A Course in BASIC Programming

sinclair ZX80

ZX80 Operating Manual

sinclair

CONTENTS

©1980

CHAPTER

1

INTRODUCTION

INTRODUCTION - READ CHAPTER TWO FIRST!

This book is the user's manual for the ZX-80 personal computer. In it is everything you need to know in order to set up the ZX-80 hardware and to use the ZX-80 BASIC language for writing your own programs.

For most of the book the chapters are grouped in pairs. The first (odd-numbered) chapter of a pair is purely descriptive and prepares the reader for what is covered in the second (even-numbered) chapter of the pair. The second chapter of a pair is intended to take you through the use of ZX-80 BASIC as painlessly as possible while you're actually sitting at the keyboard.

If you are an experienced BASIC user, you won't need to read very much — probably only chapter *2*. the ZX-80 BASIC language summary, and the Index.

If you're less experienced and patient, it's probably best to read through the whole book, chapter by chapter. If you're less experienced but impatient you can skip the odd numbered chapters 3, 5, 7, and 9 (and probably will!).

OK. NOW READ CHAPTER TWO - AGAIN!

CHAPTER

2

GETTING STARTED

GETTING STARTED

This is the most important chapter in the book. It tells you how to connect up and switch on your ZX-80.

The ZX-80 consists of two units:

(1)	The computer
(2)	The power supply

The power supply used should be 9 volts DC @ 600 mA unregulated, terminated with a 3.5 mm jack plug. The tip of the jack plug must be positive.

DON'T PLUG IN TO THE MAINS SUPPLY YET

Look at the back of the ZX-80 computer. You will see four sockets. Three of these are 3.5 mm jack sockets marked MIC, EAR and POWER. The fourth is a phono socket for the video cable.

The connection diagram shows how to connect up the ZX-80 to the power supply and to a domestic television which will be used to display the output for the ZX-80.

A video cable with a coaxial plug at each end is provided to connect the TV to the computer.

A twin cable with 3.5 mm jack plugs at both ends is provided to connect the ZX-80 to a cassette recorder. Don't worry about this for the **moment.**

Connect the power supply to the ZX-80 by plugging the 3.5 mm jack plug into the socket marked 9V DC IN (if you do get it in the wrong jack socket you won't damage your ZX-80 even if you switch on the power. It won't work until you get the plug in the right socket, though!)

SEE DIAGRAM

Connect the TV using the coax lead provided. You will have to tune the TV to the ZX-80 frequency approximately channel 36 on UHF, so check that you can do this — most TV sets either have a continuously variable tuning control or, if they select channels with push-buttons, have separate tuning controls for each channel. If your set is a push-button set select an unused channel (ITV 2, perhaps). Turn down the volume control.

OK—we can't put off the moment of truth any longer!

Switch on the TV.

When it has warmed up ...
SWITCH ON THE ZX-80.

Connecting up the ZX80

You will be disappointed to see a horrible grey mess on the screen!!

Try tuning in the TV set. At some position of the tuning control the screen will suddenly clear. At the bottom left hand corner of the screen you will see a curious symbol — a black square with a white letter K in it.

If you can't see the K, turn the brightness control up until you can see it; you may find that adjusting the contrast control improves legibility. You may be planning to use your ZX-80 a lot. If so it may be worth considering buying a second-hand black-and-white TV to use with the ZX-80. They can be bought cheaply - in the UK, at least.

STORING PROGRAMS ON TAPE

Before you can store programs on tape you'll have to write a program. The rest of this book is all about writing programs, so the first thing to do is to read on until you've got a program to store. . .

. . . and now that you have we can start.

The twin coax cable with 3.5 mm jack plugs is used to connect the tape recorder to the ZX-80. Most cassette recorders have 3.5 mm jack sockets for MICROPHONE and EARPIECE. If you have one of those, just connect the 3.5 mm jack plugs as shown in the diagram. On other recorders DIN sockets are used. DIN plug to 3.5 mm jack plug connecting leads are available from most Hi-Fi shops. Consult the handbook to find out how to connect up the plug or plugs.

Once you have done this, connect up the cassette recorder to the ZX-80. Set the tone control (if any) to MAXIMUM. Some recorders have separate TREBLE and BASS control. In this case set TREBLE to MAXIMUM, BASS to MINIMUM. set the volume control to about 1/2 or 3/4 of MAXIMUM.

