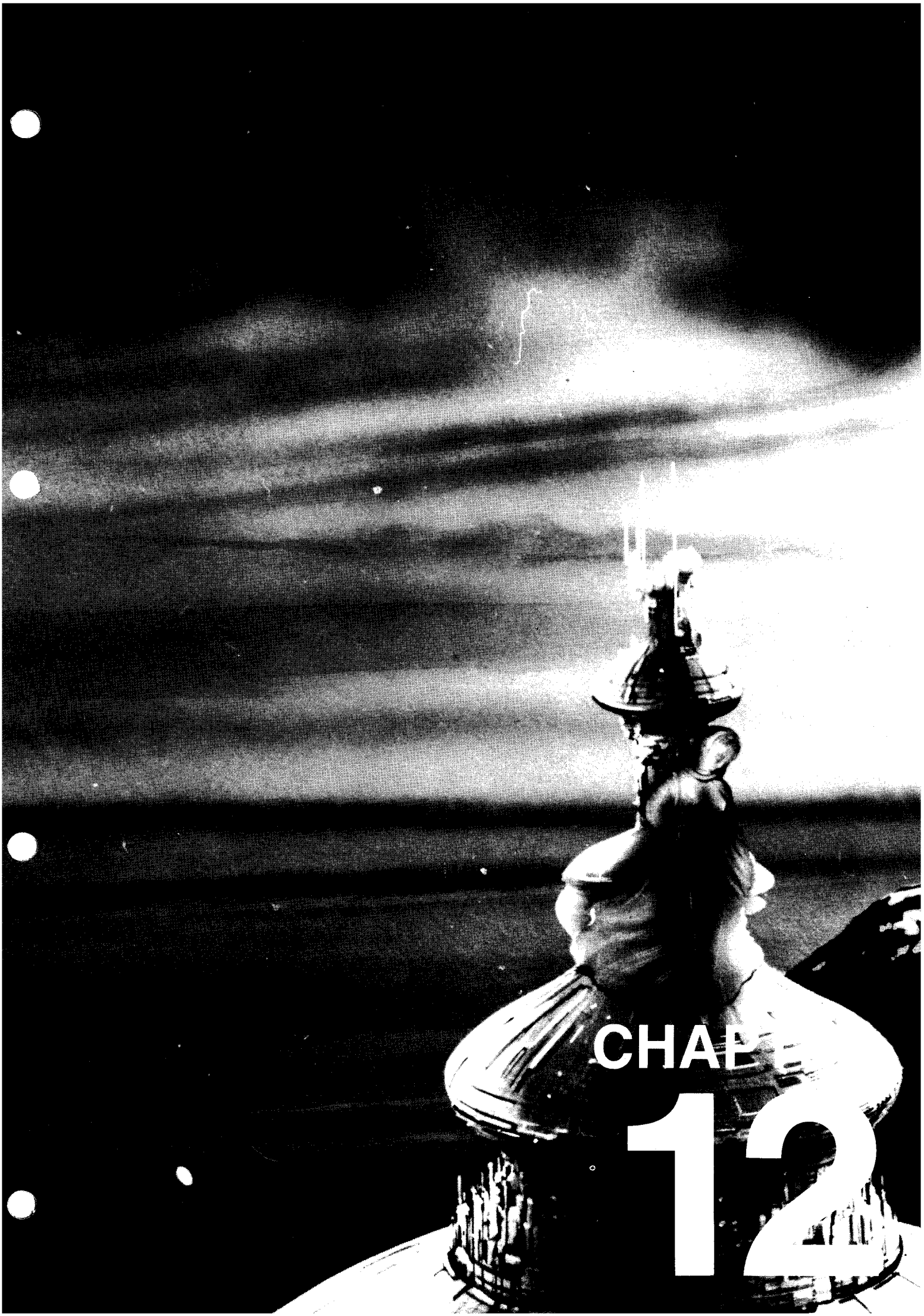
Next we'll round off the course with some hints and tips, plus a few problems to keep you going. Meanwhile, these are the various points covered in this chapter:

– that all your programs so far have been written in a language called BASIC.

– how the memory of the ZX81 is divided up into bytes, and how each of these bytes is addressed by a unique number which can be used in the **PEEK** and **POKE** keywords.

– that you may find extra hidden depths in your ZX81 by learning about "machine code programming".

There are no further exercises for this chapter.


CHAP
12

And Finally . . .

Well, here you are at the end of this course. Before you embark on a lifetime of programming, there are a few final hints and tips that I would like to pass on to you. I have deliberately left them until the end in order to make life simple for you to start your first programs. Let me give you some hints. I'll go through them in the same order as the Chapters.

Chapter 2

In this chapter you were introduced to the **REM** command. This is an important command. It helps you to remember what a program does and so I cannot stress enough how much better your programs will be (not only for yourself) if you use plenty of **REM** statements in any program you write. Try to put **REM**s at the start of the program (telling what it is), at the start of each subroutine (so that you can see what it does), and before any lines in a program that are particularly important or complicated.

This leads to a dilemma on the basic ZX81 since **REM** statements eat into your available program memory space. If a program is large, then obviously the **REM** statements should be the first items to go – but still try to write your programs with **REM** in mind.

If you already have a 16K RAM pack then it's highly unlikely that you'll run out of program space – keep putting those lovely **REM**s in!

Chapter 3

Throughout the course I have consistently told you that the value of "true" is 1. This is not strictly correct – when testing for "true" in a condition (or **IF** statement), "true" is actually taken to be anything other than 0, so that a statement such as **IF** 33.5 **THEN PRINT** "TRUE" will work as seen. Try it!

A conditional expression, such as **LET** C=(H>5), will, however, only give a value of 0 if false, or 1 if true.

The following hints cover several chapters, but I've lumped them together under Chapter 3 since they all revolve around the use of **GOTO**.

This command can get your programs into quite a bit of trouble if you're not careful with it.

Avoid (like the plague) these conditions:

NEVER USE **GOTO**:

(a) to jump into a **FOR** loop. If the **FOR** command isn't actually executed you might get some strange things happening. It is possible that you'll get an error, but under certain conditions you will not – so watch out!

(b) to leave a subroutine. ALWAYS use **RETURN**, then sort out what you want to do after you've come back from the subroutine. This especially applies if you have nested your subroutines, since the next **RETURN** would be treated as if it were from the last **GOSUB**, and the program would end up in a horrible mess.

(c) to jump *into* a subroutine. The same rules apply. Since the ZX81 keeps a track of each **GOSUB**, if you suddenly jump into a subroutine, the ZX81 will find an extra **RETURN**. Where will it go to? Who knows . . . So take my advice and don't do it.

Chapter 4

This chapter showed you how to use **FOR** loops and also how they can be nested. A common fault in programs crops up when nested **FOR** loops overlap each other – like so:

correct	***incorrect***
FOR Y=1 **TO** 10	**FOR** Y=1 **TO** 10
FOR X=1 **TO** 32	**FOR** X=1 **TO** 32
PRINT X;	**PRINT** X;
NEXT X	**NEXT** Y
NEXT Y	**NEXT** X

Can you see why? If you work them through in the way we did in Chapter 4, it'll stand out a mile.

Chapter 7
Subroutines – use lots of them! They really help to make your programs more manageable. Look to see how other people write programs and how they've worked subroutines into the program structure.

General Comments
Using meaningful variable names can make programs easier to decipher – once again the basic ZX81 can quickly run out of program space; just like **REM**, this is one area that needs to be carefully traded off.

There are several places that you may find further writings on the ZX81. You can join one of the ZX81 User Groups (see the monthly magazines for details – there are often articles written about the ZX81 which will give you an address). Alternatively, many book shops and mail order book companies now stock books written specifically for the ZX81.

BASIC comparisons

This appendix sets out to highlight some of the features found in other versions of BASIC, and how they may be coded to work on the ZX81.

