sinclair

ZX81 BASIC PROGRAMMING

sinclair

ZX81 BASIC PROGRAMMING

by Steven Vickers

First Edition 1980

Second Edition 1981

© 1980 by Sinclair Research Limited

Front cover illustration by John Harris of Young Artists, specially commissioned by Sinclair Research Limited.

CHAPTER 1
Setting up the ZX81 *Page 5*
& how to use this manual, whether or not you know BASIC.

CHAPTER 2
Telling the computer what to do *Page 11*
How to type things in to the computer
⇦, ⇨, **RUBOUT**, **NEWLINE**

CHAPTER 3
A History lesson *Page 19*

CHAPTER 4
The Sinclair ZX81 as a pocket calculator *Page 23*
Statement: **PRINT**, with commas & semicolons
Operations: +, -, *, /, **
Expressions & scientific notation

CHAPTER 5
Functions *Page 29*
Statement: **RAND**
Functions: **ABS**, **SGN**, **SIN**, **COS**, **TAN**, **ASN**, **ACS**, **ATN**, **LN**, **EXP**, **SOR**, **INT**, **PI**, **RND**, **FUNCTION**

CHAPTER 6
Variables *Page 35*
Statements: **LET**, **CLEAR**
Simple Numeric variables

CHAPTER 7
Strings *Page 41*
Operation: + (for strings)
Functions: **LEN**, **VAL**, **STR$**
Strings, simple string variables

CHAPTER 8
Computer Programming *Page 47*
Statements: **RUN**, **LIST**
Programs
Editing programs using ⇧, ⇩ & **EDIT**

CHAPTER 9
More computer programming *Page 55*
Statements: **GOTO**, **CONT**, **INPUT**, **NEW**, **REM**, **PRINT**
STOP in **INPUT** data
BREAK

CHAPTER 10
If... *Page 65*
Statements: **IF**, **STOP**
Operations: =, <, >, **<=**, **>=**, **<>**, **AND**, **OR**
Function: **NOT**

CHAPTER 11
The character set *Page 75*
Functions: **CODE**, **CHR$**
The character set is thoroughly non-standard.
GRAPHICS

CHAPTER 12
Looping *Page 81*
Statements: **FOR**, **NEXT**, **TO**, **STEP**

CHAPTER 13
SLOW & FAST *Page 87*
Statements: **SLOW**, **FAST**
The ZX81 operates at two speeds: one neat, the other fast.

CHAPTER 14
Subroutines *Page 91*
Statements: **GOTO**, **RETURN**

CHAPTER 15
Making your programs work *Page 99*
Flowcharts & debugging

CHAPTER 16
Tape storage *Page 105*
Statements: **SAVE**, **LOAD**

(cont. over)

CHAPTER 17
Printing with frills *Page 113*
Statements: **CLS**, **SCROLL**
PRINT items: **AT**, **TAB**

CHAPTER 18
Graphics *Page 117*
Statements: **PLOT**, **UNPLOT**

CHAPTER 19
Time & motion *Page 125*
Statement: **PAUSE**
Function: **INKEY$**

CHAPTER 20
The ZX81 printer *Page 131*
Statements: **LPRINT**, **LLIST**, **COPY**

CHAPTER 21
Substrings *Page 135*
Slicing, using **TO**

CHAPTER 22
Arrays *Page 141*
Statement: **DIM**

CHAPTER 23
When the computer gets full *Page 147*
Odd things happen.

CHAPTER 24
Counting on your fingers *Page 153*
Binary & hexadecimal counting

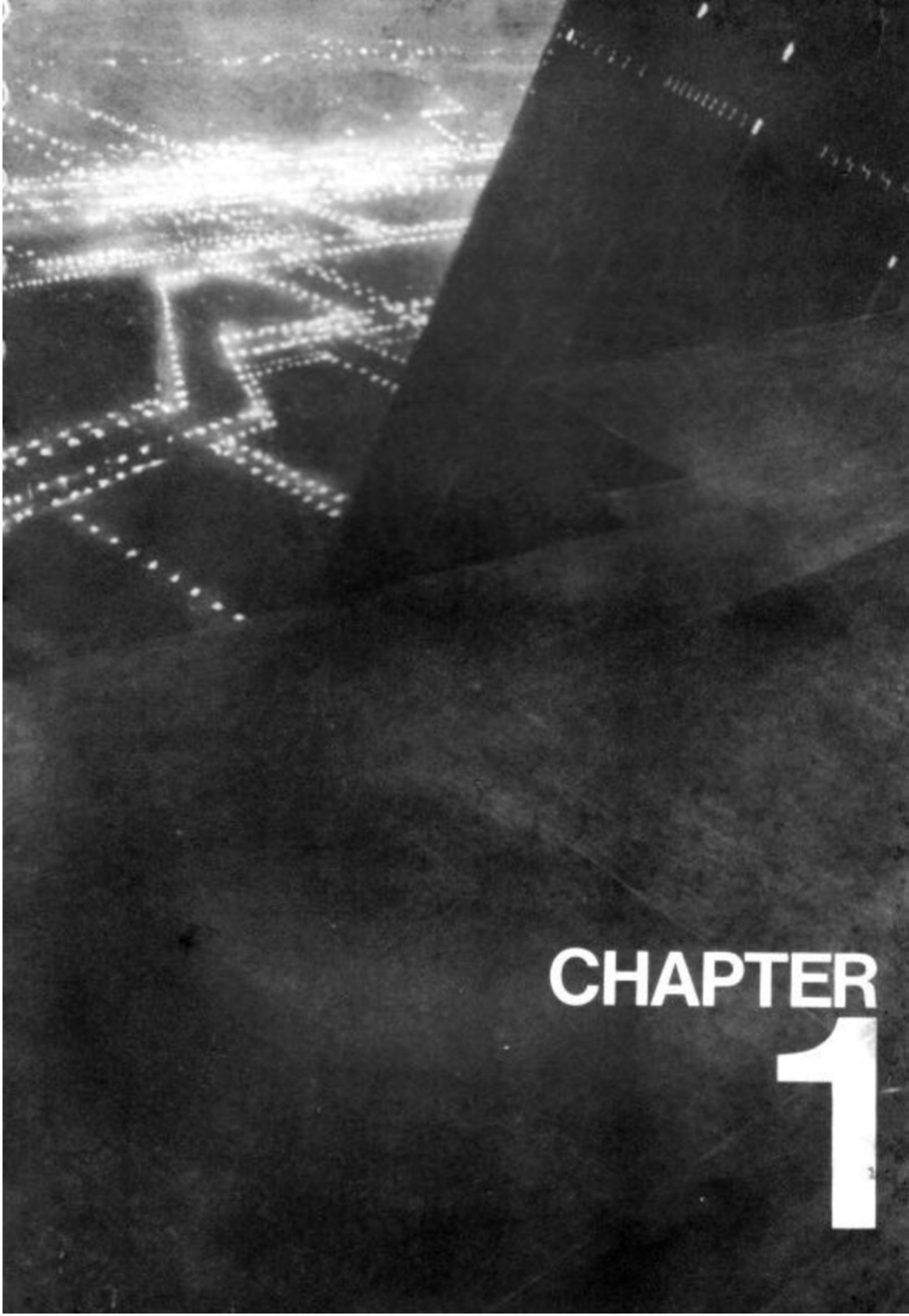
CHAPTER 25
How the computer works *Page 159*
What the individual chips do.
Statement: **POKE**
Function: **PEEK**

CHAPTER 26
Using machine code *Page 165*
Statement: **NEW**
Function: **USR**

CHAPTER 27
Organization of memory *Page 169*

CHAPTER 28
System variables *Page 175*

APPENDICES
A The character set *Page 181*
B Report codes *Page 189*
C The ZX81 for those that understand BASIC *Page 191*
Index *Page 203*



CHAPTER 1

Setting up the ZX81

Unpack the ZX81, you will have found

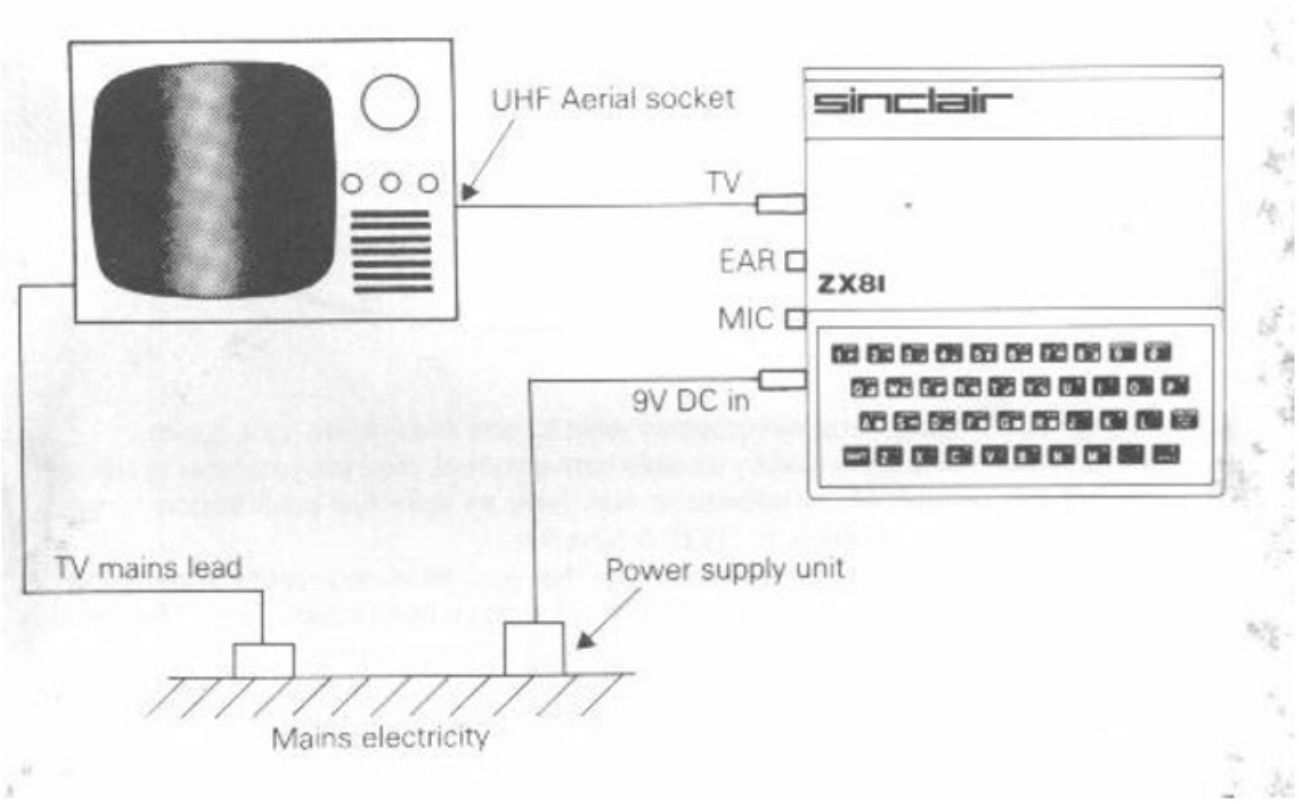
1. This manual.

2. The computer. This has three jack sockets (marked 9V DC IN, EAR & MIC), one aerial socket, & an exposed part of its circuit board where you can plug extra equipment. It has no switches - to turn it on you just connect it to the power supply.

3. A power supply.

This converts mains electricity into the form that the ZX81 uses. If by accident you plug it into the wrong socket in the computer you will do no damage. If you want to use your own power supply, it should give 9 volts DC at 700mA unregulated, & end in a 3.5mm jack plug with positive tip.

4. An aerial lead about 4 feet (120cm) long, which connects the computer to a television.



5. A pair of leads about a foot (30cm) long with 3.5mm jack plugs at both ends. These connect the computer to a tape recorder.

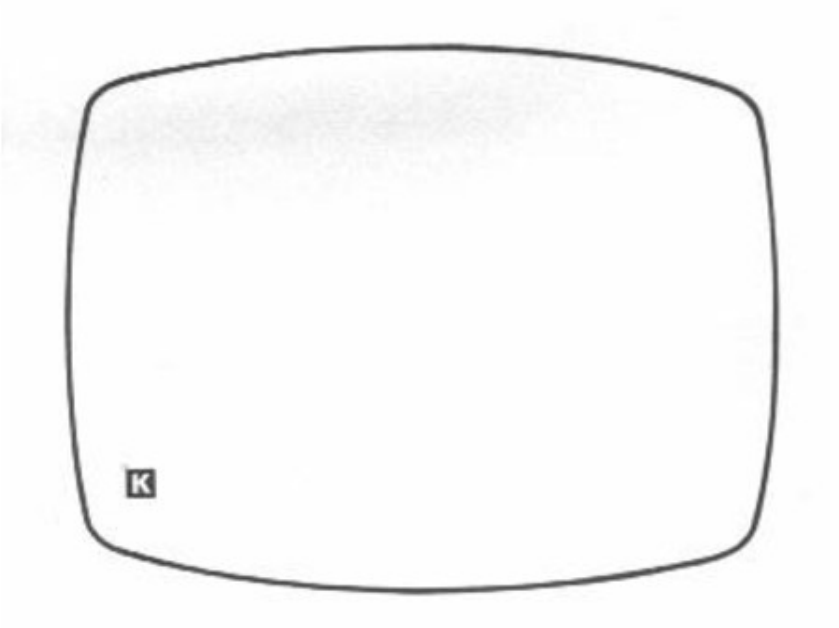
You will also need a television - the ZX81 can work without one, but you won't be able to see what it's doing! It must be a UHF television - if it's not built to receive BBC2 (in the UK) then it's no good.

Later you will need a cassette recorder. This is because when you turn a ZX81 off, all the information stored in it is permanently lost. The only way to keep it for later is by recording it on a cassette tape - you'll see how to do this in chapter 16. You can also buy tapes that other people have prepared, & so run their programs.

When you've got everything together (except the cassette recorder) connect them as shown on previous page.

If your television has two aerial sockets, marked UHF & VHF, then use the UHF one.

Turn the power on, & switch on the television. You now need to tune the television in. The ZX81 operates on channel 36 UHF, & when it is first plugged in and properly tuned it gives a picture like this:



When using the computer, you will probably want to turn the volume right down.

If your television has a continuously variable tuning control, then you just have to adjust it until you get this picture. Many televisions now have an individual push button for each station. Choose an unused one (e.g. ITV2), & tune it in.

If you get stuck with the computer, remember that you can always reset the computer & get back this picture by taking out the '9V DC IN' plug and putting it back again. This should be a last resort, because you lose all the information in the computer.

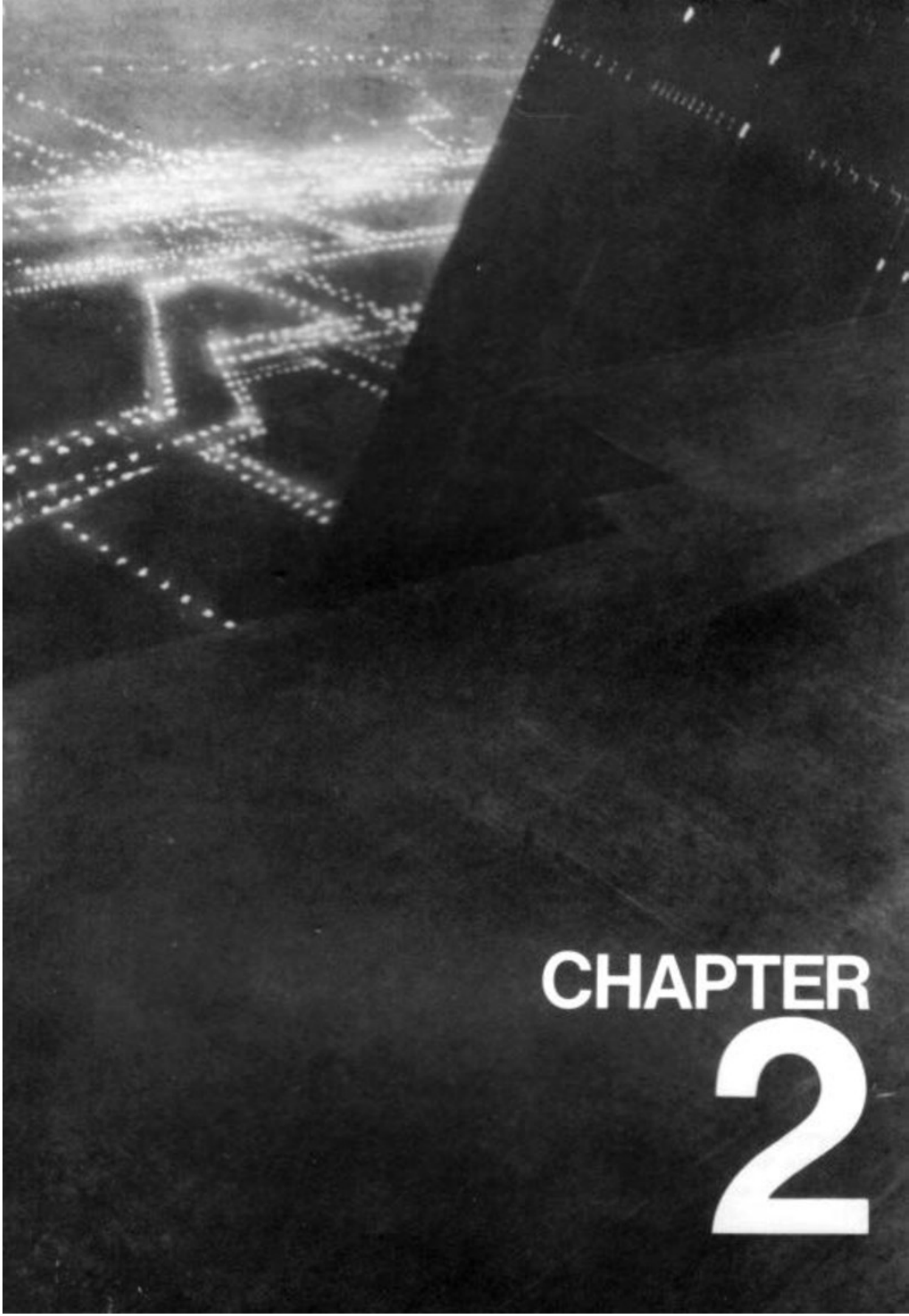
Note: This description of the television applies to Britain, where there is a UHF system using 625 lines at 50 frames per second. This will work in some other countries (for

instance most Western European countries except France). The USA, uses VHF & 525 lines at 60 frames per second.

Now that you've set up the computer, you'll want to use it. If you already know the computer language BASIC, then read appendix C & use the rest of the manual only to clarify the obscure points.

If you're a novice, then the main part of the manual has been written for you. Don't ignore the exercises; many of them raise interesting points that are not dealt with in the text. Look through them, & do any that take your fancy, or that seem to cover ground you don't understand properly.

Whatever else you do, keep using the computer. If you have a question 'What does it do if I tell it such & such?' then the answer is easy: type it in & see. Whenever the manual tells you to type something in, always ask yourself, 'What could I type instead?' & try out your replies. The more of your own stuff you write, the better you will understand the ZX81. (This is called unprogrammed learning.) Regardless of what you type in, you cannot damage the computer.



CHAPTER 2

Telling the computer what to do

Turn the computer on (by plugging it in) and get the blank screen with the write-on-black K, as in the picture in chapter 1. To make it do something, you have to type in a message that it understands; for instance the message

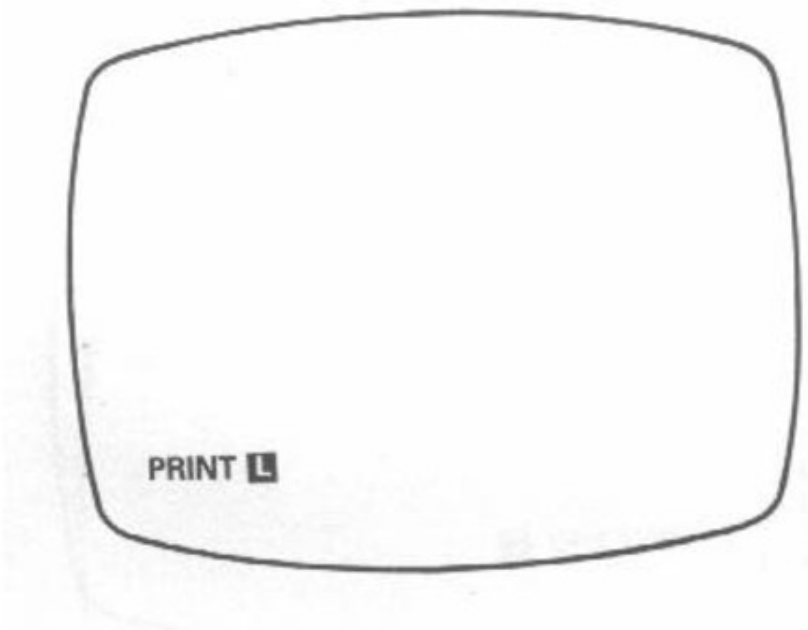
PRINT 2+2

tells it to work out the sum 2+2 and display the answer on the TV screen.

A message like this, telling the computer to do something straight away, is a command; this particular one is a **PRINT** command, but also a **PRINT** statement. Calling it a **PRINT** statement just specifies its form without referring to how the computer is going to use it. Thus every command takes the form of a statement, but so do some other things - program lines do, as we shall see in chapter 8.

To type in this command,

1. First type **PRINT**. But, although as you can see the keyboard has a key for each letter, you do not spell the word out P, R, I, N, T. As soon as you press P the whole word will come up on the screen, together with a space to make things look nice, and the screen will look like this:



The reason for this is that at the beginning of each command the computer is expecting a keyword - a word that specifies what kind of command it is. The keywords are written above the keys, and you will see that '**PRINT**' appears above the P key, so that to get '**PRINT**' you have to press P.

The computer lets you know that it expects a keyword by the K that you had to start off with. There is almost always some white-on-black (inverse video) letter, either K or L (or, we shall see later, F or G), called the cursor. The K means 'whatever key

you press, I shall interpret it as a keyword'. As you saw, after you had pressed P for **PRINT**, the K changed to an L.

This system of pressing just one key to get more than one symbol is used a lot on the ZX81. In the rest of this manual, words with their own keys are printed in **BOLD TYPE**.

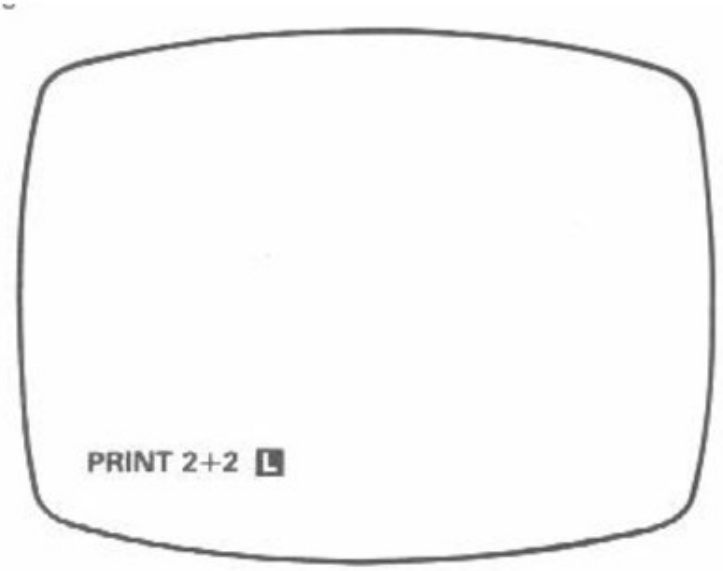
You must remember that it is useless trying to spell these words out in full, because the computer just won't understand.

2. Now type 2. This should cause no problems. Again, you should see 2 appear on the screen, and the L move along one place.

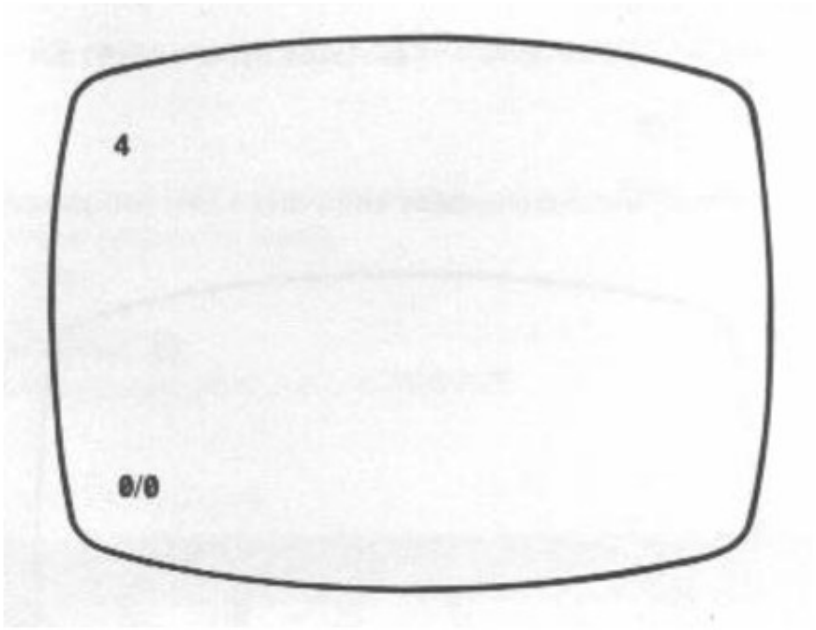
Note also how much space is automatically put in between **PRINT** & 2 to make it look neat. This is done as much as possible, so that you hardly ever have to type a space. If you do type a space, it will appear on the screen, but it will not affect the meaning of the message at all.

3. Now type +. This is a shifted character (they are marked in red - the colour of **SHIFT** itself on its key - in the top right hand corner of each key), and to get '+' you must hold down the key SHIFT and while you are still doing that, press the key K+ (LIST / LEN).

4. Now type 2 again. The screen will look like this:



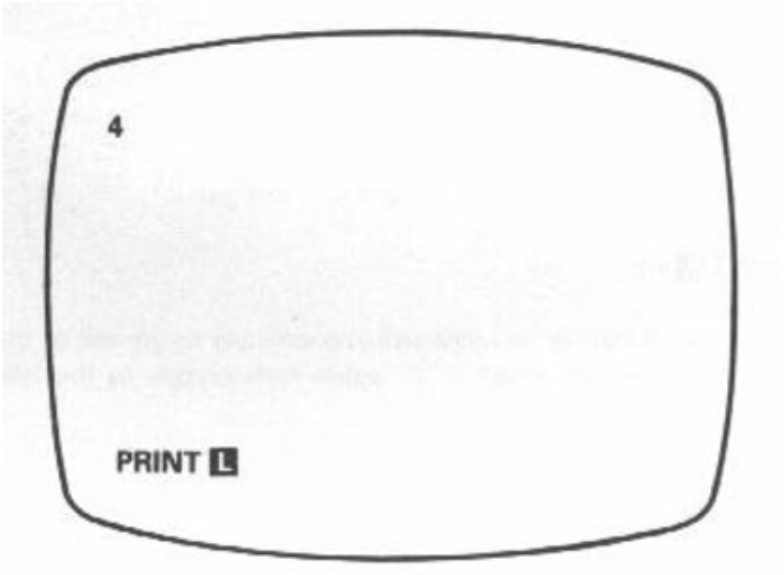
5. Now - and you must always remember this - press **NEWLINE**, the key FUNCTION / NEW LINE. This means 'message complete', or 'all right, computer, lets see some action'. The computer will now read the message, work out what has to be done, and do it. In this case, the screen will change to



4 is the answer - but of course you do not need to buy a computer to work that out.

0/0 (Note how zero is written with a slash to distinguish it from capital O. This is fairly common in computing circles.) is the report in which the computer tells you how it got on. The first 0 means 'OK, no problems'. (In appendix B there is a list of the other report codes that can arise, for instance if something goes wrong.) The second 0 means 'the last thing I did was line 0'. You will see later - when you come to write programs - that a statement can be given a number & stored away for execution later: it is then a program line. Commands do not actually have numbers, but for the sake of reports the computer pretends that they are line 0.

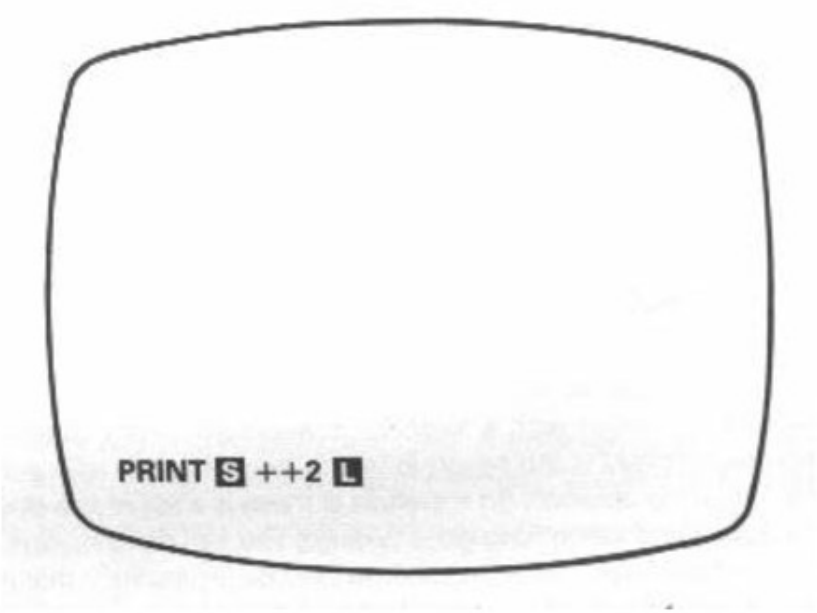
You should imagine a report as hiding a K cursor - if you press P for **PRINT** now, the report will disappear and the screen will change to



The cursor can also be used for correcting mistakes: type ++2, to get

PRINT ++2**L**

on the bottom line, Pretty incomprehensible stuff, and when you press **NEWLINE** you get



The **S** is the syntax error marker (the syntax is the grammar of the message, saying which are allowed and which are not), and shows that the computer got as far as '**PRINT**', but after that decided that it was not a proper message.

What you want to do of course is rub out the first +, and replace it by - let us say - 3. First you have to move the cursor so that it is just to the right of the first +; there are two keys, ⇦ and ⇨ (shifted 5 and shifted 8), that move the cursor left and right. Holding **SHIFT** down, press the ⇦ key twice. This moves the cursor left two places to give you

PRINT +**L**+2

Now press the **RUBOUT** key (shifted 0), and you will get

PRINT**L**+2

RUBOUT rubs out the character (or keyword) immediately to the left of the cursor.

If you now press 3 this will insert a '3', again immediately to the left of the cursor, giving

PRINT 3**L**+2

and pressing **NEWLINE** gives the answer (5).

The ⇨ key (shifted 8) works just like the ⇦ key, except that it moves the cursor right instead of left.

Summary

This chapter has covered how to type messages in for the ZX81, explaining the single keystroke system for words,

the K & L cursors,

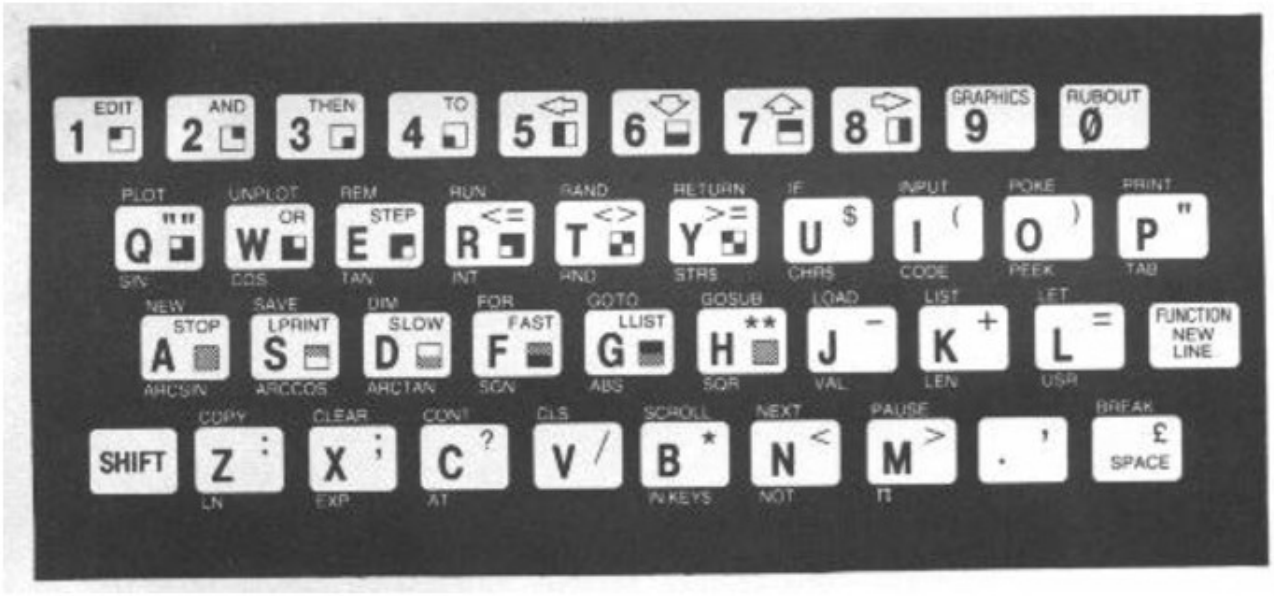
reports,

the syntax error marker, S

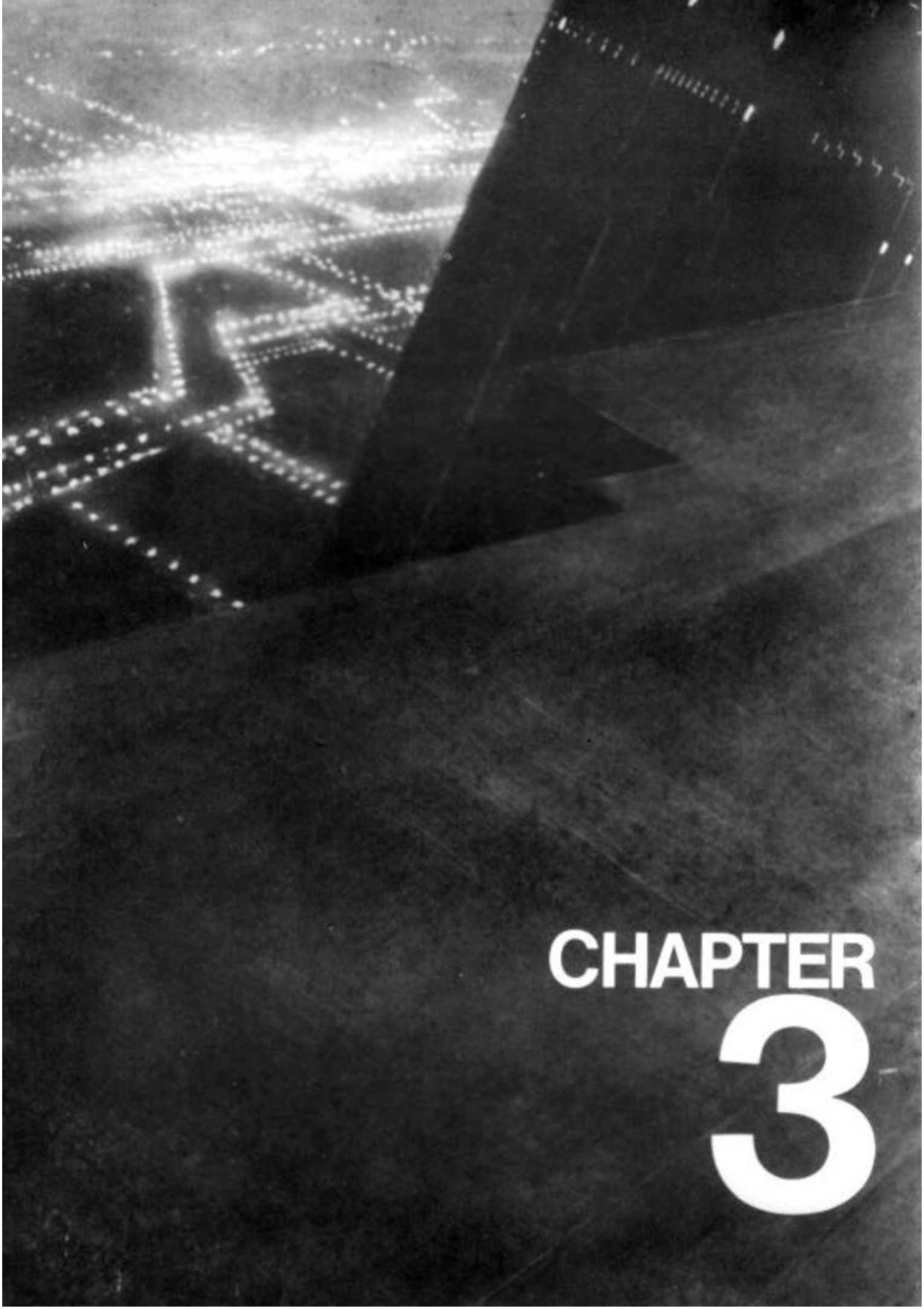
& how to correct mistakes using ⇦, ⇨ and **RUBOUT**.

The keyboard

Here is a picture of the keyboard.



Remember that to use **SHIFT**, you have to hold it down at the same time as you press another key. Do not confuse digit 0 with letter O.



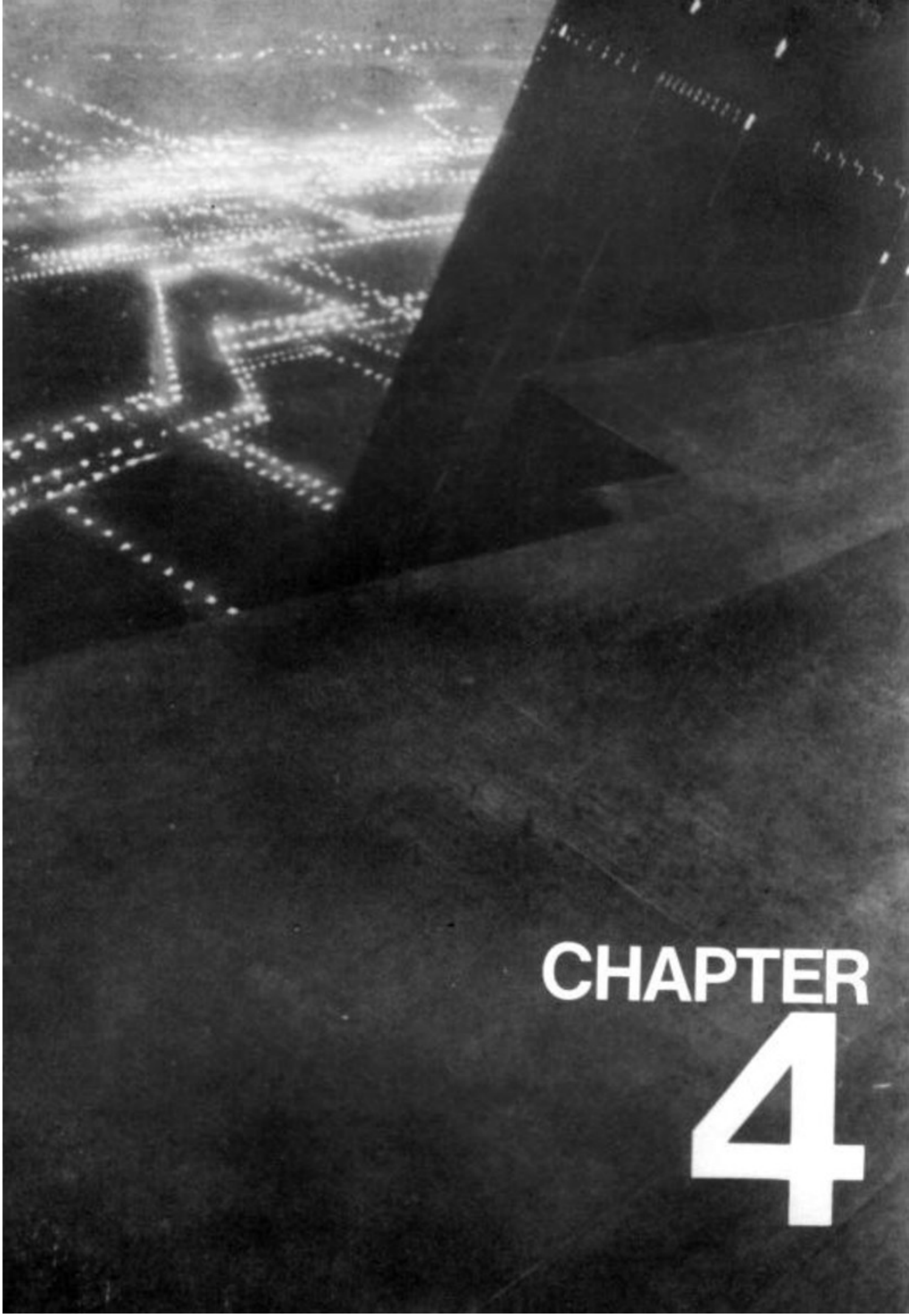
CHAPTER 3

A History lesson

The messages that you type in are in a computer language called BASIC (standing for Beginners' All-purpose Symbolic Instruction Code). It actually takes the computer quite a lot of effort to break down the BASIC messages into its own rudimentary operations, but, after all, that's what it's paid to do. The BASIC messages contain enough English words (like **PRINT**) to make them fairly easy for an English speaking human to learn.

BASIC was designed at Dartmouth College in New Hampshire, USA in 1964, & since then it has come to be by far the most widely used computer language by beginners & hobbyists. This is largely because it is very well adapted for on-line use where the user types something in & the computer answers straight away. There are other languages - such as ALGOL (in fact a whole family of ALGOLs) & PASCAL - with a much neater structure & greater power than BASIC, but only a few - such as the relatively unknown APL & POP-2 - are as easy to use on-line. Some others that must be mentioned are FORTRAN, PL1 & COBOL.

Many personal computing magazines publish programs in BASIC, & are well worth looking through for ideas. You will almost certainly have to adapt them slightly because every computer that uses the BASIC language has its own dialect, different from all the others.



CHAPTER 4

The Sinclair ZX81 as a calculator

Turn the computer on. You can now use it as a calculator, along the lines of chapter 2: type **PRINT**, then whatever it is that you want working out, & then **NEWLINE**. (We shan't usually bother to tell you to type **NEWLINE**.)