Have you got that program?

Unplug the MIC plug or DIN plug from the recorder and use the recorder's microphone to record the program's title; if you haven't recorded your voice saying the program's title, you will have trouble in finding programs when you've got a lot of them on tape. Stop the recorder. Reconnect the plug to the recorder. Get into command mode. Start recording and then type SAVE (E key) and NEWLINE.

The screen will go grey for about 5 seconds, then you'll see a series of horizontal streaks across the screen. After a few seconds the screen will clear and will show the program listing.

You will want to check that the program has been SAVED. Unplug the EAR plug. Rewind the tape until you get back to your voice. Now play back. You will hear your title, then you may (or may not) hear a short buzz followed by about 5 seconds of silence. You will then hear a peculiar sound, rather like a supercharged bumble bee. This is the program being played back via the loudspeaker. After a while the sound will change to a loud buzz. Rewind until you are at the beginning of the 5 second silence. Reconnect the EAR plug. Start playing back and then type LOAD (W key) NEWLINE. The screen will go grey or black and after a few seconds it may start looking grey but agitated. After a few more seconds the screen should clear to show a listing of the program.

Note: when you SAVE a program you also SAVE all the data and variables. You can avoid deleting these (RUN clears them) by using the command GO TO 1.

If it doesn't, use BREAK (SHIFT SPACE) to stop LOADING. If this doesn't work, unplug the power from the ZX-80 for a few seconds. Try repeating the procedure using different volume control settings. If this doesn't work, are you using the cassette recorder on mains

power? Try again using battery power. If this fails, check the wiring to MIC and EAR.

Good luck!

CHAPTER

3

NOT SO BASIC

NOT SO BASIC

Nearly all digital computers (such as the ZX-80) talk to themselves in binary codes. We talk to each other in English. This means that either we have to learn binary codes, or we have to teach the computer English. Binary codes are about as difficult to learn as Chinese, but it can be done. Forty years ago you <u>had</u> to learn them if you wanted to use a digital computer at all. But why should we do the work when we've got a computer to do it for us?

So we teach the computer English. There is-a snag, though. Computers, so far, aren't bright enough to learn English, something which a child of two does relatively easily. This is mainly because we have so many words and ways of saying things in English. A computer would have to learn every single one. We have to compromise. half-way between binary and English. Sometimes we compromise at a <u>low level</u> by using assembly <u>codes</u> which are more like binary than English. Sometimes we compromise at a <u>high level</u> by using languages such as ALGOL, PL/1, PASCAL, FORTRAN, and BASIC. These languages are much closer to English than binary. The ZX-80 uses BASIC because it is easy to learn and more than adequate for most purposes. BASIC stands for Beginner's All-purpose Symbolic Instruction Code and was devised at Dartmouth College, New Hampshire in 1964 as a simple beginners' programming language. Although it was intended just for beginners, it has since become one of the most widespread and popular high level languages. This isn't very surprising when you think about it, because even scientists and engineers prefer to concentrate on science and engineering instead of trying to talk to computers in complex and difficult languages.

Like English, BASIC has a variety of dialects, depending on which computer is being used. The ZX-80 BASIC differs from other BASIC'S in some respects. These differences are listed in the appendix "Summary of ZX-80 BASIC" at the end of this book.

As we said before, computers aren't that bright. We have to tell them exactly what to do by giving them a step-by-step list of instructions. This is called a program. Each instruction must be given in a clear and un-ambiguous way — the <u>syntax</u> must be right (in ZX-80 BASIC the computer will tell you if it thinks that the syntax is wrong <u>before</u> you can run the program. Some BASIC'S wait until you've tried to run the program before they say "nuts!"). Even if the syntax is right, the computer will be confused if you give it a badly thought out list of instructions. Chapters 5 and 7 give some ideas for making sure that you're telling the computer to do the right things in the right order.

STATEMENTS AND COMMANDS

ZX-80 BASIC allows you to use 22 instructions or statements.

These are divided into several categories.