In the world of microcomputer systems, Microsoft BASIC tends to be the most popular version available, and most comparisons made of any versions of BASIC tend to be made against the Microsoft version. This does not necessarily mean that the Microsoft BASIC is the "best", but merely that it is the most widely-used and therefore the version you are most likely to come into contact with on another system.

The various features are each noted, with a brief explanation as to its use, then a possible way of programming the feature on the ZX81.

DEF FN

this feature allows you to **DEF**ine your own **F**unctio**N**s within BASIC, being made up of a combination of any numeric or string expressions (the combination must be syntactically correct!). A single letter usually follows the **FN** to identify the new function name. A parameter normally follows the definition in brackets, allowing the function argument to be entered into the definition. Wherever **FN**x is then used throughout the program, the new function is invoked. This can be written on the ZX81 by expanding all **FN**x statements back into their original form. E.g.:

```
10 DEF FNR (X)=INT(RND*X)+1
20 LET A=FNR(100)
```

. . . can be written on the ZX81 as:

```
20 LET A=INT(RND*100)+1
```

DATA

allows numeric and string valued expressions to be stored within a program, e.g.:

```
10 DATA 21, "FREDDIE",2.5E03
```

The data can be assigned to variables using the **READ** command.

READ

takes the next data item from the **DATA** statement(s) and assigns it to the specified variable, e.g.:

```
10 DATA 21, "FREDDIE",2.5E03
20 READ NUM1 (takes 21)
30 READ N$   (takes "FREDDIE")
40 READ NUM2 (takes 2.5E03)
```

DATA and **READ** can be combined on the ZX81 by using **LET** statements to assign the variables directly. Although this is not as convenient as **DATA** & **READ**, it works just as well in practice.

RESTORE

resets the **DATA** queue back to the beginning, so that the next item to be **READ** will be the first **DATA** item again. This would be done on the ZX81 by resetting the variables to be assigned once more.

TRON/TROFF (or **TRACE**)

allows the line numbers of a running program to be displayed on the screen as each line is obeyed. This is extremely useful for debugging programs and has no equivalent on the ZX81.

LET

this can normally be omitted in most other versions of BASIC, so that:

```
10 LET X=23
20 LET Y=X*255
```

. . . could be written as . . .

```
10 X=23
20 Y=X*255
```

You must always use **LET** on the ZX81.

ON x **GOTO** . . . this statement is found in many other versions of BASIC, allowing a program to transfer control to several different line numbers depending on the value of a single variable:

```
100 ON N1 GOTO 200,300,400
```

This will cause BASIC to go to line 200 if variable N1 has the value 1, line 300 if N1=2, line 400 if N1=3. You can very easily rewrite this on the ZX81 by using a conditional **GOTO**, e.g.:

```
100 GOTO 100+(N1*100)
```

Where line numbers become more complex, you may need to write this as a series of **IF** statements:

```
100 ON N2 GOTO 27,33,14
```

. . . would become . . .

```
100 IF N2=1 THEN GOTO 27
101 IF N2=2 THEN GOTO 33
102 IF N2=3 THEN GOTO 14
```

. . . or alternatively . . .

```
100 GOTO (N2=1)*27+(N2=2)*33+(N2=3)*14
```

ON x **GOSUB** . . . this can be treated in a similar fashion, except that you should use **GOSUB** instead of **GOTO** throughout.

WHILEx
WEND allows a program to obey all instructions in between the **WHILE** and **WEND** statements while condition x is true, e.g.

```
20 WHILE X>0
30 PRINT X
40 X=X-1
50 WEND
```

This can be written on the ZX81 by specifying the *reverse* of condition x within a **FOR** statement, and using a **GOTO** to replace the **WEND**:

```
20 IF NOT X>0 THEN GOTO 60
30 PRINT X
40 LET X=X-1
50 GOTO 20
60 . . .
```

FRE(0) this is a numeric function which gives the amount of available memory at the time of using the function.

INPUT$(n) is a string function which asks for n characters to be input from the keyboard. Once n characters have been accepted, the program continues with no need to press newline. You will need to write this using the **INPUT** command, or a loop containing the **INKEY$** function, e.g.:

```
 5 LET X$=""
10 FOR X=1 TO N
20 LET C$=INKEY$
30 IF C$="" THEN GOTO 20
40 LET X$=X$+C$
50 IF INKEY$<>"" THEN GOTO 50
60 NEXT X
```

LEFT$(s,n) a string function which supplies the leftmost n characters from string expression s. You can easily write this as:

```
200 LET X$=S$( TO n)
```

MID$(s,n,m) also a string function which gives a multi-character slice from string expression s. The function takes m characters starting from character n. This can be written as:

```
200 LET X$=S$(n TO n+m-1)
```

RIGHT$**(s,n) similar to **LEFT$, this string function supplies n rightmost characters from string expression s. Can be written on the ZX81 as:

```
200 LET X$=S$(LEN S$-n TO )
```

String handling

You have already seen above that Microsoft **BASIC** allows the use of the **LEFT$ MID$** and **RIGHT$** functions to obtain string slices. The ZX81 is particularly unconventional in the way strings are manipulated since it uses the qualifier (n **TO** m) to obtain a string slice.

Some BASICs do not allow string arrays at all, while others adopt a similar method to that of the ZX81, although all strings must be dimensioned – even if they are not arrays. String slices are then used by the qualifier (n,m) instead of (n **TO** m). For example:

```
10 DIM S$ (10)
20 LET S$ (2,5)="ABCDE"
```

This sets the second to fifth characters of string variable S$ to the expression "ABCDE".

File handling

Most BASICs allow the use of floppy disk or cassette tape storage. When these facilities are available, there are usually three or four commands that allow manipulation of the records:

OPEN
which allows you to access records under a name of your own choice (called a *file name*)

READ or **INPUT** #n or **GET**
which allows a data record to be transferred from the disk or tape into the computer

WRITE or **PRINT** #n or **PUT**
which allows a data record to be transferred from the computer onto tape or disk.

CLOSE
which shuts the "file" from further access.

It would be inappropriate to go into further detail at this stage as most computer systems have slightly different methods for storing and retrieving data records.

PEEK and **POKE**

These two features are commonly found on other computer systems, but programs using them are normally not able to run on a system of a different type.

Beware of trying to convert a program using these features to run on your ZX81 (and vice versa) without first understanding what effect they are having upon the program.

Multiple statements

Most versions of **BASIC** allow you to write more than one statement on the same line. Each statement is then separated usually by a colon, thus:

```
10 FOR X=1 TO 20 : PRINT X : NEXT X
```

If you are converting a program which uses multiple statements, you will need to expand these into individual lines once more. Be careful of the line numbers that you use!

```
10 FOR X=1 TO 20
11 PRINT X
12 NEXT X
```

These multiple statements can cause some confusion when an **IF** statement is used:

```
250 IF X>0 THEN LET Y=5 : PRINT "HELLO" : GOTO 270
260 . . .
```

If the condition is true, then *all* the following statements are obeyed, while if the condition is false, *none* of the statements are obeyed. On the ZX81, this would be written as:

```
250 IF NOT X>0 THEN GOTO 260
251 LET Y=5
252 PRINT "HELLO"
253 GOTO 270
260 . . .
```

Common Problems and Solutions

This section will hopefully sort out any troubles you might have while you're getting going. A quick index is given below so that you can try to identify your problem

B.1 Cassette tape loading

Initially, this can be quite a problem but once you have got your ZX81 and tape recorder talking to each other, your problems will be unlikely to recur.

Here are a few things to check:

1. Make sure that the leads are plugged into the correct sockets. This is quite an easy mistake to make. The sockets are marked on the ZX81 to show you where each end of the leads should go to.

2. Are you using jack sockets (3.5mm) or have you bought an adaptor to plug into a DIN socket? If you're running from a DIN socket, then that's the reason why you're having trouble. DIN sockets do not give enough signal strength for the ZX81. Try using the jack sockets (if your recorder has them) or the loudspeaker extension socket instead of the earphone socket. Whatever you do, though, do **NOT** connect the ZX81 up to the speaker output from a hi-fi amplifier.