As you would hope, the ZX81 can not only add, but also subtract, multiply using a star * instead of the usual times sign - this is fairly common on computers) & divide (using / instead of ÷). Try these out.

+, -, * and / are operations, & the numbers they operate on are their operands.

The computer can also raise one number to the power of another using the operation ** (Shifted H. Do not type * - shifted B - twice): type

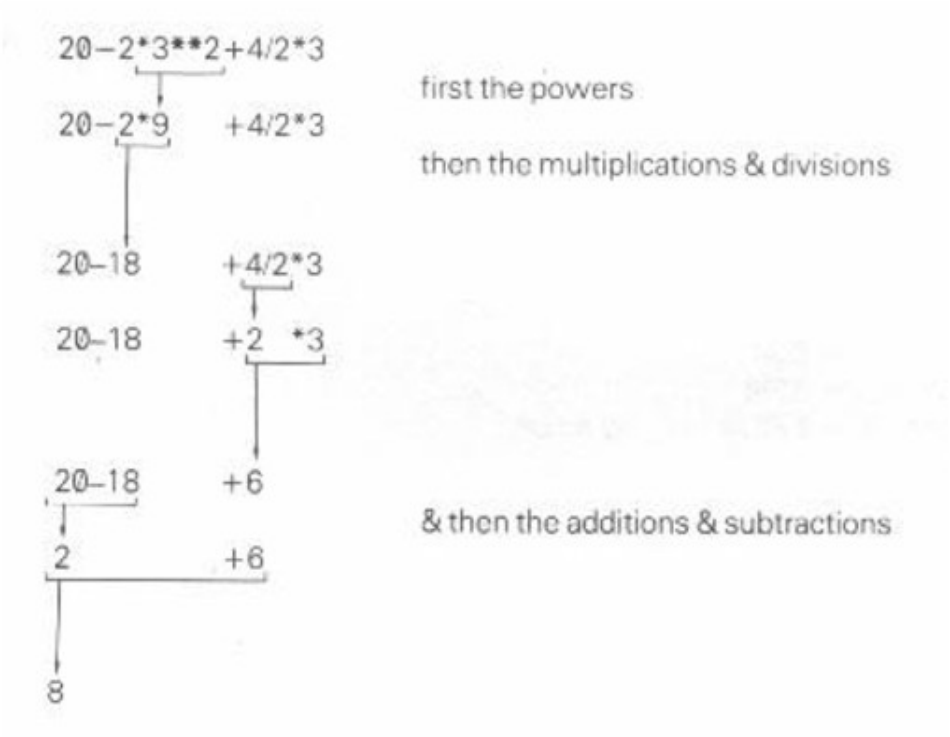
PRINT 2**3 (Remember the **NEWLINE**.)

& you will get the answer 8 (2 raised to the power 3, or 2^3, or 2 cubed)

The ZX81 will also work out combinations of the operations. For instance.

PRINT 20-2*3**2+4/2*3

gives the answer 8. It goes all round the houses to get this, because first it works out all the powers (**) in order from left to right, & then all the multiplications & divisions (* & /), again from left to right, & then the additions & subtractions (+ & -), yet again from left to right. Thus our example is worked out in the following stages:



We formalize this by giving each operation a priority, a number between 1 & 16. The operations with highest priority are evaluated first, & operations with equal priority are evaluated in order from left to right.

** has priority 10

* and / have priority 8

+ & - have priority 6

When - is used to negate something, as when you write -1, then it has priority 9. (This is unary minus, as opposed to the binary minus in 3-1: a unary operation has one operand, while a binary operation has two. Note that on the ZX81 you cannot use + as a unary operation.)

This order is absolutely rigid, but you can circumvent it by using brackets: anything in brackets is evaluated first & then treated as a single number, so that

PRINT 3*2+2

gives the answer 6+2 = 6, but

PRINT 3*(2+2)

gives the answer 3*4 = 12.

A combination like this is called an expression - in this case, an arithmetic or numeric expression because the answer is a number. In general, whenever the computer is expecting a number from you, you can give it an expression instead and it will work out the answer.

You can write numbers with decimal points (use the full stop), & you can also use scientific notation - as is quite common on pocket calculators. In this, after an ordinary number (with or without a decimal point), you can write an exponent part consisting of the letter E, then maybe + or -, & then a number without a decimal point. The E here means '*10**' ('times ten to the power of'), so that

2.34E0 = 2.34 * 10**0 = 2.34

2.34E3 = 2.34 * 10**3 = 2340

2.34E-2 = 2.34 * 10**-2 = 0.0234 & so on.

(Try printing these out on the ZX81.)

The easiest way of thinking of this is to imagine the exponent part shifting the decimal point along to the right (for a positive exponent) or to the left (for a negative exponent).

You can also print more than one thing at once, separating them either with commas (,) or semicolons (; or shifted X). If you use a comma, then the next number will be displayed starting either at the left hand margin, or in the middle of the line in the 16th column. If you use a semicolon, then the next number will be displayed immediately following the last one.

Try

PRINT 1;2;3;4;5;6;7;8;9;10

&

PRINT 1,2,3,4,5,6,7,8,9,10

to see the differences. You can mix commas & semicolons within a single **PRINT** statement if you want.

Summary

Statements: **PRINT**, with commas & semicolons

Operations: +,-,*,/,**

Expressions, scientific notation

Exercises

1. Try

PRINT 2.34E0

PRINT 2.34E1

PRINT 2.34E2

and so on up to

PRINT 2.34E15

You will see that after a while the ZX81 also starts using scientific notation. This is because it never takes more than 14 spaces to write a number in. Similarly, try

PRINT 2.34E-1

PRINT 2.34E-2

& so on.

2. Try

PRINT 1,,2,,3,,,4,,,,5

A comma always moves you on a bit for the next number. Now try

PRINT 1;;2;;3;;;4;;;;5

Why is a string of semicolons no different from a single one?

3. **PRINT** gives only 8 significant digits. Try

PRINT 4294967295, 4294967295 -429E7

This proves that the computer can hold all the digits of 4294967295, even though it is not prepared to display them all at once.

4. If you've got some log tables, then test out this rule:

Raising 10 to the power of a number is the same as taking the antilog of that number.

For instance, type

PRINT 10**0.3010

& look up the antilog of 0.3010. Why are the answers not exactly equal?

5. The ZX81 uses floating point arithmetic, which means that it keeps separate the digits of a number (its mantissa) and the position of the point (the exponent). This is not always exact, even for whole numbers. Type

PRINT 1E10+1-1E10,1E10-1E10+1

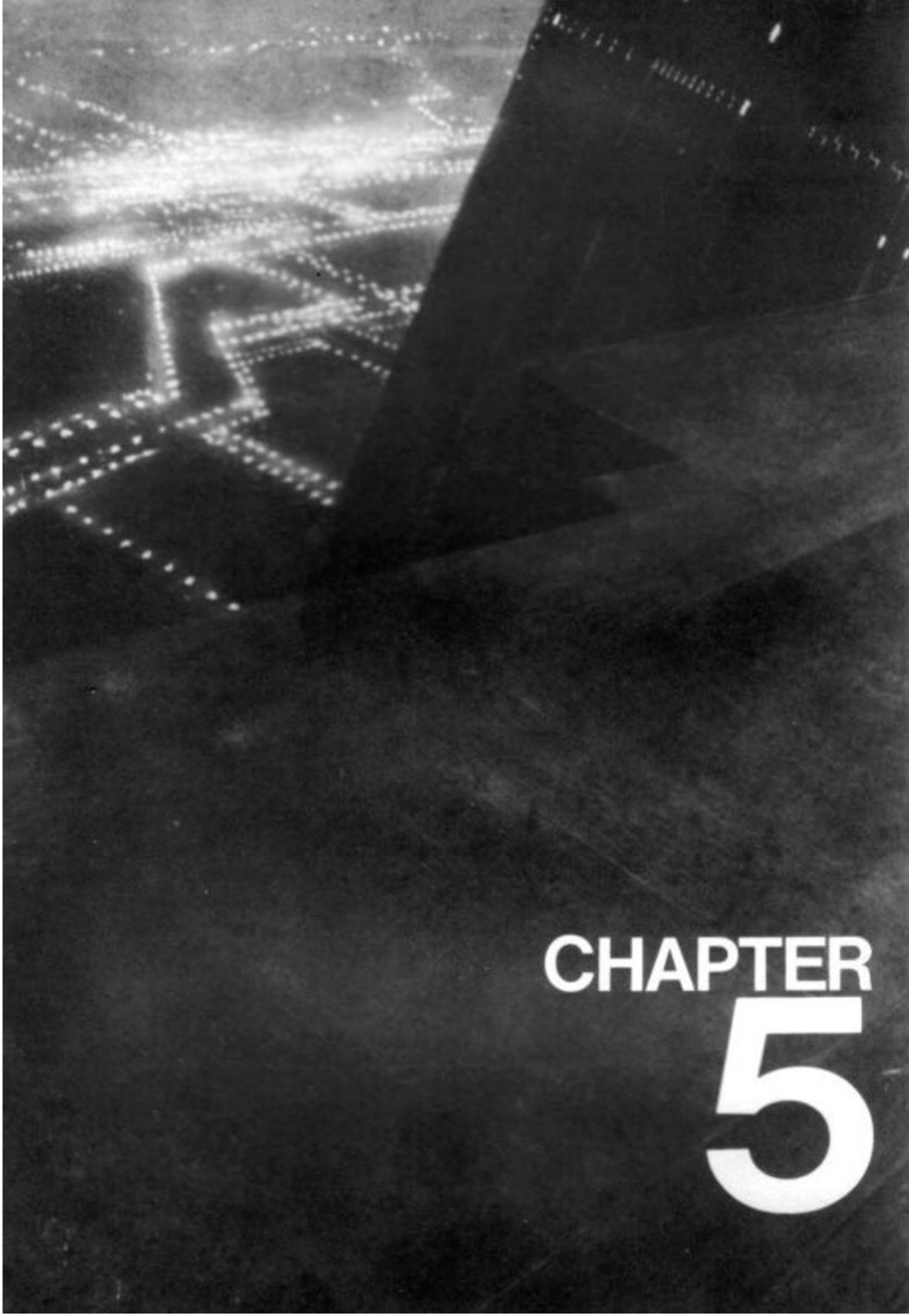
Numbers are held to about 9 1/2 digits accuracy, so 1E10 is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers 1E10 & 1E10+1 appear to the computer to be equal.

For an even more peculiar example, type

PRINT 5E9+1-5E9

Here the inaccuracy in 5E9 is only about 1, & the 1 to be added on in fact gets rounded up to 2. Here the numbers 5E9+1 & 5E9+2 appear to the computer to be equal.

The larger integer (whole number) that can be held completely accurately is 2^{32}-1 (4,294,967,295).



CHAPTER 5

Functions

Mathematically, a function is a rule for giving a number (the result) in exchange for another (the argument, or operand) & so is really a unary operation. The ZX81 has some of these built into it & their names are the words written under the keys. **SQR**, for instance, is the familiar square root function, and

PRINT SQR 9

gives 3, the square root of 9. (To get **SQR**, you first press the **FUNCTION** key - shifted **NEWLINE**. This changes the cursor to **F**. Now press the **SQR** key (H): **SQR** appears on the screen and the cursor changes back to **L**. The same method works for all the words that are written underneath the keys, almost all of which are function names.)

Try

PRINT SQR 2

You can test the accuracy of the answer by

PRINT SQR 2***SQR** 2

which ought to give 2. Note that both **SQR**s are worked out before the *, and in fact all functions (except one - **NOT**) are worked out before the five operations +, -, *, / and **. Again, you can circumvent this rule using brackets -

PRINT SQR (2*2)

gives 2.

Here are some functions (there is a complete list in appendix C). If your maths is not up to understanding some of these, it does not matter - you will still be able to use the computer.

SGN The sign function (sometimes called signum to avoid confusion with **SIN**). The result is -1, 0 or +1 according as the argument is negative, zero or positive.

ABS The absolute value, or modulus. The result is the argument made positive, so that

ABS -3.2 = **ABS** 3.2 = 3.2

SIN *

COS *

TAN *

ASN arcsin *

ACS arccos *

ATN arctan *

LN natural logarithm (to base 2.718281828459045..., alias e)

EXP exponential function

SQR square root

INT integer part. This always rounds down, so **INT** 3.9 = 3 & **INT** -3.9 = -4. (An integer is a whole number, possibly negative.)

PI π = 3.1415265358979..., the girth in cubits of a circle one cubit across. **PI** has no argument. (Only ten digits of this are actually stored in the computer, & only eight will be displayed.)

RND Neither has **RND** an argument. It yields a random number between 0 (which value it can take) & 1 (which it cannot).

* The trigonometrical functions. These work in radians, not degrees.

Using the jargon of the last chapter, all these except **PI** & **RND** are unary operations with priority 11. (**PI** & **RND** are nullary operations, because they have no operands.)

The trigonometrical functions, & **EXP**, **LN** & **SQR**, are generally calculated to 8 digits accuracy.

RND & **RAND**: These are both on the same key, but whereas **RND** is a function, **RAND** is a keyword, like **PRINT**. **RAND** is used for controlling the randomness of **RND**.

RND is not truly random, but follows a fixed sequence of 65536 numbers that happen to be so jumbled up as to appear random (**RND** is pseudo-random). You can use **RAND** to start **RND** off at a definite place in this sequence by typing **RAND**, & then a number between 1 & 65535, & then **NEWLINE**. It's not so important to know where a given number starts **RND** off, as that the same number after **RAND** will always start **RND** off at the same place. For instance, type

RAND 1 (& **NEWLINE**)

& then

PRINT RND

& type both these in turn several times. (Remember to use **FUNCTION** to get **RND**.) The answer from **RND** will always be 0.0022735596, not a very random sequence.

RAND 0

(or you can miss out the 0) acts slightly differently: it judges where to start **RND** off by how long the television has been on, & this should be genuinely random.

Note: In some dialects of BASIC you must always enclose the arguments of a function on brackets. This is not the case in ZX81 8K BASIC.

Summary

Statement: **RAND**

Functions: **SGN**. **ABS**, **SIN**, **COS**, **TAN**, **ASN**, **ACS**, **ATN**, **LN**, **EXP**, **SQR**, **INT**, **PI**, **RND**

FUNCTION

Exercises

1. To get common logarithms (to base 10), which are what you'd look up in log tables, divide the natural logarithm by **LN** 10. For instance, to find log 2,

PRINT LN 2/**LN** 10

which gives the answer 0.30103.

Try doing multiplication & division using logs, using the ZX81 as a set of log tables in this way (for antilogs, see chapter 2, exercise 3). Check the answer using * and / - which are easier, quicker, more accurate, & much to be preferred.

2. **EXP** & **LN** are mutually inverse functions in the same sense that if you apply one & then the other, you get back to your original number. For instance,

LN EXP 2 = **EXP LN** 2 = 2

The same also holds for **SIN** & **ASN**, for **COS** & **ACS**, & for **TAN** & **ATN**. You can use this to test how accurately the computer works out these functions.

3. π radians are 180°, so to convert from degrees to radians you divide by 180 & multiply by π: thus

PRINT TAN (45/180***PI**)

gives tan 45° (1).

To get from radians to degrees, you divide by π & multiply by 180.

4. Try

PRINT RND

a few times to see how the answer varies. Can you detect any pattern? (Unlikely.)

How would you use **RND** & **INT** to get a random whole number between 1 & 6, to represent the throw of a die? (Answer: **INT** (**RND** *6) +1.)

5. Test this rule:

Suppose you choose a number between 1 & 872 & type

RAND & then your number (& **NEWLINE**)

Then the next value of **RND** will be

(75 * (your number + 1) - 1)/65536

6. (For mathematicians only.)

Let p be a [large] prime, & let a be a primitive root modulo p.

Then if b_i is the residue of a^i modulo p ($1 \leq b_i < p-1$), the sequence

$$\frac{b_i - 1}{p - 1}$$

is a cylindrical sequence of p-1 distinct numbers in the range 0 to 1 (excluding 1). By choosing a suitably, these can be made to look fairly random.

65537 is a Mersenne prime, $2^{16}-1$. Use this & Gauss' law of quadratic reciprocity, to show that 75 is a primitive root modulo 65537.

The ZX81 uses p=65537 & a=75, & stores some b_i-1 in memory. The function RND involves replacing b_i-1 in memory by $b_{i+1}-1$, & yielding the result $(b_{i+1}-1)/(p-1)$. RAND n (with $1 \leq n \leq 65535$) makes b_i equal to n+1.

7. **INT** always rounds down. To round to the nearest integer, add 0.5 first. For instance,

INT (2.9+0.5) = 3 INT (2.4+0.5) = 2

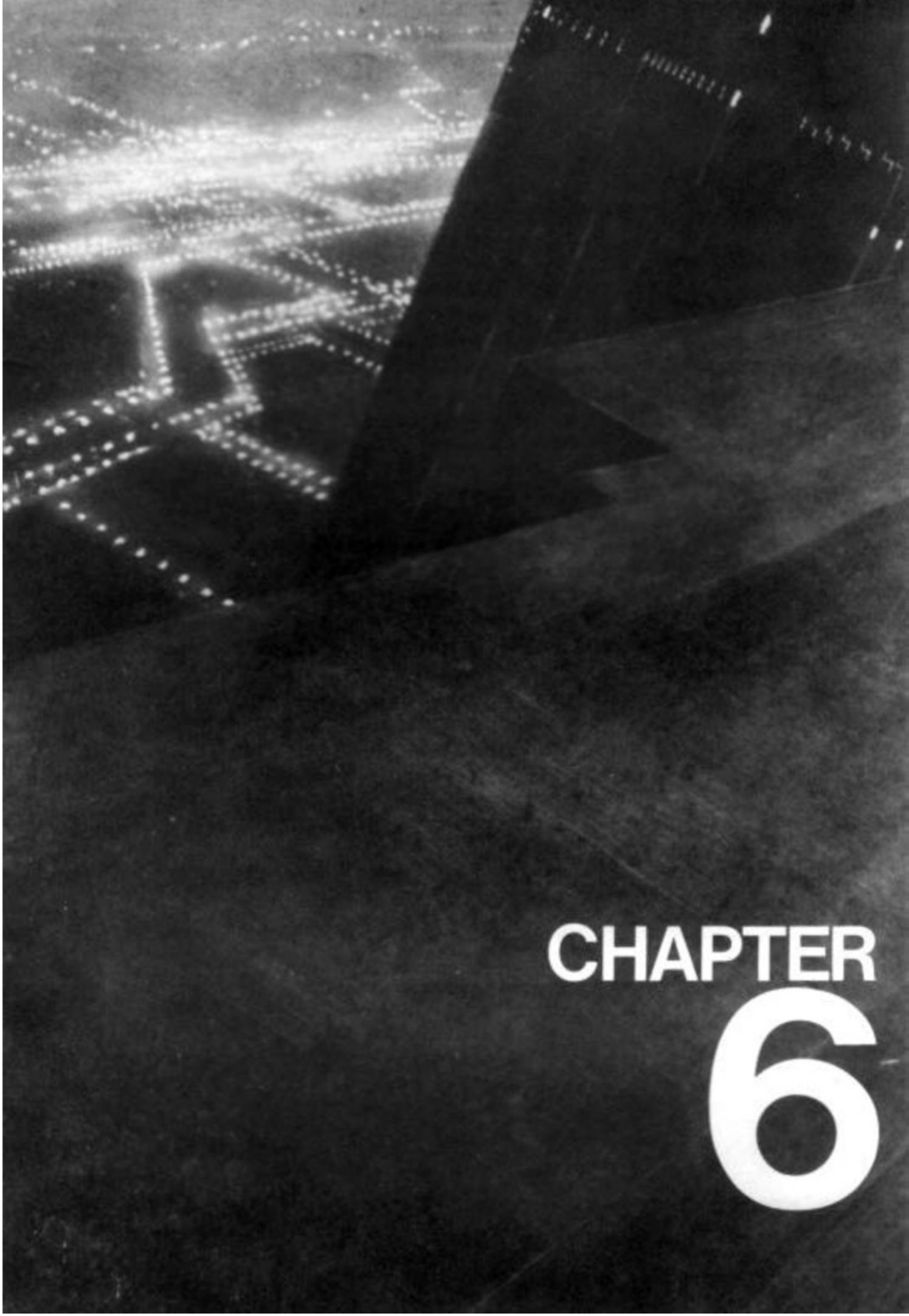
INT (-2.9+0.5) = -3 INT (-2.4+0.5) = -2

Compare these with the answers you get when you don't add 0.5

8. Try

PRINT PI, **PI** -3, **PI** -3.1, **PI** -3.14, **PI** -3.141

This shows how accurately the computer stores π.



CHAPTER 6

Variables

'My pocket calculator,' you will be saying, 'can store a number away & remember it later. Can your ZX81 do that?'

Yes. In fact it can store away literally hundreds, using the **LET** statement. Suppose that eggs cost 58p a dozen, & you want to remember this. Type

LET EGGS=58 (& **NEWLINE**, of course)

Now first, the computer has reserved a place inside itself where you can store a number, & second, it has given this place the name 'EGGS' so you can refer to it later. This combination of storage space & name is called a variable. Third, it has stored the number 58 in the space: we say that it has assigned the value 58 to the variable [whose name is] EGGS. EGGS is a numeric variable, because its value is a number.

Do you want to know how much eggs cost? Type

PRINT EGGS

If you want to know the cost of half a dozen eggs, then type

PRINT EGGS/2

In fact, should you want to know the square of the cosine of the price of one egg, you can type

PRINT COS (EGGS/12)**2

'How very easy,' you must think, & you will be wondering what to do next, when in rushes your housekeeper saying 'Glory be, eggs have just gone up to 61p a dozen.'

Well. There is no time to lose. Type

LET EGGS=61

This does not reserve any extra storage space, but replaces the old value of 58 with 61. Now you can type

PRINT EGGS

confident in the expectation of getting the most up-to-date price available.

Now type

PRINT MILK

You will get a report 2/0, & looking up 2 in appendix B, you will see that it means 'variable not found' - the computer hasn't the faintest idea how much milk costs, because you haven't told it. Type

LET MILK=18.5

& all will be all right.

(Type

PRINT MILK

again.)

A variable need not be named after groceries - you can use any letters or digits as long as the first one is a letter. You can put spaces in as well to make it easier to read, but they won't count as part of the name.

For instance, these are allowed to be the names of variables:

TWO POUNDS OF APPLES BUT NOT GOLDEN DELICIOUS
RADIO 3
RADIO 33
X
K9P

but these are not:

3 BEARS (begins with a digit)
TALBOT? (? is not a letter or a digit)
WHITE ON BLACK (inverse video characters not allowed)
FOTHERINGTON-THOMAS (- is not a letter or a digit)

Now type

CLEAR

&

PRINT EGGS

You will get report 2 (variable not found) again. The effect of **CLEAR** is to release all the storage space that had been reserved for variables - then every variable is as though it had never been defined. Turning the computer off & on will also do this - but then it doesn't remember anything at all when it is turned back on.

Expressions can contain the name of a variable anywhere they can have a number.

Note: In some versions of BASIC you are allowed to omit **LET** & just type in (say)

EGGS=58

This is not allowed on the ZX81. In any case, you'd find it rather difficult to type in.

Also in some versions, only the first two characters in a name are checked, so that RADIO 3 & RADIO 33 would count as the same name; & in some others a variable name must be a letter followed by a digit. Neither of these restrictions applies to the ZX81.

Yet again, in some versions of BASIC, if a variable has not yet appeared on the left-hand side of a **LET** statement then it is assumed to have a value 0. As you saw above with **PRINT** MILK, this is not so on the ZX81.

Summary

Variables

Statements: **LET**, **CLEAR**

Exercises

1. Why do variable names (that is to say, names of variables) have to begin with a letter?

2. If you're unfamiliar with raising to powers (**, shifted H) then do this exercise.

At its most elementary level, 'A**B' means 'A multiplied by itself B times', but obviously this only makes sense if B is a positive whole number. To find a definition that works for other values of B, we consider the rule

A ** (B+C) = A**B * A**C

You should not need much convincing that this works when B & C are both positive whole numbers, but if we decide that we want it to work even when they are not, then we find ourselves compelled to accept that

A**0 = 1

A**(-B) = 1/A**B

A**(1/B) = the Bth root of A

&

A**(B*C) = (A**B)**C

If you've never seen any of this before then don't try to remember it straight away; just remember that

A**-1 = 1/A

&

A**(1/2) = the square root of A

& maybe when you're familiar with these the rest will begin to make sense.

Experiment with all this by telling the computer to print various expressions containing **: e.g.

PRINT 3**(2+0),3**2*3**0

PRINT 4**-1,1/4

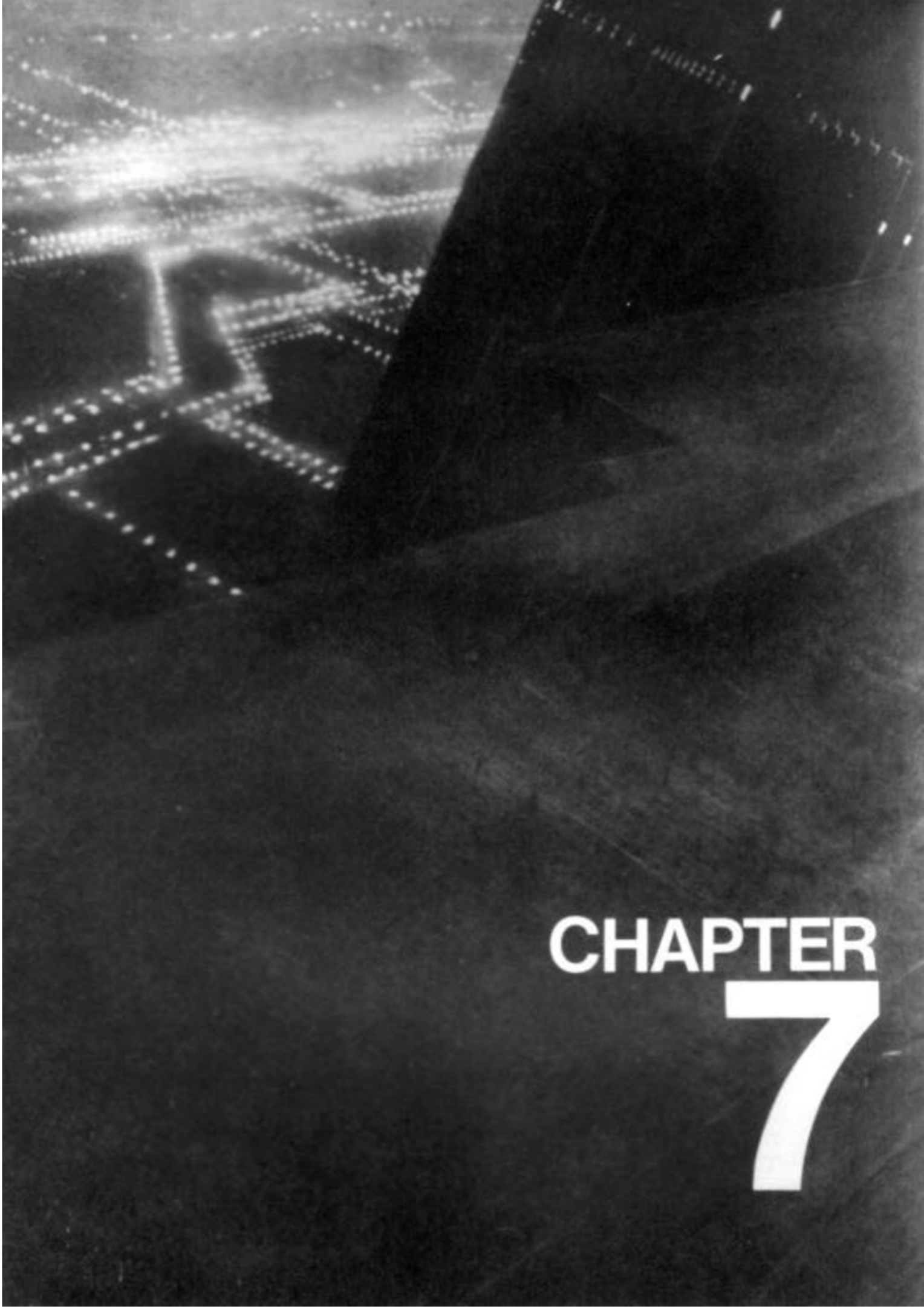
3. Type

LET E=**EXP** 1

Now E has the value 2.718281828..., the base of natural logarithms. Test the rule

EXP a number = E ** the number

for various numbers.



CHAPTER 7

Strings

One thing the ZX81 can do that pocket calculators cannot is deal with text. Type

PRINT "HI THERE. I AM YOUR ZX81." (" is shifted P.)

The chilling greeting inside the quotes is called a string (meaning a string of characters), & can contain any characters you like except the string quote,". (But you can use the so-called quote image, "" (shifted Q), & this will be printed as " by a **PRINT** statement.)

A common typing error with strings is to miss out one of the quotes - this will give you the S marker.

If you are printing numbers out, you can use these strings to explain what the numbers mean. For instance, type

LET EGGS=61

& then

PRINT "THE PRICE OF EGGS IS ";EGGS;" NEW PENCE A DOZEN."

(Don't worry about this going over the end of the line.)

This statement displays three things (**PRINT** items), namely the string "THE PRICE OF EGGS IS ", the number 61 (the value of the variable EGGS), & then the string " NEW PENCE A DOZEN." In fact you can PRINT any number of items you like, & any mixture of strings & numbers (or expressions), note how the spaces in a string are just as much part of it as the letters. They are not ignored even at the end.

There are lots of things you can do with strings.

1. You can assign them to variables. However, the name of the variable must be special to show that its value is a string & not a number: it must be a single letter followed by $ (shifted U). For instance, type

LET A$="DOUBLE GLOUCESTER"

&

PRINT A$

2. You can add them together. This is often called concatenation, meaning 'chaining together', & that is exactly what it does. Try

PRINT "GAMMO" + "N RASHERS"

You cannot subtract, multiply or divide strings, or raise them to powers.

3. You can apply some functions to strings to get numbers, & vice versa.

LEN This is applied to a string, & the result is its length. For instance **LEN** "CHEESE" = 6.

VAL This applied to a string, & the result is what that string gives when evaluated as an arithmetic expression. For instance (if A = 9), **VAL** "1/2+**SQR**A" = 3.5. If the string to which **VAL** is applied contains variables, then two rules must be obeyed.

(i) If the **VAL** function is part of a larger expression, it must be the first item; e.g. 10 **LET** X = 7+**VAL** "Y" must be changed to 10 **LET** X = **VAL** "Y" +7.

(ii) **VAL** can only appear in the first coordinate of a **PRINT AT**, **PLOT** or **UNPLOT** statement (see Chapter 17 and 18) e.g. 10 **PLOT** 5, **VAL** "X" must be changed to

10 **LET** Y = **VAL** "X"

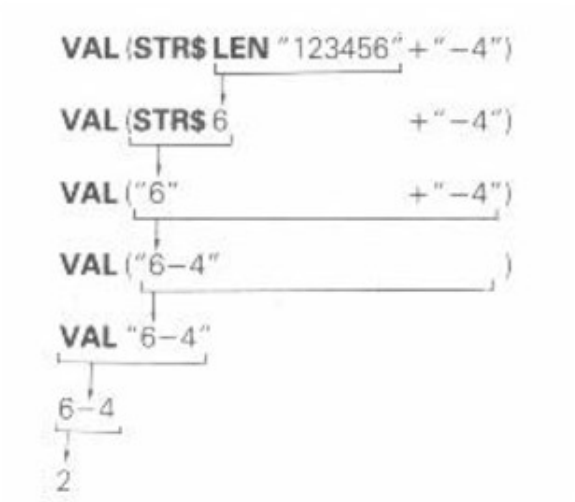
15 **PLOT** 5, Y

STR$ This is applied to a number, & the result is what would appear on the screen if the number were displayed by a **PRINT** statement. For instance **STR$** 3.5 = "3.5".

4. Just as for numbers, you can combine these to make string expressions, like

VAL (**STR$ LEN** "123456"+"-4")

which is evaluated as



Summary

Strings

Operation: + (for strings)

Functions: **LEN**, **VAL**, **STR$**

Exercises

1. Type

```
LET A$="2+2"
```

& then

```
PRINT A$;" = ",VAL A$
```

Try changing A$ to more complicated things & doing the same, e.g.

```
LET A$="ATN 1*4"
```

(The answer here should be π.)

2. The string "" with no characters is called the empty or null string. It is only string whose length is 0. Remember that spaces are significant and an empty string is not the same as one containing spaces.

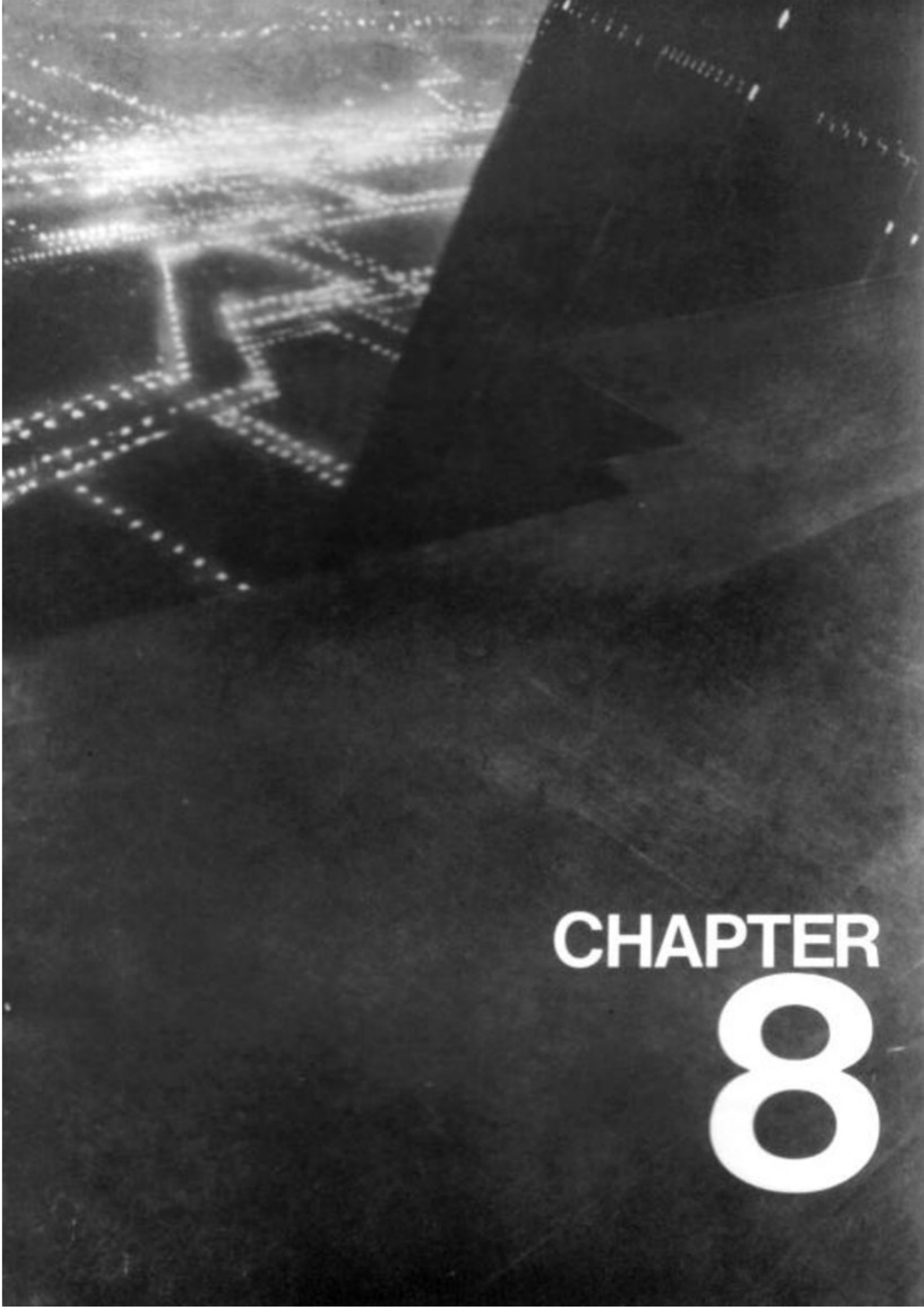
Do not confuse it with the quote image, **""** (a single token, shifted Q). This is a special device to get over the fact that you cannot write an ordinary string quote in the middle of a string (why not?). When the quote image appears in a string that has its quotes at the end (for instance in the listing of a program), it shows up as two quote symbols, to distinguish it from the ordinary quote; but when it is displayed by a **PRINT** statement, it is as just one quote symbol.

Try

```
PRINT "X";"";"X","""".""""";"";""""
```

3. If you enjoy humiliating computers, type

```
PRINT "2+2 = ";2+1
```



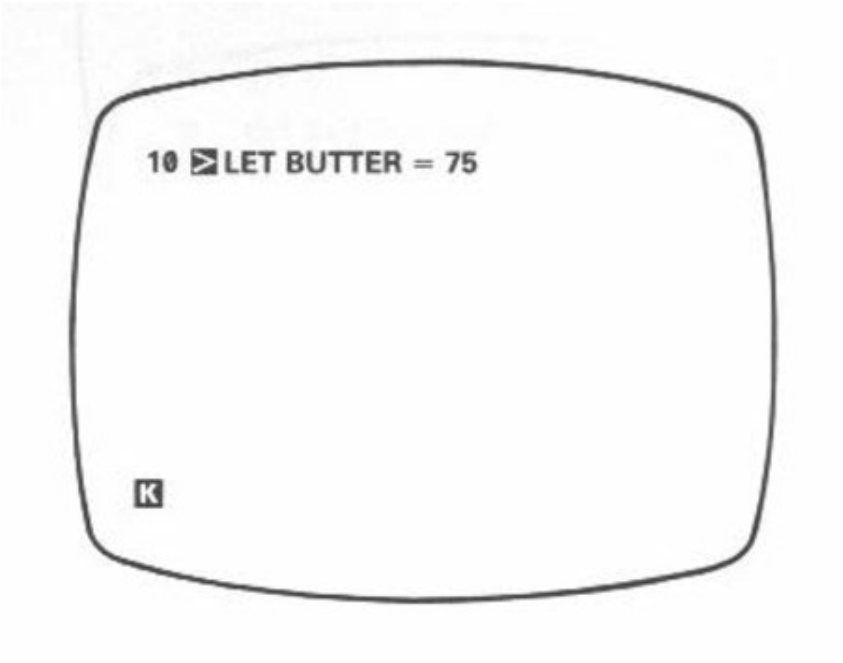
CHAPTER 8

Computer programming

And now at last you shall write a computer program. Turn the computer off & on, just to make sure that it is clear. Now type

10 **LET** BUTTER=75 (& **NEWLINE**)

& the screen will look like this:



This is different from what happened with EGGS in chapter 6; if you type

PRINT BUTTER

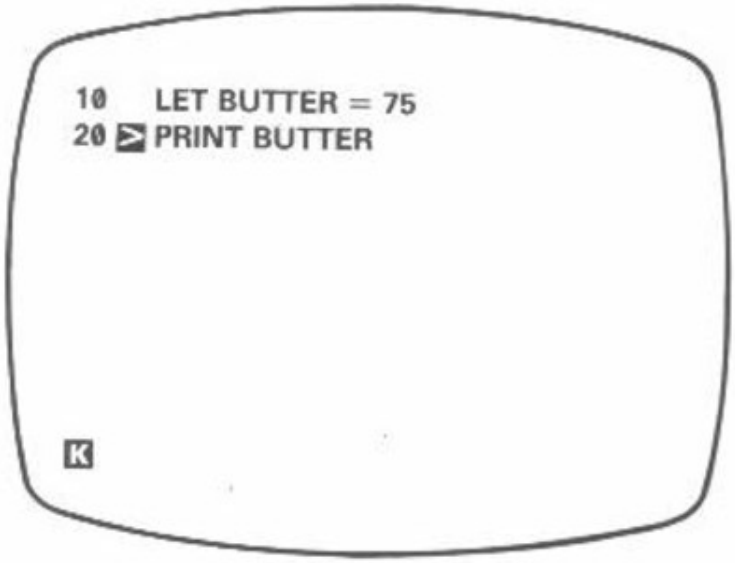
you will see (from the report 2) that the variable BUTTER has not been set up. (Press **NEWLINE** again & the screen should go back to looking like the picture.)

Because the **LET** statement had a number, 10, in front of it, the computer did not execute it straight away, but saved it for later. 10 is its line number, & is used to refer to it rather in the same way that names are used to refer to variables. A set of these stored statements is called a

program. Now type

20 **PRINT** BUTTER

& the screen should look like this:



This is a listing of your program. To have the program carried out (or executed or run), type

RUN (don't forget the **NEWLINE**)

& the answer 75 will appear in the top left-hand corner of the screen. At the bottom left-hand corner you will see the report 0/20. 0, as you know, means 'OK, no problems', & 20 is the number of the line where it finished. Press **NEWLINE**, & the listing will come back.

Note that the statements were executed in the order of their line numbers.

Now suppose you suddenly remember that you also need to record the price of yeast. Type

15 **LET** YEAST=40

& in it goes. This would have been much harder if the first two lines had been numbered 1 & 2 instead of 10 & 20 (line numbers must be whole numbers between 1 & 9999), so that is why, when first typing in a program, it is good practice to leave gaps in the line numbers.

Now you need to change line 20 to

20 **PRINT** BUTTER, YEAST

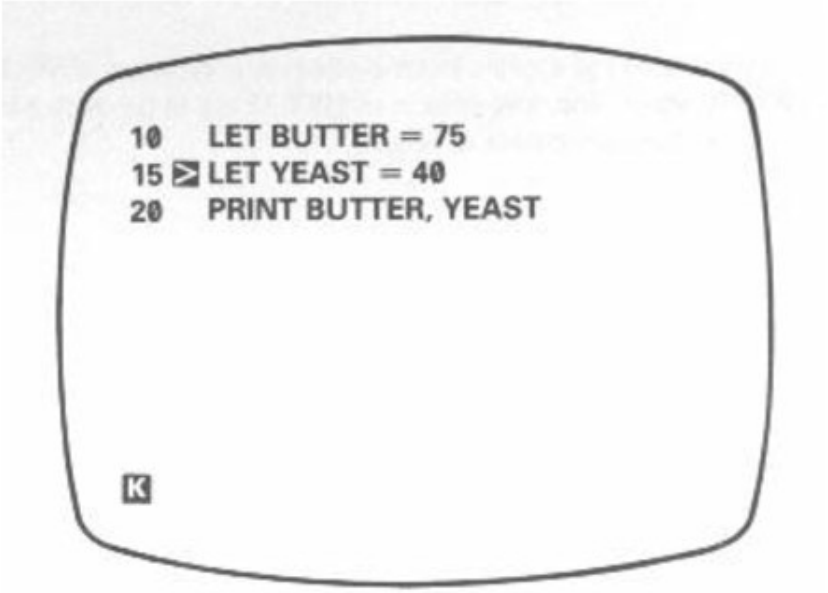
You could type out the replacement in full, but there is a way to use what is there already. You see that little > by line 15? This is the program cursor, & the line it points to is the current line. Press the ⇩ key (shifted 6), & it will move down to line 20. (⇧ moves it up again.) Now press the **EDIT** key (shifted 1), & a copy of line 20 will be displayed at the bottom of the screen. Press the ⇨ key 7 times so that the L cursor moves to the end of the line, & then type

,YEAST (without **NEWLINE**)

The line at the bottom should now read

20 PRINT BUTTER, YEAST**L**

Press **NEWLINE** & it will replace the old line 20. The screen will now look like this:



RUN this program & both prices will be displayed.