1. System commands:

   NEW-   Clears ZX-80 ready for a new program

   RUN —   Runs the current program

   LIST —   Lists the current program

   LOAD —   Loads a program from tape

   SAVE —   Saves a program on tape

2. Control statements:
   GOTO
   IF . . .THEN . . .
   GOSUB
   STOP
   RETURN
   FOR ... TO
   NEXT
   CONTINUE

These statements allow the programmer to control the order in which instructions are carried out. They are described in detail in chapters 7-10.

3. Input/output statements;

   PRINT — allows computer to output data

   INPUT— allows user to enter data

I/O statements provide a means for getting data into and out from BASIC programs.

4. Assignment statement;
   LET ...=...
This versatile statement is used whenever an arithmetic operation is to be performed.

5. Other statements:
   CLEAR — clears the stored values of variables

   CLS —   clears the screen

   DIM —   sets the size of an array

   REM —   indicates that what follows is a remark

   RANDOMISE — sets up random number generator

   POKE —   allows user to talk to the computer in binary code

There are some normal BASIC statements which are not included in ZX-80 BASIC. These are:

READ

DATA

RESTORE

END

ON

Nearly everything that can be done by using these statements can be done in ZX-80 BASIC by using other statements. (The END statement is **not** needed at all in ZX-80 BASIC).

In a BASIC program all statements or lines are preceded by a <u>statement number</u> or <u>line number</u> which labels that particular line. Line numbers can vary between 1 and 9999. The computer carries out, or <u>executes,</u> statements in the order in which they are numbered. It makes life easier by always displaying programs in order of increasing statement numbers, so that the order in which the program is listed is also the order in which it would be executed.

VARIABLES

All the pieces of information stored in the computer for use in a program are labelled so that the computer can keep track of them. Each piece of information, or <u>variable,</u> has a name.

There are two sorts of variable:

1.   Numbers <u>(integer variables)</u> which can take any whole number value from —32768 to 32767 inclusive. Example: <u>142</u>

2.   <u>Strings</u> which can be any sequence of any characters (except ") of any length. Example: <u>"AROUND THE WORLD IN 80 DAYS"</u>

Integer variables have names which must always start with a letter and only contain letters and digits.

Integer variable names can be any length, so if you wish the name could be a mnemonic for the variable.

Some allowable variable names for integer variables are:

A   A2   AB   AB3   ANSWER   A4X

Y   Y8   YZ   YZ9   FREDBLOGS (N.B. no space)

AAAA   Z123ABC   QTOTAL

Some illegal names are:

4   4   4AD   (name <u>must</u> start with a letter)

FRED BLOGGS   (space not allowed)

A.B   (other characters not allowed)

FRED-BLOGGS   (not a variable, but two variables, one subtracted from the other)

String variables have names of the form: letter $

The dollar sign tells you (and the computer) that the variable is a string variable. Because only one letter is allowed, this means that you can only have up to 26 string variables in a program.

Some allowable names for string variables:

A$ P$ X$

Some illegal names:

2$ (name must start with letter)

A2$ (too many characters)

AC$ (too many characters)

All this may seem a little daunting at first. The good news is that, like so many things, it's easier than it sounds.

CHAPTER

4

TALKING TO THE ZX-80

The Keyboard

LET'S TALK

Set up the computer as described in chapter 2 and switch on. You will see a **[K]** sign appear at the bottom left hand comer of the screen. This is the <u>cursor</u> and shows you where you are on the screen.

THE KEYBOARD

If you look at the keyboard you will see that it looks quite like a typewriter keyboard but there are some differences. There is no carriage return key (instead there's a key called NEW LINE) and the space bar is replaced by a key labelled SPACE. All the letters are upper case (capitals) so that if you use the SHIFT key you get the symbol printed on the top right hand corner of the key. You'll also notice that quite a lot of keys have labels above them - such as NEW above the Q key. These will save you a lot of time from now on.

One vital point: it is very important to distinguish between the letter O and figure 0 or Zero. In the book we will use O to mean letter O and 0 to mean zero. On the screen the computer uses a square O [O] for letter O and a hexagonal 0 [0] for zero. (The reason that we've used 0 in the book is because it is easier for printing and because all other books on programming use it).