3. Can you "see" the programs on the TV screen as the ZX81 tries to load them in from the tape, or does the screen just stay the same all the time? If you can't see any change in the screen, then either the leads are faulty (unscrew them to check) or the tape itself is blank, or the output socket from your recorder is not working. Try listening through an earphone to find out if you can hear the "super-charged bumblebee" sound. If, however, the screen *does* change as it goes past a program on tape, then it probably means that your cassette recorder heads are out of alignment. Take the unit to a reputable dealer for checking.

4. Have you tried to load programs at various volume settings? Sometimes it helps – once you've established a setting that works, mark it and keep to it.

Chapter 16 of the Sinclair Handbook gives more information.

B.2 Saving programs on cassette

First of all, what makes you think that saving programs (as opposed to loading) is giving you a problem? Can you hear anything on the tape when it is played back? If you can, then read the section on loading problems.

If you can't, then check that you've got the leads in properly, and also unscrew them to make sure all the connections are sound.

Make sure you're plugged into a MIC socket and not the AUX input to a hi-fi cassette.

B.3 Errors that won't go away

This problem comes in many disguises. If you're having troubles when editing a program (nothing happens when you press EDIT) then you're up against "Error 4 – not enough memory". See the next section.

Have you looked at the section on Error Report Codes to see what exactly any report code is? If you can't see what's wrong with a particular line of program yet still the ZX81 throws it out, try to write it differently – it may take two or three lines instead of one, but at least it'll work.

If your problem is a syntax error and you can't see why, then the same applies. RUBOUT the line and try a different method.

A common cause of errors is forgetting that many words have a single key and trying to type the word out in full. This causes a syntax error — but the line *looks* OK.

B.4 Error Report 4 – Not enough memory

This most often occurs when you're trying to edit a program – all of a sudden nothing happens when you press **EDIT**. There are several ways of getting round this, but first a word in your ear.

Whenever you see the bottom of the screen starting to rise up when you're entering a new program, you should realise that you're running out of room. Save the program on tape at this point, as it's starting to become an awful amount of typing to have to put back in again.

The only way to get a line for editing is to find a way of making space on the screen for the line to be put at the bottom. You can do this in several ways. The easiest two methods (I find) are:

1. First press **NEWLINE** so that the ZX81 does not have any rubbish around that you can't see. Then type **CLEAR** (obviously, you can't use this method if you want to save some variables – try **CLS**). This gives you a nice blank screen with just Ø/Ø at the bottom. Now type **EDIT**. The line pointed to by the cursor will now appear (on its own) at the foot of the screen. You can edit it and put it back again quite happily.

2. Once again, press **NEWLINE** to clear out any rubbish. Then type **LIST** nnnn (where nnnn is the line number you're trying to edit). You'll get some sort of report code at the foot of the screen. Now type **EDIT**. The line will now appear for you to edit as normal.

Whenever a program gets into this condition, you're probably beginning to overstep the memory limits of your ZX81. If you haven't got a 16K RAM expansion pack, then now is the time to consider getting hold of one. If you *have* got one, then it must be an enormous program you're writing! Perhaps it's reached the point where you should sit down to see if you've written it in the most efficient manner.

If methods 1 and 2 fail, it means you haven't even got enough room for a single command to be obeyed – in which case, I'm sorry to have to say that the program is just too big. Maybe it could be written in a different way.

BASIC Command Summary

Each paragraph identifies a Basic command or function. Examples of use are given, and also any useful "tricks" associated with it. The list is in alphabetic sequence.

Some of the information given in this section has not been covered within the main body of the course (e.g. making use of **POKE** and **PEEK**), but is intended as a source of reference.

The paragraph headings show:

– Command/function name
– Command type
– Full syntax, if not obvioius

Throughout this section, the following abbreviations are used:

n	numeric expression
m	numeric expression (to avoid clashes)
s	string expression
v	variable name
<S>	statement
e	expression (numeric or string)
[ ]	indicates an optional item

ABS
Numeric function
Syntax: **ABS** n

The **ABS** n function returns the absolute value of numeric expression n. The result is always positive. The main use of this function is to check numeric values regardless of sign.

Examples:

(a)
```
100 INPUT Q
110 IF ABS Q<10 THEN GOTO 200
120 PRINT "ERROR – OUT OF RANGE"
130 PRINT "MUST BE–10<Q<10"
140 . . . .
```

(b)
```
500 LET S=A(ABS X)
```

this example ensures that the reference to array A does not go negative.

ARCCOS
Numeric function
Syntax: **ACS** n

Supplies arccos value of the argument, which must be in radians.

Example:
```
100 LET B1=ACS Q
```

ARCSIN
Numeric function
Syntax: **ASN** n

Returns the arcsin value of the argument in radians.

Example:
```
250 PRINT ASN (SQR X2)
```

ARCTAN
Numeric function
Syntax: **ATN** n

Supplies the arctan value of the argument in radians.

Example:

```
360 IF ATN E>0.56 THEN GOTO 200
```

CHR$
String function
Syntax: **CHR$** n

The **CHR$** n function provides the character representation of the numeric argument. This is normally used when printing items that have been converted (using **CODE** function for example) for manipulation. It is quite normal to find routines that convert data strings into numeric quantities to allow easier programming (it can also save a lot of program space at times!).

Error B will result if the argument is outside the range 0<=n<256.

Examples:

(a)
```
100 IF CHR$ G="Y" THEN GOTO 370
```

(b) This example prints random letters of the alphabet.

```
100 FOR X=1 TO 100
110 PRINT CHR$ (INT (RND*26)+38);
120 NEXT X
```

CLEAR
Command

Resets all variables in a program. This is automatically done when "RUN" is given, but is a "quick" way of resetting all variables when they are no longer required. On a 1K RAM ZX81, unused variables (i.e. those that have been assigned but are no longer required) can take up a noticeable proportion of the space available; thus judicious use of the **CLEAR** command can keep these to a minimum.

Example:

```
560 PRINT "ANOTHER GAME?"
570 INPUT Y$
580 IF Y$="NO" THEN STOP
590 IF Y$="YES" THEN GOTO 620
600 PRINT "ANSWER YES OR NO"
610 GOTO 560
620 CLEAR
630 GOTO 100
```

CLS
Command

Clears the screen display area and resets the print position to the top left-hand corner of the screen (line 0 column 0).

Unless your programs "scroll" during program execution, you need to monitor closely the number of lines that have been displayed and ensure that this does not exceed 22, otherwise an error condition occurs (screen full). A program will usually contain some error checking of data input, and display a

message if the check fails. This can lead to screen overflow if care is not taken to ensure that the **CLS** statement is used during the loop back to accept more input.

Example:

```
20 PRINT "ENTER A NUMBER BETWEEN 1 AND 10"
30 INPUT N
40 CLS
50 IF N>0 AND N<11 THEN GOTO 200
60 PRINT "TRY READING THIS TIME . . ."
70 GOTO 20
200 . . . .
```

If the **CLS** statement were omitted here, the screen would overflow after 11 attempts to get a correct number in!

CODE
Numeric function
Syntax: **CODE** s

This function returns the numeric code value of the first character in the string argument. Useful for testing keyboard input and breaking strings into individual characters.

Code 0 is set up as the end of string marker so that **CODE** s applied to a null string returns the value 0.

Examples:

(a)

```
10 INPUT Y$
15 REM JUMP IF FIRST LETTER IS"Y"
20 IF CODE Y$=62 THEN GOTO 100
30 REM ANYTHING ELSE COMES HERE . . .
40 . . . .
```

Better still, line 20 could say:

```
20 IF CHR$ CODE Y$="Y" THEN GOTO 100
```

This avoids a problem whereby error 3 is given if an out-of-range slicer is specified, e.g.

```
20 IF Y$(1)="Y" THEN GOTO 100
```

could result in error 3 if an empty string was entered as Y$.