(Here is a useful trick involving **EDIT**, to use when you want to clear the bottom part of the screen altogether. Press **EDIT,** & the current line will be brought down from the program, replacing what you wanted deleting. If you now press **NEWLINE**, the line will be put back in the program, making no difference to it, & the bottom part of the screen will be cleared leaving just the cursor.)

Now type - in a fit of absent-mindedness –

12 **LET** YEAST=40

This will go up into the program & you will realise your mistake. To delete this unnecessary line, type

12 (with **NEWLINE**, of course)

You will notice with surprise that the program cursor has gone. You should imagine it as being hidden in between lines 10 & 15, so if you press ⇧ it will move up to line 10, while if you press ⇩ it will move down to line 15.

Last, type

LIST 15

You will now see on the screen

15>LET YEAST=40

20 PRINT BUTTER, YEAST

Line 10 has vanished from the screen, but it is still in your program - which you can prove by pressing **NEWLINE** again. The only effect of **LIST** 15 are to produce a listing that starts at line 15, & to put the program cursor at line 15.

LIST

on its own makes the listing start at the beginning of the program.

Summary

Programs

Editing programs using ⇧, ⇩ & **EDIT**.

Statements: **RUN**, **LIST**

Exercises

1. Modify the program so that it displays not only the two prices, but also messages to show which is which.

2. Use the **EDIT** key to change the price of butter.

3. Run the program & then type

PRINT BUTTER, YEAST

The variables are still there, even though the program has finished.

4. Type

12 (& **NEWLINE**)

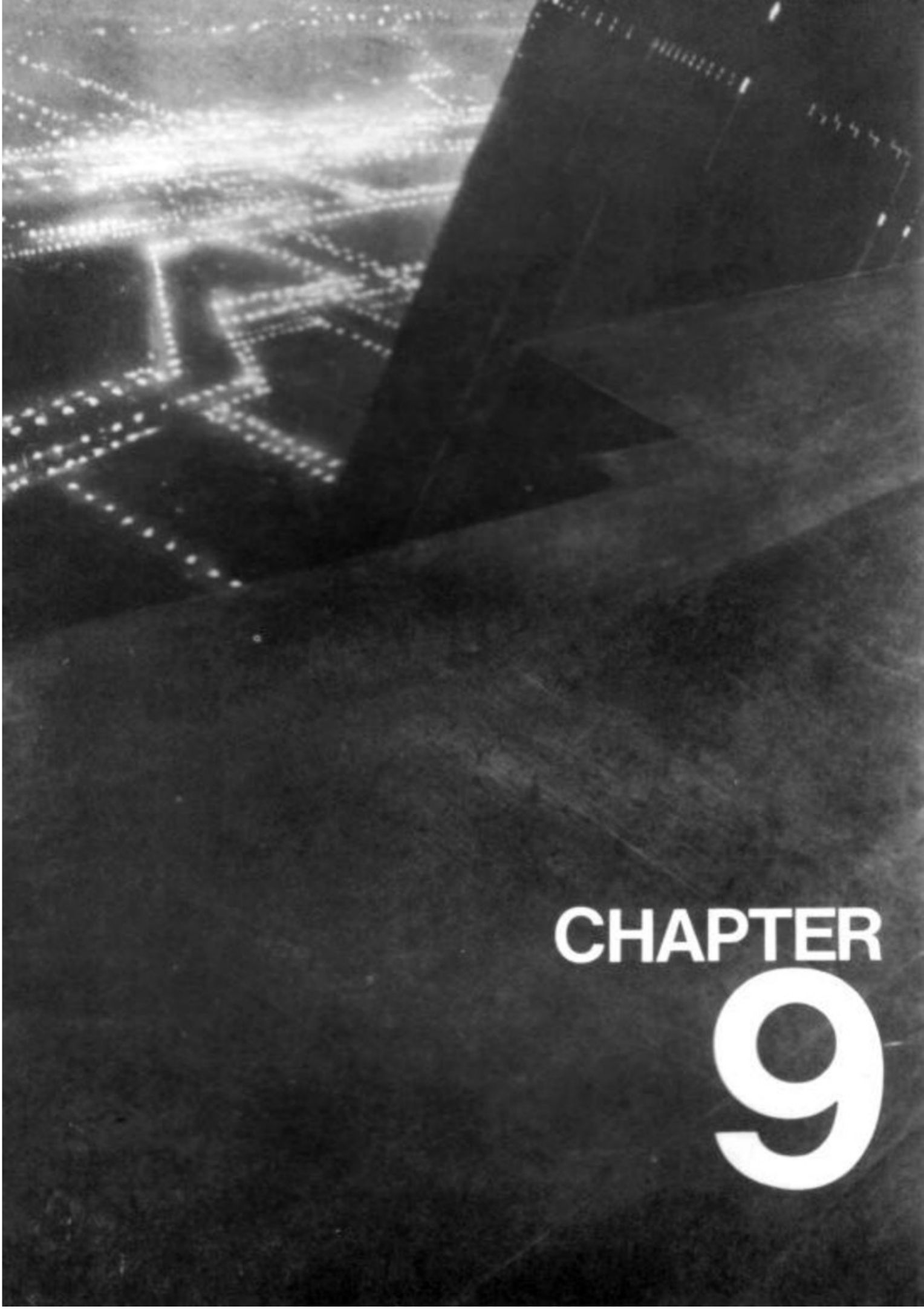
Again, the program cursor will be hidden between lines 10 & 15. Now press **EDIT**, & line 15 will come down: when the program cursor is hidden between two lines, **EDIT** brings down the second one. Type **NEWLINE** to clear the bottom part of the screen.

Now type

30

This time, the program cursor is hidden after the end of the program; & if you press **EDIT**, then line 20 will be brought down.

5. Put a **LIST** statement in the program so that when you run it, it lists itself.



CHAPTER 9

More computer programming

Type

NEW

This will erase any old programs & variables in the ZX81. (This is rather like **CLEAR**, but **CLEAR** only erases the variables.) Now carefully type in this program:

```
10 REM THIS PROGRAM EXTRACTS SQUARE ROOTS
20 INPUT A
30 PRINT A,SQR A
40 GOTO 20
```

(You will need to type in most of the spaces yourself in line 10).

Now run it. Apparently the screen goes blank, & nothing happens, but look at the cursor in the bottom left-hand corner: where you might have expected a K there is instead an L - the machine has gone into input mode. This is the effect of the **INPUT** statement in line 20. The machine is waiting for you to type in a number (or even an expression), & it won't carry on until you have. After that, it will have the effect of

20 **LET** A=... whatever you typed

Just a minute though, what happened to line 10? It looks as though the computer has completely ignored it. Well, it has. **REM** in line 10 stands for remark, or reminder, & there solely to remind you of what the program does. A **REM** statement consists of **REM** followed by anything you like, & the computer will ignore it.

All right, so we're in input mode for line 20. Type some number, 4, say, & then **NEWLINE**. The 4 & its square root appear on the screen, & you might think that was that. But, no - it seems to want another number. This is because of line 40, **GOTO** 20, which means exactly what it says. Instead of running out of program & stopping, the computer jumps back to line 20 & starts again. So, type another number (2, say; at any rate you had better make it positive).

After a few more of these you might be wondering if the machine will ever get bored with this game; it won't. Next time in its instability it asks for another number, type **STOP** (shifted A) instead; it will take the hint. The computer reports back with report D/20 - look up D in the list of reports (appendix B). 20 is the line where it was waiting for some input when you stopped it.

Have you suddenly remembered some more numbers that you wanted the square root of? Then type

CONT

(short for CONTINUE) & the computer will clear the screen & ask you for another number.

For **CONT**, the computer remembers the line number in the last report that it sent you that had a code other than 0, & jumps to that line. Since the last report was D/20 (& D is not 0), in our case **CONT** is identical to **GOTO** 20.

Now type in numbers until the screen starts getting full. When it is full, the computer will stop with the report 5/30. 5 means 'screen full', & 30 is the number of the **PRINT** statement it was trying to execute when it discovered there was no room. Again,

CONT

will clear the screen & carry on - this time, **CONT** means **GOTO** 30.

Note that the screen is cleared not because this is a **CONT** statement, but because it is a command. All commands (except **COPY**, which appears in chapter 20) clear the screen first.

When you're tired of this, stop the program using **STOP** & get the listing by pressing **NEWLINE**.

Look at the **PRINT** statement on line 30. Each time the pair of numbers A & **SQR** A is printed, it is on a new line, & this is because the **PRINT** statement does not end with a comma or semicolon. Whenever this is the case, then the next **PRINT** statement starts printing on a new line. (Thus to put in a blank line, use a **PRINT** statement in which there is nothing to be printed - just **PRINT** on its own.)

However, a PRINT statement can end in a comma or semicolon, & then the next **PRINT** statement carries on printing as though the two had been one long statement.

For instance, with commas, replace the 30 by

30 **PRINT** A,

& run the program to see how successive **PRINT** statements can print on the same line but spread out in two columns.

With semicolons, on the other hand, with

30 **PRINT** A;

everything is jammed together.

Try also

30 **PRINT** A

Now type in these extra lines.

100 **REM** THIS PROGRAM MEASURES STRINGS

110 **INPUT** A$

120 **PRINT** A$,LEN A$

130 **GOTO** 110

This is a separate program from the last one, but you can keep them both in at the same time. To run the new one, type

RUN 100

This program inputs a string instead of a number, & prints it & its length. Type

CAT (& **NEWLINE**, as usual)

Because the computer is expecting a string, it prints out two string quotes - this is a reminder to you, & it usually saves you some typing as well. But you don't have to restrict yourself to string constants (explicit string with opening & closing quotes & all their characters in between); the computer will evaluate any string expression, such as one with string variables. In this case you might have to rub out the quotes that the computer has displayed. Try this. Rub them out (with ⇨ & **RUBOUT** twice), & type

A$

Since A$ still has the value "CAT", the answer is CAT 3 again.

Now input

A$

again, but this time without rubbing out the string quotes. Now A$ has the value "A$", & the answer is 2.

If you want to use **STOP** for string input, you must first move the cursor back to the beginning of the line, using ⇦.

Now look back at the **RUN** 100 we had earlier on. That jumps to line 100, so couldn't we have said **GOTO** 100 instead? In this case, it so happens that the answer is yes, byt there is a difference. **RUN** 100 first of all clears the variables (like **CLEAR** in chapter 6), & after that works just like **GOTO** 100. **GOTO** 100 doesn't clear anything. There may well be occasions when you want to run a program without clearing any variables. Here **GOTO** is necessary & **RUN** could be disastrous, so it is best to get into the habit of automatically typing **RUN** to run a program.

Another difference is that you can type RUN without a line number, & it starts off at the first line in the program. **GOTO** must always have a line number.

Both these programs stopped because you typed **STOP** in the **INPUT** line; but sometimes - by mistake - you write a program that you can't top & won't stop itself. Type

200 **GOTO** 200

RUN 200

This looks all set to go on for ever unless you pull the plug out; but there is a less drastic remedy. Press the **SPACE** key, which has 'BREAK' written above it. The program will stop, saying D/200.

At the end of every program line, the computer looks to see if this key is pressed; & if it is, then it stops. The **BREAK** key can also be used when you are in the middle of using the tape recorder or the printer.

You have now seen the statements **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GOTO**, **CONT**, **CLEAR**, **NEW** & **REM**, & most of them can be used either as commands or as program lines - this is true of almost all statements in BASIC. The only real exception is **INPUT**, which cannot be used as a command (you get report 8 if you try; the reason is that the same area inside the computer is used for both commands & for input data, & for an **INPUT** command there would be a muddle). **RUN**, **LIST**, **CONT**, **CLEAR** & **NEW** are not usually much use in a program, but they can be used.

Summary

Statements: **GOTO**, **CONT**, **INPUT**, **NEW** , **REM**, **PRINT**

STOP in input data

BREAK

Exercises

1. In the square root program, try replacing line 40 by **GOTO** 5, **GOTO** 10 or **GOTO** 15 - it should make no perceptible difference to the running of the program. If the line number in a **GOTO** statement refers to a non-existent line, then the jump is to the next line after. The same goes for **RUN**; in fact **RUN** on its own actually means **RUN** 0.

2. Run the string length program, & when it asks you for input type

X$ (after removing the quotes)

Of course, X$ is an undefined variable & you get report 2/110.

If you now type

LET X$="SOMETHING DEFINITE"

(which has its own report of 0/0) &

CONT

you will find that you can use X$ as input data without any trouble.

The point about this exercise is that **CONT** has the effect of **GOTO** 110. It disregards the report 0/0 because that had code 0, & takes its line number from the previous report, 2/110. This is intended to be useful. If a program stops over some error then you can do all sorts of things to fix it, & **CONT** will start work afterwards.

3. Try this program:

```
10 INPUT A$
20 PRINT A$;" = ";VAL A$
30 GOTO 10
```

(c.f. chapter 7, exercise 1).

Put in extra print statements so that the computer announces what it is going to do, & asks for the input string with extravagant politeness.

4. Write a program to input prices & print out the VAT due (at 15%). Again, put in **PRINT** statements so that it tells you what it's doing. Modify the program so that you can also input the VAT rate (to allow for zero rating or future budgets).

5. Write a program to print a running total of numbers you input. (Suggestion: have two variables TOTAL - set to 0 to begin with - & ITEM. Input ITEM, add it to TOTAL, print them both, & go round again.)

6. The automatic listings (the ones that are not the result of a **LIST** statement) may well have you puzzled. If you type in a program with 50 lines, all **REM** statements,

```
1 REM
2 REM
3 REM
: :
```

: :

49 **REM**

50 **REM**

then you will be able to experiment.

The first thing to remember is that the current line (with >) will always appear on the screen, & preferably near the middle.

Type

LIST (& **NEWLINE**, of course)

& then press **NEWLINE** again. You should get lines 1 to 22 on the screen. Now type

23 **REM**

& you should get lines 2 to 23 on the screen; type

28 **REM**

& you get lines 27 to 48. (In both cases, by typing in a new line you have moved the program cursor so that a new listing has to be made.)

Does this look a little arbitrary to you? It is actually trying to give you exactly what you want, although, humans being unpredictable creatures, it doesn't always guess right.

The computer keeps a record not only of the current line, the one that has to appear on the screen, but also the top line on the screen. When it tries to make a listing, the first thing it does is compare the top line with the current line.

If the top line comes after, then there is no point in starting there, so it uses the current line for a new top line & makes its listing.

Otherwise, it first tries to make the listing starting at the top line. If the current line gets on the screen then all is well; if the current line is only just off the bottom of the screen then it moves the top line down one & tries again; & if the current line is way off the bottom of the screen then it changes the top line to be the line before the current line.

Experiment with moving the current line about by typing

line number **REM**

LIST moves the cursor line but not the top line, so subsequent listings might be different. For instance, type

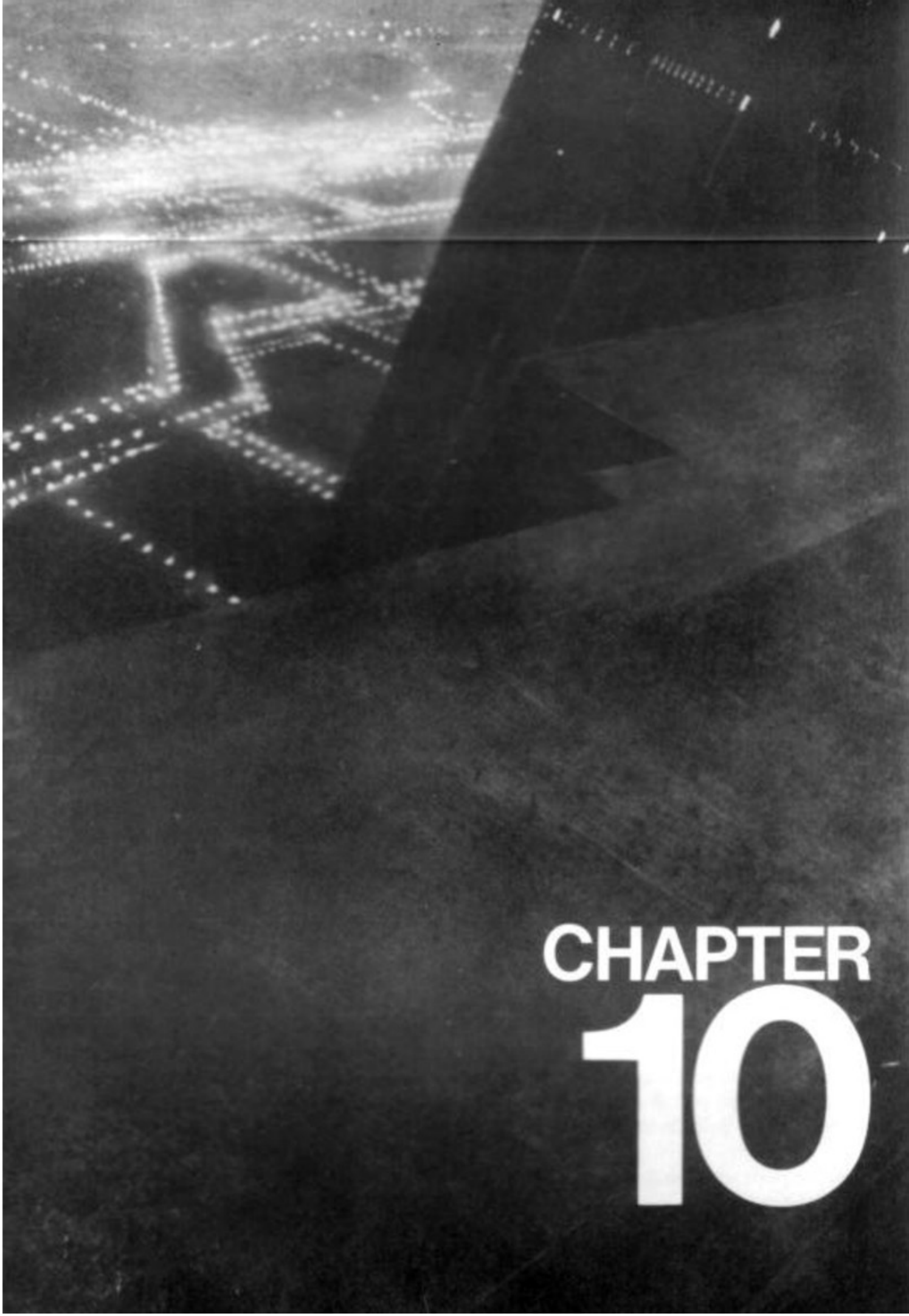
LIST

to get the LIST listing, & then press NEWLINE again to make line 0 the top line. You should have lines 1 to 22 on the screen. Type

LIST 22

which gives you lines 22 to 43; when you press **NEWLINE** again, you get back lines 1 to 22. This tends to be more useful for short programs than for long ones.

7. What would **CONT**, **CLEAR** & **NEW** do in a program? Can you think of any uses at all for this?



CHAPTER 10

If...

All the program we've seen so far have been pretty predictable - they went straight through doing all the instructions, & then maybe went back to the beginning again. This is not all that useful. In practice the computer would be expected to make decisions & act accordingly; it does this using the **IF** statement.

Clear the computer (using **NEW**), & type in & run this terribly amusing little program:

```
10 PRINT "SHALL I TELL YOU A JOKE?"
20 INPUT A$
30 IF A$="GET LOST" THEN GOTO 200
40 PRINT "HOW MANY LEGS HAS A HORSE GOT?"
50 INPUT LEGS
60 IF LEGS=6 THEN GOTO 100
70 PRINT "NO, 6, FORE LEGS IN FRONT","AND TWO BEHIND."
80 STOP
100 PRINT "YES",,"SHALL I TELL YOU IT AGAIN?"
110 GOTO 20
200 PRINT "ALL RIGHT, THEN, I WONT."
```

Before we discuss the **IF** statement, you should first look at the **STOP** statement in line 80: a **STOP** statement stops the execution of the program, giving report 9.

Now as you can see, an **IF** statement takes the form

IF condition **THEN** statement

The statements here are **GOTO** statements, but they could be anything at all, even more **IF** statements. The condition is something that is going to be worked out as either true or false; if it comes out as true then the statement after **THEN** is executed, but otherwise it is skipped over.

The most useful conditions compare two numbers or two strings: they can test whether two numbers are equal, or whether one is bigger than the other; & can test whether two strings are equal, or whether one comes before the other in alphabetical order. They use the relations =, <, >, **<=**, **>=** and **<>**.

=, which we have used twice in the program (once for numbers & once for strings) means 'equals'. It is not the same as the = in a **LET** statement.

< means 'is less than', so that

1<2

-2<-1

& -3<1

are all true, but

1<0

& 0<-2

are false.

To see how this works, let us write a program to input numbers & display the biggest so far.

```
10 PRINT "NUMBER","BIGGEST SO FAR"
20 INPUT A
30 LET BIGGEST=A
40 PRINT A,BIGGEST
50 INPUT A
60 IF BIGGEST<A THEN LET BIGGEST=A
70 GOTO 40
```

The crucial part is line 60, which updates BIGGEST if its old value was smaller than the new input number A.

> (shifted M) means 'is greater than', & is just like < but the other way round. You can remember which is which, because the thin end points to the number that is supposed to be smaller.

<= (shifted R - do not type it as < followed by =) means 'is less than or equal to', so that it is like < except that it holds even if the two numbers are equal: this 2<=2 holds, but 2<2 does not.

>= (shifted Y) means 'is greater than or equal to' & is similarly like >.

<> (shifted T) means 'is not equal to' the opposite in meaning to =.

All six of these relational operations have priority 5.

Mathematicians usually write <=, >= and <> as ≤, ≥ and ≠. They also write things like '2<3<4' to mean '2<3 and 3<4', but this is not possible in BASIC.

These relations can be combined using the logical operations **AND**, **OR** & **NOT**.

one relation **AND** another relation

is true whenever both relations are true.

one relation **OR** another

is true whenever one of the two relations is true (or both are).

NOT relation

is true whenever the relation is false and is false whenever the relation is true.

Logical expressions can be made with relations & **AND**, **OR** & **NOT** just as numerical expressions can be made with numbers & +, - and so on; you can even put in brackets if necessary. **NOT** has priority 4, **AND** 3 & **OR** 2.

Since (unlike other functions) **NOT** has fairly low priority, its argument does not need brackets unless it contains **AND** or **OR**; so **NOT** A = B means **NOT** (A = B) (which is the same as A <> B).

To illustrate this, clear the computer & try this program.

```
10 INPUT F$
20 INPUT AGE
30 IF F$="X" AND AGE<18 OR F$="AA" AND AGE<14 THEN PRINT "DONT";
40 PRINT "LET IN."
50 GOTO 10
```

Here F$ is supposed to be the category if a film - X for 18 years old & over, AA for 14 & over, & A or U for anyone, & the program works out whether a person of a given age is to be allowed to see the film.

Lastly, we can compare not only numbers, but also strings. We have seen '=' used in 'F$="X"', & you can even use the other five, < & so on.

So what does 'less than' mean for strings? One thing it does not mean is 'shorter than', so don't make that mistake. We make the definition that one string is less than another if it comes first in alphabetical order: thus

"SMITH"	< "SMYTHE"
"SMYTHE"	> "SMITH"
"BLOGGS"	< "BLOGGS-BLACKBERRY"
"BILLION"	< "MILLION"
"TCHAIKOVSKY"	< "WAGNER"
"DOLLAR"	< "POUND"

all hold. **<=** means 'is less than or equal to', & so on, just as for numbers.

Note: In some versions of BASIC - but not on the ZX81 -, the **IF** statement can have the form

IF condition **THEN** line number

This means the same as

IF condition **THEN GOTO** line number

Summary

Statements: **IF**, **STOP**

Operations: =, <, >, **<=**, **>=**, **<>**, **AND**, **OR**

Function: **NOT**

Exercises

1. **<>** and = are opposites in the sense that **NOT** A=B is the same as A**<>**B

&

NOT A**<>**B is the same as A=B

Persuade yourself that **>=** is opposite to <, and **<=** is opposite to > so that you can always get rid of **NOT** from in front of a relation by changing the relation to its opposite.

Also,

NOT (a first logical expression **AND** a second)

is the same as

NOT (the first) **OR NOT** (the second),

&

NOT (a first logical expression **OR** a second)

is the same as

NOT (the first) **AND NOT** (the second).

Using this you can work **NOT**s through brackets until eventually they are all applied to relations, & then you can get rid of them. Thus, logically speaking, **NOT** is unnecessary. You might still find that using it makes a program clearer.

2. BASIC can sometimes work along different lines from English. Consider, for instance, the English clause 'if A doesn't equal B or C'. How would you write this in BASIC? [The answer is not

'**IF** A<>B **OR** C' nor '**IF** A<>B **OR** A<>C']

Don't worry if you don't understand exercises 3, 4 & 5, the points covered in them are rather refined.

3. (For experts.)

Try

PRINT 1=2,1<>2

which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value.

(i) =, <, >, **<=**, **>=**, and **<>** are all number valued binary operations, with priority 5. The result is 1 (for true) if the relation holds, & 0 (for false) if it does not.

(ii) In

IF condition **THEN** statement

the condition can actually be any numeric expression. If its value is 0, then it counts as false, & any other value counts as true. This the **IF** statement means exactly the same as

IF condition **<>**0 **THEN** statement

(iii) **AND**, **OR** & **NOT** are also number valued operations.

X **AND** Y has the value	{ X if Y is non-zero (counting as true) { 0 if Y is zero (counting as false)
X **OR** Y has the value	{ 1 if Y is non-zero { X if Y is zero
NOT X has the value	{ 0 if X is non-zero { 1 if X is zero

Read through the chapter again, in the light of this revelation, making sure that it all works.

In the expressions X **AND** Y, X **OR** Y & **NOT** X, X and Y will each usually take the value 0 or 1, for false or true. Work out the ten different combinations & check that they do what you expect **AND**, **OR** & **NOT** to do.

4. Try this program:

```
10 INPUT A
20 INPUT B
30 PRINT (A AND A>=B)+(B AND A<B)
40 GOTO 10
```

Each time it prints the larger of the two numbers A & B - why?

Convince yourself that you can think of

X AND Y

as meaning

'X if Y (else the result is 0)'

& of

X **OR** Y

as meaning

'X unless Y (in which case the result is 1)'

An expression using **AND** & **OR** like this is called a conditional expression. An example using **OR** could be

```
LET RETAIL PRICE=PRICE LESS VAT*(1.15 OR V$="ZERO RATED")
```

Notice how **AND** tends to go with addition (because its default value is 0), & **OR** tends to go with multiplication (because its default value is 1).

5. You can also make string valued conditional expressions, but only using **AND**.

X$ **AND** Y$ has the value { X$ if Y is non-zero
{ if Y is zero

so it means 'X$ if Y (else the empty string)'.

Try this program, which inputs two strings & puts them in alphabetical order.

```
10 INPUT A$
20 INPUT B$
30 IF A$<=B$ THEN GOTO 70
40 LET C$=A$
50 LET A$=B$
60 LET B$=C$
70 PRINT A$;" ";("<" AND A$<B$)+("=" AND A$=B$);" ";B$
80 GOTO 10
```

6. Try this program:

```
10 PRINT "X"
20 STOP
30 PRINT "Y"
```

When you run it, it will display "X" & stop with report 9/20. Now type

CONT

You might expect this to behave like '**GOTO** 20', so that the computer would just stop again without displaying "Y"; but this would not be very useful, so things are arranged so that for reports with code (**STOP** statement executed), the line number is increased by 1 for a **CONT** statement. This in our example, '**CONT**' behaves like '**GOTO** 21' (which, since there are no lines between 20 & 30, behaves like '**GOTO** 30').

7. Many versions of BASIC (but not the ZX81 BASIC) have an ON statement, which takes the form

ON numeric expression GOTO line number, line number,...,line number In this the numeric expression is evaluated; suppose its value is n then the effect is that of.

GOTO the nth line number

For instance

ON A GOTO 100, 200, 300, 400, 500

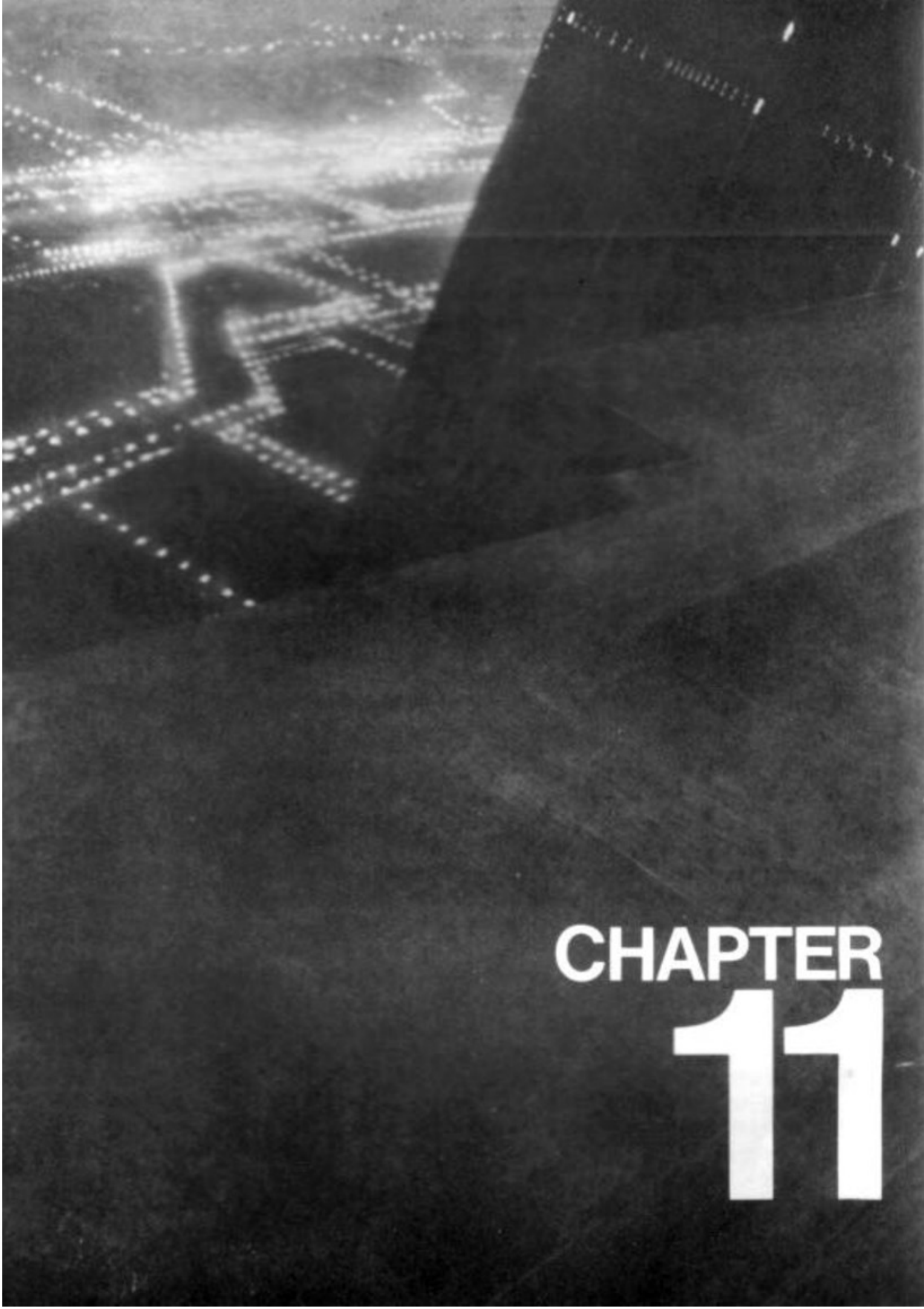
Here, if A has the value 2, then 'GOTO 200' is executed. In ZX81 BASIC this can be replaced by

GOTO 100*A

In case the line numbers don't go up neatly by hundreds like this, work out how you could use

GOTO a conditional expression

instead.



CHAPTER 11

The character set

The letters, digits, punctuation marks & so on that can appear in strings are called characters, & they make up the alphabet, or character set, that the ZX81 uses. Most of these characters are single symbols, but there are some more, called tokens, that represent whole words, such as **PRINT**, **STOP**, **, & so on.

There are 256 characters altogether, & each one has a code between 0 & 255. There is a complete list of them in appendix A. To convert between codes & characters, there are two functions, **CODE** & **CHR$**.

CODE is applied to a string, & gives the code of the first character in the string (or 0 if the string is empty).

CHR$ is applied to a number, & gives the single character string whose code is that number.

This program prints out the entire character set.

```
10 LET A=0
20 PRINT CHR$ A;
30 LET A=A+1
40 IF A<256 THEN GOTO 20
```

At the top you can see the symbols ", £, $ and so on up to Z, which all appear on the keyboard & can be typed in when you have the L cursor. Further on, you can see the same characters, but in white on black (inverse video); these are also obtainable from the keyboard. If you press **GRAPHICS** (shifted 9) then the cursor will come up as G: this means graphics mode. If you type in a symbol it will appear in its inverse video form, & this will go on until you press **GRAPHICS** again or **NEWLINE**. **RUBOUT** will have its usual meaning. Be careful not to lose the cursor G amongst all the inverse video characters you've just typed in.

When you've experimented a bit, you should still have the character set at the top; if not, then run the program again. Right at the beginning are space & ten patterns of black, white & grey blobs; further on there are eleven more. These are called the graphics symbols & are used for drawing pictures. You can enter these from the keyboard, using graphics mode (except for space, which is an ordinary symbol using the L cursor; the black square is inverse space). You use the 20 keys that have graphics symbols written on them. For instance, suppose you want the symbol ▚, which is on the T key. Press **GRAPHICS** to get the G cursor, & then press shifted T. From the previous description of the graphics mode, you would expect to get an inverse video symbol; but shifted T is normally **<>**, a token, & tokens have no inverses: so you get the graphics symbol ▞ instead.

Here are the 22 graphics symbols.

Symbol	Code	How obtained
	0	K or L **SPACE**
	1	G shifted 1
	2	G shifted 2
	3	G shifted 7
	4	G shifted 4
	5	G shifted 5
	6	G shifted T
	7	G shifted E
	8	G shifted A
	9	G shifted D
	10	G shifted S

Symbol	Code	How obtained
	128	G **SPACE**
	129	G shifted Q
	130	G shifted W
	131	G shifted 6
	132	G shifted R
	133	G shifted 8
	134	G shifted Y
	135	G shifted 3
	136	G shifted H
	137	G shifted G
	138	G shifted F

Now look at the character set again. The tokens stand out quite clearly in two blocks: a small group of three (**RND**, **INKEY$** & **PI**) after Z, & a larger group (starting with the quote image after Z, & carrying on from **AT** up to **COPY**).

The rest of the characters all seem to be ? This is actually just the way they get printed - the real question mark is between : and (. Of the spurious ones, some are for control characters like ⇦, **EDIT** & **NEWLINE**, & the rest are for characters that have no special meaning for the ZX81 at all.

Summary

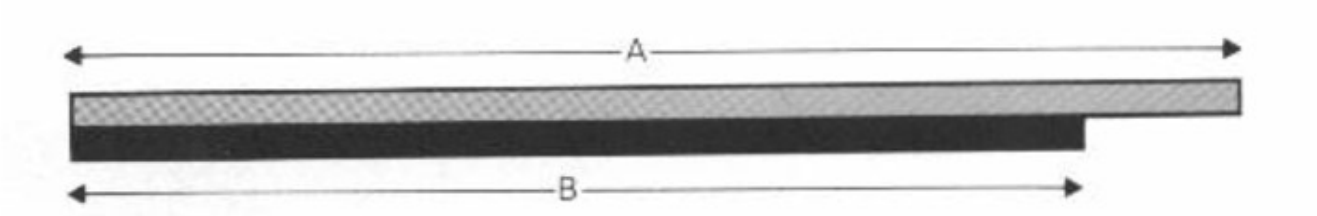
Functions: **CODE**, **CHR$**

Exercises

1. Imagine the space for one key symbol divided up into four quarters: ⊞. Then if each quarter can be either black or white, there are 2*2*2*2 = 16 possibilities. Find them all in the character set.

2. Imagine the space for one symbol divided into two horizontally: ⊟. Then if each half can be black, white or grey, there are 3*3 = 9 possibilities. Find them all.

3. The characters in exercise 2 are designed to be used in horizontal bar charts, using two colours, grey & black. Write a program that inputs two numbers A & B (both between 0 & 32), & draws a bar chart for them:



You will need to start off printing "▒", & change to either "▭" or "▬", according as A is more or less than B.

What does your program do if A & B are not whole numbers? Or if thet are not in the range 0 to 32? a good - 'user friendly' is the fashionable term - program will do something sensible & useful.

4. There are two different all grey characters on the keyboard, on A & H. If you look at them very close up, you will see that the one on H is like a miniature chessboard, while the one on A is like a sideways chessboard. Print them next to each other, & you will see that they don't join up properly. The one on A is used to join up neatly with ▭ & ▒ (on S & D), while the one on H joins up neatly with ▬ & ▒ (on F & G).

5. Run this program:

```
10 INPUT A
20 PRINT CHR$A;
30 GOTO 10
```

If you experiment with it, you will find that for CHR$, A is rounded to the nearest whole number; & if A is not in the range 0 to 255 then the program stops with report B.

6. Using the codes for the characters, we can extend the concept of 'alphabetical ordering' to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters, in the same order as their codes, then the principle is exactly the same. For instance, these strings are in ZX81 alphabetical order.

" ZACHARY"

"▟"

"(ASIDE)"

"123 TAXI SERVICE"

"AASVOGEL"

"AARDVARK"

"ZACHARY"

"AARDVARK"

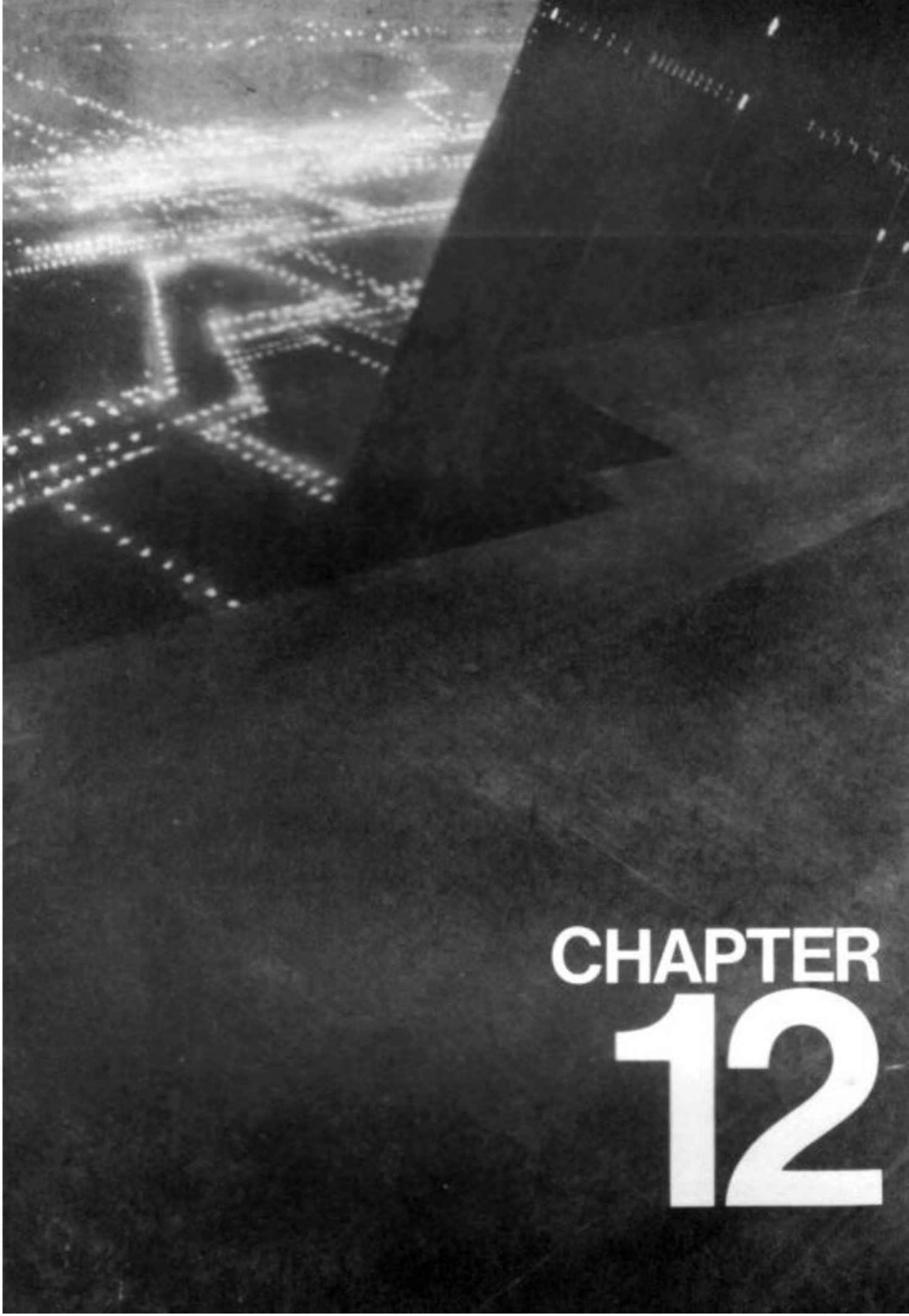
Here is the rule. First, compare the first characters in the two strings. If these are different, then one of them has its code less than the other, & the string of which it is the first character is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If in this process one of the strings runs out before the other, then that string is the earlier; otherwise they must be equal.