SINGLE-KEY KEYWORD SYSTEM

Hit the Q key. Amazing! The computer writes NEW on the screen. Now hit NEWLINE. You have cleared the computer ready to accept a new program. This illustrates the single-key keyword system. Most of the words you'll have to use with ZX-80 BASIC can be typed with only one keystroke in this way. In all this may save you up to 40% of the typing you'd otherwise have to do. Some BASIC'S allow you to use single keystrokes to input words like NEW, PRINT, RUN etc but only the ZX-80 prints out the word in full. This makes it much easier to keep track of what's going on in programs. You'll see how it works as you go along.

The labels above the numeral keys — NOT, AND etc — are not keywords. They are called <u>tokens</u> and, when needed, are obtained by using the SHIFT key. This also applies to EDIT on the NEWLINE key.

<u>Getting data in and out of the ZX-80 computer.</u>

The PRINT statement allows you to get data out of the computer. This can be done while a program is running — or afterwards.
Type exactly:

10"THIS IS A STRING"

using SHIFT Y for"

That should have come out as

10 PRINT "THIS IS A STRING"

The computer provided the word PRINT, as well as the space between 10 and PRINT. Did you notice what happened to the **[K]** cursor?. First of all, it followed what you were typing in, appearing just after the last character typed. After PRINT it changed to a **[L]** sign - this showed that the computer was not expecting any more single-key keywords just then. After you typed " you may have noticed that another character appeared as well — a **[S]** sign. What was actually on the screen was

10 PRINT "**[S]**

This **[S]** is the <u>syntax error marker</u> and occurs if there is a syntax error in the line being typed in. In this case it came up because the line needs the closing quotation marks if it is to be correct. The **[S]** followed the cursor as you typed in THIS IS A STRING and vanished when you typed the second ". Now erase the " by using the RUBOUT key. The **[S]** reappears. Try hitting the NEWLINE key — nothing happens because the computer won't accept a line containing a syntax error.

Type " and hit the NEWLINE key.
The computer accepts the line and displays it at the top of the screen together with a **[>]** cursor which shows that it is the last (and in this case the first) line entered. The **[>]** points to the <u>current line.</u>

Although there is only one line entered it is a program consisting of only one instruction. Type R (RUN) and NEWLINE and the program will run — after a brief flicker on the screen the words

THIS IS A STRING

appear at the top of the screen. At the bottom you'll see 0/10 which tells you that the program ran successfully with no errors.

Type any key to get back into <u>command mode.</u>

What the computer did was to print all the characters between the quotation marks. Any

character (except for quotation marks) is legal when PRINT is used in this way.

In general this form of the PRINT statement is useful for things like titles which only have to be printed once in a program. There are other forms which allow more complicated things to be done; you'll come across these the whole way through this book.

Now let's try getting some data into the computer. This is done by using the INPUT statement, which is basically of the form:

INPUT variable name.

Hit NEW (key **Q) and** NEWLINE. This tells **the** computer that a new program is being entered.
Type in the following program, using NEWLINE at the end of each line. (If you make a mistake, use SHIFT 0 (RUBOUT) to delete the mistake; every time you hit RUBOUT the character, keyword, or token to the left of the cursor is deleted).

10 PRINT "ENTER YOUR STRING"
20 INPUT A$ (A$ is the name of a string variable)
30 PRINT A$ (You can use PRINT to print out a variable)

Now run the program by typing RUN (key R) and NEWLINE.
The computer responds with

  ENTER YOUR STRING
  [L].

and waits. The "L" means that it is waiting for you to do something; in this case the quotation marks show that is is expecting you to enter a string variable.

Type in a short message or random string of letters (don't use quote marks) and press NEWLINE. The computer prints out what you have just put in, which was a string variable called A$.

Get back into command mode by pressing any key. Try running the program several times using more and more characters in your string. Keep an eye on the cursor. On the 15th line it will vanish. RUBOUT (shift 0) - the cursor will reappear. Now carry on adding characters — the display will gradually get SMALLER as you use up the storage capacity of the processor. Now hit NEWLINE—a 4/20 error message appears. The 4 tells you that the ZX-80 ran out of space to put the string variable into, and the 20 tells you that this happened at line 20 of the

program.

Get back into command mode.

You may have wondered why the line numbers have been 10, 20, 30 instead of. (say) 1,*2*. 3.

This is because you may want to add bits to the middle of your program.