(b) split a word into letters in a numeric array

```
10 DIM W(20)
20 INPUT W$
30 IF LEN W$=0 THEN GOTO 9999
40 FOR X=1 TO LEN W$
50 LET W(X)=CODE W$(X)
60 NEXT X
9999 STOP
```

CONT
Command

Allows execution of a program to resume after a break. The break could have occurred for a variety of reasons – pressing the **BREAK** key, a **STOP** command encountered, or after correcting a line that has given an error report code. Execution resumes at the next line number unless the program was stopped during an **INPUT** command.

COPY
Command

This command allows the contents of the display file (or a copy of the current screen) to be sent to the printer.

An extremely useful command for both program debugging and for taking copies of displayed results.

Although the command would normally be used in immediate mode, there is no restriction on its use within a program.

Since the plotting facilities cannot be used within **PRINT** and **LPRINT** commands, the only way of obtaining a plotted graph on the ZX printer is by using the **COPY** command after plotting on the screen.

Example:

This program will print the graph of a sine wave:

```
10 FOR X=0 TO 63
20 PLOT X,20
30 NEXT X
40 FOR Y=0 TO 43
50 PLOT 0,Y
60 NEXT Y
70 FOR X=0 TO 63
80 PLOT X,SIN (X/5)*15+20
90 NEXT X
100 COPY
```

COS
Numeric function
Syntax: **COS** n

Gives the cosine value of the argument in radians.

Example:

```
20 LET C=COS X
```

DIM
Command
Syntax: **DIM** v[$](n[,m])

Defines the maximum size of an array v to be n items at the lowest dimension.

Array elements start at element number 1 and go through to n.

The variable v may be either a single character numeric variable name, or a single character string variable name (followed by $).

String variables become *fixed length* over the length of the lowest array subscript – e.g. **DIM** S$(5,12) will allocate a string array S$ of five strings, each with a fixed length of 12 characters.

Whenever the **DIM** statement is executed, all elements of a numeric array are set to zero, while those of a string array are set to spaces.

Elements of the array are addressed by subscripting the array name with a numeric expression equivalent to the element number. For example, A(5) will give the value of the fifth element within array A.

Examples:

(a)

```
10 REM DEFINE AN ARRAY OF 5 ITEMS
20 DIM Q(5)
```

(b)

```
20 DIM X(9,4)
30 FOR X=1 TO 4
40 LET X(2,X)=X
50 NEXT X
```

This last example shows that numeric array names are distinct from ordinary individual variables (but note that the same is *not* true for string names and string arrays).

<Expression>

An expression is a combination of constants, variables and logical or arithmetical operators. Two types of expressions exist – numerical and string.

The order of priority in evaluating any logical or arithmetical operators is (from highest [i.e. most binding] to lowest):

12 ()	bracketed expressions
11 any function	
10 **	exponentiation
9 –n	unary minus
8 *	multiplication
7 /	division
6 + –	addition and subtraction
5 = <> < > >= <=	equality and inequality
4 **NOT**	inversion
3 **AND**	logical
2 **OR**	logical

The only operators relevant to string expressions are:

=, <, >, +, >=, <=

where equality/inequality tests are made using the character equivalents as listed in the ZX81 Handbook.

Concatenation of strings can be undertaken using the + operator.

The equality and inequality tests result in a value of 1 if the test result is ''true'' and 0 (zero) if the test result is ''false''.

The **NOT** operator has the effect of *inverting* the value of the following expression, so that **NOT** 1 becomes 0, while **NOT** 0 becomes 1.

Logical operators are effective only for numeric expressions. **AND** requires that both expressions be true for the whole expression to be true, while **OR** requires that only one of the two expressions be true for the whole expression to be true. A complete discussion of arithmetic and logical operators can be found as follows:

Arithmetic operators	Chapter 1 section 1.1 (1.1/3)
Conditional operators	Chapter 3 section 3.2 (3.2/2)
Logical operators	Chapter 3 section 3.2 (3.2/3)

Numeric expressions

These can be represented by:

– a numeric quantity
– a numeric variable
– the arithmetic result of any combination of the above
– the logical result of any combination of the above

Examples:

(a) 5
(b) A(3)
(c) ((Q*3)+(R/7))
(d) (X=1 **AND** Y=5)
(e) T$=''QUIT'' since the equality test results in a value of 0 or 1.

String expressions

These can be represented by:

– a string of text enclosed in quotes
– a string variable
– any concatenated combination of the above
– any of the above qualified by a *slicer*

Examples:

(a)	"JOE FELL OVER"
(b)	Q$
(c)	Q$+"JOE FELL OVER"
(d)	Q$(3)
	Q$(2 **TO** 3)
	Q$(**TO** 3)
	Q$(2 **TO**)
	Q$+"JOE FELL OVER"(5 **TO** 1Ø)
	(Q$+"JOE FELL OVER")(5 **TO** 1Ø)

Study these last examples carefully. They can be extremely powerful and useful.

EXP

Numeric function

Syntax: **EXP** n

Supplies the value of the constant e raised to the power of the argument, i.e. e^x. This is effectively the natural anti-log of a number.

Example:

```
6Ø PRINT EXP S
```

The anti-log of a value $\log_{10}$ can be obtained by specifying:

```
PRINT 1Ø**V
```

FAST

Command

Puts the ZX81 into fast mode, where the screen display is not refreshed during computation.

The screen is displayed under the following conditions:

(a) when an **INPUT** statement is encountered
(b) when a **PAUSE** statement is encountered
(c) when an error report code is generated
(d) when a **STOP** command is encountered in the program or the **BREAK** key is pressed during program execution, although these amount to the same as (c) since report D will be given.

Fast mode is four times faster than slow mode and should be used for program entry and editing, or for programs which do not require the constant display of results.

FOR

Command

Syntax: **FOR** v=n **TO** m [**STEP** n]

Initiates the control variable of a block loop.

v may be any single character numeric control variable
n and m are any numeric expressions

A **FOR** loop is not executed if the "finish" value is exceeded on entry to the loop. Full details of these conditions can be found in Section 4.2 "Iteration (2)".

The **STEP** value is optional, and if omitted, a value of +1 is assumed. The **STEP** value is added at each encounter with a corresponding **NEXT** statement.

FOR loops can be nested to any depth (although only 26 single character variable names exist to allow nesting!).

Examples:

(a)

```
50 REM PRINT THE LETTERS A TO Z
60 FOR X=38 TO 63
70 PRINT CHR$ X;
80 NEXT X
```

(b)

```
100 REM*SHUFFLE A PACK OF CARDS
110 DIM C(52)
160 FOR X=1 TO 52
170 LET Y=INT (RND*52)+1
180 REM*FIRST CARD SKIPS OVER TESTS
190 IF X=1 THEN GOTO 300
200 FOR W=1 TO X-1
210 REM*CHECK ALL PREVIOUS CARDS
220 REM*TO SEE IF THIS ONE IS
230 REM*ALREADY IN USE.
240 IF Y=C(W) THEN GOTO 170
250 NEXT W
300 REM*CARD HAS NOT BEEN SELECTED
310 REM*BEFORE, SO ENTER IT INTO
320 REM*THE ARRAY.
330 LET C(X)=Y
340 NEXT X
```

GOSUB
Command
Syntax: **GOSUB** n

Causes processing to continue at the line specified by the argument of the statement. The line number of the statement following the **GOSUB** is retained and upon executing a **RETURN**, control resumes with this statement.

GOSUBs are the lifeblood of modular programming, and also a useful way of keeping your program sizes to a minimum. Any duplicate lines of program can normally be grouped together and referenced by a **GOSUB** command.

Since the argument of the statement can be any variable, conditional **GOSUB**s can be written (found in other BASICs as **ON** × **GOSUB** n,m . . .).

If the line number specified in the argument of a **GOSUB** statement doesn't exist, the program jumps to a subroutine at the next highest line number.