Type in again the program in exercise 4 of chapter 10 (the one that inputs two strings & prints them in order), & use it to experiment.

7. This program prints a screenful of random black & white graphics characters:

```
10 LET A=INT (16*RND)
20 IF A>=8 THEN LET A=A+120
30 PRINT CHR$ A;
40 GOTO 10
```

(How does it work?)



CHAPTER 12

Looping

Suppose you want to input five numbers & add them together. One way (don't type this in unless you're feeling dutiful) is to write

```
10 LET TOTAL=0
20 INPUT A
30 LET TOTAL=TOTAL+A
40 INPUT A
50 LET TOTAL=TOTAL+A
60 INPUT A
70 LET TOTAL=TOTAL+A
80 INPUT A
90 LET TOTAL=TOTAL+A
100 INPUT A
110 LET TOTAL=TOTAL+A
120 PRINT TOTAL
```

This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add ten numbers would be, & a hundred would be just impossible.

Much better is to set up a variable to count up to five & then stop the program, like this (which you should type in):

```
10 LET TOTAL=0
20 LET COUNT=1
30 INPUT A
40 REM COUNT = NUMBER OF TIMES THAT A HAS BEEN INPUT SO FAR
50 LET TOTAL=TOTAL+A
60 LET COUNT=COUNT+1
70 IF COUNT<=5 THEN GOTO 30
80 PRINT TOTAL
```

Notice how easy it would be to change line 70 so that this program adds ten numbers, or even a hundred.

This sort of counting is so useful that there are two special statements to make it easier: the **FOR** statement, & the **NEXT** statement. They are always used together. Using these, the program you have just typed in does exactly the same as

```
10 LET TOTAL=0
```

```
20 FOR C=1 TO 5
30 INPUT A
40 REM C = NUMBER OF TIMES THAT A HAS BEEN INPUT SO FAR
50 LET TOTAL=TOTAL+A
60 NEXT C
80 PRINT TOTAL
```

(To get this program from the previous one you just have to edit lines 20, 40, 60 & 70. **TO** is shifted 4.)

Note that we have changed COUNT to C. The counting variable - or control variable - of a **FOR-NEXT** loop must have a single letter for its name.

The effect of this program is that C runs through the values 1 (the initial value), 2, 3, 4 & 5 (the limit), & for each one, lines 30, 40 & 50 are executed. Then, when C has finished its five values, line 80 is executed.

An extra subtlety to this is that the control variable does not have to go up by 1 each time: you can change this 1 to anything else you like by using a **STEP** part in the **FOR** statement. The most general form for a **FOR** statement is

FOR control variable = initial value **TO** limit **STEP** step

where the control variable is a single letter, & the initial value, limit & step are all numeric expressions. So, if you replace line 20 in the program by

```
20 FOR C=1 TO 5 STEP 3/2
```

then C will run through the values 1, 2.5 & 4. Notice that you don't have to restrict yourself to whole numbers, & also that the control value does not have to hit the limit exactly - it carries on looping as long as it is less than or equal to the limit (but see exercise 4).

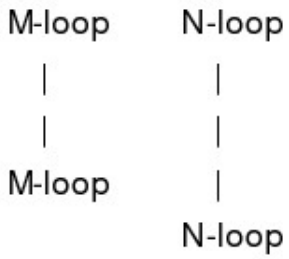
You must be careful if you are running two **FOR-NEXT** loops together, one inside the other. Try this program, which prints out a complete set of 6-spot dominoes.

```
10 FOR M=0 TO 6                    N-loop      M-loop
20 FOR N=0 TO M                      |           |
30 PRINT M;":";N;" ";              N-loop        |
40 NEXT N                                        |
50 PRINT                                         |
60 NEXT M                                      M-loop
```

You can see that the N-loop is entirely inside the M-loop - they are properly nested. What must be avoided is two **FOR-NEXT** loops that overlap without either being entirely inside the other, like this:

WRONG

```
10 FOR M=0 TO 6
20 FOR N=0 TO M
30 PRINT M;":";N;" ";
40 NEXT M
50 PRINT
60 NEXT N
```



The **FOR-NEXT** loops must either be one inside the other, or be completely separate.

Another thing to avoid is jumping into the middle of a **FOR-NEXT** loop from the outside. The control variable is only set up properly when its **FOR** statement is executed, & if you miss this out the **NEXT** statement will confuse the computer. You might get error report 1 or 2 (meaning that a **NEXT** statement does not contain a recognised control variable) if you're lucky.

Summary

Statements: **FOR**, **NEXT**, **TO**, **STEP**

Exercises

1. Rewrite the program in chapter 11 that prints out the character set, using a **FOR-NEXT** loop (Answer in chapter 13.)

2. A control variable has not just a name & a value, like an ordinary variable, but also a limit, a step, & a line number for looping back to (the line after the **FOR** statement where it was set up). Persuade yourself first, that when the **FOR** statement is executed all this information is available (using the initial value as the first value it takes), & second, that (using as an example our second & third programs), this information is enough to

convert the one line.

NEXT C

into the two lines

LET C=C+1

IF C<=5 **THEN GOTO** 30

(Actually we have cheated slightly here: it should really be **GOTO** 21 instead of **GOTO** 30. This will have the same effect in our program.)

3. Run the program, & then type

PRINT C

Why is the answer 6, & not 5?

[Answer: the **NEXT** statement in line 60 is executed 5 times, and each time 1 is added to C.] What happens if you put **STEP** 2 in line 20?

4. Change the third program so that instead of automatically adding five numbers, it asks you to input how many numbers you want adding. When you run this program, what happens if you input 0, meaning that you want no numbers adding? Why might you expect this to cause problems to the computer, even though it is clear what you mean? (The computer has to make a search for the statement **NEXT** C, which is not usually necessary.) In fact this has all been taken care of.

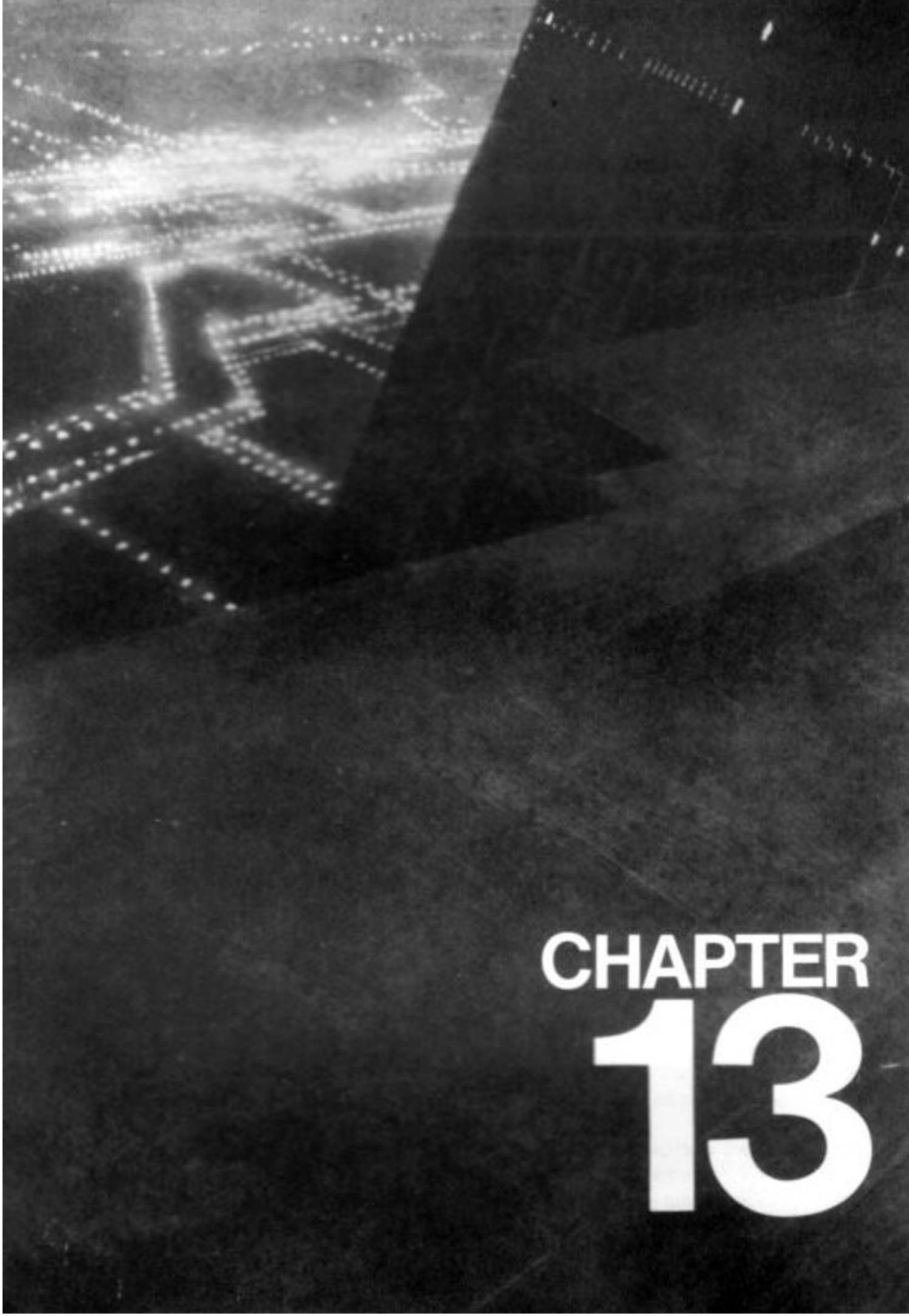
5. Try this program, to print out the numbers from 1 to 10 in reverse order.

10 **FOR** N=10 **TO** 1 **STEP** -1

20 **PRINT** N

30 **NEXT** N

Convert this into a program that does not use **FOR-NEXT** loops in the same way that you would convert program 3 into program 2 (see exercise 2). Why does the negative step make is slightly different?



CHAPTER 13

Slow & fast

The ZX81 can run at two speeds - SLOW and FAST. When first switched on, the computer runs in the SLOW mode and can compute and display information on the screen simultaneously. This mode is ideal for animation displays.

However, it can go about four times as fast, & it does this by forgetting about the picture except when it has nothing else to do. To see this working, type

FAST

Now whenever you press a key, the screen will blink - this is because the computer stops displaying a picture while it works out what key you pressed.

Type in a program, say

```
10 FOR N=0 TO 255
20 PRINT CHR$ N;
30 NEXT N
```

When you run this, the whole screen will go an indeterminate grey until the end of the program, when the output from the PRINT statement will come up on the screen.

The picture is also displayed an INPUT statement, while the computer is waiting for you to type the INPUT data. Try this program:

```
10 INPUT A
20 PRINT A
30 GOTO 10
```

To get back into the normal (compute & display) mode, type

SLOW

It will often be just a matter of taste whether you want compute & display mode for neatness, or fast mode for speed, but in general you will use the fast mode when

(i) your program contains a lot of numerical calculation, especially if it doesn't print much but in compute and display mode time doesn't seem to linger quite as much is you can see output coming up on the screen fairly frequently, or

(ii) you are typing in a long program. You will already have noticed how the listing gets remade every time you enter a new program line, & this can get annoying.

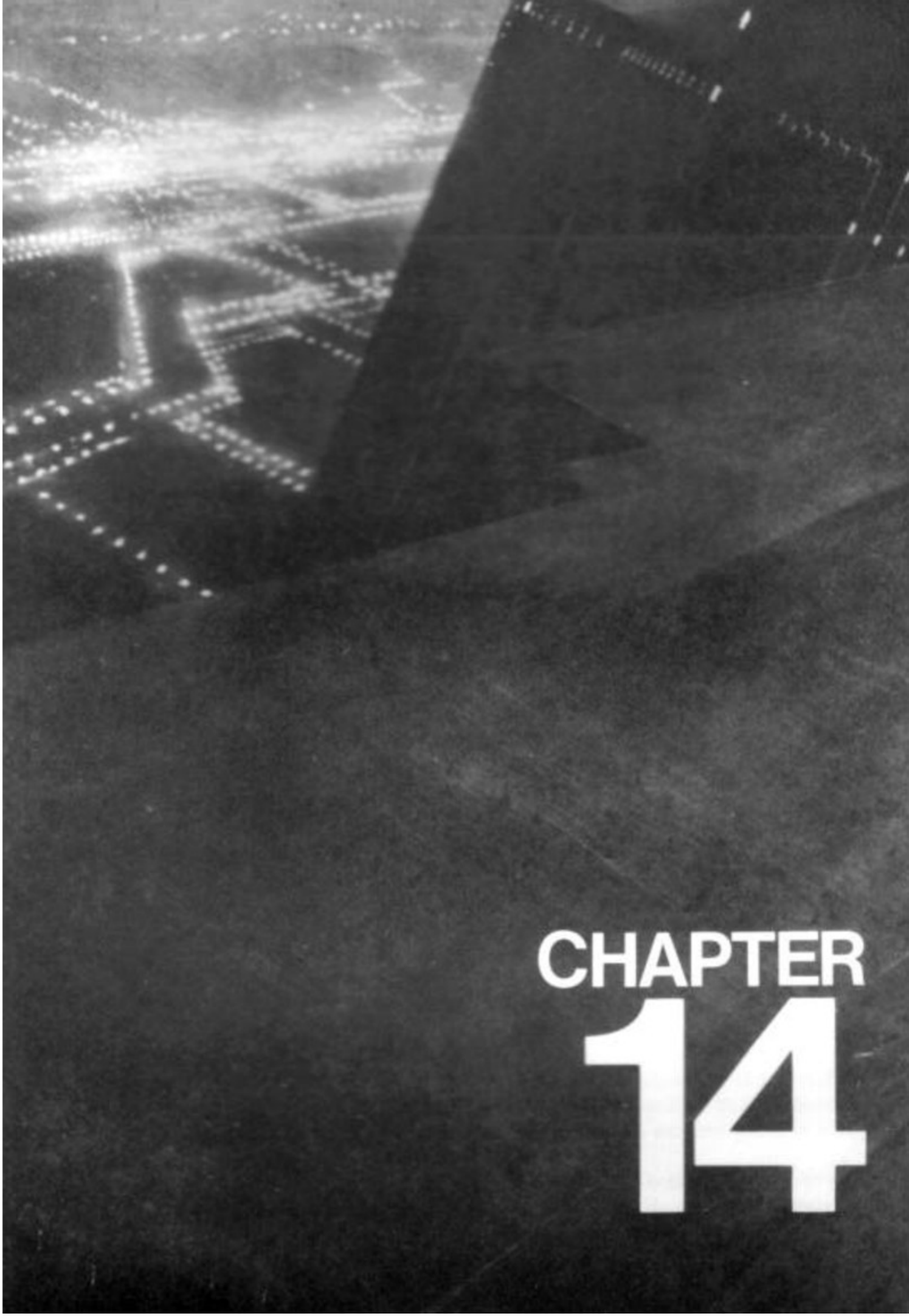
You can use **SLOW** & **FAST** statements in programs without any problems.

For example,

```
10 SLOW
20 FOR N=1 TO 64
30 PRINT "A";
40 IF N=32 THEN FAST
50 NEXT N
60 GOTO 10
```

Summary

Statements: **FAST**, **SLOW**



CHAPTER 14

Subroutines

Sometimes different parts of your program will have rather similar jobs to do, & you will find yourself typing the same lines in twice or more; however this is not necessary. You can type the lines in once, in the form known as a subroutine, & then use, or call, them anywhere else in the program without having to type them in again.

To do this, you use the statements **GOSUB** (GO to SUBroutine) & **RETURN**.

GOSUB n

where n is the line number of the first line in the subroutine, is just like **GOTO** n except that the computer remembers the line number of the **GOSUB** statement so that it can come back again after doing the subroutine. It does this remembering by putting the line number (the return address) on top of a pie of them (the GOSUB stack).

RETURN

takes the top line number off the GOSUB stack, & goes to the line after it.

As a first example,

```
10 PRINT "THIS IS THE MAIN PROGRAM",
20 GOSUB 1000
30 PRINT "AND AGAIN";
40 GOSUB 1000
50 PRINT "AND THAT IS ALL."
60 STOP
1000 REM SUBROUTINE STARTS HERE
1010 PRINT "THIS IS THE SUBROUTINE,"
1020 RETURN
```

The **STOP** statement in line 60 is very important because otherwise the program will run on into the subroutine & cause error 7 when the **RETURN** statement is reached.

For a less trivial example, suppose you want to write a computer program to handle pounds, shillings and pence. Those with long memories will remember that before 1971 a pound was divided into twenty shillings - so a shilling is 5p - & a shilling was subdivided into twelve old pence; d was the abbreviation for an old penny.) You will have three variables L, S & D (any maybe others - L1, S1, D1 & so on), and arithmetic is dead easy. First you do it separately on the pounds, shillings and pence - for instance, to add two sums of money, you add the pence, add the shillings and add the pounds; to double a sum of money you double the pence, double the shillings and double the pounds; and so on. When all that is done, adjust it to the correct form so that the pence are between 0 & 11, and the shillings between 0 & 19. This last stage is common to all the operations, so we can make it into a subroutine.

Laying aside the notion of subroutines for a moment, it is worth your while trying to write the program yourself. Give the arbitrary numbers L, S & D, how do you convert them into proper pounds, shillings & pence? Part of the problem is that you will start thinking of odder & odder cases.

What first springs to mind will probably be something like £1..25s..17d, which you want to convert to £2..6s..5d. Not so difficult. But suppose you have negative numbers? A dept of £1..25s..17d, or £-1..-25s..-17d, might well turn out as £-3..13s..7d, which is rather an odd way of expressing it (as though people only ever lend each other whole pounds). And what about fractions? If you divide £1..25s..17d by two, you get £5..12.5s..8.5d, & although this has the pence, 8.5, between 0 & 11; the shillings, 12.5, between 0 & 19, it is certainly not as good as £1..3s..2.5d. Try & work out your own answers to all this - & use them in a computer program - before you read any further.

Here is one solution.

```
1000 REM SUBROUTINE TO ADJUST L.S.D. TO THE NORMAL FORM FOR
POUNDS, SHILLINGS AND PENCE
1010 LET D=240*L+12*S+D
1020 REM NOW EVERYTHING IS IN PENCE
1030 LET E=SGN D
1040 LET D=ABS D
1050 REM WE WORK WITH D POSITIVE, HOLDING ITS SIGN IN E
1060 LET S=INT (D/12)
1070 LET D=(D-12*S)*E
1080 LET L=INT (S/20)*E
1090 LET S=S*E-20*L
1100 RETURN
```

On its own, this is not much use because there is no program to set up L, S & d beforehand, nor to do anything with them afterwards. Type in the main program, & also another subroutine to print out L, S & D.

```
10 INPUT L
20 INPUT S
30 INPUT D
40 GOSUB 2000
45 REM PRINT THE VALUES
50 PRINT
60 PRINT "□□□ = ";
70 GOSUB 1000
```

```
75 REM THE ADJUSTMENT
80 GOSUB 2000
85 REM PRINT THE VALUES
90 PRINT
100 GOTO 10
2000 REM SUBROUTINE TO PRINT L,S AND D
2010 PRINT "£";L;"..";S;"S..";D;"D";
2020 RETURN
```

(Recall from chapter 9 that the empty **PRINT** statement in line 50 prints a black line.)

Clearly we have saved on program by using the printing subroutine at 2000, & this in itself is a very common use for subroutines: to shorten programs. However, the adjustment subroutine in fact makes the program longer - by a **GOSUB** & a **RETURN**; so program length is not the only consideration. Used with skill, subroutines can make programs easier to understand for the ones that matter, humans.

The main program is simplified by its using more powerful statements: each **GOSUB** represents some complicated BASIC, but you can forget that - only the net result matters. Because of this, it is much easier to grasp the main structure of the program.

The subroutines, on the other hand, are simplified for a very different reason, namely that they are shorter. They still use the same old plodding **LET** & **PRINT** statements, but they only have to do a part of the whole job & so are easier to write.

The skill lies in choosing the level - or levels - at which to write the subroutines. They must be big enough to have a significant impact on the main program, yet small enough to be significantly easier to write than a complete program without subroutines. These examples (not recommended) illustrate this.

First,

```
10 GOSUB 1000
20 GOTO 10
1000 INPUT L
1010 INPUT S
1020 INPUT D
1030 PRINT " ";L;"..";S;"S..";D;"D";TAB 8;"=";
1040 LET D=240*L+12*S+D
  :  :
  :  :
2000 RETURN
```

& second

```
10 GOSUB 1010
20 GOSUB 1020
30 GOSUB 1030
40 GOSUB 1040
50 GOSUB 1050
  :  :
  :  :
30 GOTO 10
1010 INPUT L
1015 RETURN
1020 INPUT S
1025 RETURN
1030 INPUT D
1035 RETURN
1040 PRINT " ";L;"..";S;"S..";D;"D";TAB 8; "=";
1045 RETURN
1050 LET D=240*L+12*S+D
1055 RETURN
  :  :
  :  :
```

The first, with its single powerful subroutine, & the second, with its many trivial ones, demonstrate quite opposite extremes, but with equal futility.

A subroutine can happily call another, or even itself (a subroutine that calls itself is recursive), so don't be afraid to having several layers.

Summary

Statements: **GOSUB**, **RETURN**

Exercises

1. The example program is virtually a universal LSD calculator. How would you use it.

   (i) to convert pounds & new pence into pounds, shillings & pence?

(ii) to convert guineas into pounds & shillings? (1 guinea = 1..1s)

(iii) to find fractions of a pound? (e.g. a third of a pound, or a mark, is 6s..8d.

Put in a line to round the pence off to the nearest farthing (1/4d).

2. Add two statements to the program:

4 **LET** ADJUST=1000

7 **LET** LSDPRINT=2000

& change

GOSUB 1000 to **GOSUB** ADJUST

GOSUB 2000 to **GOSUB** LSDPRINT

This works exactly as you'd hope; in fact the line number in a **GOSUB** (or **GOTO** or **RUN**) statement can be any numerical expression. (Don't expect this to work on computers other than the ZX81, because it is not standard BASIC.)

This sort of stuff can work wonders for the clarity of your programs.

3. Rewrite the main program in the example to do something quite different, but still using the same subroutines.

4. ... **GOSUB** n

... **RETURN**

in consecutive lines can be replaced by

... **GOTO** n

Why?

5. A subroutine can have several entry points. For instance, because of the way our main program uses them, with **GOSUB** 1000 followed immediately by **GOSUB** 2000, we can replace our two subroutines by one big one that adjusts L, S & D & then prints them. It has two entry points: one at the beginning for the whole subroutine, & another further on for the printing part only.

Make the necessary rearrangements.

6. Run the program:

```
10 GOSUB 20
20 GOSUB 10
```

The return addresses are pushed on to the **GOSUB** stack in droves, but they never get taken off again & eventually there is no room for any more in the computer. The program then stops with error 4 (see appendix B).

You might have difficulty in clearing them out again without losing everything, but this will work.

(i) Delete the two **GOSUB** statements.

(ii) Insert two new lines

```
11 RETURN
21 RETURN
```

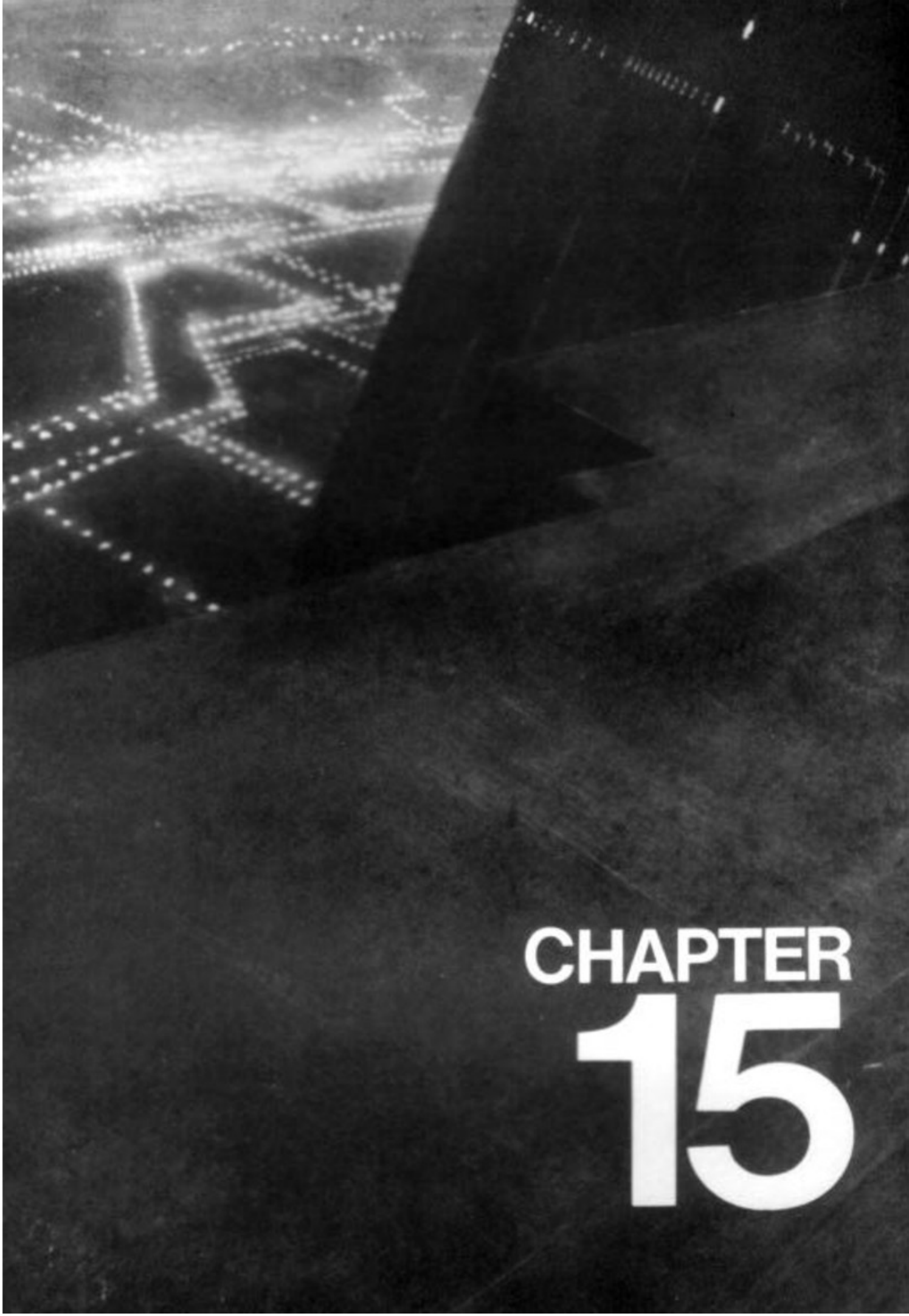
(iii) Type

```
RETURN
```

The return addresses will be stripped off until you get error 7.

(iv) Change your program so you don't get the same thing happening again.

How does this work?



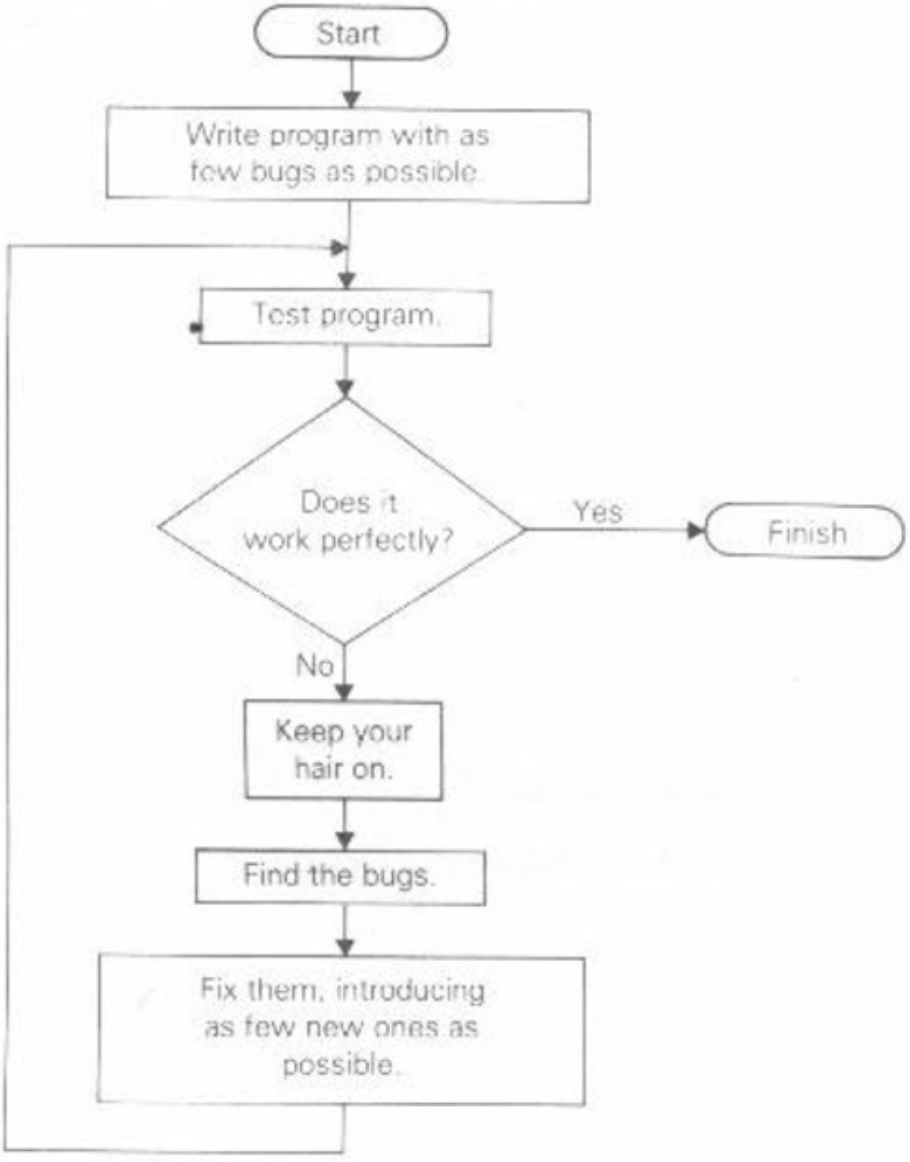
CHAPTER 15

Making your programs work

There is more to the art of programming computers than just knowing which statement does what. You will probably already have found that most of your programs have what are technically known as bugs when you first type them in: maybe just typing errors, or maybe mistakes in your own ideas of what the program should do. You might put this down to inexperience, but you would be deluding yourself.

EVERY PROGRAM STARTS OFF WITH BUGS.

Many programs finish up with bugs as well. There are two corollaries to this; first, you must test all your programs straight away; & second, there's no point in losing your temper every time they don't work. The general plan can be illustrated with a flowchart:



The idea is that you follow the arrows from box to box, doing what it says at each one. We have used different sorts of boxes for different sorts of instructions:

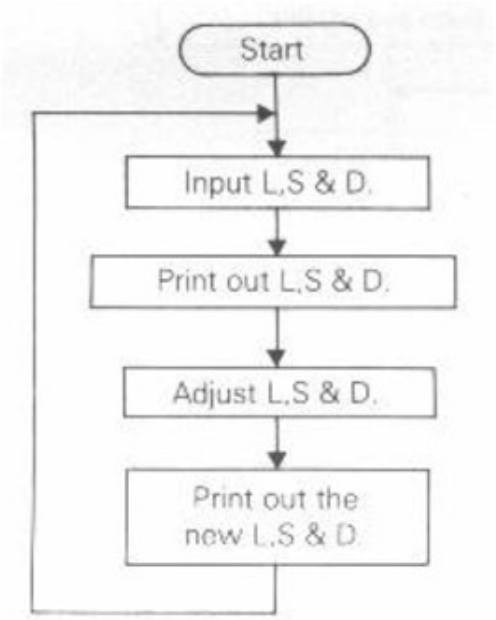
A rounded box is start or finish.

A rectangular box is a straightforward instruction.

A diamond asks you to make some kind of decision before carrying on.

(These shapes are fairly widely used, but nothing earth-shattering depends on them.)

Of course, these flowcharts are ill-adapted for describing human activities; thinking along fixed straight lines like this is not good for creativeness or flexibility. For computers, however, they are just the job. They are best at describing the large-scale structure of programs, with a subordinate in almost every box, so a flowchart for our sterling example in the last chapter might be.



Conceptually, then, rectangular boxes correspond to **GOSUB**s; although in practice some boxes - like the one above that inputs L, S & D - are translated directly into BASIC statements, without a subroutine.

Anything - like flowcharts, subroutines, & also **REM** statements - that makes the program clearer gives you a better understanding of it; & then you are bound to write fewer bugs. But subroutines also help you get out the bugs you've written anyway, by making the program easier to test. You will find it much easier to test the subroutines

individually & make sure that they fit together properly, than to test a whole unstructured program.

Subroutines, then, help with the box 'find the bugs' & this is the box where you need all the help you can get, for it is often the most exasperating. Other hints for finding bugs are

(i) Check that there are no typing errors. Always do this.

(ii) Try to work out what all the variables should be at each stage - & if possible explain this in **REM** statements. You can check the variables at a given point in the program by inserting a **PRINT** statement there.

(iii) If the effect of the program is to make the program stop with an error report, then use this information as thoroughly as you can. Look up the report code, & work out why it stopped on the line where it did. Print out the values of the variables, if necessary.

(iv) You might be able to step through a program line by line by typing in its lines as commands.

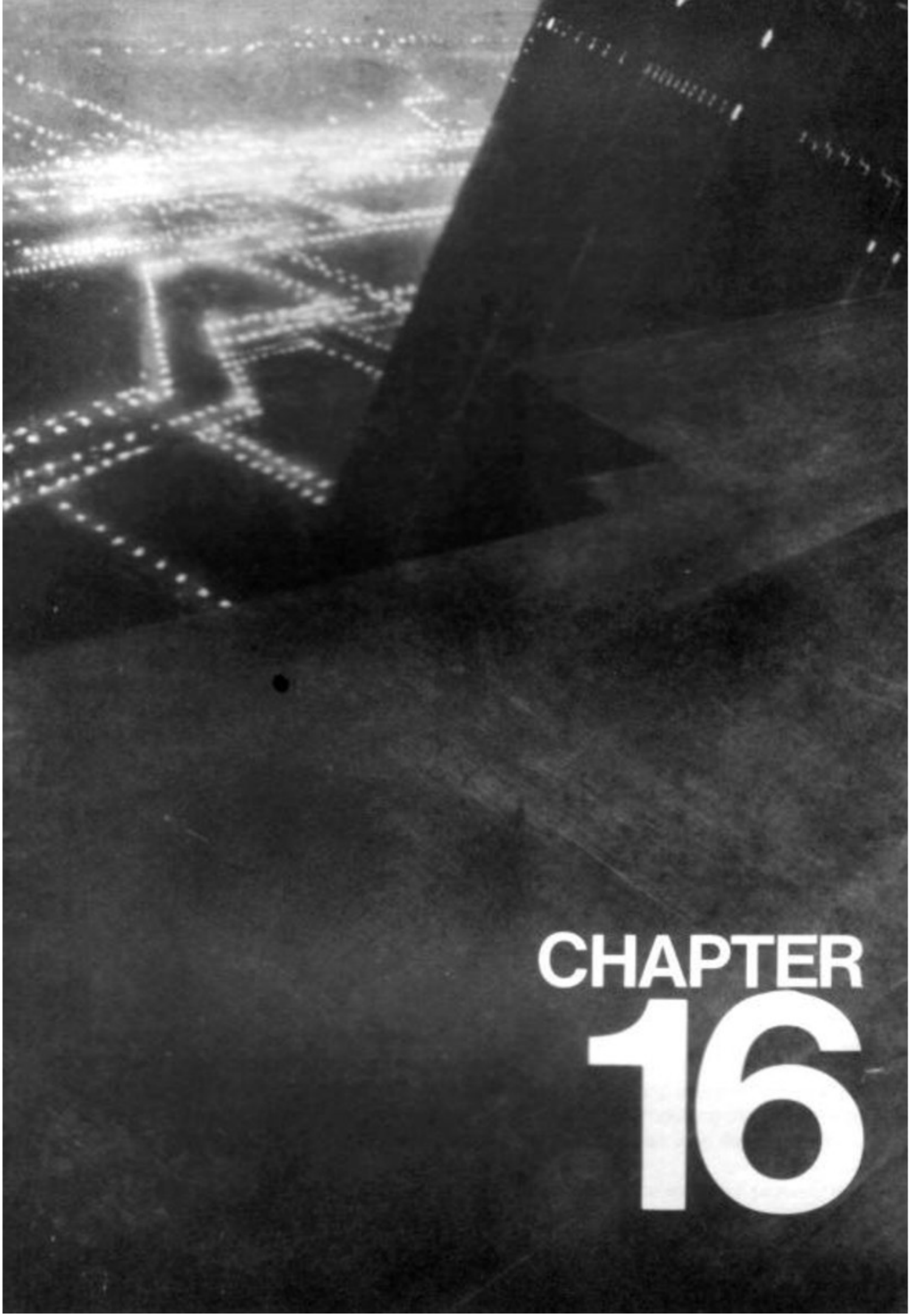
(v) Pretend to be the computer: run the program on yourself using pencil & paper to note down the values of the variables.

Once you've found the bugs, fixing them is much like writing the original program, but you must test the program again. It is surprisingly easy to fix one bug, only to introduce another.

Exercises

1. The flowchart for the LSD calculator has no 'finish' box. Does this matter? Where would you put one if you wanted to?

2. Write flowcharts for the looping programs in chapter 12.



CHAPTER 16

Tape storage

As was mentioned in chapter 1, & you have no doubt found out from experience anyway, when you turn the ZX81 off you lose all the program & variables that were stored inside it. The only way to save these is to have the computer record them onto a cassette tape; & then later you can load them back in & the computer will be restored to practically the same state as it was in when it made the recording.

You will have found that with the ZX81 a pair of leads (article (v) in chapter 1) which connect the ZX81 to a cassette recorder. You must provide your own tape recorder, & some work better than others.

First, as far as the ZX81 is concerned, the cheap, portable mono cassette recorders are at least as good as expensive stereo ones, & give less trouble as well. You will find a tape counter very useful.

Second, the tape recorder must have an input socket for use with microphones, & an output socket for use with earphones if there isn't one, try the external loudspeaker socket.) They should preferably be 3.5 mm jack sockets i.e. to fit the jack plugs on the leads provided), because other sorts often do not give a signal powerful enough for the ZX81.

Any cassette tape should work, although low noise tapes are preferable.

Now, having acquired a suitable cassette recorder, connect it to the computer: one lead should connect the microphone input socket on the recorder to the socket marked 'MIC' on the side of the ZX81, & the other connects the earphone output socket on the recorder to the 'EAR' socket on the ZX81. Make sure that the leads are not crossed over (although you won't damage the ZX81 if they are.)

Type some program into the computer, say the character set program in chapter 11. You are going to have to give the program a name when you save it, & it is a good idea to put this name into the program so that it appears in listings - the easiest way is with a REM statement. So type

```
5 REM "CHARACTERS"
```

Now - & this is just a dry run, so that you can see what happens, - type

```
SAVE "CHARACTERS"
```

& watch the television. For five seconds it will just be a greyish colour, then for about six seconds there will be a distinctive pattern of thin black & white stripes, & then the screen will go white with the report 0/0. The computer was sending a signal to the 'MIC' socket; but it was also sending the same signal to the television, producing the picture that you saw. The grey part was a silent lead-in, & the black & white part was the program.

What you want to do, of course, is catch that signal on tape, so let's do it properly this time.

Saving a program

1. Position the tape in a part either that is blank, or that you are prepared to overwrite.

2. Using a microphone, record yourself saying 'characters'. This is not essential, but it will make it easier to find the program afterwards. Reconnect the computer to the tape recorder.

3. Type

SAVE "CHARACTERS" (without **NEWLINE**)

4. Start the tape recorder recording.

5. Press **NEWLINE**.

6. Watch the television as before. When it has finished (with the report 0/0), stop the tape recorder.

To make sure that this has worked, you should now listen to the tape through the tape recorder's own loudspeaker. (You will probably have to unplug the lead from the earphone socket on the tape recorder.) Rewind the tape to where you started before, & play it back.

First, you will hear your own voice saying 'characters'.

After that, comes a soft, humming buzz. This is not really part of the recording, but the end of the signal for the television (before you pressed **NEWLINE**), which happens to have been sent to the tape recorder as well.

Next come five seconds of silence, the beginning of the proper tape signal. This corresponds to the period when the television screen went grey.

Next come about six seconds of a very harsh high-pitched buzz, & at full volume this should be very unpleasantly loud. This is a recording of the program, & corresponds to the black & white patterns on the television screen.

Finally, the soft humming buzz will return.

If you did not hear these various hums & buzzes, then check that you had the computer & tape recorder properly connected up. It happens with some tape recorders that the jack plug dies not make contact if it is pushed right in. Try pulling it out about a tenth of an inch - you can sometimes feel it settling down into a more natural position.

Now let us suppose that the recoding sounds all right to the human ear, & you want to try to load it back into the computer.

Loading a program with a name

1. Rewind the tape to the place where you started.

2. Make sure that the 'EAR' socket on the computer is connected to the earphone socket on the tape recorder.

3. Turn the volume control on the tape recorder to about three quarters of the maximum volume; if it has tone control then adjust them so that treble is high & bass is low (so that it sounds hissy).

4. Type

LOAD "CHARACTER" (again, without **NEWLINE**)

5. Start the tape recorder playing.

6. Press **NEWLINE**.

Again, you will be able to see pictures of the recording in the television, but they will look different this time, everything being a black & white pattern. The two parts, the silence & the program, will be less easily distinguished, but you should be able to see that the program part has much broader, more defined lines. (Try exercise 1 some time.)

After fifteen seconds it should have loaded & stopped with report 0/0. Otherwise, press the **BREAK** key (space), which should let it out of its misery.

The most likely thing to have been wrong is the volume level: this should be

(i) loud enough for the program part to be picked up by the computer,

(ii) not so loud that the program part is distorted (this is actually fairly rare),

& (iii) quiet enough for the silent part to be recognized as silent by the computer.

The best adjustment is to turn the volume up as loud as it will go without making the silent part at all noisy; you can do this while listening to the recording through the loudspeaker. If the silence is incorrigibly noisy, then you may have other problems:

Some tape recorders form a feedback loop with the ZX81. This can only happen when the EAR & MIC leads are both in at the same time, so the cure is to **SAVE** with the EAR lead disconnected.