For instance, enter

22 PRINT "ENTER A NUMBER"
24 INPUT A
40 PRINT A

The program displayed at the top of the screen now reads

10 PRINT "ENTER YOUR STRING"
20 INPUT A$
22 PRINT "ENTER A NUMBER"
24 INPUT A
30 PRINT A$
40 PRINT A

because the computer always sorts out the lines in numerical order. By leaving gaps between line numbers you can always do this. It also makes it easy to see how you've altered the program since the original lines will have line numbers in multiples of 10 (unless the lines were added at the end of the program, like statement 40). Now try running the program again.

FIRST it asks you for a string. Enter a string.

THEN it asks you for a number. Notice that when the ZX-80 is waiting for you to enter a number it displays **[L][S]** and not **"[L]".** Enter a number.

FINALLY it prints out what you put in.

In chapter 3 the difference between string variables and integer variables is mentioned. The program has two variables in it — a string variable A$ and an integer variable A. How does the computer react when you give it a different sort of variable to the one that it's expecting?

Easy enough to find out — just run the program again. When the computer asks you for a

string, give it a number. What happened?. It was accepted — numbers are acceptable inside strings. Now — when it asks you for a number, type in a letter. Nothing much happened except that 2/24 appeared at the lower left hand corner of the screen. This is another error code. The 2 stands for the type of error — VARIABLE NAME NOT FOUND - because the computer knew that the letter was the name of an integer variable and you hadn't assigned a value to it. The 24 stands for the line number at which the error was found.

EDITING PROGRAMS

Suppose that you want to alter something in a program that you've already entered. (This is called editing a program.)

It's easy on a ZX-80.

Get into command mode.

Look at the listing at the top of the screen. The **[>]** cursor (current line pointer) is at line 40. Now see what happens when you hit SHIFT 7 (⇑) a few times - the cursor moves up line by line. SHIFT 6 (⇓) moves it down line by line. These keys allow you to select any line in your program to edit.

A further key, HOME (SHIFT 9) sets the line pointer to line 0. Because there isn't a line 0 the cursor will vanish if you use HOME — but when you hit SHIFT 6 the cursor will jump to the next line — in this case line 10 — and reappear.

Select a line — line 22 for example.

Now hit SHIFT NEWLINE. Line 22 (or any other one you might have selected) appears at the bottom of the screen.

Try SHIFT 8 (⇒) — you'll see the cursor jump past PRINT. SHIFTS (⇐) moves the cursor left, to where it was before. Try moving the cursor right a few times — say to the position after ENTER. Now type a few A's — they are inserted into the string to the immediate left of the cursor. A few RUBOUTS and you can erase them again.

This editing facility is very useful if you want to alter individual characters within lines. However, if you want to delete a whole line it would be rather tedious to go through the procedure mentioned above. There is a simpler way to delete lines, though.

Type the line number, then NEWLINE — and you'll see the line vanish from the listing at the top of the screen. The cursor **[>]** will also vanish. It will not reappear until you type a new line, use the control keys ⇑ and ⇓, or use the LIST command.

THE LIST COMMAND

As you get more experienced with the ZX-80 you'll want to write programs longer than the 24 lines which will fit onto the screen. This could pose a problem — or could it? We can find out by writing a really long program.

Type NEW then NEWLINE.

Enter the following program (which prints a large number of blank lines).

    10 PRINT

    20 PRINT

    30 PRINT

    40 PRINT . . . and so on to ...

    230 PRINT   (keep an eye on the listing in the upper part of the screen)
    240 PRINT
    250 PRINT
    260 PRINT

We seem to have lost the first few lines. Or have we?

Hit LIST (key A) followed by NEWLINE - the program is now displayed from line 10 down.

Now try LIST 200 — the program is displayed from line 200. You can list the program starting from any line you want in this way. Notice that the current line cursor is altered to the line you selected.

Try using the up/down cursor control keys — go on hitting the ⇑ key and see what happens.

In this way you can display any part of the program you want.

By now you will have realised that the ZX-80 has some powerful (and very convenient) facilities to help you write programs.

From now on we can concentrate on how to write programs which actually do something — like solving arithmetic problems, for instance.

CHAPTER

5

SO YOU'VE GOT PROBLEMS

SO YOU'VE GOT PROBLEMS

People are very good at disentangling confused instructions and solving complex problems. A computer isn't — all it can do is to follow a list of instructions and carry out the instructions as it comes to them.