Examples:

(a)

```
  20 REM CHECK FOR YES/NO INPUT
  30 GOSUB 8000
  40 REM*AFTERWARDS COME BACK HERE
  50 REM*NOW YOU CAN TEST VARIABLE Y
  60 IF Y THEN GOTO 270
     . . . .
 100 GOSUB 8000
 110 IF NOT Y THEN GOTO 90
     . . . .
8000 REM*ASK FOR YES OR NO INPUT
8010 PRINT "ANSWER YES OR NO"
8020 INPUT Y$
8025 LET Y=1
8030 IF Y$="YES" THEN RETURN
8035 LET Y=0
8040 IF Y$="NO" THEN RETURN
8050 REM*TRY AGAIN . . .
8060 GOTO 8000
```

(b)

```
 100 IF I<0 OR O>2 THEN STOP
 110 REM*THAT ENSURES THINGS DONT GO WRONG
 120 GOTO 1000+(I*10)
     . . . .
1000 REM*HERE IF I=0
1005 RETURN
1010 REM*HERE IF I=1
1015 RETURN
1020 REM*HERE IF I=2
1025 RETURN
```

GOTO
Command
Syntax: **GOTO** n

Transfers flow of program to the line number specified in the argument of the command. This is similar to **GOSUB**, except that control is not conditional upon a **RETURN** (or any other) statement.

As with **GOSUB**, the argument can be a complex variable, thus allowing conditional branching (as with the **ON** × **GOTO** statement found in other BASICs).

Examples:

(a)
```
250 INPUT A
260 IF A>0 AND A<10 THEN GOTO 300
270 PRINT "WHAT DO YOU CALL THAT?"
280 GOTO 250
300 . . . .
```

(b)
```
100 IF I<0 OR O>2 THEN STOP
110 REM*CHECK THAT ALLS O.K.
120 GOTO 2000+(I*20)
    . . . .
2000 REM*HERE IF I=0
2020 REM*HERE IF I=1
2050 REM*HERE IF I=2
```

note that **GOTO** uses the next higher line number if the one specified does not exist.

IF
Command
Syntax: **IF** e **THEN** <S>

Tests the value of the expression and obeys the **THEN** clause if the evaluation is true.

In this BASIC, "true" expressions are indicated by +1 and "false" expressions by 0.

The evaluation logic reduces all expressions to a true/false condition, and hence a single variable can be used instead of a full expression e.g.

If an expression such as **IF** T$(2)="Y" **OR** T$(2)="N" . . . is required continually throughout a program (or even more than once is sufficient) it becomes more practical to write –

```
LET T=(T$(2)="Y" OR T$(2)="N")
```

and then test the value of T:

```
IF T THEN . . .
```

but remember that T reflects the value of the expression at the time of assignment. This may not be the same as if the full expression were written, since the variable T$ could have altered.

Expressions can be made up from: **AND**, **OR**, +, –, *, /, =, >, <, <=, >=, or <>. See "Expressions" for more details of these.

The action argument can be *any BASIC statement* – it is not restricted to **GOTO**s.

Examples:

(a)
```
10 LET Y=0
20 INPUT Y$
30 IF Y$="NO" THEN GOTO 200
40 IF Y$="YES" THEN LET Y=1
50 IF NOT Y THEN GOTO 20
200 . . . .
210 IF Y THEN PRINT "YOU ANSWERED YES"
```

(b)
```
10 IF X=23 AND NOT Q-6<0 THEN STOP
```

(c)
```
350 IF LEN Q$ THEN IF Q$(1)="Y" THEN STOP
```

Think about that one!

INKEY$
String function
Syntax: **INKEY$**

Gives the character waiting at the keyboard, or the "empty string" if there is none.

An extremely useful function for allowing a program to be stopped at convenient points. Whereas the **BREAK** key stops execution of the program, **INKEY$** does not need to. The character read can be used to modify program operation.

Example:
This program teaches you to touch-type (if you want!) by displaying a random letter of the alphabet and random numbers, and waiting until you type the same character in reply:

```
10 SCROLL
20 FOR X=1 TO 10
30 LET C=INT (RND*36)+28
40 PRINT TAB X;CHR$ C;
50 IF INKEY$<>CHR$ C THEN GOTO 50
60 NEXT X
70 GOTO 10
```

(why must the program run in **SLOW** mode?)

INPUT
Command
Syntax: **INPUT** v

The **INPUT** statement allows data entry during program execution. Either numeric or string variables may be assigned data in the argument. Program execution is suspended and the screen buffer displayed (in fast mode). A cursor "prompt" indicates that data is required and if the data is a string, then the cursor is contained in quotes.

Any expression may be legally entered as data (for example the name of another variable, or even an arithmetic calculation!), although if the input is for a string expression, the quotes surrounding the prompt must be rubbed out first (use **EDIT** to remove these).

If **STOP** is entered as the first character of the input data (remove the quotes from a string expression), then report D will be given.

Examples:

(a)
```
20 PRINT "WHAT IS YOUR NAME?"
30 INPUT N$
40 IF N$="" " THEN GOTO 20
50 PRINT "O.K. ";N$;" MINE IS STANLEY"
```

(b)
```
210 LET N=1E38
220 PRINT "HOW MANY DAYS IN JULY?"
230 PRINT "(TYPE N IF YOU DONT KNOW)"
240 INPUT DAYS
250 IF DAYS=N THEN GOTO 300
```

In this second example, an answer of N is legal, since the variable N has been declared. This is quite a useful trick for allowing mixed numeric and textual responses.

INT
Numeric function
Syntax: **INT** n

Provides the integer value of the argument. The result is therefore truncated, and no rounding occurs.

The *next lower integer value* is always given, so that **INT** performed on a negative value will supply the next lower value also – e.g. **INT** −1.5 gives −2.

When numbers are manipulated in BASIC, tests of equality can sometimes be ineffective, due to small fractional components preventing the test from being evaluated properly. This can be particularly

noticed when a variable is being used as part of a computed **GOTO** or **GOSUB** statement.

Do not assume that a variable is integer after a calculation involving division or keyboard input – it may not be. Use the **INT** function to remove the fractional components.

Examples:

(a)
```
100 PRINT "ENTER A WHOLE NUMBER BETWEEN 1 AND 10"
110 INPUT N
120 IF N<1 OR N>10 OR N<>INT N THEN GOTO 100
```

(b)
```
 30 GOTO INT (X/20+1000)
 40 . . . .
```

LEN

Numeric function

Syntax: **LEN** s

Gives the number of characters contained in the string argument.

A very useful tool to allow strings to be manipulated with ease. The length of an empty string is zero.

Examples:

(a) Convert a string to code equivalent in array W.

```
100 REM CONVERT S$ INTO W ARRAY.
110 IF LEN S$=0 THEN GOTO 200
120 FOR X=1 TO LEN S$
130 LET W(X)=CODE S$(X)
140 NEXT X
200 . . . .
```

(b) Check input data validity

```
100 PRINT "PLEASE ENTER YOUR FIRST NAME"
110 INPUT N$
120 IF LEN N$<2 OR LEN N$>20 THEN GOTO 200
130 PRINT N$;", EH?"
    . . . .
200 PRINT ". . . WHAT SORT OF NAME IS THAT?"
210 GOTO 100
```

LET

Command

Syntax: **LET** v[$]=e

The **LET** command allows variables to be assigned with expression values, either numeric or string.

The expression on the right-hand side of the equals symbol is *fully evaluated* before the result is assigned to the variable declared on the left-hand side of the equals symbol. This allows a variable to be modified within a single statement (see example b below).

A variable cannot be used within an expression unless it has previously been assigned a value, otherwise error report 2 is given. Variables can be assigned in two ways:

(a) use the **LET** command with the variable declared on the left-hand side of the equals sign

(b) use the **DIM** command to create an array of the appropriate name

Rules for variable names and restrictions on their values can be found under the heading "Variables".