Some tape recorders can record a mains hum. This is cured by operating them on batteries.

Some tape recorders - especially old, worn ones - are intrinsically noisy. This may be helped by using a better quality tape, although this should not be necessary.

Try cleaning the tape head in the cassette recorder, in case it is dirty.

Finally, there may be the same problem about pushing the plug right into the earphone socket as was mentioned for the microphone socket.

If you've got a program on tape & you can't remember its name, you can still load it. (Try this with the program "CHARACTERS" that you were using before.)

Loading a program without a name

1. Position the tape in the silent lead-in.

2. Check everything & adjust the controls as before. You might well find that you have to be more careful with the volume level than you did when you knew the name.

3. Type

LOAD "" (without **NEWLINE**)

4. Start the tape recorder playing.

5. Press **NEWLINE**.

6. The rest is as before.

The idea is that if the name of the program you ask to be loaded is the empty string, then the computer loads the first program it comes across. Note that when you save a program, you cannot make its name the empty string - if you try then you will get error F.

LOAD & **SAVE** can also be used in programs. With **SAVE**, the program will save itself in such a state then when loaded it will immediately carry on executing from the line after the **SAVE** statement.

For example, type in

```
5 REM "USELESS"
10 PRINT "THIS IS ALL IT DOES"
20 STOP
100 SAVE "USELESS"
110 GOTO 10
```

Connect up the tape recorder, type

RUN 100 (without **NEWLINE**),

start the tape recorder recording, & press **NEWLINE**. When the program has saved itself, it will continue running. You will discover afterwards that the last S in USELESS in line 100 has changed to inverse video, but this is nothing to worry about.

To load it, rewind the tape to somewhere before the beginning of the program, type

LOAD "USELESS" (without **NEWLINE**),

start playing back the tape into the computer, & press **NEWLINE**. When it has loaded, it will carry on with line 110 & execute itself without any effort on your part.

Note how putting the **SAVE** statement at the end of the program means that to run it without the **SAVE** you just type **RUN** - you don't have to jump round the **SAVE** statement.

Don't **SAVE** from within a **GOSUB** routine - it won't work properly.

Don't put inverse video characters in a program name. Any part of the name after the inverse video character gets lost.

The name should not contain more than 127 characters.

The name in a **LOAD** or **SAVE** does not have to be a string constant, it can be any string-valued expression, like A$ or CHR$ 100.

Summary

Saving a program on tape

Loading a named program from tape

Loading the first available program from tape

Saving a program so that it will load & then run itself

Statements: **SAVE**, **LOAD**

Note

You cannot load programs that were saved from any other kinds of computer or from the ZX80 with its own integer BASIC. Your saved programs cannot be loaded into other kinds of computer or the ZX80 with integer BASIC. The ZX80 with ZX81 BASIC, on the other hand is compatible with the ZX81; saved programs from either can be loaded to the other. After loading a ZX80 program, the ZX81 will be in fast mode.

Exercises

1. Make a tape with loads of short programs, start playing it back into the computer, & type

LOAD "NOT THE NAME OF A PROGRAM"

You should easily be able to see the difference on the television between the empty stretches on the tape (with a fairly unstructured black & white pattern) & the programs (with more definite lines). Both patterns are different from what you see when you save. If you turn the volume down while a program is going past, you can see the picture switching to the empty space pattern while the signal goes too quiet to look like program.

2. Make a tape on which the first program, when loaded, prints a menu (a list of the other programs on the tape), asks you to choose a program, & then loads it.

3. Type in the program "CHARACTERS" again, & then type

LET X=7

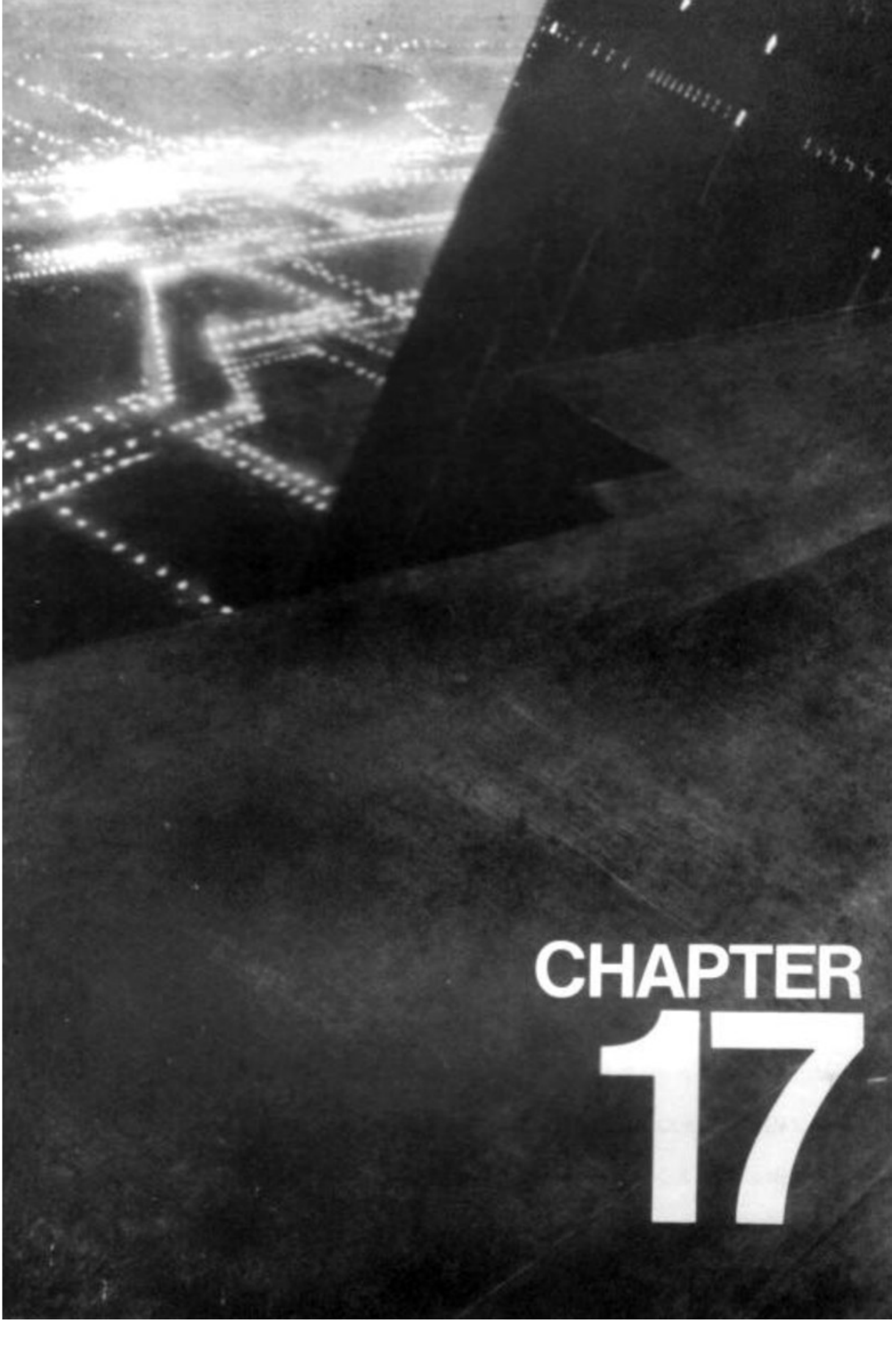
so that - although it doesn't appear in the program - the computer now contains a variable X with value 7. Now save the program, turn the computer off & on (to make sure there's no cheating), & load the program back again. Type

PRINT X

& you will get the answer 7. The **SAVE** statement saved not only the program, but also all the variables - including X.

If you want to keep these variables when you execute the program, you must remember to use **GOTO** & not **RUN** (as was mentioned in chapter 9). You can avoid having to remember this by making the program execute itself (using **SAVE** as a program line).

4. Type in a very long program, & them momentarily disconnect the power supply. This sort of thing sometimes happens spontaneously; it is not a bug, but a glitch. There is nothing you can do about it except cry. It if happens more often than you can bear then there is probably something wrong, but it would be worth saving the incomplete program on tape half-way through.



CHAPTER 17

Printing with frills

You will recall that a **PRINT** statement has a list of items, each one an expression (or possibly nothing at all), & that they are separated by commas or semicolons. There are two more kinds of **PRINT** item which are used to tell the computer not what, but where to print. For example, **PRINT** AT 11,16; "*" prints a star in the middle of the screen.

AT line, column

moves the **PRINT** position (the place where the next item is to be printed) to the line & column specified. Lines are numbered from 0 (at the top) to 21, & columns from 0 (on the left) to 31.

TAB column

moves the **PRINT** position to the column specified. It stays on the same line, or, if this would invoke back-spacing, moves on to the next one. Note that the computer reduces the column number modulo 32 (it divides by 32 & takes the remainder); so **TAB** 33 means the same as **TAB** 1.

For example

PRINT TAB 30,1;**TAB** 12;"CONTENTS";**AT** 3,1;"CHAPTER";**TAB** 24;"PAGE"

(This is how you might print out the heading of a Contents page, with 1 as the page numbers.)

Some small points:

(i) These new items are best terminated with semicolons, as we have done above. You can use commas (or nothing, at the end of the statement), but this means that after having carefully set up the PRINT position you immediately move it on again - not usually terribly useful.

(ii) Although **AT** & **TAB** are not functions, you have to type the function key (shifted **NEWLINE**) to get them.

(iii) You cannot print on the bottom two lines (22 & 23) of the screen because they are reserved for commands, **INPUT** data, reports and so on. References to the 'bottom line' usually mean line 21.

(iv) You can use **AT** to put the **PRINT** position even where there is already something printed; the old stuff will be overwritten.

There are two more statements connected with **PRINT**, namely **CLS** & **SCROLL**.

CLS Clears the Screen (but nothing else).

SCROLL moves the whole display up one line (losing the top line) & moves the **PRINT** position to the beginning of the bottom line.

To see how this works, run this program:

```
10 SCROLL
20 INPUT A$
30 PRINT A$
40 GOTO 10
```

Summary

PRINT items: **AT**, **TAB**

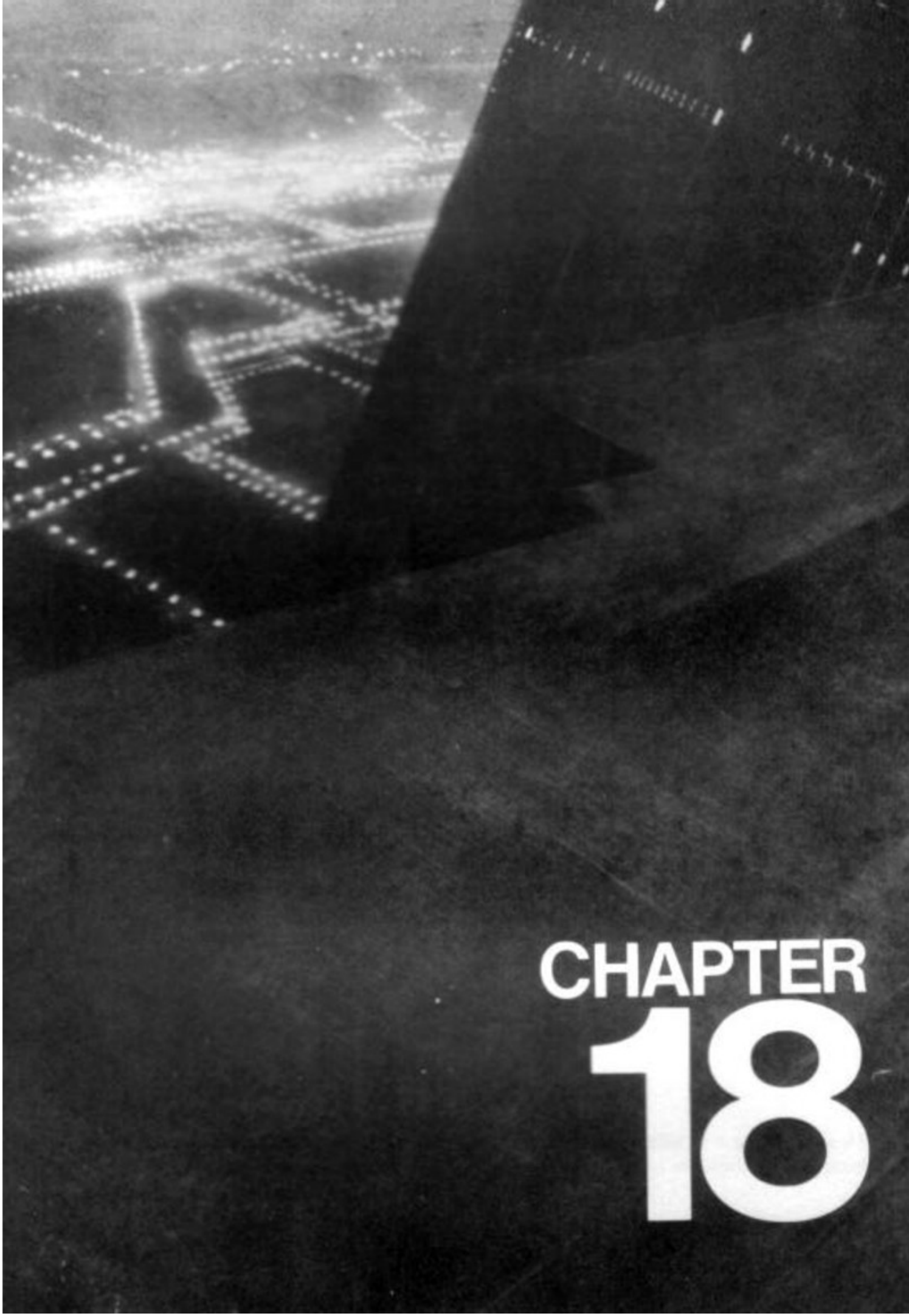
Statements: **CLS**, **SCROLL**

Exercises

1. Try running this:

```
10 FOR I=0 TO 20
20 PRINT TAB 8*I;I;
30 NEXT I
```

This shows what is meant by the **TAB** number's being reduced modulo 32. For a more elegant example, change the 8 in line 20 to a 6.



CHAPTER 18

Graphics

Here are some of the most elegant features of the ZX81; they use what are called pixels (picture elements). The screen you can display on has 22 lines & 32 columns, making 22*23 = 704 character positions, & each of these contains 4 pixels, divided up like a slice of Battenburg cake.

A pixel is specified by two numbers, its coordinates. The first, its x-coordinate, says how for it is across from the extreme left-hand column (remember, X is ACROSS), & the second, its y-coordinate, says how far if is up from the bottom. These coordinates are usually written as a pair in brackets, so (0,0), (63,0), (0,43) & (63,43) are the bottom left-, bottom right-, top left-, & top right-hand corners.

The statement

PLOT x-coordinate, y-coordinate

blacks in the pixel with these coordinates, while the statement

UNPLOT x-coordinate, y-coordinate

blanks it out.

Try this measles program:

```
10 PLOT INT (RND*64),INT (RND*44)
20 INPUT A$
30 GOTO 10
```

This plots a random point each time you press **NEWLINE**.

Here is a rather more useful program. It plots a graph of the function **SIN** (a sine wave) for values between 0 & 2ð.

```
10 FOR N=0 TO 63
20 PLOT N, 22+20*SIN (N/32*PI)
30 NEXT N
```

This next one plots a graph of **SQR** (part of a parabola) between 0 & 4:

```
10 FOR N=0 TO 63
20 PLOT N,20*SQR (N/16)
30 NEXT N
```

Notice that pixel coordinates are rather different from the line & column in an **AT** item. At the end of this chapter is a diagram which you may find useful in working out pixel coordinates & line & column numbers.

Exercises

1. There are three differences between the numbers in an **AT** item & pixel coordinates; what are they?

Suppose that the **PRINT** position corresponds to **AT** L,C (for line & column). Prove to yourself that the four pixels in that position have x-coordinates 2*C or 2*C+1, & y-coordinates 2*(21-L) or 2*(21-L)+1. (Look at the diagram.)

2. Make a cheese nibbler by altering the measles program so that it first fills the screen with black (a black square is an inverse video space), & then unplots random points, if you have only 1K of memory - that is to say, the standard machine without any extra memory, - you will find yourself running out of store, so you will have to fix the program so that it uses only part of the screen.

3. Modify the **SIN** graph program so that before plotting the graph itself it prints a horizontal line of "-"s for an x-axis, & a vertical line of "/"s for a y-axis.

4. Write programs to plot graphs of more functions, e.g. **COS**, **EXP**, **LN**, **ATN**, **INT** & so on. For each one you will have to make sure that the graph fits the screen, so you will need to consider

(i) over what range you are going to take the functions (corresponding to the range 0 to 2ð for the **SIN** graph).

(ii) whereabouts on the screen to put the x-axis (corresponding to 22 in line 20 in the **SIN** graph program).

(iii) how to scale the y-axis of the graph (corresponding to 20 in line 20 of the **SIN** graph program).

You will find that **COS** is the easiest - it's just like **SIN**.

5. Run this:

```
10 PLOT 21,21
20 PRINT "HEAVY QUOTES"
30 PLOT 46,21
```

PLOT moves on the **PRINT** position. (**UNPLOT** does too.)

6. This subroutine draws a (fairly) straight line from the pixel (A,B) to the pixel (C,D).

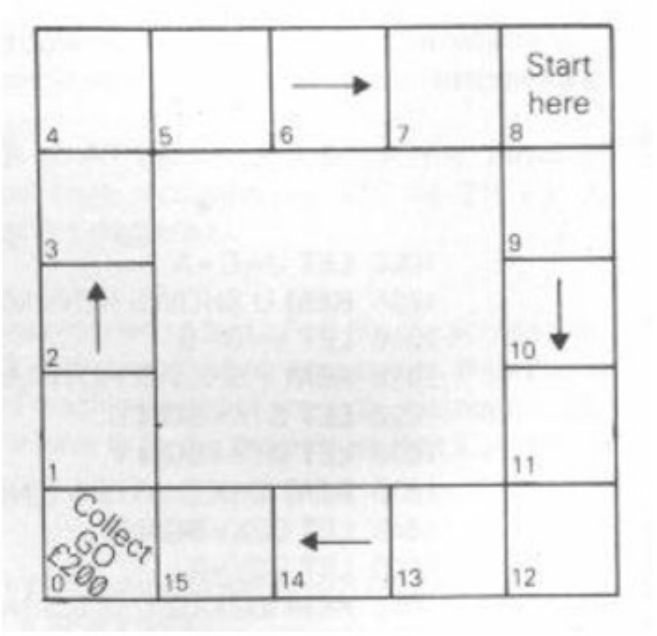
Use it as part of some main program that supplies the value of A, B, C & D.

(If you have not got a memory expansion board then you'll probably need to omit the **REM** statements.)

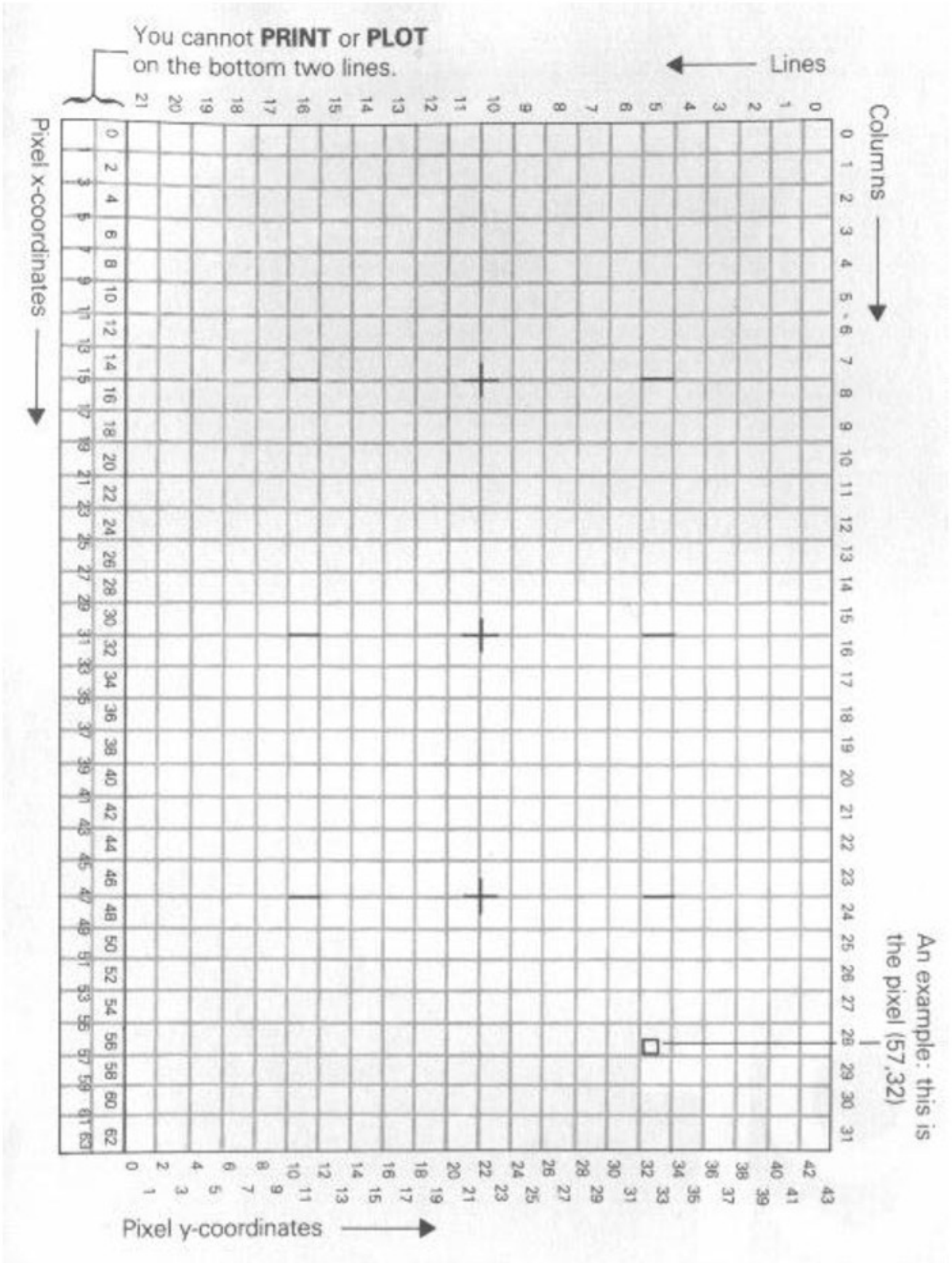
```
1000 LET U=C-A
1005 REM U SHOWS HOW MANY STEPS ALONG WE NEED TO GO
1010 LET V=D-B
1015 REM V SHOWS HOW MANY STEPS UP
1020 LET D1X=SGN U
1030 LET D1Y=SGN V
1035 REM (D1X,D1Y) IS A SINGLE STEP IN A DIAGONAL DIRECTION
1040 LET D2X=SGN U
1050 LET D2Y=0
1055 REM (D2X,D2Y) IS A SINGLE STEP LEFT OR RIGHT
1060 LET M=ABS U
1070 LET N=ABS V
1080 IF M>N THEN GOTO 1130
1090 LET D2X=0
1100 LET D2Y=SGN V
1105 REM NOW (D2X,D2Y) IS A SINGLE STEP UP OR DOWN
1110 LET M=ABS V
1120 LET N=ABS U
1130 REM M IS THE LARGER OF ABS U & ABS V, N IS THE SMALLER
1140 LET S=INT (M/2)
1145 REM WE WANT TO MOVE FROM (A,B) TO (C,D) IN M STEPS USING N UP- DOWN OR RIGHT-LEFT STEPS D2, & M-N DIAGONAL STEPS D1, DISTRIBUTED AS EVENLY AS POSSIBLE
1150 FOR I=0 TO M
1160 PLOT A,B
1170 LET S=S+N
1180 IF S<M THEN GOTO 1230
1190 LET S=S-M
1200 LET A=A+D1X
1210 LET B=B+D1Y
1215 REM A DIAGONAL STEP
```

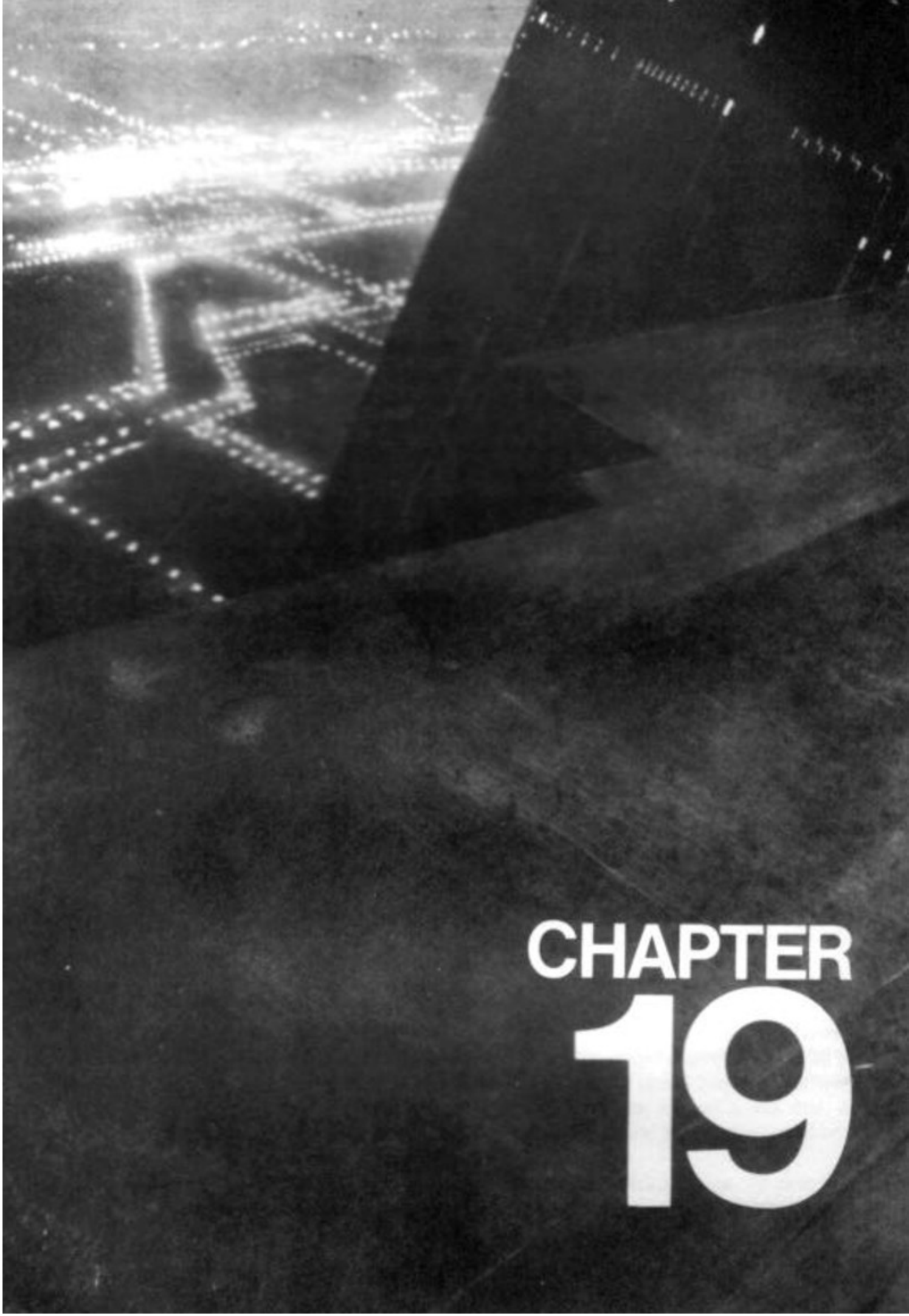
```
1220 GOTO 1250
1230 LET A=A+D2X
1240 LET B=B+D2Y
1245 REM AN UP-DOWN OR RIGHT-LEFT STEP
1250 NEXT I
1260 RETURN
```

The last part (lines 1150 on) mixes the M-N step D1 evenly with the N steps D2. Imagine a Monopoly board with M squares round the edge, numbered from 0 to M-1. The square you are on at any time is number S, starting at the corner opposite G0. Each move takes you N squares round the board, & in the straight line on the screen you make either a left-right/ up-down step (if you pass GO on the board), or a diagonal step otherwise. Since your total journey on the board is M*N steps, or right round N times, you pass GO N times & evenly spaced out in your M steps are N left-right/up-down steps.



Adjust the program so that if another parameter, E, is 1 then the line is drawn in black (as here), & if it is 0 then the line is drawn in white (using **UNPLOT**). You can then rub out a line you've just drawn by undrawing it.


You cannot **PRINT** or **PLOT** on the bottom two lines.
Lines
Columns
Pixel x-coordinates
Pixel y-coordinates
An example: this is the pixel (57,32)



CHAPTER 19

Time & motion

Quite often you will want to make the program take a specified length of time, & for this you will find the **PAUSE** statement useful (especially in fast mode):

PAUSE n

stops computing & displays the picture for n frames of the television (at 50 frames per second, or 60 in America). n can be up to 32767, which gives you just under 11 minutes; if n is any bigger then it means '**PAUSE** for ever'.

A pause can always be cut short by pressing a key (note that a space, or £, will cause a break as well). You have to press the key down after the pause has started.

At the end of the pause, the screen will flash.

If a **PAUSE** statement is used in a program that will be run in FAST mode or on the old ZX80 with the new 8K ROM, then the **PAUSE** statement must be followed by **POKE** 16437,355. The **PAUSE** will appear to work without doing this, but it will probably result in your program being wiped out.

This program works the second hand (here just a single dot on the edge) of a clock.

```
5 REM FIRST WE DRAW THE CLOCK FACE
10 FOR N=1 TO 12
20 PRINT AT 10-10*COS (N/6*PI),10+10*SIN (N/6*PI);N
30 NEXT N
35 REM NOW WE START THE CLOCK
40 FOR T=0 TO 10000
45 REM T IS THE TIME IN SECONDS
50 LET A=T/30*PI
60 LET SX=21+18*SIN A
70 LET SY=22+18*COS A
200 PLOT SX,SY
300 PAUSE 42
310 POKE 16437,255
320 UNPLOT SX,SY
400 NEXT T
```

(Miss out the **REM** statements unless you have a memory expansion board.)

This clock will run down after about 2 3/4 hours because of line 40, but you can easily make it run longer. Note how the timing is controlled by line 300. You might expect **PAUSE** 50 to make it tick once a second, but the computing takes a bit of time as well & has to be allowed for. This is best done by trial & error, timing the computer clock against

a real one, & adjusting line 300 until they agree. (You can't do this very accurately; an adjustment of one frame in one second is 2% or half an hour a day.)

The function **INKEY$** (which has no argument) reads the keyboard. If you are pressing exactly one key (or the shift key and one other key) then the result is the character that the key gives in **L** mode; otherwise the result is the empty string. The control characters do not have their usual effect, but give results like **CHR$** 118 for newline - they are printed as "?".

Try this program, which works like a typewriter.

```
10 IF INKEY$<>"" THEN GOTO 10
20 IF INKEY$="" THEN GOTO 20
30 PRINT INKEY$;
40 GOTO 10
```

Here line 10 waits for you to lift your finger off the keyboard & line 20 waits for you to press a new key.

Remember that unlike **INPUT**, **INKEY$** doesn't wait for you. So you don't type newline, but on the other hand if you don't type anything at all then you've missed your chance.

Exercises

1. What happens if you miss out line 10 in the typewriter program?

2. Why can you not type space or £ in the typewriter program?

Here is a modified program that gives you a space if you type cursor right (shifted 8).

```
10 IF INKEY$<>"" THEN GOTO 10
20 IF INKEY$="" THEN GOTO 20
30 LET A$=INKEY$
40 IF A$=CHR$ 115 THEN GOTO 110
90 PRINT A$;
100 GOTO 10
110 PRINT " ";
120 GOTO 10
```

Note how we read **INKEY$** into A$ in line 30. It would be possible to miss this out & replace A$ by **INKEY$** in lines 40 & 0, but there would always be a chance that **INKEY$** could change between the lines.

Add some more programs so that if you type **NEWLINE** (**CHR$** 118) it gives you a new line.

3. Another way of using **INKEY$** is in conjunction with **PAUSE**, as in this alternative typewriter program.

```
10 PAUSE 40000
20 POKE 16437,255
30 PRINT INKEY$
40 GOTO 10
```

To make this work, why is it essential that a pause should not finish if it finds you already pressing a key when it starts?

This method has the disadvantage that the screen flashes, but in fast mode it is the only way of doing it. Run the program in fast mode, & notice that the computer takes the opportunity of a pause to display the television picture.

4. This one will drive you mad. The computer displays a number, which you (or an innocent victim) shall type back. To begin with you have a second to do it in, but if you get it wrong you get a longer time for the next number, while if you get it right you get less time for the next one. The idea is to get it going as fast as possible, & then press Q to see your score - the higher the better.

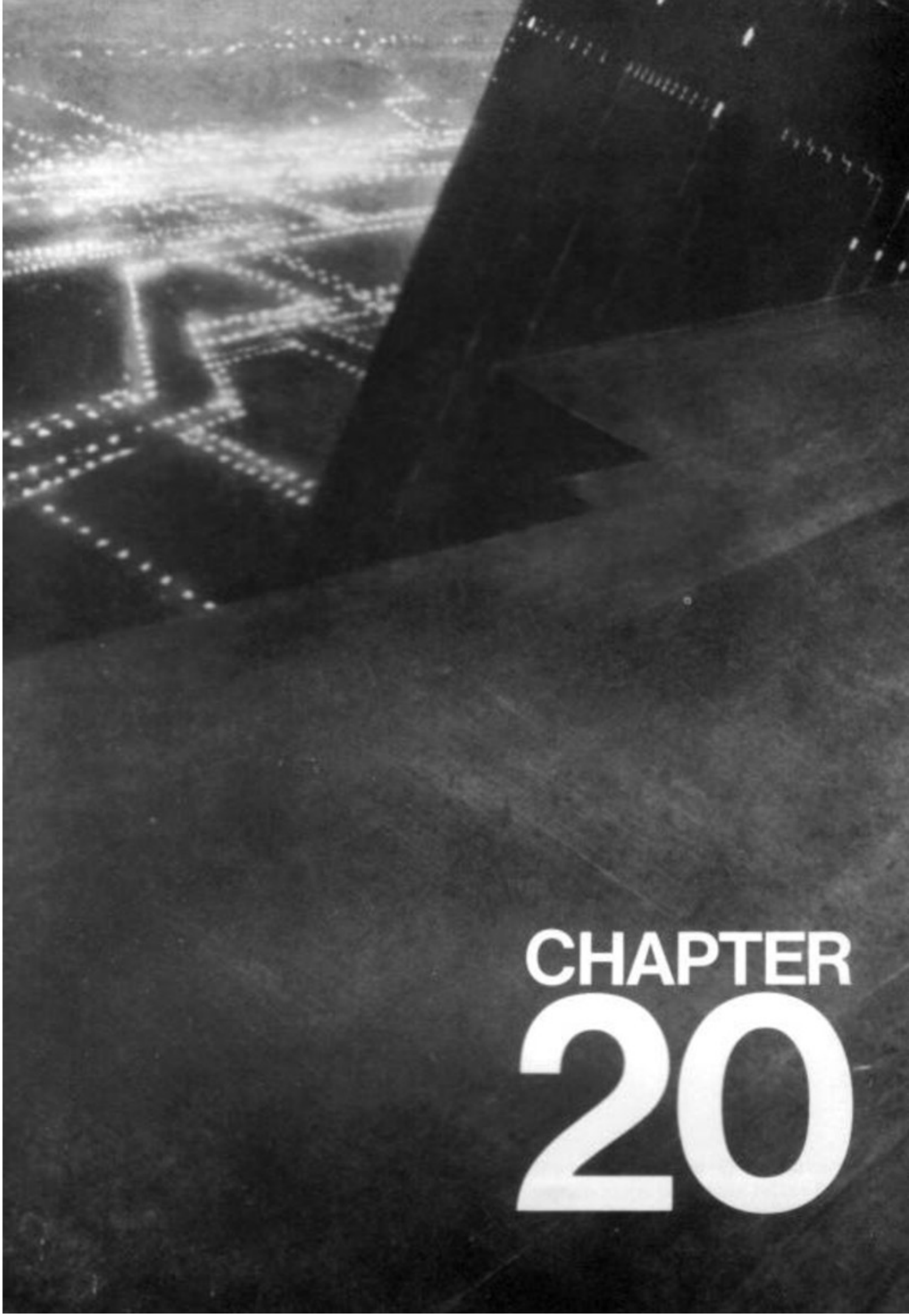
```
10 LET T=50
15 REM T=NUMBER OF FRAMES PER GO-INITIALLY 50 FOR 1 SECOND
20 SCROLL
30 LET A$=CHR$ INT (RND*10+CODE "0")
35 REM A$ IS A RANDOM DIGIT
40 PRINT A$
45 PAUSE T
50 POKE 16437,255
60 LET B$=INKEY$
70 IF B$="Q" THEN GOTO 200
80 IF A$=B$ THEN GOTO 150
90 PRINT "NO GOOD"
100 LET T=T*1.1
```

```
110 GOTO 20
150 PRINT "OK"
160 LET T=T*0.9
170 GOTO 20
200 SCROLL
210 PRINT "YOUR SCORE IS "; INT (500/T)
```

5. (Only for those with extra RAM.) Using the straight line routine in chapter 15, change the second hand program so that it also shows minute & hour hands, drawing them every minute. (Make the hour hand shorter.) If you're feeling ambitious, arrange so that every quarter of an hour it puts on some kind of a show.

6. (For fun.) Try this:

```
10 IF INKEY$ = "" THEN GOTO 10
20 PRINT AT 11.14; "OUCH"
30 IF INKEY$<>"" THEN GOTO 30
40 PRINT AT 11,14; "    "
50 GOTO 10
```



CHAPTER 20

The ZX81 Printer

If you have got a ZX81 printer, you will have some operating instructions with it. This chapter covers the BASIC statements needed to make it work.

The first two, **LPRINT** & **LLIST**, are just like **PRINT** and **LIST**, except that they use the printer instead of the television. (The L is a historical accident. When BASIC was invented it usually used an electronic typewriter instead of a television, so **PRINT** really did mean print. If you wanted messages of output you would use a very fast line printer attached to the computer, & an **LPRINT** statement meaning 'Line printer **PRINT**'.)

Try this program for example.

```
10 LPRINT "THIS PROGRAM:",,,,
20 LLIST
30 LPRINT ,,"PRINTS OUT THE CHARACTER SET.",,,
40 FOR N=0 TO 255
50 LPRINT CHR$ N;
60 NEXT N
```

The third statement, **COPY**, prints out a copy of the television screen. For instance, get a listing on the screen of the program above, & type

```
COPY
```

You can always stop the printer when it is running by pressing **BREAK** key (space).

If you execute these statements without the printer attached, it should just lose all the output & carry on with the next statement. However, sometimes it might get stuck, & when this happens the break key will bring it out.

Summary

Statements: **LPRINT**, **LLIST**, **COPY**

Note: None of these statements is standard BASIC, although **LPRINT** is used by some other computers.

Exercises

1. Try this:

```
10 FOR N=31 TO 0 STEP -1
20 PRINT AT 31-N,N;CHR$ (CODE "0" +N);
30 NEXT N
```

You will see a pattern of letters working down diagonally from the top right-hand corner until it reaches the bottom of the screen, when the program stops with error report 5.

Now change '**AT** 31-N,N' in line 20 to '**TAB** N'. The program will have exactly the same effect as before.

Now change **PRINT** in line 20 to **LPRINT**. This time there will be no error 5, which should not occur with the printer, & the pattern will carry on an extra ten lines with the digits.

Now change '**TAB** N' to '**AT** 21-N,N' still using **LPRINT**. This time you will get just a single line of symbols. The reason for the difference is that the output from the **LPRINT** is not printed straight away, but arranged in a buffer store a picture one line long of what the computer will print when it gets round to it. The printing takes place

(i) when the buffer is full,

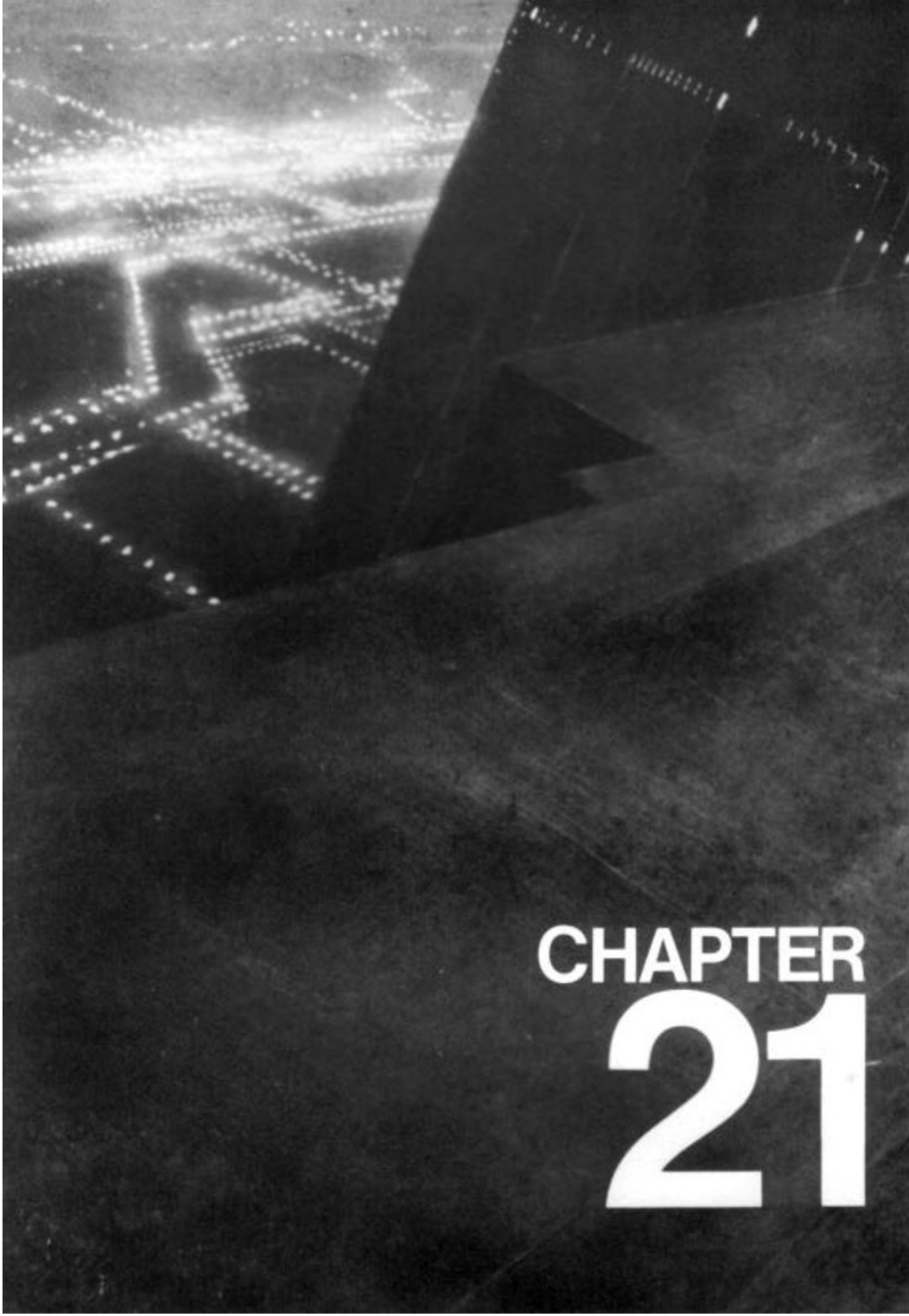
(ii) after an **LPRINT** statement that does not end in a comma or semicolon.