Examples:

(a)
```
 60 LET GROATS=52
 70 LET COST=GROATS*10
```

(b)
```
100 LET N$="MY NAME IS"+N$
```

This example relies on variable N$ being previously assigned, since it is referred to on the right-hand side of the equals symbol.

LIST
Command
Syntax: **LIST** [n]

The **LIST** command allows portions of the program to be viewed on the screen. The program cursor is positioned at the line number given in the argument (or at the beginning if no argument given), and the program is displayed on the screen with this line at the top.

When editing a program, **LIST** can be very useful. By typing:

LIST 20 (say)
EDIT

the appropriate line is presented (in this case 20) ready for editing at the foot of the screen. This can save time-consuming repeated cursor movement commands.

LLIST
Command
Syntax: **LLIST** [n]
This command is identical to **LIST** except the program listing is sent to the printer instead of the display.

The listing can be stopped by pressing the **BREAK** key.

LN
Numeric function
Syntax: **LN** n

This function supplies the natural logarithm of the expression following, i.e. $\log_e$ n.

Logarithms to base 10 can be easily obtained from the formula:

$$\log_{10} x = \log_e x / \log_e 10$$

e.g. $\log_{10}$ 2 would be calculated by:

PRINT LN 2/**LN** 10

Anti-logarithms are given by the **EXP** function (refer to this function for details of anti-logs to base 10).

LOAD
Command
Syntax: **LOAD** s

Allows a program with name s to be loaded from the cassette unit. The ZX81 does not have internal stop/start controls and so the unit must be controlled by hand. The tape will be searched until program s is found.

The command can be stopped at any time by using the **BREAK** key.

The statement is normally used in direct mode, but can be part of a program to allow "chaining" of programs

If s is the empty string, then the first program found on tape will be loaded.

Example:

```
9000 PRINT "LOAD TAKE-2 TAPE AND"
9010 PRINT "PRESS NEWLINE WHEN IT"
9020 PRINT "IS LOADED AND RUNNING"
9030 INPUT Y$
9040 LOAD "TAKE-2"
```

LPRINT
Command
Syntax: **LPRINT** [e] [,e] [;e] [**AT** n,m] [**TAB** n]

The **LPRINT** command is identical to the **PRINT** command, except that results are sent to the printer instead of the display. Full details of the syntax are found under **PRINT**, but you should note the following exception:

The **AT** facility usually specifies a line and column, but this feature is restricted to the column number only with **LPRINT**, although the line number must be present. An error report may be given if the line number is invalid even though it is not used, so it should be set at zero.

Examples of use of the **PRINT** command should be studied.

NEW
Command

This command clears the current program from memory, including all variables. It is used in direct mode (a bit self-defeating if used in a program) to remove a program ready for another.

Memory beyond the system variable **RAMBOT** is not affected, so this can be used to store ''resident'' machine code subroutines.

Be careful not to use this command when you've just spent 30 minutes typing in a new version of Star Trek . . .

NEXT
Command
Syntax: **NEXT** v

Causes the control variable in a **FOR** loop to take its next value (incrementing the variable by the **STEP** value), and returns the program control back to the line number following the **FOR** statement containing the corresponding variable definition.

If the control variable has exceeded its terminal limit specified in the **FOR** statement, then control passes to the line following the **NEXT**.

Example:

```
250 FOR A=1 TO 7
260 LET D(A)=0
270 NEXT A
```

Further examples of **FOR/NEXT** can be found under **FOR**.

PAUSE
Command
Syntax: **PAUSE** n

Allows the screen buffer to be displayed for n frames (0<n<32768), where 50 frames per second are displayed. Program execution resumes after (n/50) seconds.

If any key is pressed during this period, the **PAUSE** statement is terminated, and the program continues.

n may be set to any value within the range 0<n<65536 otherwise error report B is given. If the value of n is greater than 32767, the **PAUSE** command will cause program execution to be suspended until a key is pressed (i.e. it will not automatically resume).

A **PAUSE** statement should be followed by the statement **POKE** 16437,255

Example:

```
100 REM PRINT SQUARES UNTIL STOPPED
110 LET A=1
120 SCROLL
130 PRINT A;'' SQUARED= '';A**2
140 REM WAIT 5 SECS
150 PAUSE 250
160 POKE 16437,255
170 LET A=A+1
180 IF INKEY$='' '' THEN GOTO 120
190 STOP
```

PEEK
Numeric function
Syntax: **PEEK** n

PEEK allows the contents of the byte at memory address n to be accessed as a normal variable.

To use **PEEK**, it is necessary to know the layout of system memory, and the Sinclair Handbook has included chapters identifying the way that the ZX81 memory is organised (Chapters 27 and 28).

PEEK only looks at one byte at a time (a standard single numeric variable occupies five bytes), a byte containing 8 bits with maximum value of +127/−128 or modulus 256. Thus to study some of the system variables which occupy two bytes, a routine to convert two adjacent bytes into a single numeric quantity is required. Such a routine is:

```
8000 REM SET X TO ADDRESS OF
8010 REM FIRST BYTE IN PAIR
8020 REM TO BE STUDIED.
8030 REM RESULT IS GIVEN IN
8040 REM VARIABLE Y.
8050 LET Y=PEEK (X+1)*256+PEEK x
8060 RETURN
```

One useful item to **PEEK** is location 16389, which will tell whether a 16K RAM pack is fitted:

```
10 IF PEEK 16389>68 THEN LET RAMPACK=TRUE
```

Another is location 16437, which alters its value every 5.12 seconds (only in **SLOW** mode!), and can be used to time events (either internally to the program or externally):

```
 10 LET TICK=PEEK 16437          (start the "clock")
 20 LET ELAPSED TIME=0           (keep count of secs)
    . . . .
100 LET NTICK=PEEK 16437         (see if it's altered)
110 IF NTICK<>TICK THEN GOSUB 1000   (occurs every 5.12s)
120 LET TICK=NTICK
    . . . .
    . . . .
1000 REM COME HERE EVERY 5.12 SECONDS
1010 LET ELAPSED TIME=ELAPSED TIME+5.12
1020 IF ELAPSED TIME>120 THEN STOP
1030 RETURN
```

PI
Numeric value

Gives the value of pi to 7 decimal places – 3.1415927.

No argument is required.

Example:

```
100 PRINT "ENTER CIRCLE RADIUS ";
110 INPUT R
120 PRINT R
130 PRINT "AREA IS ";R**2*PI
```

PLOT
Command
Syntax: **PLOT** m,n

Puts a black quarter graphic character at the coordinates given in the **PLOT** argument.

The coordinates follow Cartesian conventions – i.e. horizontal address is specified first.

The horizontal address may be 0<=m<=63, and vertical address may be 0<=n<=43.

Coordinate 0,0 represents the bottom left-hand corner of the screen display, while 63,43 represents

the top right-hand corner.

Subsequent **PRINT** statements will print data following the plotted character.

Example:
This program "draws" a die face on the screen:

```
 20 FOR X=22 TO 38        draw horizontal lines
 30 PLOT X,6
 50 PLOT X,22
 70 NEXT X
 80 FOR Y=6 TO 22         draw vertical lines
100 PLOT 22,Y
120 PLOT 38,Y
140 NEXT Y
160 PLOT 26,10   }
180 PLOT 34,10   }
210 PLOT 30,14   }  these lines draw the spots
240 PLOT 26,18   }
260 PLOT 34,18   }
300 STOP
```

POKE
Command
Syntax: **POKE** n,m

POKE allows the contents of a memory location to be altered to any desired value.

n specifies the address of a byte to be changed, and m the value desired in that location. Note that m can only have a range of 0<=m<=255 (unsigned) since **POKE** will only operate on one byte. As with **PEEK**, it is necessary to have routines to split a two byte variable into composite parts ready for **POKE**ing.

The common use for **POKE** is to allow machine code routines to be inserted in spare memory and executed via the **USR** function. Because **POKE** operates in decimal values, a chart of the Z80 instruction set is required which gives the machine code equivalents in both hex and decimal.

It is not the purpose of this book to make the world proficient in machine code programming – refer to the reading list in Chapter 11.

One way of creating some available RAM for your routine is to include a **REM** statement as the FIRST command in a program. The **REM** should contain just a series of full-stops (one for each byte of program space required). The RAM address of the first full-stop under these conditions is 16514.

Another method is to alter the value of system variable **RAMBOT** (see Sinclair Handbook) and **POKE** the routine into the reserved area.

Example:

```
  1 REM .....
 10 LET X=16514
 20 POKE X,1
 30 POKE X+1,100
 40 POKE X+2,0
 50 POKE X+3,201
 60 LET J=USR X
```

This silly routine causes variable J to be assigned the value 100 (it would have been quicker to write **LET** J=100 but this is really to demonstrate how **POKE** and **USR** work). The equivalent assembly code set up by these **POKE**s is:

```
LD    BC,100   ;1,100,0
RET            ;201
```

By using variable X to contain the base address, several benefits are gained; it takes less program space to refer to X+n rather than a numeric literal such as 16514 and should the system RAM addresses alter in future versions of BASIC, you will only have one or two places to alter your programs rather than every single **POKE**.

PRINT
Command
Syntax: **PRINT** [e] [,e] [;e] [**AT** n,m] [**TAB** n]

PRINT causes information to be displayed on the screen, by building up a picture of the screen in a buffer area. Both numeric and string arguments are acceptable, in any mix or quantity. Arguments are separated by either a comma or semi-colon (see below).

Each argument can be a complex expression, which gives **PRINT** probably more power than any other single command.

If an expression is terminated by a semi-colon, then the next printed expression will be appended to this with no intervening spaces. If an expression is terminated by a comma, then the next item printed will be positioned at the next print zone on the screen, where zones are placed at intervals of 16 characters. This gives a line length of 2 fields (2 fields of 16 characters each). If a field is 15 or more characters in length when converted for display, then the next item will be positioned at the following print zone. This ensures that at least one space is left between each displayed field.

If the last expression (if any is present) is terminated by a space, then this indicates that there is no more data to be printed on this line, and further **PRINT**ed items are to start a new line.

The **TAB** n option forces the print position to be moved to column n of the current line. If the print position has passed column n, then the print position is moved to the same column on the next line down.

The **AT** n,m option allows a line n and column m to be specified. The print position is moved to this address on the display. Both n and m must be valid line (0<=n<=21) and column (0<=m<=31) numbers, else error report B is given.

Note that it would be fairly meaningless to follow the **AT** and **TAB** options by anything other than a semi-colon, since the print position would be altered again.

Examples:

(a)
```
300 FOR X=1 TO 10
310 PRINT X;" SQUARED IS ";X**2
320 NEXT X
```

In this example, the text "SQUARED IS" contains a space at the beginning and end. This is to allow for the fact that the semi-colon field separator does not leave a blank position between fields.

(b)
```
100 REM PRINT ALL THE GRAPHICS
110 PRINT "NORMAL","INVERSE"
120 FOR A=1 TO 10
130 PRINT " ";CHR$(A)," ";CHR$(A+128)
140 NEXT A
```

Here, the "space" strings are to position the graphic character centrally underneath the column headings. A comma after the first **CHR$** function ensures that the next field falls into line at a print zone underneath the next heading.

(c)
```
100 PRINT AT 0,0;"TOP";AT 21,26;"BOTTOM"
```

(d)
```
200 FOR X=24 TO 0 STEP -2
210 PRINT TAB X;"DIAGONAL"
220 NEXT X
```

RAND
Command
Syntax: **RAND** [n]

This command causes the random number generator seed to be initialised from the screen frame counter. If a non-zero numeric argument is present, then the seed is taken as this value.

Random numbers are taken from a series one-by-one as the **RND** function is called. However, this can mean that every time the computer is switched on, an identical series is obtained. The **RAND** command

forces a new seed to be used, and therefore causes the series to be entered at a different point each time.

Example:

```
 5 REM ROLL A DIE
10 RAND
20 LET D=INT (RND*6)+1
30 CLS
40 PRINT "YOU ROLLED A ";D
50 PRINT
60 PRINT "PRESS NEWLINE TO ROLL"
70 PAUSE 40000
80 GOTO 20
```

If you alter line 10 to:

```
10 RAND 5
```

then every time you run the program, the die rolls will give the same sequence of numbers.

REM
Command

The **REM** command allows remarks or comments to be entered into the body of the program to serve as a form of annotation as to the workings of the code.

It is good practice to sprinkle a program liberally with **REM** statements so that not only you (trying to debug your game in five years' time), but others may understand your train of thought in the flow of logic.

Unfortunately, **REM** statements eat up the available RAM and in the basic 1K RAM system this can be quite a problem. However, there are two solutions to this:

(a) Annotate your program listings on paper and ensure that a copy of this paper accompanies the cassette containing the program.

(b) Use **PRINT** commands (again, watch the memory limits) to inform the program user as to the way in which the program is to be used. This will normally prevent people from wanting to poke around inside your coding!

As more memory becomes available, the number of **REM** commands found in a program should increase dramatically.

RETURN
Command

Returns program control to the line following the last **GOSUB** encountered. This is normally the last line found in a subroutine.

Study the examples under **GOSUB**.

RND
Numeric value
Syntax: **RND**

Supplies the next number from a series of random numbers. This uses a seed to generate the series (which can be assigned using **RANDOMISE**). No argument is required.

The value returned is in the range 0<=n<1, so it is necessary to multiply this value by a number m (m>1) in order to obtain a random number 0<=r<m.

Examples:

(a)
```
10 LET N=INT (RND*100)+1
20 PRINT "I AM THINKING OF A"
30 PRINT "NUMBER BETWEEN 1 AND 100"
```

(b)
```
100 IF INT (RND*4)+1=1 THEN PRINT "THIS MESSAGE IS ONE IN FOUR"
```

This last example is a handy way of supplying information to a program user from time to time (for example if stuck in a maze game).

RUN
Command
Syntax: **RUN** [n]

Forces program execution to commence at line number n, or if n is absent or zero, then at the first line of program.

All variables are cleared when this command is entered. If you need to retain the variables, then use **GOTO** n rather than **RUN**.

Examples:

(a) **RUN** 0 (starts at first program line)

(b) **RUN** 200 (starts at line 200, or the first one encountered after if 200 is not present)

SAVE
Command
Syntax: **SAVE** s

Allows a program in memory to be saved on cassette tape with name s. The tape should be positioned ready, and running in record mode when the **NEWLINE** is entered.

The command can be used in immediate mode or within a program.

All assigned variables are stored with the program, and so data can be "stored" from one run to another. See Chapter 6 for full details of using the cassette.

Example:
```
9900 SAVE "GRAPH"
9910 GOTO 1
```

This program will automatically start running when reloaded from cassette – see Chapter 6.

SCROLL
Command

Moves the screen buffer up one line, creating a blank line at the bottom (i.e. line 21) and moving the print position to the first column of the blank line created. This allows "teletype" facilities of a sort to be used.

Example:

see example under **PAUSE**.

SGN
Numeric function
Syntax: **SGN** n

Answers with the sign of the argument.

If the argument is greater than zero, then the reply is 1. If the argument is less than zero, the reply is

−1. If the argument is exactly zero, then the reply is zero.

Example:
This routine rounds a value to the *nearest* integer:

```
100 LET V=INT (ABS V+0.5)*SGN V
```

SIN
Numeric function
Syntax: **SIN** n

Gives the sine value of the argument in radians.

Example:

```
3610 PRINT SIN A
```

SLOW
Command

This command puts the ZX81 into slow mode, or ''compute-and-display'' mode, where computed results are displayed simultaneously to program execution.

The main penalty paid for this luxury is that of speed of execution, which is four times slower than fast mode.