(iii) when a comma or **TAB** item requires a new line.

or (iv) at the end of a program, if there is anything left unprinted.

(iii) explains why our program with **TAB** works the way it does. As for **AT**, the line number is ignored & the **LPRINT** position (like the **PRINT** position, but for the printer instead of the television) is changed to the column number. An **AT** item can never cause a line to be sent to the printer. (Actually, the line number after **AT** is not completely ignored; it has to be between -21 & +21 or an error will result. For this reason it is safest always to specify line 0. The item '**AT** 21-N,N' in the final version of our program would be much better albeit less illustrative if replaced by '**AT** 0,N'.)

2. Make a printed graph of **SIN** by running the program in chapter 18 & then using **COPY**.



CHAPTER 21

Substrings

Given a string, a substring of it consists of some consecutive characters from it, taken in sequence. Thus "STRING" is a substring of "BIGGER STRING", but "B STRING" & "BIG REG" are not.

There is a notation called slicing for describing substrings, & this can be applied to arbitrary string expressions. The general form is

string expression (start **TO** finish)

so that, for instance,

"ABCDEF" (2 **TO** 5) = "BCDE"

If you omit the start, then 1 is assumed; if you omit the finish then the length of the string is assumed. Thus

"ABCDEF" (**TO** 5) = "ABCDEF" (1 **TO** 5) = "ABCDE"

"ABCDEF" (2 **TO**) = "ABCDEF" (2 **TO** 6) = "BCDEF"

&

"ABCDEF" (**TO**) = "ABCDEF" (1 **TO** 6) = "ABCDEF"

(you can also write this last one as "ABCDEF" (), for what it's worth.)

A slightly different for misses out the **TO** & just has one number:

"ABCDEF" (3) = "ABCDEF" (3 **TO** 3) = "C"

Although normally both start & finish must refer to existing parts of the string, this rule is overridden by one other: if the start is more than the finish, then the result is the empty string. So

"ABCDEF" (5 **TO** 7)

gives error 3 (subscript error) because, the string only contains 6 characters, & 7 is too many, but

"ABCDEF" (8 **TO** 7) = ""

&

"ABCDEF" (1 **TO** 0) = ""

The start & finish must not be negative, or you get error B.

This next program makes B$ equal to A$, but omitting any trailing spaces.

```
10 INPUT A$
20 FOR N=LEN A$ TO 1 STEP -1
30 IF A$(N)<>"" THEN GOTO 50
40 NEXT N
50 LET B$=A$( TO N)
60 PRINT """";A$;""""","""";B$;""""
70 GOTO 10
```

Note how if A$ is entirely spaces, then in line 50 we have N = 0 & A$(**TO** N) = A$(1 **TO** 0) = "".

For string variables, we can not only extract substrings, but also assign to them. For instance type

```
LET A$="LOR LOVE A DUCK"
```

& then

```
LET A$(5 TO 8)="*******"
```

&

```
PRINT A$
```

Notice how since the substring A$(5 **TO** 8) is only 4 characters long, only the first four stars have been used. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short - this is called Procrustean assignment after the innkeeper Procrustes who used to make sure that his guests fitted the bed by either stretching them out on a rack or cutting their feet off.

If you now try

```
LET A$()="COR BLIMEY"
```

&

```
PRINT A$;"."
```

you will see that the same thing has happened again (this time with spaces put in) because A$() counts as a substring.

```
LET A$="COR BLIMEY"
```

will do it properly

Slicing may be considered as having priority 12, so, for instance

```
LEN "ABCDEF"(2 TO 5) = LEN("ABCDEF"(2 TO 5)) = 4
```

Complicated string expressions will need brackets round them before they can be sliced. For example,

"ABC"+"DEF"(1 **TO** 2) = "ABCDE"

("ABC"+"DEF")(1 **TO** 2) = "AB"

Summary

Slicing, using **TO**. Note that this notation is non-standard.

Exercises

1. Some BASICs (not the ZX81 BASIC) have functions called LEFT$, RIGHT$, MID$ & TL$.

LEFT$(A$,N) gives the substring of A$ consisting of the first N characters.

RIGHT$(A$,N) gives the substring of A$ consisting of the characters from the Nth on.

MID$(A$,N1,N2) gives the substring of A$ consisting of N2 characters starting at the N1th.

TL$(A$) gives the substring of A$ consisting of all its characters except the first.

How would you write these in ZX81 BASIC? Would your answers work with strings of length 0 or 1?

2. Try this sequence of commands:

LET A$="X*+*Y"

LET A$(2)=**CHR$** 11 [the string quote character]

LET A$(4)=**CHR$** 11

PRINT A$

A$ is now a string with string quotes inside it! So there is nothing to stop you doing this if you are persevering enough, but clearly if you had originally typed

LET A$="X"+"Y"

the part to the right of the equals sign would have been treated as an expression, giving A$ the value "XY".

Now type

LET B$="X""+""Y"

You will find that although A$ & B$ look the same when printed out, they are not equal - try

PRINT A$=B$

Whereas B$ contains mere quote image characters (with code 192), A$ contains genuine string quote characters (with code 11).

3. Run this program:

10 **LET** A$="**LEN** """ABDC"""

100 **PRINT** A$;" = ";**VAL** A$

This will fail because **VAL** does not treat the quote image **""** as a string quote.

Insert some extra lines between 10 & 100 to replace the quote images in A$ by string quotes (which you must call **CHR$** 11), & try again.

Make the same modifications to the program in chapter 9, exercise 3, & experiment with it.

4. This subroutine deletes every occurrence of the string "CARTHAGO" from A$.

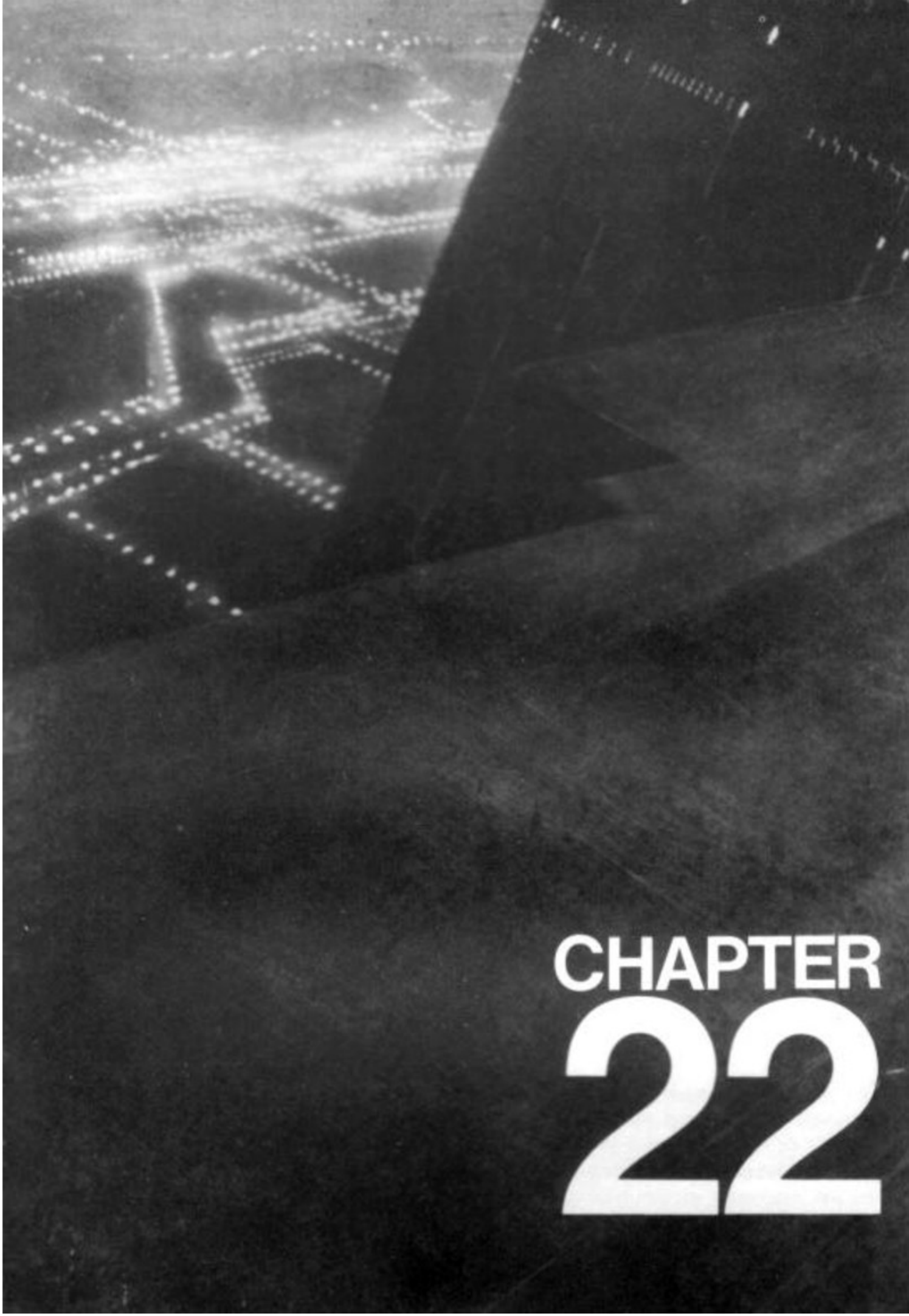
1000 **FOR** N=1 **TO LEN** A$-7

1020 **IF** A$(N **TO** N+7)="CARTHAGO" **THEN LET** A$(N TO N+7)="*********"

1030 **NEXT** N

1040 **RETURN**

Write a program that gives A$ various values (e.g. "DELENDA EST CARTHAGO.") & applies the subroutine.



CHAPTER 22

Arrays

Suppose you have a list of numbers, for instance the number of income tax collectors that have died each month in the current financial year. To store them in a computer you could set up a single variable for each month, but you would find this very awkward. You might decide to call the variables ALAS NO MORE 1, ALAS NO MORE 2, & so on up to ALAS NO MORE 12, but the program to print out these twelve numbers would be rather long & boring to type in.

How much nicer it would be if you could type this:

```
5 REM THIS PROGRAM WILL NOT WORK
10 FOR N=1 TO 12
20 PRINT ALAS NO MORE N
30 NEXT N
```

Well, you can't.

However, there is a mechanism by which you can apply this idea, & it uses arrays. An array is a set of variables, or elements, all with the same name, & distinguished only by a number (the subscript) written in brackets after the name. In our example the name would be A (like control variables for **FOR-NEXT** loops, the name of an array must be a single letter), & the twelve variables would then be A(1), A(2), & so on up to A(12).

The elements of an array are called subscripted variables, as opposed to the simple variables that you are already familiar with.

Before you can use an array, you must reserve some space for it inside the computer, & you do this using a **DIM** (for dimension) statement.

```
DIM A(12)
```

sets up an array called A with dimension 12 (i.e. there are 12 subscripted variables A(1),...,A(12)), & initializes the 12 values to 0. It also deletes any array called A that existed previously. (But not a simple variable. An array & a simple variable with the same name can coexist, & there shouldn't be any confusion between them because the array variable always has a subscript.)

The subscript can be an arbitrary numerical expression, so now you can write

```
10 FOR N=1 TO 12
20 PRINT A(N)
30 NEXT N
```

You can also set up arrays with more than one dimension. In a two-dimensional array you need two numbers to specify one of the elements - rather like the line & column numbers to specify a character position on the television screen - so it has the form of a table. Alternatively, if you imagine the line & column numbers (two dimensional) as referring to a page printed, you could have an extra dimension for the page numbers. Of

course, we are talking about numeric arrays; so the elements would not be printed characters as in a book, but numbers. Think of the elements of a three-dimensional array C as being specified by C (page number, line number, column number).

For example, to set up a two-dimensional array B with dimensions 3 & 6, you use a **DIM** statement

DIM B(3,6)

This then gives you 3*6 = 18 subscripted variables

B(1,1), B(1,2),..., B(1,6)

B(2,1), B(2,2),..., B(2,6)

B(3,1), B(3,2),..., B(3,6)

The same principal works for any number of dimensions.

Although you can have a number & an array with the same name, you cannot have two arrays with the same name, even if they have different numbers of dimensions.

There are also string arrays. The strings in an array differ from simple strings in that they are of fixed length & assignment to them is always

Procrustean - another way of thinking of them is as arrays (with one extra dimension) of single characters. The name of a string array is a single letter followed by $, & a string array & a simple string variable cannot have the same name (unlike the case for numbers).

Suppose then, that you want an array A$ of five strings. You must decide how long these strings are to be - let us suppose that 10 characters each is long enough. You then say

DIM A$(5,10) (type this in)

This sets up a 5*10 array of characters, but you can also think of each row as being a string:

A$(1)=A$(1,1) A$(1,2) ... A$(1,10)

A$(2)=A$(2,1) A$(2,2) ... A$(2,10)

: : : : : :

A$(5)=A$(5,1) A$(5,2) ... A$(5,10)

If you give the same number of subscripts (two in this case) as there were dimensions in the **DIM** statement, then you get a single character; but if you miss the last one out, then you get a fixed length string. So, for instance, A$(2,7) is the 7th character in the string A$(2); using the slicing notation, we could also write this as A$(2)(7). Now type

LET A$(2)="1234567890"

&

PRINT A$(2),A$(2,7)

You get

1234567890 7

For the last subscript (the one you can miss out), you can also have a slice, so that for instance

A$(2,4 **TO** 8) = A$(2)(4 **TO** 8) = "45678"

Remember:

In a string array, all the strings have the same, fixed length.

The **DIM** statement has an extra number (the last one) to specify this length.

When you write down a subscripted variable for a string array, you can put in an extra number, or a slicer, to correspond with the extra number in the **DIM** statement.

Summary

Arrays (the way the ZX81 handles string arrays is slightly non-standard.)

Statements: **DIM**

Exercises

1. Set up an array M$ of twelve strings in which M$(N) is the name of the Nth month. (Hint: the **DIM** statement will be **DIM** M$(12,9).) Test it by

printing out all the M$(N) (use a loop). Type

PRINT "NOW IS THE MONTH OF ";M$(5);"ING";"WHEN MERRY LADS ARE PLAYING"

What can you do about all those spaces?

2. You can have string arrays with no dimensions. Type

DIM A$(10)

& you will find that A$ behaves just like a string variable, except that it always has length 10, & assignment to it is always Procrustean.

3. READ, DATA & RESTORE; who needs them?

Most BASICs (but not the ZX81 BASIC) have three statements called READ, DATA & RESTORE.

A DATA statement is a list of expressions, & taking all the DATA statements in the program gives one long list of expressions, the DATA list.

READ statements are used to assign these expressions, one by one, to variables:

READ X

for instance, assigns the current expression in the DATA list to the variable X, & moves on to the next expression for the next READ statement.

[RESTORE moves back to the beginning of the DATA list.]

In theory, you can always replace READ & DATA statements by **LET** statements; however, one of their major uses is in initializing arrays, as in this program:

```
5 REM THIS PROGRAM WILL NOT WORK IN ZX81 BASIC
10 DIM M$(12,3)
20 FOR N=1 TO 12
30 READ M$(N)
40 NEXT N
50 DATA "JAN","FEB","MAR","APR"
60 DATA "MAY","JUN","JUL","AUG"
70 DATA "SEP","OCT","NOV","DEC"
```

If you only want to run this program once, you might as well replace line 30 with an **INPUT** statement thus:

```
10 DIM M$(12,3)
20 FOR N=1 TO 12
30 INPUT M$(N)
40 NEXT N
```

& you will have no extra typing to do. However, if you want to save the program, you will certainly not want to type the months in every time you

run it.

We suggest that you use this method:

(i) Initialize the array using a program like the one above.

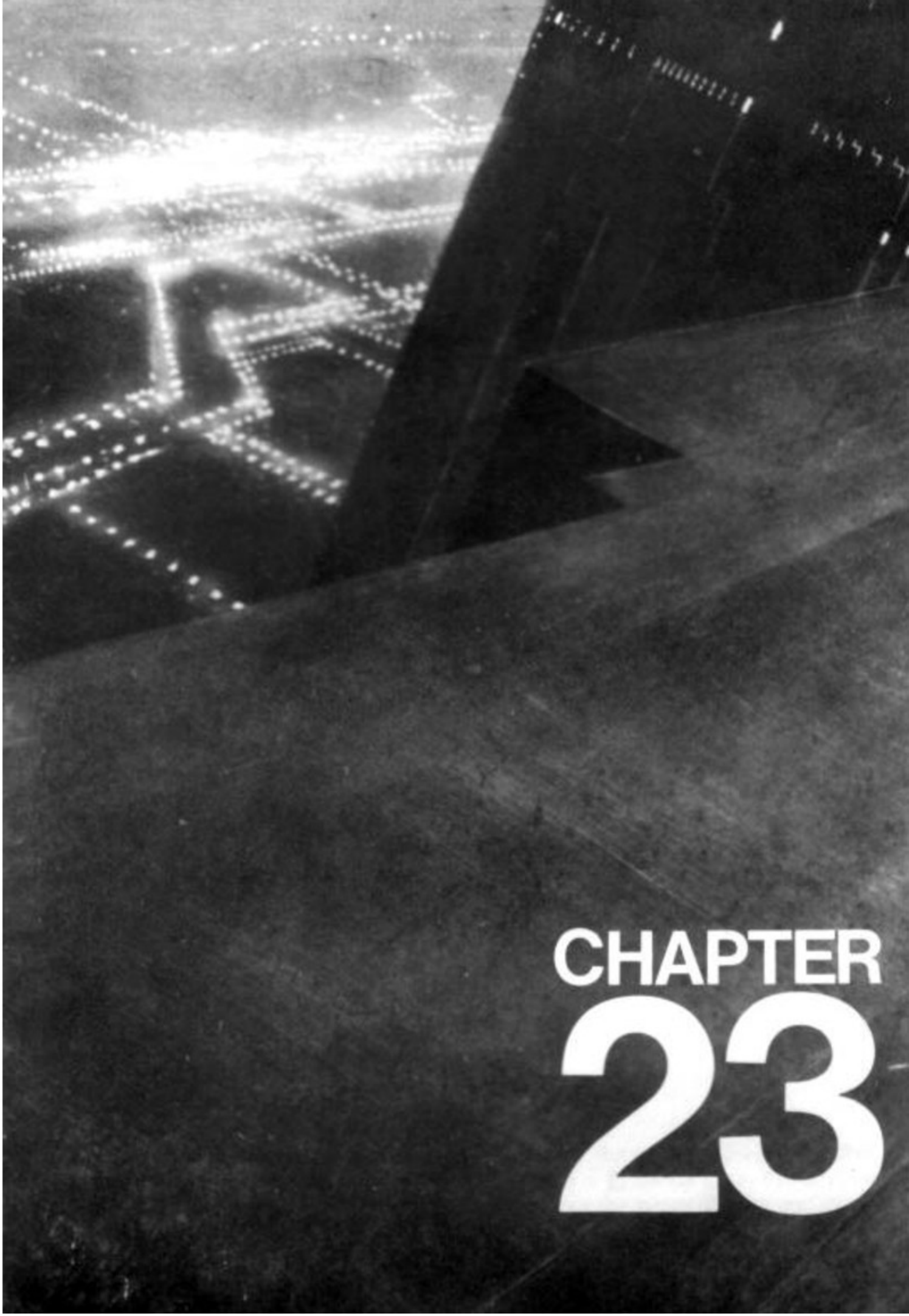
(ii) Edit out the initialization program. (Don't use **NEW**, because you want to preserve the array.)

(iii) Type in the rest of the program, & save it. This will save the variables as well, including the array.

(iv) When you load the program back, you will also load the array.

(v) When you execute the program, do not use **RUN**, which clears the variables. Use **GOTO** with a line number instead.

You may alternatively be able to use the **LOAD** & execute technique of chapter 16, & its exercise 3. Then in stage (iii) above you will use a **SAVE** statement in the program, & stage (v) will be omitted altogether.



CHAPTER 23

When the computer gets full

The ZX81 has a limited amount of internal storage, and it is not hard to fill it up. The best sign of this happening is usually an error report 4, but other things can happen and some of them are rather strange. The exact behaviour depends on whether you have a memory expansion board attached, so first let us assume that you have not. If you have, take it off (after first switching off the computer).

The display file, that is to say the area inside the computer where it stores the television picture, is cunningly designed so that it only takes up space for what has been printed so far: a line in the display consists of up to 32 characters and then a **NEWLINE** character. This means that you can run out of memory by printing something, and the most obvious place is while making a listing. Type

```
NEW
DIM A(150)
 10 FOR N=1 TO 15
 20 PRINT N
```

Here comes the first surprise: line 10 disappears from the listing. The listing is bound to include the current line, 20, and there is not room for both lines. Now type

```
 30 NEXT N
```

Again, there is only room for line 30 in the listing. Now type

```
 40 REM X  (without NEWLINE)
```

and you will see line 30 disappear and line 40 jump to the top of the screen. It has not been entered in the program - you still have the L cursor and can move it about. All you have seen is some obscure mechanism that gives the bottom half of the screen 24 lines to give it priority over the top half. Now type

```
 XXXXXX  (still without NEWLINE)
```

and the cursor will disappear - there is no room to display it. Type another X, without **NEWLINE**, and one of the Xs will disappear. Now type **NEWLINE**. Everything will disappear, but the program is still in the computer, as you can prove by deleting line 10 and using ⇩ & ⇧. Now type

```
 10 FOR N=1 TO 15
```

again - it will move up to the top of the screen as line 40 did. But when you press **NEWLINE**, it will not be entered, although there is no error message or S marker to say that anything is wrong. This is the result of there being no room to check the syntax of a line, and usually happens only for lines that contain numbers (other than the line number at the beginning).

The only cure is to make some space somehow, but first delete the line 10 that won't go in. Press **EDIT**: the screen will go blank, because there is no room to bring the line down.

When **EDIT** does not work it is sometimes possible to make space by typing a number of spaces until the cursor moves up the screen.

Press **NEWLINE**, and you will get part of the listing back. Now delete the line 40 (which you didn't really want anyway) by typing

```
40   (& NEWLINE)
```

Now try typing in line 10 again - and it still won't go. Rub it out again. You must still find some extra space somewhere. Bear in mind that the reason that line 10 was rejected was probably that there was no room to check the syntax of the two numbers, 1 & 15: so if you delete line 20 in the program you might have room to enter line 10, and still have room to re-enter line 20 (which contains no number) afterwards. Try this. Type

```
20
10 FOR N=1 TO 15
20 PRINT N
```

and the program is entered properly.

Type

```
GOTO 10
```

and again you will find that this line is rejected because its syntax cannot be checked; however, of you rub it out and type

```
RUN
```

it will work. (**RUN** clears out the array, making plenty of space.)

Now type in the same as before from **NEW** up to line 30, and then

```
40 REM XXXXXXXXXXXX
```

(12 Xs), which will end up looking like 40 RE. When you press **NEWLINE**, the listing will just consist of line 30, and in fact line 40 has been completely lost. This is because it was simply too long to fit in the program. The effect is a bit worse when the line is a lengthened version of a line that is already in the program, for you will lose both the old line from the program and the new line that was to replace it.

The ultimate cure for this is to buy a RAM pack, which fits on the back of the computer. The Sinclair 16K RAM pack gives the computer sixteen times as much memory as it has in its unexpanded form.

If you have an old ZX80 3K RAM pack, it will not work on the ZX81.

The behaviour with the RAM pack is rather different, because the display file is filled out with spaces to make each line 32 characters long (note that **SCROLL** upsets this - see chapter 27). Now printing and listing will not make the computer run out of memory, and you will not see all these shortened listings, and jumping around; but you will see the lines sticking or getting lost, and again the only cure is to find some spare space.

If you have a memory expansion board, put it on and go through the typing in this chapter, using

DIM A(3069)

to replace **DIM** A(150).

To summarize, this is a tangled tale and the moral is to avoid getting an absolutely jam-packed computer if you can. However, the second moral is that things are not usually as bad as they look.

1. If the listing starts shortening or things start jumping around, then the space is getting tight.

2. If **NEWLINE** seems to have no effect at the end of a line, then there is probably no room to deal with a number. Rub out the line using **EDIT**-**NEWLINE** or **RUBOUT**.

3. **NEWLINE** might lose a line altogether.

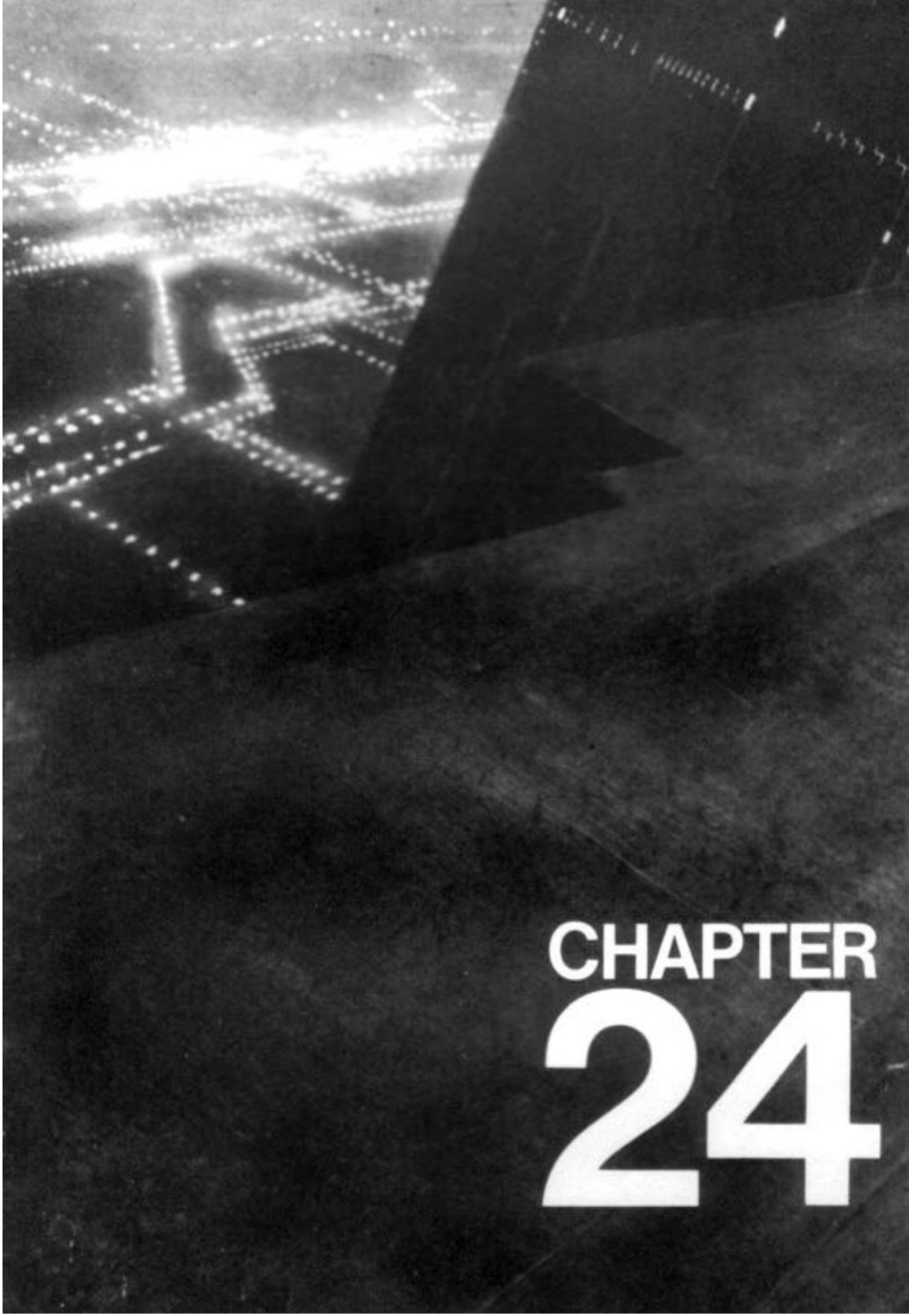
For all these oddities, the cure is the same. Don't panic, and look for some spare space.

The first thing to consider is **CLEAR**. If you have some variables and you do not mind losing any of them, then this is the thing to do.

Failing this, look for unnecessary statements in the program, such as **REM** statements, and delete some of those.

Summary

When the memory fills up odd things can happen; but they are not usually fatal.



CHAPTER 24

Counting on your fingers

The next chapter digs inside the computer a bit, but before we look at that it would be as well to describe how computers count: they do it using the binary system, which means that they have no fingers - they are all thumbs.

Most European languages count using a more or less regular pattern of tens - in English, for example, although it starts off a bit erratically, it soon settles down into regular groups:

twenty, twenty one, twenty two,...,twenty nine

thirty, thirty one, thirty two,...,thirty nine

forty, forty one, forty two, ...,forty nine

& so on, & this is made even more systematic with the Arabic numerals that we use. However, the only reason for using ten is that we happen to have ten fingers & thumbs.

Now suppose Martians have three extra fingers on each hand (in so far as one can call them fingers): so instead of using our decimal system, with ten as its base, they use a hexadecimal (or hex, for short) system, based on sixteen. They need six extra hex digits in addition to the ten that we use, & they happen to write them as A, B, C, D, E & F. And what comes after F? Just as we, with ten fingers, write 10 for ten, so they, with sixteen, write 10 for sixteen. Their number system starts off:

Hex	*English*
0	nought
1	one
2	two
:	:
:	:
9	nine

just as ours does, but then it carries on

A	ten
B	eleven
C	twelve
D	thirteen
E	fourteen
F	fifteen
10	sixteen
11	seventeen
:	:
:	:
19	twenty five
1A	twenty six

1B	twenty seven
:	:
:	:
1F	thirty one
20	thirty two
21	thirty three
:	:
:	:
9E	a hundred & fifty eight
9F	a hundred & fifty nine
A0	a hundred & sixty
A1	a hundred & sixty one
:	:
:	:
FE	two hundred & fifty four
FF	two hundred & fifty five
100	two hundred & fifty six

If you are using hex notation & you want to make the fact quite plain, then write 'h' at the end of the number, & say 'hex'. For instance, for a hundred & fifty eight, write '9Eh' & say 'nine E hex'.

You will be wondering what all this has to do with computers. In fact, computers behave as though they had only two digits, represented by a low voltage, or off (0), & a high voltage, or on (1). This is called the binary system, & the two binary digits are called bits: so a bit is either 0 or 1.

In the various systems, counting starts off

English	*Decimal*	*Hexadecimal*	*Binary*
nought	0	0	0 or 0000
one	1	1	1 or 0001
two	2	2	10 or 0010
three	3	3	11 or 0011
four	4	4	100 or 0100
five	5	5	101 or 0101
six	6	6	110 or 0110
seven	7	7	111 or 0111
eight	8	8	1000
nine	9	9	1001
ten	10	A	1010
eleven	11	B	1011
twelve	12	C	1100
thirteen	13	D	1101

fourteen	14	E	1110
fifteen	15	F	1111
sixteen	16	10	10000

The important point is that sixteen is equal to two raised to the fourth power, & this makes converting between hex & binary very easy.

To convert hex to binary, change each hex digit into four bits, using the table above.

To convert binary to hex, divide the binary number into groups of four bits, starting on the right, & then change each group into the corresponding hex digit.

For this reason, although strictly speaking computers use a pure binary system, humans often write the numbers stored inside a computer using hex notation.

The bits inside the computer are mostly grouped into sets of eight, or bytes. A single byte can represent any number from nought to two hundred & fifty five (11111111 binary or FF hex), or alternatively any character in the ZX81 character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is technically called a word. A word can be written using sixteen bits of hex digits, & represents a number from 0 to (in decimal) $2^{16}-1 = 65535$.

A byte is always eight bits, but words vary from computer to computer.

Summary

Decimal, hexadecimal & binary systems.

Bits & bytes (don't confuse them) & words.

Exercises

1. The Martian unit of currency is the pound, & it is divided into sixteen ounces. How would you convert from pounds & ounces to ounces & back again

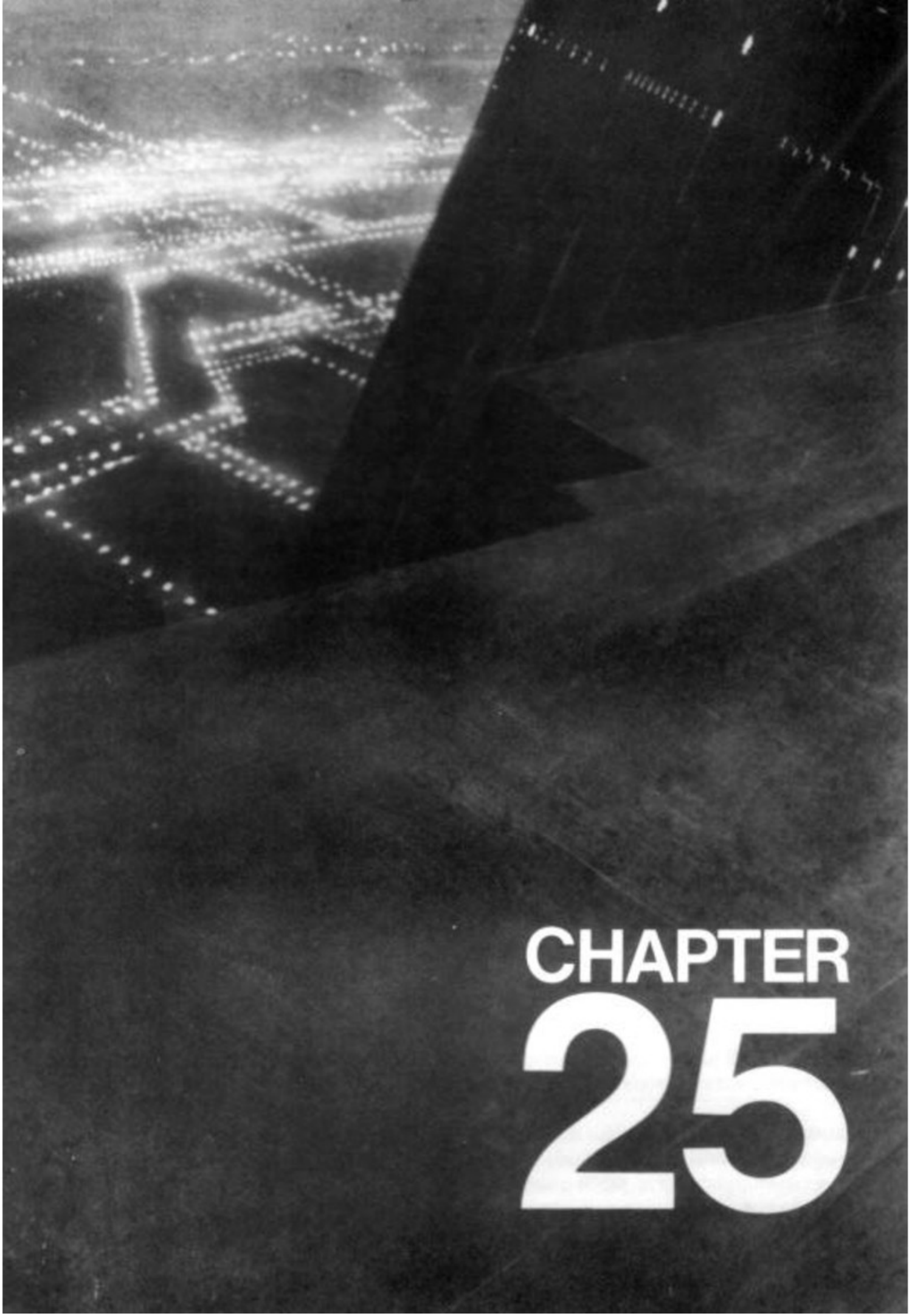
(i) when all the numbers are written in decimal?

(ii) when all the numbers are written in hex?

2. How would you convert between decimal & hex? (Hint: exercise 1.)

When programs on the ZX81 to convert numerical values into the strings giving their hex representation, & vice versa. (This is what **STR$** & **VAL** do with decimal representations.)

3. Suppose people from Venus have a total of eight fingers, without thumbs, how would their octal (base eight) counting be useful with computers?



CHAPTER 25

How the computer works

The illustration overleaf shows the inside of the ZX81 (but don't take it apart yourself because it can be a tricky business putting it together again).

As you can see, everything has a three letter abbreviation (TLA).

The pieces of plastic with lots of metal legs are the wondrous silicon chips, which have brought you not only digital watches, but also the Sinclair ZX81. Inside each piece of plastic is a piece of silicon about the size of the K cursor, joined by wires to the metal legs.

The brains behind the operation is the processor chip, often called the CPU (Central Processor Unit). This particular one is a Z80 processor (actually a Z80A, which goes faster).

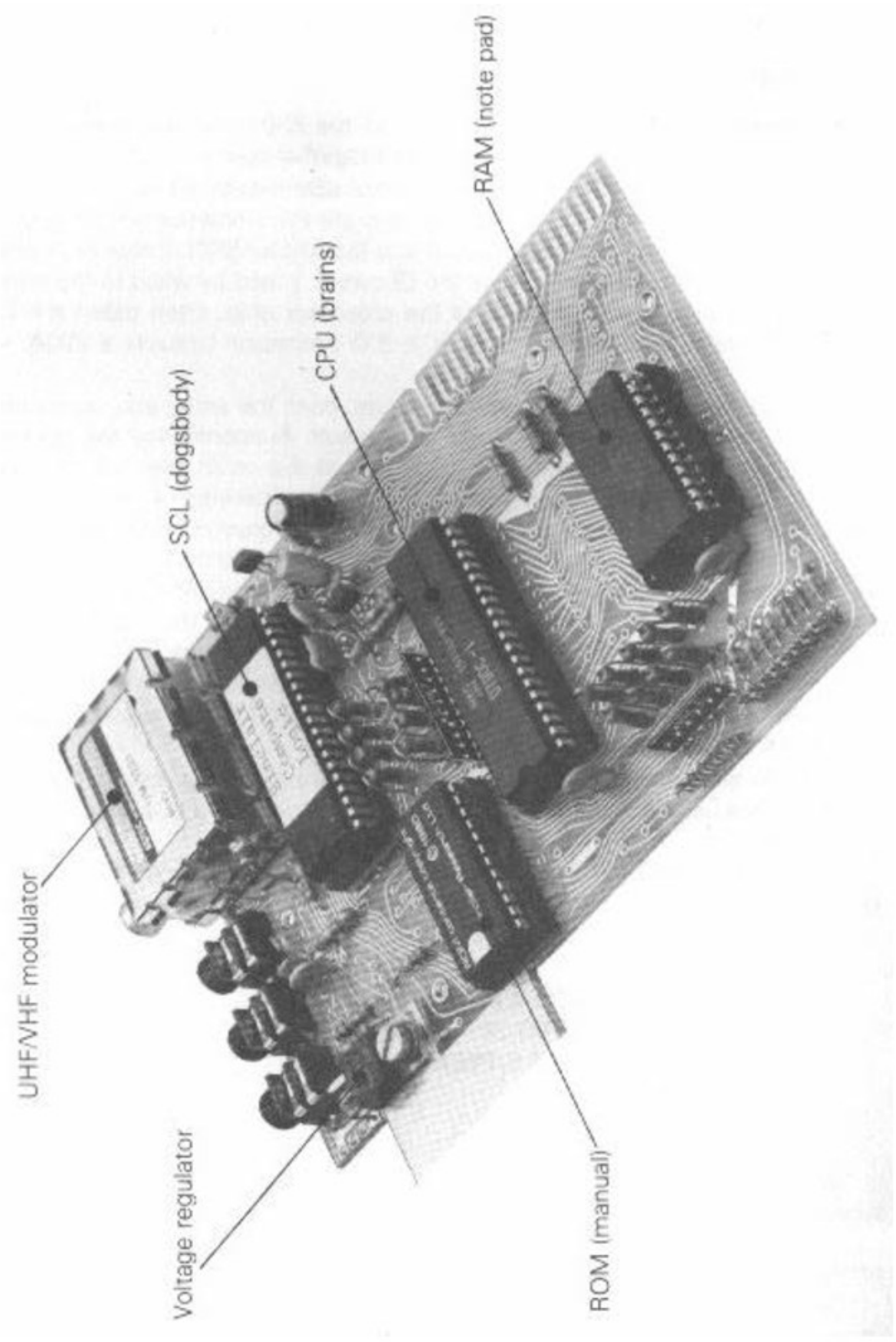
The processor controls the whole computer, does the arithmetic, considers what keys you've pressed, decides what to do as a result, & coordinates the television picture. However, for all its cleverness, it could not do all this on its own. Left to itself, it knows nothing about BASIC, floating point arithmetic, or televisions, & it has to get all its instructions from another chip, the ROM (Read Only Memory). The ROM is just a long list of instructions that make a complete computer program telling the processor what to do under all foreseeable circumstances. This program is written not in BASIC, but in what is called Z80 machine code, & takes the form of a long sequence of bytes. (Remember that a byte is just a number between 0 & 255.) Each byte has an address showing whereabouts in the ROM it is; the first one has address 0, the second has address 1, & so on up to 8191. That makes altogether 8192 = 8*1024 bytes, which is why this ZX81 BASIC is sometimes called an 8K BASIC. 1K is 1024, or 2^{10}.

Although there are similar chips in many different machines, this particular sequence of instructions is unique to the ZX81 & was written specially for it by a small firm of Cambridge mathematicians.

You can see what byte is at a given address using the function **PEEK**. For example, this program prints out the first 21 bytes in the ROM (& their addresses).

```
10 PRINT "ADDRESS";TAB 8;"BYTE"
20 FOR A = 0 TO 20
30 PRINT A;TAB 8;PEEK A
40 NEXT A
```

The next chip to consider is the memory, or RAM (Random Access Memory) chip. This is where the processor stores the information that it wants to keep - your BASIC program, variables, the picture for the television, & various notes (the system variables) of what sort of state the computer is in.


RAM (note pad)
CPU (brains)
SCL (dogsbody)
UHF/VHF modulator
Voltage regulator
ROM (manual)

Like the ROM, the memory is arranged into bytes, each with an address: the addresses range from 16384 to 17407 (or possibly up to 32767 if you have a 16K RAM pack.) Again as with the ROM you can find the values of these bytes using **PEEK**, but the big difference from the ROM is that you can also change them. (The ROM, of course, is fixed & unalterable.)

Type

POKE 17300,57

This makes the byte at address 17300 have the value 57. If you now type

PRINT PEEK 17300

you get your number 57 back. (Try poking in other values, to prove that there's no cheating.)

Note that the address has to be between 0 & 65535; & most of these will refer to bytes in ROM or nowhere at all, & so have no effect. The value must be between -255 & +255, & if it is negative it gets 256 added to it.

The ability to poke gives you immense power over the computer if you know how to use it; however, the necessary knowledge is rather more than can be imparted in an introductory manual like this.

The last big chip is the logic chip, or SCL (Sinclair Computer Logic) chip. Again, this was specially designed & made for the ZX81, & it connects the other chips together in rather a clever way to make them do more than they normally would.

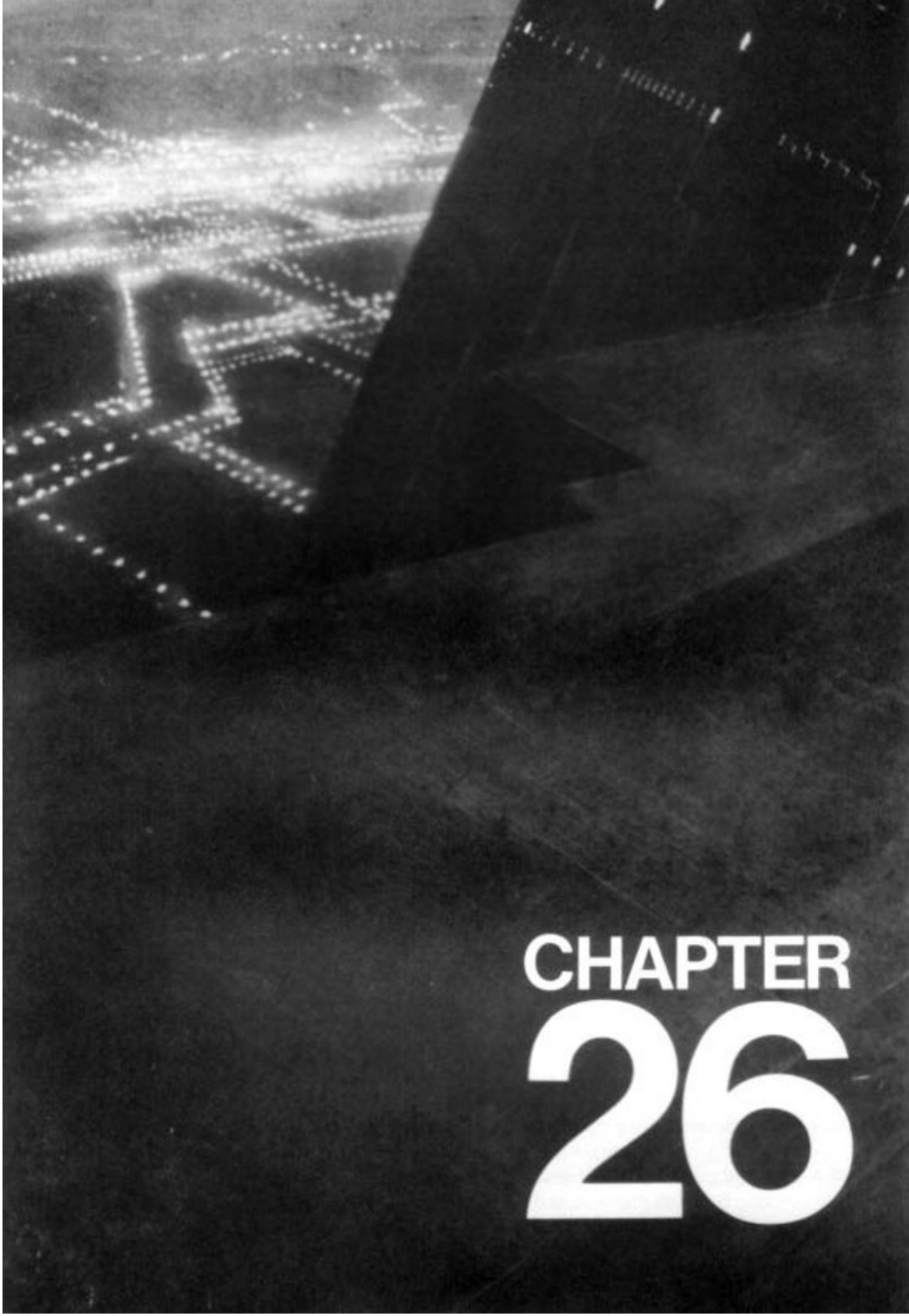
The modulator converts the computer's television output into a form suitable for the television, & the regulator converts the smoothed, but unregulated 9 volts of the power supply to a regulated 5 volts.

Summary

Chips

Statements: **POKE**

Functions: **PEEK**



CHAPTER 26

Using machine code

This chapter is written for those that understand Z80 machine code, the set of instructions that the Z80 processor chip uses. If you do not, but you would like to, there are books about it; two introductory ones are 'Programming the Z80' by Rodnay Zaks, published by Sybex at about £10 and 'Z80 and 8080 Assembly language programming' by Rathe Spracklen, published by Hayden at £5.25.

The ultimate authority is the 'Z80 Assembly Language Programming Manual' together with the 'Z80-CPU, Z80A-CPU Technical Manual', published by Zilog at about £6 for the pair, but these could hardly be recommended for beginners.

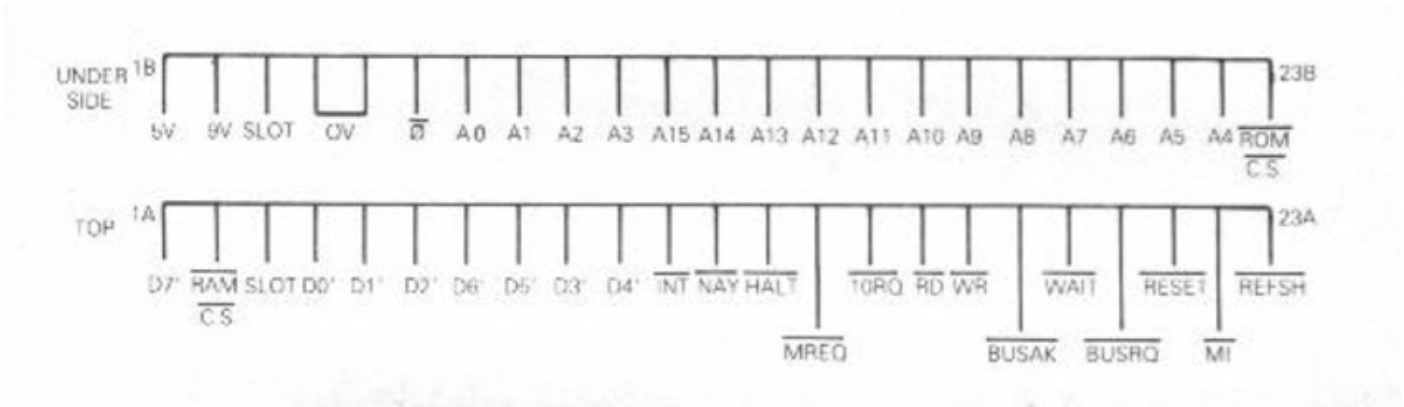
Machine code routines can be executed from within a BASIC program using the function **USR**. The argument of **USR** is the starting address of the routine, and its result is a two byte unsigned integer, the contents of the bc register pair on return. The return address to the BASIC is tacked in the usual way, so return is by a Z80 ret instruction.

There are certain restrictions on **USR** routines:

(i) On return, the iy & i registers must have the values 4000h & 1 Eh.

(ii) The display routine uses the a', f', ix, iy & r registers, so a **USR** routine should not use these if compute & display is operating. (It is not even safe to read the af' pair.)

The control, data & address busses are all exposed at the back of the ZX81, so you can do almost anything with a ZX81 that you can with a Z80. The ZX81 hardware might sometimes get in the way, though, especially in compute and display. Here is a diagram of the exposed connections at the back.



A piece of machine code in the middle of memory runs the risk of being overwritten by the BASIC system. Some safer places are

(i) In a **REM** statement: type in a **REM** statement with enough characters to hold your machine code, which you then poke in. Make this the first line in the program, or it might move about. Avoid halt instructions, since these will be recognized as the end of the **REM** statement.

(ii) In a string: set up a long enough string, and then assign a machine code byte to each character. Strings are always liable to move about in the memory.

In appendix A, the character set, you will find the characters and Z80 instructions written down side by side in order, & you may well find this useful when entering code.

(iii) At the top of the memory. When the ZX81 is switched on, it tests to see how much memory there is and puts the machine stack right at the top so that there is no space for **USR** routines there. It stores the address of the first non-existent byte (e.g. 17K, or 17408, if you have 1K of memory) in a system variable known as RAMTOP, in the two bytes with addresses 16388 & 16389. **NEW** on the other hand, does not do a full memory test, but only checks up as far as just before the address in RAMTOP. Thus if you poke the address of an existing byte into RAMTOP, for **NEW** all the memory from that byte on is outside the BASIC system and left alone. For instance, suppose you have 1K of memory and you have just switched on the computer.

PRINT PEEK 16388+256***PEEK** 16389

tells you the address (17408) of the first non-existent byte.

Now suppose you have a **USR** routine 20 bytes long. You want to change RAMTOP to 17388 = 236 + 256*67 (how would you work this out on the computer?), so type

POKE 16388,236

POKE 16389,67

and then **NEW**. The twenty bytes of memory from address 17388 to 17407 are now yours to do what you like with. If you then type **NEW** again it will not affect these twenty bytes.

The top of memory is a good place for **USR** routines, safe (even from **NEW**) and immobile. Its main disadvantage is that it is not saved by **SAVE**.

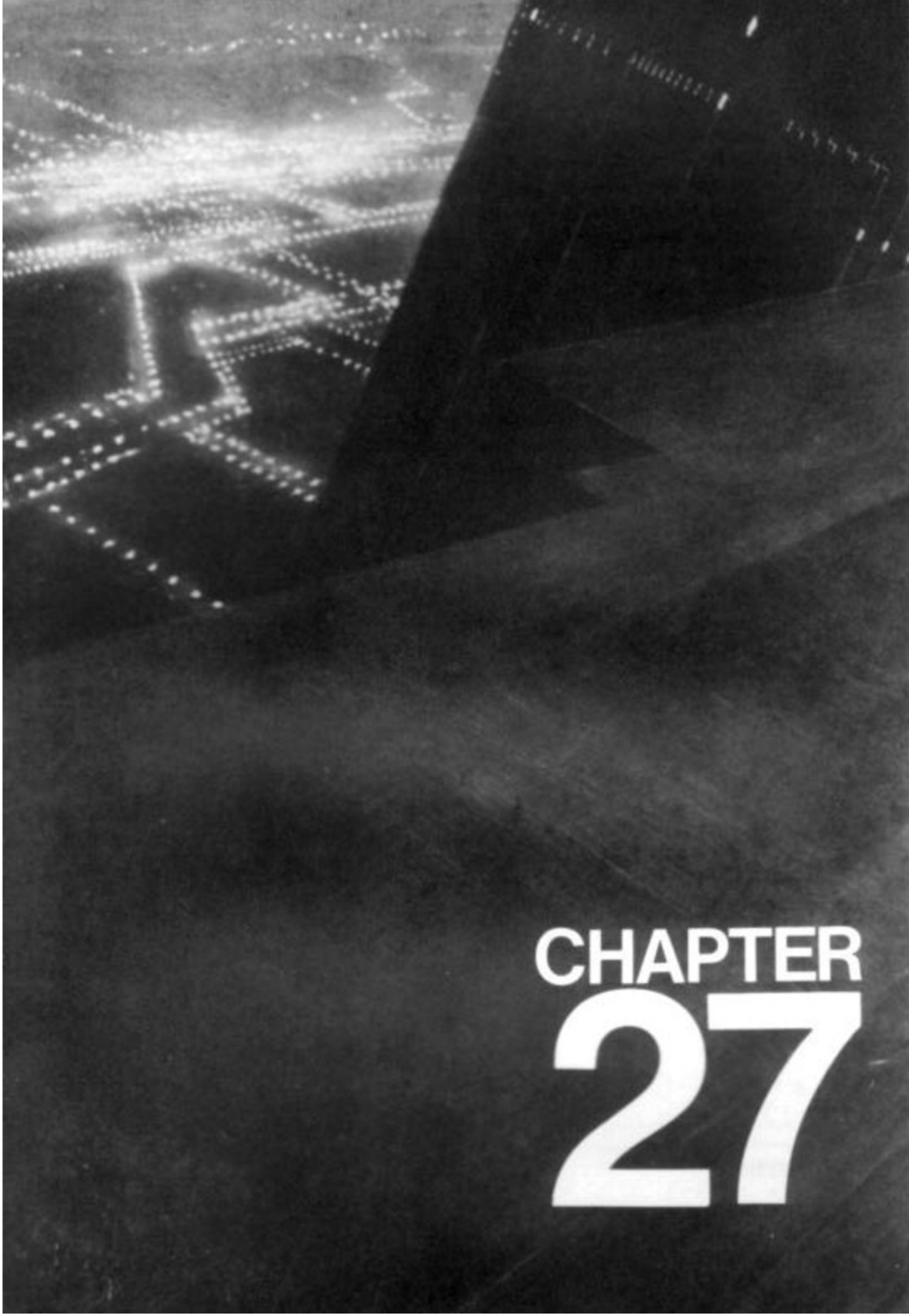
Summary

Functions: **USR**

Statements: **NEW**

Exercises

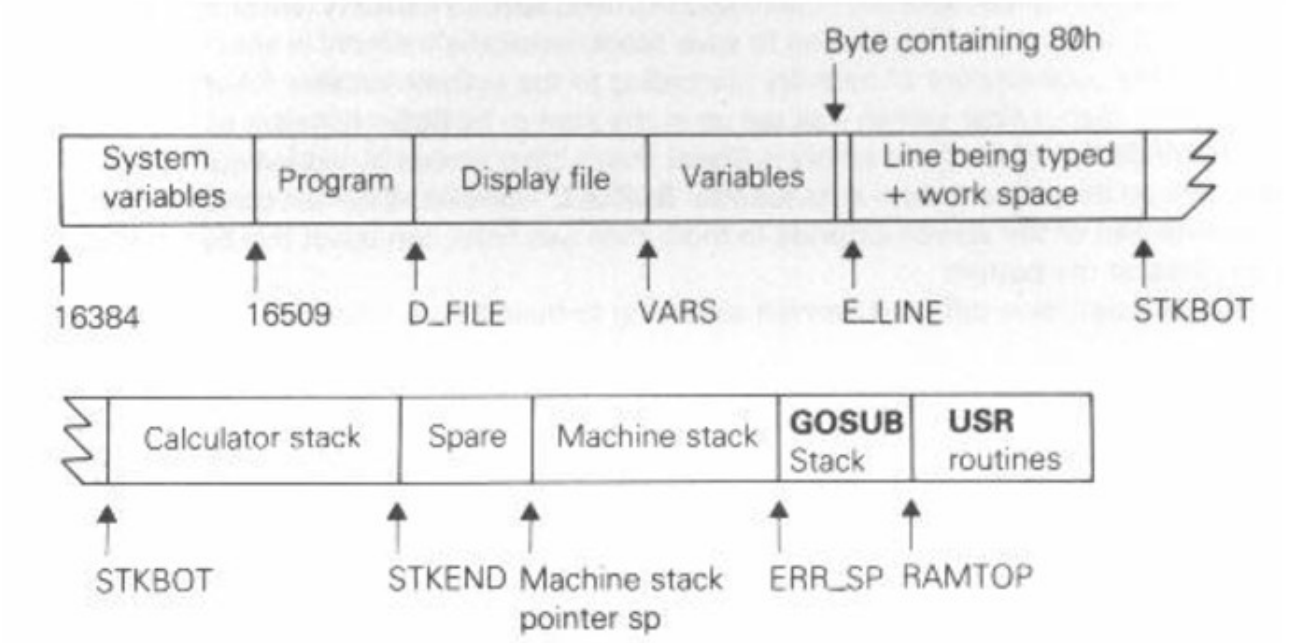
1. Make RAMTOP equal to 16700 and then execute **NEW**. You will get an idea of what happens when the memory gets full.



CHAPTER 27

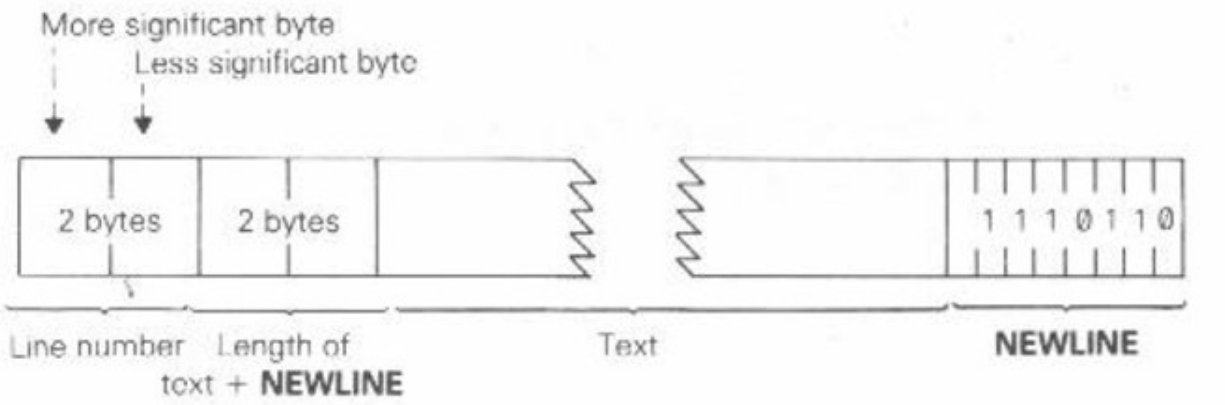
Organization of memory

The memory is sorted into different areas for storing different kinds of information. The areas are only large enough for the information that they actually contain, & if you insert some more at a given point (for instance by adding a program line or variable) space is made by shifting up everything above that point. Conversely, if you delete information then everything above it is shifted down.



The system variables contain various pieces of information that tell the computer what sort of state the computer is in. They are listed fully in the next chapter, but for the moment note that there are some (called D_FILE, VARS, E_LINE & so on) that contain the addresses of the boundaries between the various areas in the memory. These are not BASIC variables, & their names will not be recognized by the computer.

Each line in the program is stored as:



Note that, in contrast with all other cases of two-byte numbers in the Z80, the line number here (& also in a **FOR-NEXT** control variable) is stored with its most significant byte first: that is to say, in the order that you would write them down in.

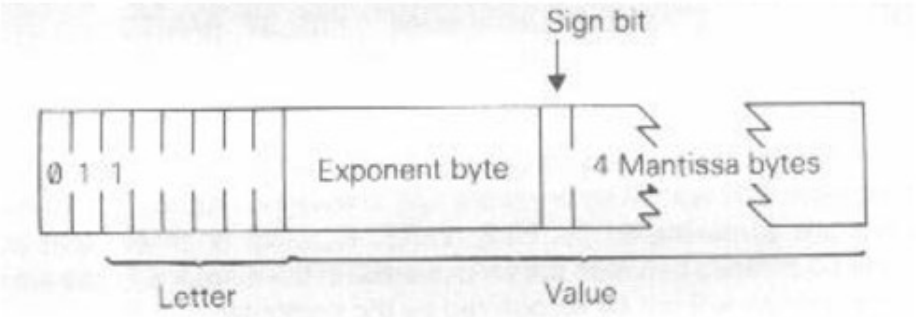
A numerical constant in the program is followed by its binary form, using the character **CHR$** 126 followed by five bytes for the number itself.

The display file is the memory copy of the television picture. It begins with a **NEWLINE** character, & then has the twenty four lines of text, each finishing with a **NEWLINE**. The system is so designed that a line of text does not need space a full thirty two characters: final spaces can be omitted. This is used to save space when the memory is small.

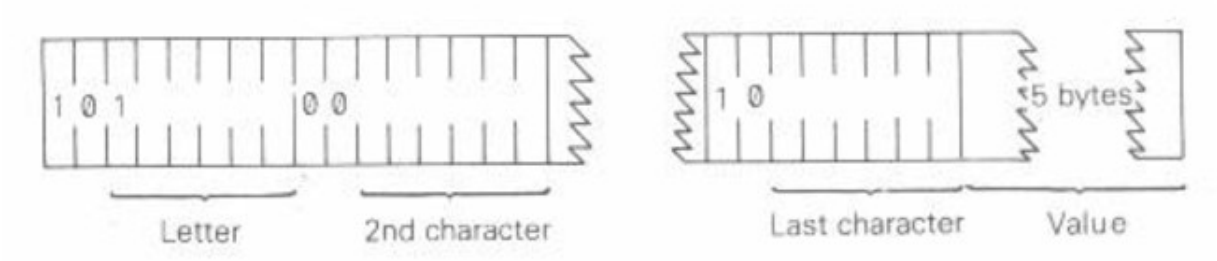
When the total amount of memory (according to the system variable RAMTOP) is less than 3 1/4 K, then a clear screen - as set up at the start or by **CLS** - consists of just twenty five **NEWLINE**s. When the memory is bigger than a clear screen is padded out with 24*32 spaces & on the whole it stays at its full size; **SCROLL**, however, & certain conditions where the lower part of the screen expands to more than two lines, can upset this by introducing short lines at the bottom.

The variables have different formats according to their different natures.

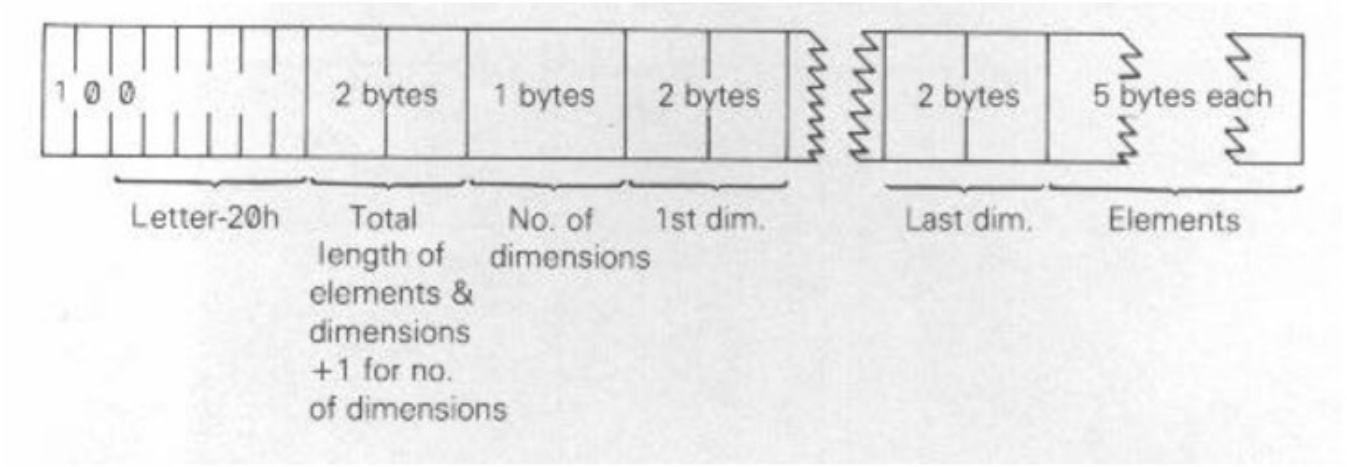
Number whose name is one letter only:



Number whose name is longer than one letter:



Array of numbers:



The order of the elements is:

first, the elements for which the first subscript is 1

next, the elements for which the first subscript is 2

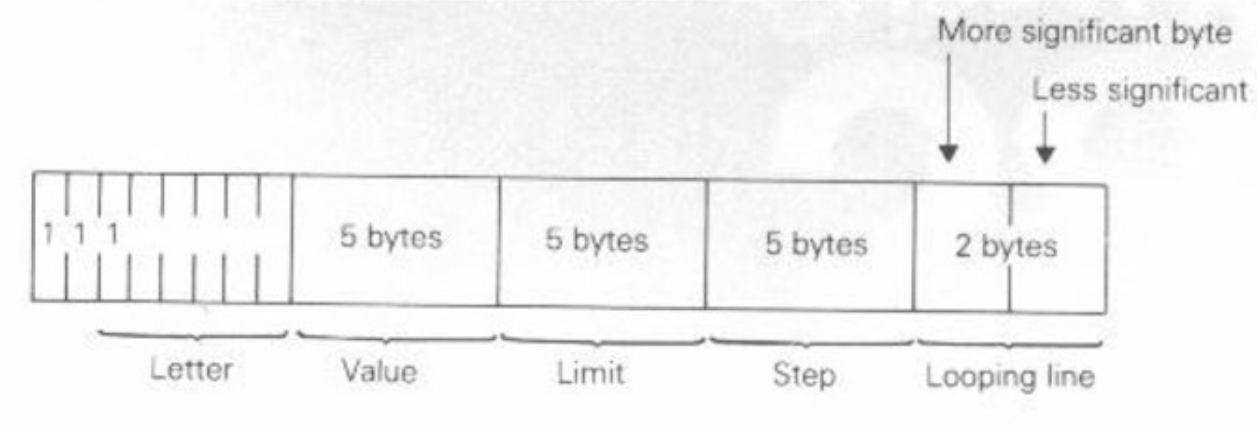
next, the elements for which the first subscript is 3

& so on for all possible values of the first subscript.

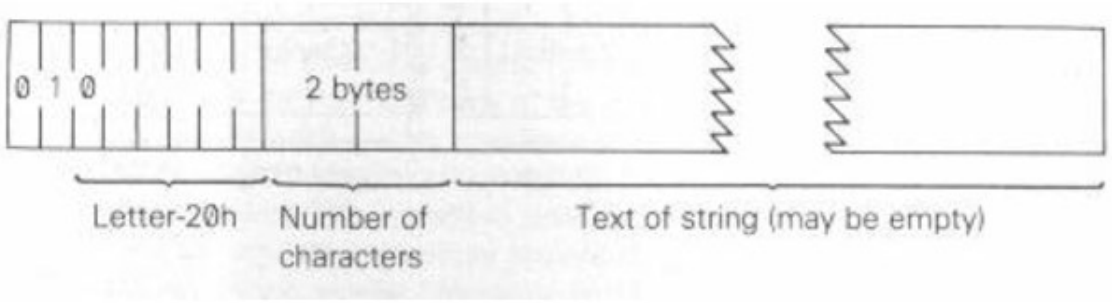
The elements with a given first subscript are ordered in the same way using the second subscript, & so on down to the last.

As an example, the elements of the 3 x 6 array B in chapter 22 are stored in the order B(1,1),B(1,2),B(1,3),B(1,4),B(1,5),B(1,6),B(2,1),B(2,2),...,B(2,6),B(3,1),B(3,2),...,B(3,6).

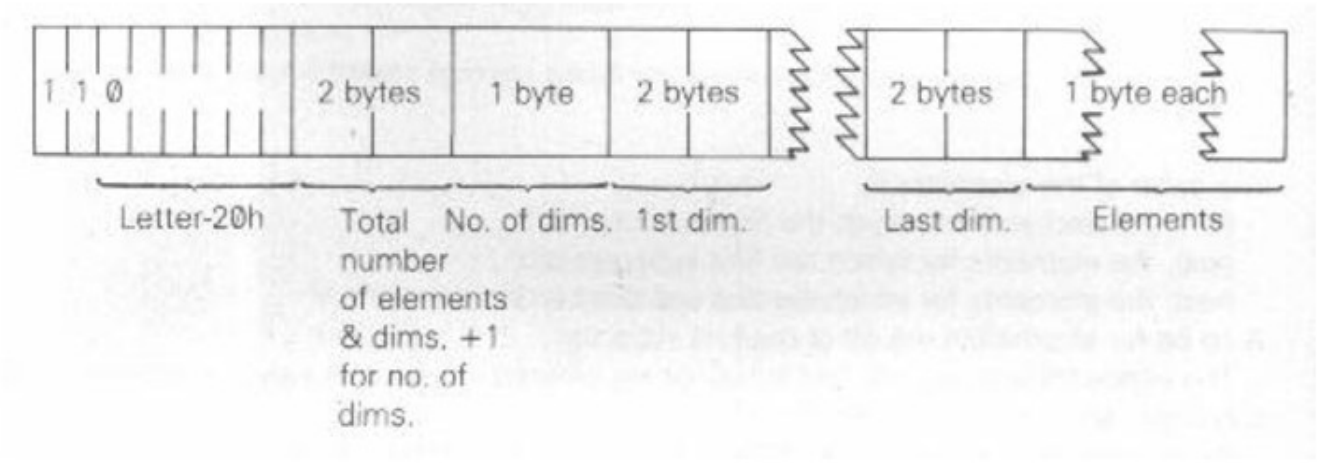
Control variable of a **FOR-NEXT** loop:



String:



Array of characters:



The part string at E_LINE contains the line being typed (as a command, a program line, or **INPUT** data) & also some work space.

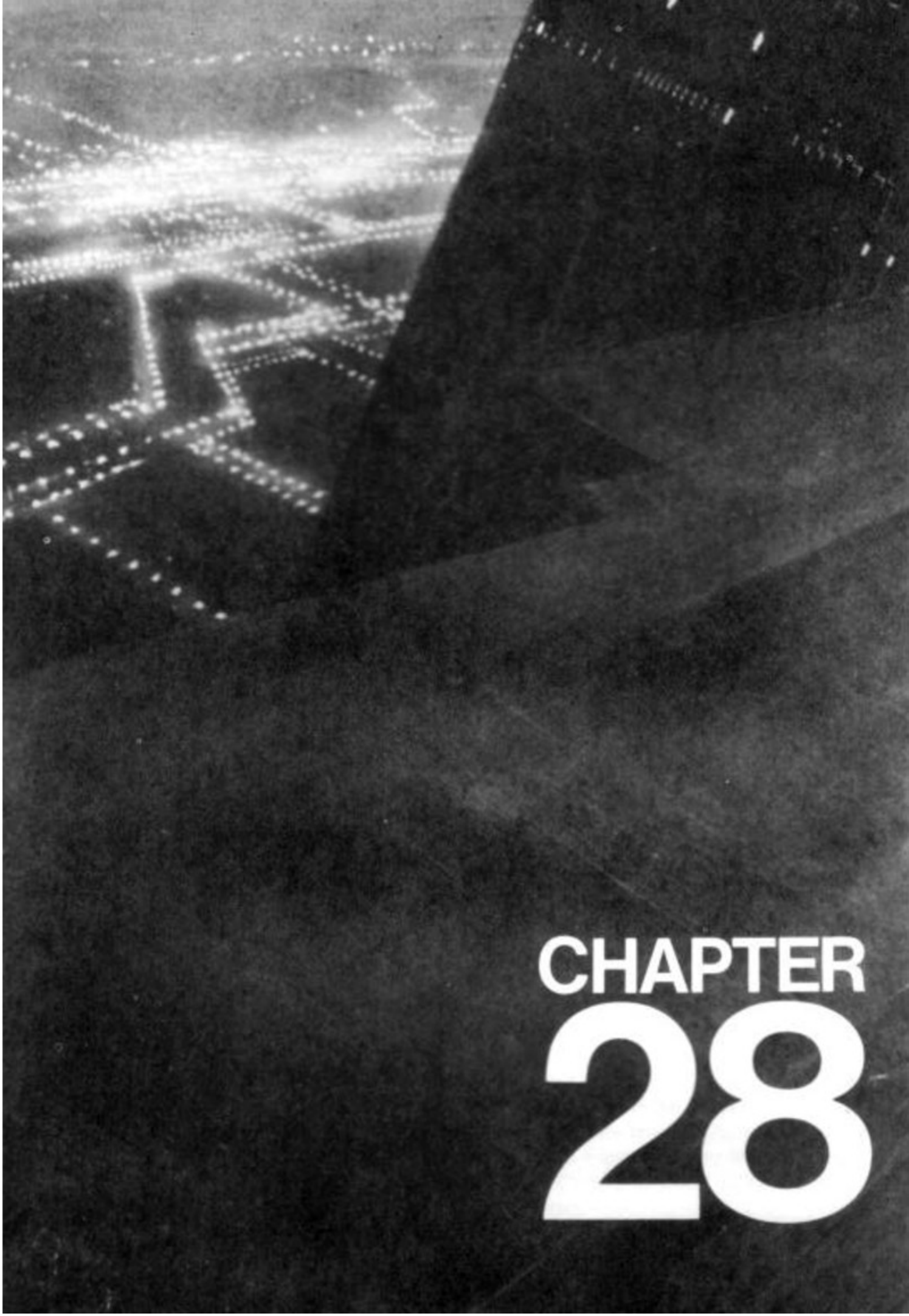
The calculator is the part of the BASIC system that deals with arithmetic, & the numbers on which it is operating are held mostly in the calculator stack.

The spare part contains the space so far unused.

The machine stack is the stack used by the Z80 chip to hold return addresses & so on.

The **GOSUB** stack was mentioned in chapter 14.

The space for **USR** routines has to be set aside by you, using **NEW** as described in the last chapter.



CHAPTER 28

The system variables

The bytes in memory from 16384 to 16508 are set aside for specific uses by the system. You can peek them to find out various things about the system, & some of them can be usefully poked. They are listed here with their uses.

These are called system variables, & have names, but do not confuse them with the variables used by BASIC. The computer will not recognize the names as referring to system variables, & they are given solely as mnemonics for you humans.

The abbreviations in column 1 have the following meanings:

X The variable should not be poked because the system might crash.

N Poking the variable will have no lasting effect.

S The variable is saved by **SAVE**.

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the less significant byte - the reverse of what you might expect. So to poke a value v to a two-byte variable at address n, use

POKE n,v-256***INT** (v/256)

POKE n+1,**INT** (v/256)

& to peek its value, use the expression

PEEK n + 256***PEEK** (n+1)

Notes	*Address*	*Name*	*Contents*
1	16384	ERR_NR	1 less than the report code. Starts off at 255 (for - 1), so **PEEK** 16384, if it works at all, gives 255. **POKE** 16384,n can be used to force an error halt: 0 ≤ n ≤ 14 gives one of the usual reports, 15 ≤ n ≤ 34 or 99 ≤ n ≤ 127 gives a non-standard report, and 35 ≤ n ≤ 98 is liable to mess up the display file.
X1	16385	FLAGS	Various flags to control the BASIC system.
X2	16386	ERR_SP	Address of first item on machine stack (after **GOSUB** returns).
2	16388	RAMTOP	Address of first byte above BASIC system area. You can poke this to make NEW reserve space above that area (see chapter 26) or to fool CLS into setting up a minimal display file (chapter 27). Poking RAMTOP has no effect until one of these two is executed.

Notes	*Address*	*Name*	*Contents*
N1	16390	MODE	Specified K, L, F or G cursor.
N2	16391	PPC	Line number of statement currently being executed. Poking this has no lasting effect except in the last line of the program.
S1	16393	VERSN	0 Identifies ZX81 BASIC in saved programs.
S2	16394	E_PPC	Number of current line (with program cursor).
SX2	16396	D_FILE	See chapter 27.
S2	16398	DF_CC	Address of **PRINT** position in display file. Can be poked so that **PRINT** output is sent elsewhere.
SX2	16400	VARS	See chapter 27.
SN2	16402	DEST	Address of variable in assignment.
SX2	16404	E_LINE	See chapter 27.
SX2	16406	CH_ADD	Address of the next character to be interpreted: the character after the argument of **PEEK**, or the **NEWLINE** at the end of a **POKE** statement.
S2	16408	X_PTR	Address of the character preceding the **S** marker.
SX2	16410	STKBOT	See chapter 27.
SX2	16412	STKEND	See chapter 27.
SN1	16414	BERG	Calculator's b register.
SN2	16415	MEM	Address of area used for calculator's memory. (Usually MEMBOT, but not always.)
S1	16417	not used	
SX1	16418	DF_SZ	The number of lines (including one blank line) in the lower part of the screen.
S2	16419	S_TOP	The number of the top program line in automatic listings.
SN2	16421	LAST_K	Shows which keys pressed.
SN1	16423		Debounce status of keyboard.
SN1	16424	MARGIN	Number of blank lines above or below picture: 55 in Britain, 31 in America.
SX2	16425	NXTLIN	Address of next program line to be executed.
S2	16427	OLDPPC	Line number of which **CONT** jumps.
SN1	16429	FLAGX	Various flags.
SN2	16430	STRLEN	Length of string type destination in assignment.
SN2	16432	T_ADDR	Address of next item in syntax table (very unlikely to be useful).
S2	16434	SEED	The seed for **RND**. This is the variable that is set by **RAND**.

Notes	*Address*	*Name*	*Contents*
S2	16436	FRAMES	Counts the frames displayed on the television. Bit 15 is 1. Bits 0 to 14 are decremented for each frame set to the television. This can be used for timing, but **PAUSE** also uses it. **PAUSE** resets to 0 bit 15, & puts in bits 0 to 14 the length of the pause. When these have been counted down to zero, the pause stops. If the pause stops because of a key depression, bit 15 is set to 1 again.
S1	16438	COORDS	x-coordinate of last point **PLOT**ted.
S1	16439		y-coordinate of last point **PLOT**ted.
S1	16440	PR_CC	Less significant byte of address of next position for **LPRINT** to print as (in PRBUFF).
SX1	16441	S_POSN	Column number for **PRINT** position.
SX1	16442		Line number for **PRINT** position.
S1	16443	CDFLAG	Various flags. Bit 7 is on (1) during compute & display mode.
S33	16444	PRBUFF	Printer buffer (33rd character is **NEWLINE**).
SN30	16477	MEMBOT	Calculator's memory area; used to store numbers that cannot conveniently be put on the calculator stack.
S2	16507	not used	

Exercises

1. Try this program

```
10 FOR N=0 TO 21
20 PRINT PEEK (PEEK 16400+256*PEEK 16401+N)
30 NEXT N
```

This tells you the first 22 bytes of the variables area: try to match up the control variable N with the description in chapter 27.

2. In the program above, change line 20 to

```
20 PRINT PEEK (16509+N)
```

This tells you the fist 22 bytes of the program area. Match these up with the program itself.

The character set

This is the complete ZX80 character set, with codes in decimal & hex. If one imagines the codes as being Z80 machine code instructions, then the right hand columns give the corresponding assembly language mnemonics. As you are probably aware if you understand these things, certain Z80 instructions are compounds starting with CBh or EDh; the two right hand columns give these.