There are few programs which cannot be written to run in either fast or slow mode (remember that ZX80 users with the ZX81 BASIC 8K ROM installed cannot use slow mode – the ZX80 *always* runs in fast mode), but two examples have been given in this course – program **TRACE**, and example (b) under **PEEK**, which uses the timer facility of the screen refresh counter.

Whenever the ZX81 is switched on, or the **NEW** command is used, slow mode is automatically selected.

Example:

```
 1 REM PROGRAM TO USE THE TIMER
10 SLOW
   . . . .
   . . . .
```

(see the example under **PEEK**)

SQR
Numeric function
Syntax: **SQR** n

Gives the square root of the argument. Error B is given if n is negative.

Example:

```
100 REM CALCULATE HYPOTENUSE
110 PRINT ''ENTER BASE''
120 INPUT B
130 PRINT ''ENTER HEIGHT''
140 INPUT H
150 PRINT ''HYPOTENUSE IS '';SQR (B**2+H**2)
```

STR$
String function
Syntax: **STR$** n

Converts a numeric expression into string form. This allows more detailed processing of data items. The

resulting string is only as long as the item would be if printed – i.e. no leading spaces or zeros are supplied.

Example:

```
 1 REM PRINT NUMBER WITH LEADING ZEROS
10 INPUT N
15 LET N=INT N
20 LET N$=STR$(N)
30 FOR X=1 TO 10-LEN N$
40 PRINT "0";
50 NEXT X
60 PRINT N$
```

STOP
Command

Forces program execution to stop at the current line. The screen buffer is displayed and an error code is given at the foot of the screen in the form 9/LLLL where LLLL is the line number containing **STOP**. The program can be restarted by using **CONT**.

Examples:

(a)

```
20 REM*CHECK IF NUMBER IN RANGE
30 IF N>0 AND N<10 THEN GOTO 100
40 PRINT "NUMBER OUT OF RANGE"
50 STOP
   . . . .
```

(b)

```
300 PRINT "WHAT NEXT?"
310 INPUT N$
320 IF N$="FINISH" THEN GOTO 9999
330 IF N$="END" THEN GOTO 9999
   . . . .
9999 STOP
```

TAN
Numeric function
Syntax: **TAN** n

Gives the tangent value of the argument, which must be in radians.

Example:

```
230 LET T=TAN X
```

UNPLOT
Command
Syntax: **UNPLOT** m,n

Erases a pixel created previously by **PLOT**. The print position indicator is updated to the new "m,n" position. See **PLOT** for details of checks made on m and n.

Example:
To erase the centre spot created in the example under **PLOT**, thus making the die face into a "four" instead of a "five" –

```
400 UNPLOT 30,14
410 STOP
```

USR
Numeric function
Syntax: **USR** n

Forces a call to a machine code routine at memory address n.

On return from the routine, the contents of the BC registers are taken as the function value.

A machine coded routine must not use (in any way) registers A' F' IX and R, and it is advisable not to use registers IY and I.

In order to set up a machine code routine, it is necessary to use **POKE** to install a byte at a time (see a full example of this under "**POKE**"). An alternative method is to create an array using **LET** statements, then calculate the address of the array using the system variable **VARS** (see ZX81 Handbook, chapters 28 and 29).

Example:

See example under **POKE**

VAL
Numeric function
Syntax: **VAL** s

The **VAL** function gives the numeric equivalent of the string expression argument that follows.

If the string expression does not evaluate to a numeric equivalent, then error report C will be given.

It is possible for the **VAL** function to be given a string expression containing variable names, although such a function must appear as the first item within any larger expression. Where the string expression is made up of more complex expressions, error reports other than report C may be given (e.g. error 2).

Examples:

(a)
```
20 LET VARA=20
30 LET M$="*100"
40 LET S$="VARA"
50 PRINT VAL (S$+M$)
```

(b)
```
200 LET NUM=53
    ....
    ....
300 PRINT VAL STR$ NUM
```

VAL and **STR$** have opposite effects.

<Variable>

A variable can represent one of four items:
(a) Simple numeric floating point value
(b) An array element
(c) Control loop variable
(d) String variable

Variable names:

Type (a) can be of any length: e.g. J, ERRORCOUNT, TOTAL

Type (b) must be a single character, optionally followed by a $ symbol, followed by a subscripting variable (which may be any numeric expression) e.g. D(5), X$(TOTAL), T((X=1)*5 **OR** (X=0)*6)

Type (c) must be a single character e.g. X, E

Type (d) must be a single character suffixed by $ e.g. T$, B$

Maximum values:
All numeric variables (types a, b and c) are accurate to 9.5 significant digits, thus the longest accurate

number that can be held is 4,294,967,295. The largest value that can be held (regardless of accuracy in lower orders) is approximately 10E38, and the smallest is approximately 4*10E−39.

Strings can be of any length (although a string variable name can only be one character followed by a $).

Report Codes

Each possible report code is shown here, with an example of how it may have arisen. Some reports are merely to tell you the status of the ZX81 (like **BREAK** key pressed) while others are as a result of programming errors and may be more difficult to detect. The line number of the error report is given as the second of the two numbers.

Report Code	Meaning
0	A program (or direct command) has completed successfully. This can also occur when a **GOTO** is used to jump to a line number that is higher than the highest line number in the program.
1	A **NEXT** statement has been found for which an appropriate control variable has not been set up, although an ordinary variable with the correct name has been found. An example: 10 **LET** X=1 20 **PRINT** "HI THERE" 30 **NEXT** X
2	An undefined variable name has been used. If the variable is an ordinary variable, then it must be assigned with **LET** before it can be used in other commands. If the variable is an array, then it should be assigned with a **DIM** statement. A common cause of this error is either absent-mindedness or a typing mistake! E.g.: 10 **LET** NMBER=23 20 **PRINT** NUMBER
3	Subscript out of range. E.g. 10 **DIM** A(5) 20 **LET** A(12)=32 or . . . 10 **LET** X$="ABCDEFGHI" 20 **PRINT** X$(12) Both will give report 3/20.
4	Not enough memory in the ZX81 to complete this command or statement. Read this section in "Common Problems and Solutions".
5	Screen display full. **CONT** will allow the program to restart with a blank screen. This is covered quite extensively in Chapters 3 and 4.
6	Arithmetic overflow. A calculation has resulted in a number larger than approximately 10^{38}. E.g. **PRINT** 10E25*10E25
7	A **RETURN** statement has been found when no **GOSUB** statement was given.
8	**INPUT** has been used as an immediate command, and this is not allowed.
9	A **STOP** statement has been found in a program. If you enter **CONT,** the ZX81 will continue from the *next* line number.
A	You have tried to use a function incorrectly, e.g. **PRINT** SQR −1. This can happen with SQR, LN, ASN and ACS.
B	An invalid number has been found. Certain statements require numbers within a range (for example **GOTO** requires a line number in the range 1 to 9999). If this

number is beyond a suitable range, then error report B is given. Look at the line that has caused the problem and refer to the chapter (use the Index) that deals with the particular command. A common mistake occurs with **PLOT** and **UNPLOT**, also **PRINT** AT y,x.

C The string expression used after a VAL function does not represent a valid numeric expression, e.g.:

LET E=VAL "32X.66"

D **BREAK** key pressed, or **STOP** used as the first item in an input expression.

E Not used.

F A **SAVE** command has been given with an empty program name string, e.g.

SAVE ""

INDEX

ABOUT THE AUTHOR

Trevor Toms has been in the computer industry since 1970; most of this time was spent with a computer bureau working on computer operating systems and commercial business programs for mainframes and microcomputers. Later he wrote the multi-user operating system for an 8080 based commercial system. He is now a partner of Phipps Associates, 3 Downs Avenue, Epsom, who specialise in microcomputer applications. He is the author of "The ZX80 Pocket Book".

His hobbies include music (playing banjo for the Boodle-um Jug Stompers), comic collecting and playing with the ZX81!

Printed by
The Leagrave Press Ltd
Luton and London