Code	*Character*	*Hex*	*Z80 assembler*	*- after CBh*
0	space	00	nop	rlc b
1		01	ld bc,NN	rlc c
2		02	ld (bc),a	rlc d
3		03	inc bc	rlc e
4		04	inc b	rlc h
5		05	dec b	rlc l
6		06	ld b,N	rlc (hl)
7		07	rlca	rlc a
8		08	ex a,af'	rrc b
9		09	add hl,bc	rrc c
10		0A	ld a,(bc)	rrc d
11	"	0B	dec bc	rrc e
12	£	0C	inc c	rrc h
13	$	0D	dec c	rrc l
14	:	0E	ld c,N	rrc (hl)
15	?	0F	rcca	rrc a
16	(	10	djnz DIS	rl b
17	)	11	ld de,NN	rl c
18	>	12	ld (de),a	rl d
19	<	13	inc de	rl e
20	=	14	inc d	rl h
21	+	15	dec d	rl l
22	-	16	ld d,N	rl (hl)
23	*	17	rla	rl a
24	/	18	jr DIS	rr b
25	;	19	add hl,de	rr c
26	,	1A	la a,(de)	rr d
27	.	1B	dec de	rr e
28	0	1C	inc e	rr h
29	1	1D	dec e	rr l
30	2	1E	ld e,N	rr (hl)
31	3	1F	rra	rr a

Code	*Character*	*Hex*	*Z80 assembler*	*-after CBh*	*-after EDh*
32	4	20	jr nz,DIS	sla b	
33	5	21	ld hl,N	sla c	
34	6	22	ld (NN),hl	sla d	
35	7	23	inc hl	sla e	
36	8	24	inc h	sla h	
37	9	25	dec h	sla l	
38	A	26	ld h,N	sla (hl)	
39	B	27	daa	la a	
40	C	28	jr z,DIS	sra b	
41	D	29	add hl,hl	sra c	
42	E	2A	ld hl,(NN)	sra d	
43	F	2B	dec hl	sra e	
44	G	2C	inc l	sra h	
45	H	2D	dec l	sra l	
46	I	2E	ld l,N	sra (hl)	
47	J	2F	cpl	sra a	
48	K	30	jr nc,DIS		
49	L	31	ld sp,NN		
50	M	32	ld (NN),a		
51	N	33	inc sp		
52	O	34	inc (hl)		
53	P	35	dec(hl)		
54	Q	36	ld (hl),N		
55	R	37	scf		
56	S	38	jr c,DIS	srl b	
57	T	39	add hl,sp	srl c	
58	U	3A	ld a,(NN)	srl d	
59	V	3B	dec sp	srl e	
60	W	3C	inc a	srl h	
61	X	3D	dec a	srl l	
62	Y	3E	ld a,N	srl (hl)	
63	Z	3F	ccf	srl a	
64	**RND**	40	ld b,b	bit 0,b	in b,(c)
65	**INKEY$**	41	ld b,c	bit 0,c	out (c),b
66	**PI**	42	ld b,d	bit 0,d	sbc hl,bc
67	not used	43	ld b,e	bit 0,e	ld (NN),bc
68	not used	44	ld b,h	bit 0,h	neg
69	not used	45	ld b,l	bit 0,l	retn
70	not used	46	ld b,(hl)	bit 0,(hl)	im 0

Code	*Character*	*Hex*	*Z80 assembler*	*-after CDh*	*-after EDh*
71	not used	47	ld b,a	bit 0,a	ld i,a
72	not used	48	ld c,b	bit 1,b	i c,(c)
73	not used	49	ld c,c	bit 1,c	out (c),c
74	not used	4A	ld c,d	bit 1,d	adc hl,bc
75	not used	4B	ld c,e	bit 1,e	ld bc,(NN)
76	not used	4C	ld c,h	bit 1,h	
77	not used	4D	ld c,l	bit 1,l	reti
78	not used	4E	ld c,(hl)	bit 1,(hl)	
79	not used	4F	ld c,a	bit 1,a	ld r,a
80	not used	50	ld d,b	bit 2,b	ld d,(c)
81	not used	51	ld d,c	bit 2,c	out (c),d
82	not used	52	ld d,d	bit 2,d	sbc hl,de
83	not used	53	ld d,e	bit 2,e	ld (NN),de
84	not used	54	ld d,h	bit 2,h	
85	not used	55	ld d,l	bit 2,l	
86	not used	56	ld d,(hl)	bit 2,(hl)	im 1
87	not used	57	ld d,a	bit 2,a	ld a,i
88	not used	58	ld e,b	bit 3,b	in e,(c)
89	not used	59	ld e,c	bit 3,c	out (c),e
90	not used	5A	ld e,d	bit 3,d	adc hl,de
91	not used	5B	ld e,e	bit 3,e	ld de,(NN)
92	not used	5C	ld e,h	bit 3,h	
93	not used	5D	ld e,l	bit 3,l	
94	not used	5E	ld e,(hl)	bit 3,(hl)	im 2
95	not used	5F	ld e,a	bit 3,a	ld a,r
96	not used	60	ld h,b	bit 4,b	in h,(c)
97	not used	61	ld h,c	bit 4,c	out (c),h
98	not used	62	ld h,d	bit 4,d	sbc hl,hl
99	not used	63	ld h,e	bit 4,e	ld (NN),hl
100	not used	64	ld h,h	bit 4,h	
101	not used	65	ld h,l	bit 4,l	
102	not used	66	ld h,(hl)	bit 4,(hl)	
103	not used	67	ld h,a	bit 4,a	rrd
104	not used	68	ld l,b	bit 5,b	in l,(c)
105	not used	69	ld l,c	bit 5,c	out (c),l
106	not used	6A	ld l,d	bit 5,d	adc hl,hl
107	not used	6B	ld l,e	bit 5,e	ld de.(NN)
108	not used	6C	ld l,h	bit 5,h	
109	not used	6D	ld l,l	bit 5,l	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CBh*	*-after EDh*
110	not used	6E	ld l,(hl)	bit 5,(hl)	
111	not used	6F	ld l,a	bit 5,a	rld
112	cursor up ⇧	70	ld (hl),b	bit 6,b	
113	cursor down ⇩	71	ld (hl),c	bit 6,c	
114	cursor left ⇦	72	ld (hl),d	bit 6,d	sbc hl,sp
115	cursor right ⇨	73	ld (hl),e	bit 6,e	ld (NN),sp
116	**GRAPHICS**	74	ld (hl),h	bit 6,h	
117	**EDIT**	75	ld (hl),l	bit 6,l	
118	**NEWLINE**	76	halt	bit 6,(hl)	
119	**RUBOUT**	77	ld (hl),a	bit 6,a	
120	**K**/**L** mode	78	ld a,b	bit 7,b	in a,(c)
121	**FUNCTION**	79	ld a,c	bit 7,c	out (c),a
122	not used	7A	ld a,d	bit 7,d	adc hl,sp
123	not used	7B	ld a,e	bit 7,e	ld sp,(NN)
124	not used	7C	ld a,h	bit 7,h	
125	not used	7D	ld a,l	bit 7,l	
126	number	7E	la d,(hl)	bit 7,(hl)	
127	cursor	7F	ld a,a	bit 7,a	
128		80	add a,b	res 0,b	
129		81	add a,c	res 0,c	
130		82	add a,d	res 0,d	
131		83	add a,e	res 0,e	
132		84	add a,h	res 0,h	
133		85	add a,l	res 0,l	
134		86	add a,(hl)	res 0,(hl)	
135		87	add a,a	res 0,a	
136		88	add a,b	res 1,b	
137		89	add a,c	res 1,c	
138		8A	add a,d	res 1,d	
139	inverse "	8B	adc a,e	res 1,e	
140	inverse £	8C	adc a,h	res 1,h	
141	inverse $	8D	adc a,l	res 1,l	
142	inverse :	8E	adc a,(hl)	res 1,(hl)	
143	inverse ?	8F	adc a,a	res 1,a	
144	inverse (	90	sub b	res 2,b	
145	inverse)	91	aub c	res 2,c	
146	inverse >	92	sub d	res 2,d	
147	inverse <	93	sub e	res 2,e	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CBh*	*-after EDh*
148	inverse =	94	sub h	res 2,h	
149	inverse +	95	sub l	res 2,l	
150	inverse -	96	sub (hl)	res 2,(hl)	
151	inverse *	97	sub a	res 2,a	
152	inverse /	98	sbc a,b	res 3,b	
153	inverse ;	99	sbc a,c	res 3,c	
154	inverse ,	9A	sbc a,d	res 3,d	
155	inverse .	9B	abc a,e	res 3,e	
156	inverse 0	9C	abc a,h	res 3,h	
157	inverse 1	9D	abc a,l	res 3,l	
158	inverse 2	9E	abc a,(hl)	res 3,(hl)	
159	inverse 3	9F	abc a,a	res 3,a	
160	inverse 4	A0	and b	res 4,b	ldi
161	inverse 5	A1	and c	res 4,c	cpi
162	inverse 6	A2	and d	res 4,d	ini
163	inverse 7	A3	and e	res 4,e	outi
164	inverse 8	A4	and h	res 4,h	
165	inverse 9	A5	and l	res 4,l	
166	inverse A	A6	and (hl)	res 4,(hl)	
167	inverse B	A7	and a	res 4,a	
168	inverse C	A8	xor b	res 5,b	ldd
169	inverse D	A9	xor c	res 5,c	cpd
170	inverse E	AA	xor d	res 5,d	ind
171	inverse F	AB	xor e	res 5,e	outd
172	inverse G	AC	xor h	res 5,h	
173	inverse H	AD	xor l	res 5,l	
174	inverse I	AE	xor (hl)	res 5,(hl)	
175	inverse J	AF	xor a	res 5,a	
176	inverse K	B0	or b	res 6,b	ldir
177	inverse L	B1	or c	res 6,c	cpir
178	inverse M	B2	or d	res 6,d	inir
179	inverse N	B3	or e	res 6,e	otir
180	inverse O	B4	or h	res 6,h	
181	inverse P	B5	or l	res 6,l	
182	inverse Q	B6	or (hl)	res 6,(hl)	
183	inverse R	B7	or a	res 6,a	
184	inverse S	B8	cp b	res 7,b	lddr
185	inverse T	B9	cp c	res 7,c	cpdr
186	inverse U	BA	cp d	res 7,d	indr

Code	*Character*	*Hex*	*Z80 assembler*	*-after CBh*	*-after EDh*
187	inverse V	BB	cp e	res 7,e	otdr
188	inverse W	BC	cp h	res 7,h	
189	inverse X	BD	cp l	res 7,l	
190	inverse Y	BE	cp (hl)	res 7,(hl)	
191	inverse Z	BF	cp a	res 7,a	
192	"""	C0	ret nz	set 0,b	
193	**AT**	C1	pop bc	set 0,c	
194	**TAB**	C2	jp nz,NN	set 0,d	
195	not used	C3	jp NN	set 0,e	
196	**CODE**	C4	call nz,NN	set 0,h	
197	**VAL**	C5	push bc	sel 0,l	
198	**LEN**	C6	add a,N	set 0,(hl)	
199	**SIN**	C7	rst 0	set 0,a	
200	**COS**	C8	ret z	set 1,b	
201	**TAN**	C9	ret	set 1,c	
202	**ASN**	CA	jp z,NN	set 1,d	
203	**ACS**	CB		set 1,e	
204	**ATN**	CC	call z,NN	set 1,h	
205	**LN**	CD	call NN	set 1,l	
206	**EXP**	CE	adc a,N	set 1,(hl)	
207	**INT**	CF	rst 8	set 1,a	
208	**SQR**	D0	ret nc	set 2,b	
209	**SGN**	D1	pop de	set 2,c	
210	**ABS**	D2	jp nc,NN	set 2,d	
211	**PEEK**	D3	out N,a	set 2,e	
212	**USR**	D4	call nc,NN	set 2,h	
213	**STR$**	D5	push de	set 2,l	
214	**CHR$**	D6	sub N	set 2,(hl)	
215	**NOT**	D7	rst 16	set 2,a	
216	**	D8	ret c	set 3,b	
217	**OR**	D9	exx	set 3,c	
218	**AND**	DA	jp c,NN	set 3,d	
219	**<=**	DB	in a,N	set 3,e	
220	**>=**	DC	call c,NN	set 3,h	
221	**<>**	DD	prefixes instructions using ix	set 3,l	
222	**THEN**	DE	sbc a,N	set 3,(hl)	
223	**TO**	DF	rst 24	set 3,a	
224	**STEP**	E0	ret po	set 4,b	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CBh*
225	**LPRINT**	E1	pop hl	set 4,c
226	**LLIST**	E2	jp po,NN	set 4,d
227	**STOP**	E3	ex (sp),hl	set 4,e
228	**SLOW**	E4	call po,NN	set 4,h
229	**FAST**	E5	push hl	set 4,l
230	**NEW**	E6	and N	set 4,(hl)
231	**SCROLL**	E7	rst 32	set 4,a
232	**CONT**	E8	ret pe	set 5,b
233	**DIM**	E9	jp (hl)	set 5,c
234	**REM**	EA	jp pe,NN	set 5,d
235	**FOR**	EB	ex de,hl	set 5,e
236	**GOTO**	EC	call pe,NN	set 5,h
237	**GOSUB**	ED		set 5,l
238	**INPUT**	EE	xor N	set 5,(hl)
239	**LOAD**	EF	rst 40	set 5,a
240	**LIST**	F0	ret p	set 6,b
241	**LET**	F1	pop af	set 6,c
242	**PAUSE**	F2	jp p,NN	set 6,d
243	**NEXT**	F3	di	set 6,e
244	**POKE**	F4	call p,NN	set 6,h
245	**PRINT**	F5	push af	set 6,l
246	**PLOT**	F6	or N	set 6,(hl)
247	**RUN**	F7	rst 48	set 6,a
248	**SAVE**	F8	ret m	set 7,b
249	**RAND**	F9	ld sp,hl	set 7,c
250	**IF**	FA	jp m,NN	set 7,d
251	**CLS**	FB	ei	set 7,e
252	**UNPLOT**	FC	call m,NN	set 7,h
253	**CLEAR**	FD	prefixes instructions using iy	set 7,l
254	**RETURN**	FE	cp N	set 7,(hl)
255	**COPY**	FF	rst 56	set 7,a

Report codes

This table gives each report code with a general description & a list of the situations where it can occur. In appendix C, a more detailed description of what error reports mean is given under each statement or function.

Code	*Meaning*	*Situations*
0	Successful completion, or jump to line number bigger than any existing. A report with code 0 does not change the line number used by **CONT**.	Any
1	The control variable does not exist (has not been set up by a **FOR** statement) but there is an ordinary variable with the same name.	**NEXT**
2	An undefined variable has been used. For a simple variable this will happen if the variable is used before it has been assigned to in a **LET** statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a **DIM** statement. For a control variable this will happen if the variable is used before it has been set up as a control variable in a **FOR** statement, when there is no ordinary simple variable with the same name.	Any
3	Subscript out of range. If the subscript is hopelessly out of range (negative, or bigger than 65535) then error B will result.	Subscripted variables
4	Not enough room in memory. Note that the line number in the report (after the /) may be incomplete on the screen, because of the shortage of memory: for instance 4/20 may appear as 4/2. See chapter 23. For **GOSUB** see exercise 6 of chapter 14.	**LET**, **INPUT**, **DIM**, **PRINT**, **LIST**, **PLOT**, **UNPLOT**, **FOR**, **GOSUB**. Sometimes during function evaluation.
5	No more room on the screen. **CONT** will make room by clearing the screen.	**PRINT**, **LIST**.

Code	*Meaning*	*Situations*
6	Arithmetic overflow: calculations have led to a number greater than about 10^{38}.	Any arithmetic
7	No corresponding **GOSUB** for a **RETURN** statement.	**RETURN**
8	You have attempted **INPUT** as a command (not allowed).	**INPUT**
9	**STOP** statement executed. **CONT** will not try to re-execute the **STOP** statement.	**STOP**
A	Invalid argument to certain functions.	**SQR**, **LN**, **ASN**, **ACS**
B	Integer out of range. When an integer is required, the floating point argument is rounded to the nearest integer. If this is outside a suitable range then error B results. For array access, see also report 3.	**RUN**, **RAND**, **POKE**, **DIM**, **GOTO**, **GOSUB**, **LIST**, **LLIST**, **PAUSE**, **PLOT**, **UNPLOT**, **CHR$**, **PEEK**, **USR** Array access
C	The text of the (string) argument of **VAL** does not form a valid numerical expression.	**VAL**
D	(i) Program interrupted by **BREAK**. (ii) The **INPUT** line starts with **STOP**.	At end of any statement, or in **LOAD**, **SAVE**, **LPRINT**, **LLIST** or **COPY**. **INPUT**
E	Not used	
F	The program name provided is the empty string.	**SAVE**

The ZX81 for those that understand BASIC

General

If you already know BASIC then you should not have much trouble using the ZX81; but it has one or two idiosyncrasies.

(i) Words are not spelled out, but have keys of their own - this is dealt with in chapter 2 (for keywords & shifted keys) & chapter 5 (for function names). In the text, these words are printed in **BOLD TYPE**.

(ii) ZX81 BASIC lacks READ, DATA & RESTORE (but see exercise 3 of chapter 22 concerning this), user-defined functions (FN & DEF; but **VAL** can sometimes be used), & multi-statement lines.

(iii) The string handling facilities are comprehensive but non-standard - see chapter 21, & also chapter 22 (for string arrays).

(iv) The ZX81 character set is completely its own.

(v) The television display is not in general memory-mapped.

(vi) If you are accustomed to using **PEEK** & **POKE** on a different machine, remember that all the addresses will be different on the ZX81.

Speed

The machine works at two speeds, called compute & display mode, & fast mode.

In compute & display, the television picture is generated continuously & computing is done during the blank parts at the top & bottom.

In fast mode, the television picture is turned off during computing, & is only displayed at the end of the program, while waiting for **INPUT** data, or during a pause (see **PAUSE**).

Fast mode runs about four times as fast, & should be used for programs with a lot of computing as opposed to output, or when typing in long programs.

Switching between speeds is done with the **FAST** & **SLOW** statements (q.v.).

The keyboard

ZX81 characters comprise not only the single symbols (letters, digits, etc.), but also the compound tokens (keywords, function names, etc.; these are printed here in **BOLD TYPE**) & all these are entered from the keyboard rather than being spelled out. To fit this in, some keys have up to five distinct meanings, given partly by shifting the keys (i.e. pressing the **SHIFT** key at the same time as the required one) & partly by having the machine in different modes.

The mode is indicated by the cursor, an inverse video letter that shows where the next character from the keyboard will be inserted.

K mode (for keywords) occurs automatically when the machine is expecting a command or program line (rather than **INPUT** data), & from its position on the line it knows it should expect a line number or a keyword. This is at the beginning of the line, or just after some digits at the beginning of the line, or just after **THEN**. If unshifted, the next key will be interpreted as either a keyword (these are mostly written above the keys), or a digit.

L mode (for letters) normally occurs at all other times. If unshifted, the next key will be interpreted as the main symbol on that key.

In both **K** & **L** modes, shifted keys will be interpreted as the subsidiary red character in the top right-hand corner of the key.

F mode (for functions) occurs after **FUNCTION** (shifted **NEWLINE**) is pressed, & lasts for one key depression only. That key will be interpreted as a function name, these appearing under the keys.

G mode for graphics occurs after **GRAPHICS** (shifted 9) is pressed, and lasts until it is pressed again. An unshifted key will give the inverse video of its **L** mode interpretation; a shifted key will as well, provided that it is a symbol, but if the shifted key would normally give a token, in graphics mode it gives the graphics symbol that appears in the bottom right hand corner of the key.

The screen

This has 24 lines, each 32 characters long, and is divided into two parts. The top part is at most 22 lines, and displays either a listing or program output. The bottom part, at least two lines, is used for inputting commands, program lines and **INPUT** data, and also for displaying reports.

Keyboard input: this appears in the bottom half of the screen as it is typed, each character (single symbol or compound token) being inserted just before the cursor. The cursor can be moved left with ⇦ (shifted 5) or right with ⇨ (shifted 8). The character before the cursor can be deleted with **RUBOUT** (shifted 0). (Note:- the whole line can be deleted by typing **EDIT** (shifted 1) followed by **NEWLINE**; or, if it is **INPUT** data, just by typing **EDIT**.)

When **NEWLINE** is pressed, the line is executed, entered into the program, or used as INPUT data as appropriate, unless it contains a syntax error. In this case the symbol **S** appears just before the error.

As program lines are entered, a listing is displayed in the top half of the screen. The manner in which the listing is produced is rather complicated, and explained more fully in chapter 9, exercise 6. The last line to be entered is called the current line and indicated by the symbol **>**, but this can be changed using the keys ⇩ (shifted 6) and ⇧ (shifted 7). If **EDIT** (shifted 1) is pressed, the current line is brought down to the bottom part of the screen and can be edited.

When a command is executed or a program run, the screen is first of all cleared, & then output is displayed in the top half of the screen and remains until a program line is entered, or **NEWLINE** is pressed with an empty line, or ⇩ or ⇧ is pressed. In the bottom part appears a report of the form m/n where m is a code showing what happened (see appendix B), & n is the number of the last line executed - or 0 for a command. The report remains until a key is pressed (and indicates **K** mode).

In certain circumstances, the **SPACE** key acts as a **BREAK**, stopping the computer with report D. This is recognized

(i) at the end of a statement while a program is running,

(ii) while the computer is looking for a program on tape,

or (iii) while the computer is using the printer (or by accident trying to use it when it is not there).

The BASIC

Numbers are stored to an accuracy of 9 or 10 digits. The largest number you can get is about 10^{38}, & the smallest (positive) number is about $4 * 10^{-39}$.

A number is stored in the ZX81 in floating point binary with one exponent byte e ($1 \leq e \leq 255$), & four mantissa bytes m ($\frac{1}{2} \leq m \leq 1$). This represents the number $m * 2^{e-128}$.

Since $\frac{1}{2} \leq m \leq 1$, the most significant bit of the mantissa m is always 1. Therefore in actual fact we can replace it with a bit to show the sign - 0 for positive numbers, 1 for negative.

Zero has a special representation in which all 5 bytes are 0.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters and digits. All these are significant, so that for instance LONGNAME & LONGNAMETOO are distinct names. Spaces are ignored.

Control variables for **FOR**-**NEXT** loops have names a single letter long.

Numeric arrays have names a single letter long, which may be the same as the name of a simple variable. They may have arbitrarily many dimensions of arbitrary size. Subscripts start at 1.

Strings are completely flexible in length. The name of a string consists of a single letter followed by $.

String arrays can have arbitrarily many dimensions of arbitrary size. The name is a single letter followed by $ and may not be the same as the name of a string. All the strings in a given array have the same fixed length, which is specified as an extra, final dimension in the **DIM** statement. Subscripts start at 1.

Slicing: Substrings of strings may be specified using slicers.

A slicer can be

(i) empty

or

(ii) numerical expression

or

(iii) optional numerical expression **TO** optional numerical expression

& is used in expressing a substring either by

(a) string expression (slicer)

or by

(b) string array variable (subscript,...,subscript, slicer)

which means the same as

string array variable (subscript,...,subscript)(slicer)

In (a), suppose the string expression has the value s$.

If the slicer is empty, then the result is s$ considered as a substring of itself.

If the slicer is a numerical expression with value m, then the result is the mth character of s$ (a substring of length 1).

If the slicer has the form (iii), then suppose the first numerical expression has the value m (the default value is 1), & the second, n (the default value is the length of s$).

If $1 \leq m \leq n \leq$ the length of s$ then the result is the substring of s$ starting with the mth character & ending with the nth.

If $0 \leq n \leq m$ then the result is the empty string.

Otherwise, error 3 results.

Slicing is performed before functions or operations are evaluated, unless brackets dictate otherwise.

Substrings can be assigned to (see **LET**).

The argument of a function does not need brackets if it is a constant or a (possibly subscripted or sliced) variable.

Function	**Type of operand** (x)	**Result**
ABS	number	Absolute magnitude.
ACS	number	Arccosine in radians. Error A if x not in the range -1 to +1.
AND	binary operation, right operand always a number	Numeric left operand: A **AND** B = A if B ≠ 0, 0 if B = 0 String left operand: A$ **AND** B = A$ if B ≠ 0, "" if B = 0
ASN	number	Arcsine in radians. Error A if x not in the range -1 to +1.

Function	Type of operand	Result
ATN	number	Arctangent in radians.
CHR$	number	The character whose code is x, rounded to the nearest integer. Error B if x not in the range 0 to 255.
CODE	string	The code of the first character in x (or 0 if x is the empty string).
COS	number (in radians)	Cosine
EXP	number	e^x
INKEY$	none	Reads the keyboard. The result is the character representing (in **L** mode) the key pressed if there is exactly one, else the empty string.
INT	number	Integer part (always rounds down).
LEN	string	Length
LN	number	Natural logarithm (to base e) Error A if $x \leq 0$
NOT	number	0 if $x \neq 0$, 1 if $x = 0$. **NOT** has priority 4.
OR	binary operation, both operands numbers	A **OR** B = 1 if $B \neq 0$, A if B = 0 **OR** has priority 2.
PEEK	number	The value of the byte in memory whose address is x (rounded to the nearest integer). Error B if x not in the range 0 to 65535.
PI	none	π (3.14159265..)
RND	none	The next pseudo-random number y in a sequence generated by taking the powers of 75 modulo 65537, subtracting 1 & dividing by 65536. $0 \leq y \leq 1$.
SGN	number	Signum: the sign (-1, 0 or +1) of x.
SIN	number (in radians)	Sine
SQR	number	Square root. Error B if $x < 0$
STR$	number	The string of characters that would be displayed if x were printed.
TAN	number (in radians)	Tangent
USR	number	Calls the machine code subroutine whose starting address is x (rounded to the nearest integer). On return, the result is the contents of the bc register pair. Error B if x is not in the range 0 to 65535.
VAL	string	Evaluates x (without its bounding quotes) as a numerical expression. Error C if x contains a syntax error, or gives a string value. Other errors possible, depending on the expression.
-	number	Negation

The following are binary operations:

+	Additional (on numbers), or concatenation (on strings).
-	Subtraction
*	Multiplication
/	Division
**	Raising to a power. Error B if the left operand is negative.
=	Equals*
>	Greater than*
<	Less than*
<=	Less than or equal to*
>=	Greater than or equal to*
<>	Not equal to*

* Both operands must be of the same type. The result is a number, 1 if the comparison holds & 0 if it does not.

Functions & operations have the following priorities:

Operation	*Priority*
Subscripting & slicing	12
All functions except **NOT** and unary minus	11
**	10
Unary minus	9
* , /	8
+ , - (binary -)	6
= , > , < , **<= , >=, <>**	5
NOT	4
AND	3
OR	2

Statements

In this list,

α	represents a single letter
v	represents a variable
x,y,z	represent numerical expressions
m,n	represent numerical expressions that are rounded to the nearest integer
e	represents an expression
f	represents a string valued expression
s	represents a statement

Note that arbitrary expressions are allowed everywhere (except for the line numbers at the beginning of a statement).

All statements except **INPUT** can be used either as commands or in programs (although they may be more sensible in one than the other).

CLEAR Deletes all variables, freeing the space they occupied.

CLS (Clear Screen) Clears the display file. See chapter 27 concerning the display file.

CONT Suppose p/q was the last report with a non-zero. Then **CONT** has the effect **GOTO** q if p ≠ 9, **GOTO** q+1 if p = 9 (**STOP** statement)

COPY Sends a copy of the display to the printer, if attached; otherwise does nothing.
Unlike all other commands, a **COPY** command does not clear the screen first. There must be no spaces before **COPY**.
Report D if **BREAK** pressed.

DIM $\alpha(n_1,...,n_k)$ Deletes any array with the name α, & sets up an array á of numbers with k dimensions $n_1,...,n_k$. Initializes all the values to 0. Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

DIM $\alpha\$(n_1,...,n_k)$ Deletes any array or string with the name $\alpha\$$, & sets up an array á$ of characters with k dimensions $n_1,...,n_k$. Initializes all the values to "". This can be considered as an array of strings of fixed length n_k, with k-1 dimensions $n_1,...,n_{k-1}$.
Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

FAST Starts fast mode, in which the display file is displayed only at the end of the program, while **INPUT** data is being typed in, or during a pause.

FOR α=x **TO** y **FOR** α=x **TO** y **STEP** 1

FOR α=x **TO** y **STEP** z Deletes any simple variable á, & sets up a control variable with value x, limit y, step z, & looping address 1 more than the line number of the **FOR** statement (-1 if it is a command). Checks if the initial value is greater (if step ≥ 0) or less (if step < 0) than the limit, & if so then skips to statement **NEXT** á at the beginning of a line.

See **NEXT** α.
Error 4 occurs if there is no room for the control variable.

GOSUB n Pushes the line number of the **GOSUB** statement onto a stack; then as **GOTO** n
Error 4 can occur if there are not enough **RETURN**s.

GOTO n Jumps to line n (or, if there is none, the first line after that).

IF x **THEN** s If x is true (non-zero) then s is executed. The form '**IF** x **THEN** line number' is not allowed.

INPUT v
Stops (with no special prompt) & waits for the user to type in an expression; the value of this is assigned to v. In fast mode, the display file is displayed. **INPUT** cannot be used as a command; error 8 occurs if you try.
If the first character in the **INPUT** line is **STOP**, the program stops with report D.

LET v=e
Assigns the value of e to the variable v.
LET cannot be omitted.
A simple variable is undefined until it is assigned to in a **LET** or **INPUT** statement.
If v is a subscripted string variable, or a sliced string variable (substring), then the assignment is Procrustean: the string value of e is either truncated or filled out with spaces on the right, to make it the same length as the variable v.

LIST
LIST 0

LIST n
Lists the program to the television, starting at line n, & makes n the current line.
Error 4 or 5 if the listing is too long to fit on the screen; **CONT** will do exactly the same again.

LLIST
LLIST 0

LLIST n
Like **LIST**, but using the printer instead of the television.
Should do nothing if the printer is not attached.
Stops with report D if **BREAK** pressed.

LOAD f
Looks for a program called f on tape, & loads it & its variables.
If f = "", then loads the first program available.
If **BREAK** is pressed or a tape error is detected, then
(i) if no program has yet been read from tape, stops with report D & old program;
(ii) if part of a program has been read in, then executes **NEW**.

LPRINT ...
Like **PRINT**, but using the printer instead of the television. A line of text is sent to the printer.
(i) when printing spills over from one line to the next,
(ii) after an **LPRINT** statement that does not end in a comma or a semicolon,
(iii) when a comma or **TAB** item requires a new line, or
(iv) at the end of the program, if there is anything left unprinted.
In an **AT** item, only the column number has any effect; the line number is ignored (except that the same error conditions arise as for **PRINT** if it is out of range). An **AT** item never sends a line of text to the printer.
There should be no effect if the printer is absent. Stops with report D if **BREAK** is pressed.

NEW Starts the BASIC system off anew, deleting program & variables, & using the memory up to but not including the byte whose address is in the system variable RAMTOP (bytes 16388 & 16389).

NEXT α (i) Finds the control variable á.
(ii) Adds its step to its value.
(iii) If the step ≥ 0 & the value > the limit; or if the step < 0 & the value < the limit, then jumps to the looping line.
Error 1 if there is a simple variable α.
Error 2 if there is no simple or control variable α.

PAUSE n Stops computing & displays the display file for n frames (at 50 frames per second) or until a key is pressed.
$0 \leq n \leq 65535$, else error B. If $n \geq 32767$ then the pause is not timed, but lasts until a key is pressed.

PLOT m,n Blacks in the pixel ($|m|$, $|n|$); moves the **PRINT** position to just after that pixel.
$0 \leq |m| \leq 63$, $0 \leq |n| \leq 43$, else error B.

POKE m,n Writes the value n to the byte in store with address m.
$0 \leq m \leq 65535$, $-255 \leq n \leq 255$, else error B.

PRINT ... The '...' is a sequence of **PRINT** items, separated by commas or semicolons, & they are written to the display file for display on the television. The position (line & column) where the next character is to be printed is called the **PRINT** position.
A **PRINT** item can be
(i) empty, i.e. nothing
(ii) a numerical expression.
First, a minus sign is printed if the value is negative. Now let x be the modulus of the value.
If $x \leq 10^{-5}$ or $x \geq 10^{13}$, then it is printed using scientific notation. The mantissa part has up to eight digits (with no trailing zeros), & the decimal point (absent if only one digit) is after the first. The exponent part is E, followed by + or -, followed by one or two digits.
Otherwise x is printed in ordinary decimal notation with up to eight significant digits, & no trailing zeros after the decimal point. A decimal point right at the beginning is always followed by a zero, so for instance .03 & 0.3 are printed as such.
0 is printed as a single digit 0.

(iii) a string expression.
The tokens in the string are expanded, possibly with a space before or after.
The quote image character prints as ".
Unused characters & control characters print as ?.
(iv) AT m,n
The **PRINT** position is changed to line |m| (counting from the top, column n (counting from the left).
$0 \leq |m| \leq 21$, else error 5 if $|m| = 22$ or 23, error B otherwise.
$0 \leq |n| \leq 31$, else error B.
(v) **TAB** n
n is reduced modulo 32. Then, the **PRINT** position is moved to column n, staying on the same line unless this would involve backspacing, in which case it moves on to the next line.
$0 \leq n \leq 255$, else error B.
A semicolon between two items leaves the **PRINT** position unchanged, so that the second item follows on immediately after the first. A comma, on the other hand, moves the **PRINT** position on at least one place, & after that, however many as are necessary to leave it in column 0 or 16, throwing a new line if necessary.
At the end of the **PRINT** statement, if it does not end in a semicolon or comma, a new line is thrown.
Error 4 (out of memory) can occur with 3K or less of memory.
Error 5 means that the screen is filled.
In both cases, the cure is **CONT**, which will clear the screen & carry on.

RAND — **RAND** 0

RAND n — Sets the system variable (called SEED) used to generate the next value of **RND**. In $n \neq 0$, then SEED is given the value n; if $n = 0$ then it is given the value of another system variable (called FRAMES) that counts the frames so far displayed on the television, & so should be fairly random.
Error B occurs if n is not in the range 0 to 65535.

REM ... — No effect. '...' can be any sequence of characters except **NEWLINE**.

RETURN — Pops a line number from the **GOSUB** stack, & jumps to the line after it.
Error 7 occurs when there is no line number on the stack. There is some mistake in your program; **GOSUB**s are not properly balanced by **RETURN**s.

RUN — **RUN** 0

RUN n — **CLEAR**, & then **GOTO** n.

SAVE f — Records the program & variables on tape, & calls it f. **SAVE** should not be used inside a **GOSUB** routine.
Error F occurs if f is the empty string, which is not allowed.

SCROLL — Scrolls the display file up one line, losing the top line & making an empty line at the bottom.
NB the new line is genuinely empty with just a **NEWLINE** character & no spaces. (See chapter 27).

SLOW — Puts the computer into compute & display mode, in which the display file is displayed continuously, & computing is done during the spaces at the top & bottom of the picture.

STOP — Stops the program with report 9. **CONT** will resume with the following line.

UNPLOT m,n — Like **PLOT**, but blanks out a pixel instead of blacking it in.

Index

The index includes the keys on the keyboard & how to obtain them (the mode - **K**, **L**, **F** or **G** - & whether shifted or not), & their codes.

Usually an entry is referenced only once per chapter, so having found one reference, look through the rest of the chapter including the exercises.

A

ABS **F**, on G. Code 210. 31
accuracy 28
ACS **F**, on S. Code 203. Arccosine. 31
addition of strings 43
address 161
- of a byte 161
return address 93
algebra 37
ALGOL 21
alphabetical order 67, 80
AND **K** or **L**, shifted 2. Code 218. 68
antilog 28
APL 21
argument 31
arithmetic expression 26
array 143
ASN **F**, on A. Code 202. Arcsine. 31
assign 37
AT **F**, on C. Code 193. 44, 115, 133
ATN **F**, on D. Code 204. Arctangent. 31

B

bar chart 79
BASIC 21, 193
binary
- operation 26, 71
- system 155
bit 156
BOLD TYPE 14, 191
bracket 26
BREAK On **SPACE**. Only recognized as **BREAK** in certain situations. 60, 109
buffer 179

bug 101
byte 157

C

call 93
cassette recorder 8, 107
character 77
 - position 119
 - set 77, 181
 control character 79
CHR$ F, on U. Code 214. 77
CLEAR K, on X. Code 253. 38, 57
CLS K, on V. Code 251. 115
COBOL 21
CODE F, on I. Code 196. 77
code 77, 161, 167
 machine code 161, 167
command 13, 60
comparison 67
 - of numbers 67
 - of strings 67
compute & display 89
concatenation 43
condition 67
conditional expression 72
CONT K, on C. Code 232. 58
control
 - character 79
 - variable 84
coordinate 119
COPY K, on Z. Code 255. 58, 133
COS F, on W. Code 200. 31
CPU 161
current line 50, 62
cursor 14
 F cursor 31
 G cursor 77
 K cursor 14
 L cursor 14
 program cursor (>) 50

D

DATA 145
decimal system 155
degree 33
DIM **K**, on D. Code 233. 143
dimension 143
display file 149, 171

E

- in exponent part 26, 40
EDIT **K** or **L**, shifted 1. Code 117. 50
element 173
empty string 45, 77
entry point 97
execute 50
EXP **F**, on X. Code 206. 32, 40
exponent 26
- byte 172
- part 26
expression 26
arithmetic expression 26
conditional expression 72
logical expression 69
numeric expression 26
string expression 44

F

F mode 31
false 68
FAST **K** or **L**, shifted F. Code 229. 89
fast mode 89
floating point 28
flowchart 101
FOR **K**, on F. Code 235. 84
FORTRAN 21
FUNCTION **K** or **L**, shifted **NEWLINE**. Code 121. 31
function 31
- mode 31

G

G mode 77

glitch 111
GOSUB **K**, on H. Code 237. 93, 102
- stack 93
GOTO **K**, on G. Code 236. 57, 95
graph 119
graphics 77, 119
- symbol 77
- mode 77
GRAPHICS **G**, **K** or **L**, shifted 9. Code 116. 77, 119
grey characters 79

H

hex 155
hexadecimal 155

I

if 67
IF **K**, on U. Code 250. 67
initial value 84
INKEY$ **F**, on B. Code 65. 127
INPUT **K**, on I. Code 238. 57, 67
- mode 57
INT **F**, on R. Code 207. 32
integer 28
inverse 33
- functions 33
- video 77
item 43
PRINT item 43

J

jack plug 7

K

K mode 14
keyboard 17
keyword 13
- mode 14

L

L mode 14
LEFT$ 139
LEN K, on K. Code 198. 43
LET K, on L. Code 241. 37
letter mode 14
limit 84
line
- number 49, 119
current line 50, 62
looping line 84
program line 15, 49, 60
top line 62
LIST K, on K. Code 240. 52, 61
listing 50, 61
LLIST K or L, shifted G. Code 226. 133
LN F, on Z. Code 205. 31
LOAD K, on J. Code 239. 108
logic
- chip 161
- al operation 68
logs 28, 31
Loop 83
- ing line 84
LPRINT K or L, shifted S. Code 225. 133

M

machine code 161, 167
main program 95
mantissa 28, 172
memory 150
- expansion board 149
menu 111
MID$ 139
mode 14
command mode 14
compute & display mode 89
fast mode 89
function mode (F) 31
graphics mode (G) 77

INPUT mode 57
keyword mode (K) 14
letter mode (L) 14
Modulo 34, 116
Modulus 31

N

Name
- of a variable 37
- of a program 107
Nesting 85
NEW K, on A. Code 230. 57, 168
NEWLINE Code 118. 14, 25
NEXT K, on N. Code 243. 84
NOT F, on N. Code 215. 31, 68
nullary operation 32
null string 45
Numeric
- expression 26
- variable 37

O

Octal 157
ON 73
on-line 21
operand 25
operation 25
binary operation 26, 71
logical operation 68
nullary operation 32
unary operation 26
OR K or L, shifted W. Code 217. 68

P

PASCAL 21
PAUSE K, on M. Code 242. 127
PEEK F, on O. Code 211. 161
PI F, on M. π. Code 66. 32
pixel 119
PL-1 21
PLOT K, on Q. Code 246. 44, 119

POKE K, on O. Code 244. 127, 163
POP-2 21
position 119
 LPRINT position 134
 PRINT position 115
power 25
 - supply 7
PRINT K, on P. Code 245. 13, 25, 115
 - item 43
 - position 115
printer 133
priority 26
processor 161
Procrustean assignment 138
program 49, 95
 - cursor 50
 - line 15, 49, 60
pseudorandom 32

Q

quote 43
 - image 45
 string quote 43, 59

R

radian 31
RAM 161
RAMTOP 168
RAND K, on T. Code 249. 32
random 32
READ 145
recursive 96
register 167
relation 67
REM K, on E. Code 234. 57, 102, 107
report 15, 103, 189
RESTORE 145
result 31
RETURN K, on Y. Code 254. 93
return address 93
RIGHT$ 139
RND F, on T. Code 64. 32

ROM		161
rounding	(inc. to nearest integer)	28, 34, 80
RUBOUT	**G**, **K** or **L**, shifted O. Code 119.	16, 77
RUN	**K**, on R. Code 247.	50, 59, 97

S

SAVE	**K**, on S. Code 248.	107
scientific notation		26
SCROLL	**K**, on B. Code 231.	115
SGN	**F**, on F. Code 209.	31
shift		14
- ed key		14
sign, signum		31
simple variable		143
SIN	**F**, on Q. Code 199.	31
slice		137
SLOW	**K** or **L**, shifted D. Code 228.	89
space		151
SQR	**F**, on H. Code 208.	32
stack		167
calculator stack		171
GOSUB stack		93, 171
machine stack		171
statement		13
STEP	**K** or **L**, shifted E. Code 224.	84
STOP	**K** or **L**, shifted A. Code 227.	57, 67
STR$	**F**, on Y. Code 213.	44
string		43
- addition		43
- expression		44
- quote		43, 59
- variable		43
subroutine		93
subscript		137, 143
- ed variable		143
substring		137
symbol		77
- graphics symbol		77
syntax		16
system variable		161, 177

T

TAB F, on P. Code 194. 115, 134
TAN F, on E. Code 201. 31
tape 8, 107
- recorder 107
- storage 8
television 67
THEN K or L, shifted 3. Code 222. 139
TL$ 84
TO K or L, shifted 4. Code 223. 77
token 62
top line 31
trigonometrical functions 68
true

U

unary operation 26
unless 72
UNPLOT K, on W. Code 252. 44, 119
USR F, on L. Code 212. 167

V

VAL F, on J. Code 197. 44
value 37
initial value 84
variable 37, 43, 84
control variable 84
numeric variable 37
simple variable 143
string variable 43
subscripted variable 143
system variable 161, 177

W

word 157

X

x-coordinate 119

Y

y-coordinate 119

Z

Zilog Z80 167
ZX80 127

. K or L. Code 27. Full stop or decimal point. 26
, K or L, shifted full stop. Code 26. Comma. 26
; K or L, shifted X. Code 25. Semicolon. 26
: K or L, shifted Z. Code 14. Colon.
? K or L, shifted C. Code 15. Question mark.
“ K or L, shifted P. Code 11. String quote. 43
“” K or L, shifted Q. Code 192. Quote image. 45
(K or L, shifted I. Code 16. Open bracket. 26
) K or L, shifted O. Code 17. Close bracket. 26
£ K or L, shifted space. Code 12. Pound. 95
$ K or L, shifted U. Code 13. Dollar. 43
+ K or L, shifted K. Code 21. Plus. 14
- K or L, shifted J. Code 22. Minus. 25
* K or L, shifted B. Code 23. Times. 25
/ K or L, shifted V. Code 24. Divide. 25
** K or L, shifted H. Code 216. To power. 25
= K or L, shifted L. Code 20. Equals. 49, 67
> K or L, shifted M. Code 18. Greater than. 67
< K or L, shifted N. Code 19. Less than. 67
<= K or L, shifted R. Code 219. Less than or equal to. 67
>= K or L, shifted Y. Code 220. Greater than or equal to. 67
<> K or L, shifted T. Code 221. Not equal to. 67
⇦ K or L, shifted 5. Code 114. Cursor left. 16
⇩ K or L, shifted 6. Code 113. Cursor down. 52
⇧ K or L, shifted 7. Code 112. Cursor up. 52
⇨ K or L, shifted 8. Code 115. Cursor right. 16

Printed at The Leagrave Press Limited, Luton & London

Sinclair Research Limited

6 King's Parade,
Cambridge CB2 1SN
England

This PDF version of

"**ZX81 BASIC Programming**" by Steven Vickers

was created by Robin Stuart (robin.stuart^nospam^@blueyonder.co.uk)

using Microsoft Word 2000 and Adobe Acrobat 4.0.

The HTML version of this document was produced with Microsoft FrontPage 2000.

The copyright for this work is owned by Amstrad plc (http://www.amstrad.com), who have given permission for it to be distributed freely on the Internet.

I would like to thank Russell Marks for his help with the HTML version of this book, which was used to construct this PDF file, and Peter Liebert-Adelt and Christopher Adams for their encouragement.

Here is a short list of URLs which you might find of interest:

ZX-TEAM Homepage – Peter Liebert-Adelt's ZX81 page:
http://home.t-online.de/home/p.liebert/f_zx81_e.htm

The ZX81 Web-Ring – For access to more ZX81 info:
http://home.t-online.de/home/sinclair_zx81/zx81_wbr.htm

The World of Spectrum – Tons of info and games for the ZX81's big brother:
http://www.worldofspectrum.org/

The BBC Lives! – Information and games for another classic 80's computer:
http://www.nvg.ntnu.no/bbc/