As an example, take the following instructions which a mother might give to her child:

"Could you run down to the shop and buy some bread? Take 50p which is in my purse on the kitchen table. And for goodness sake get dressed!"

It all seems quite clear; the child knows where to go, what to buy and where the money is.

A computer controlled robot would take the first instruction:

run down the shop, and do that.

The second instruction: buy some bread. It can't do that (no money) and at this stage would probably just stop, baffled.

The third instruction: take 50p from the purse, it can't do that.

The fourth instruction: go to the kitchen table. It returns home and goes to the kitchen table.

Then the fifth instruction: get dressed.

It gets dressed.

A child would instinctively carry out the instructions in the following order:

1. Get dressed
2. Go to the kitchen table
3. Take 50p from the purse
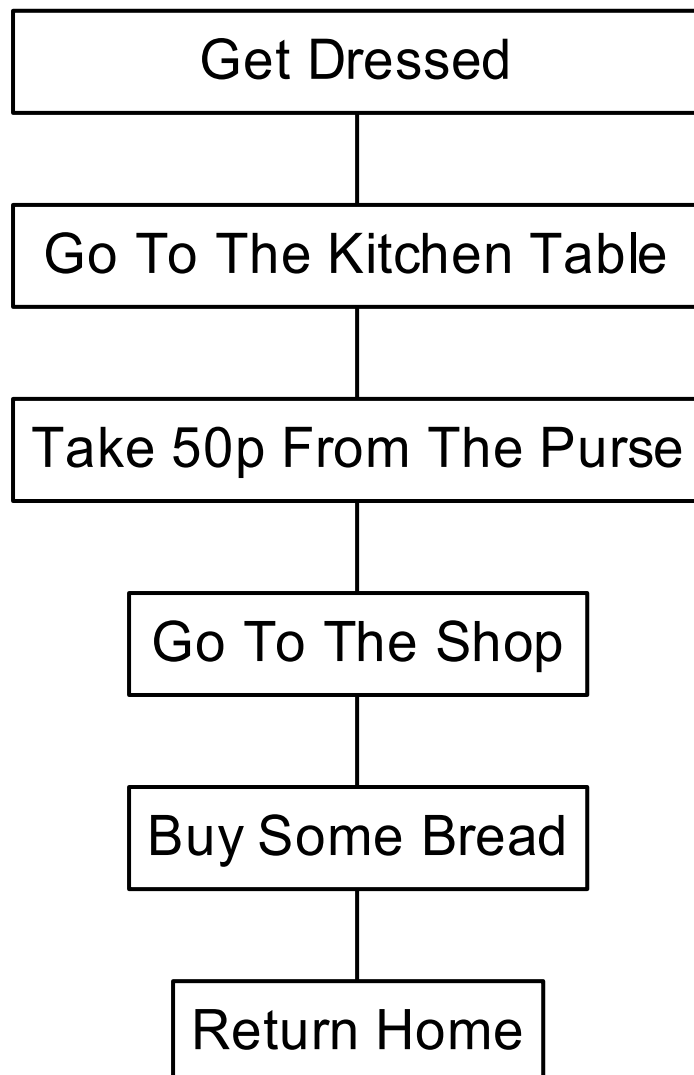4. Go to the shop
5. Buy some bread

He or she would come back home!

The mother in this example missed out one vital instruction ("come home") as well as putting the instructions in an illogical order. Children use commonsense to interpret complex instructions, but computers can't do this. You have to do all the thinking in advance when you use a computer. So remember:

 THINK STRAIGHT!

One way of making sure that you are thinking straight is by drawing a flow diagram before you start to do any actual programming. This helps you get things in the right order to begin with.

The flow diagram for the example above would look like this:

```
┌─────────────────────────────────┐
│          Get Dressed            │
└─────────────────────────────────┘
                 │
┌─────────────────────────────────┐
│      Go To The Kitchen Table    │
└─────────────────────────────────┘
                 │
┌─────────────────────────────────┐
│      Take 50p From The Purse    │
└─────────────────────────────────┘
                 │
      ┌───────────────────────┐
      │      Go To The Shop   │
      └───────────────────────┘
                 │
      ┌───────────────────────┐
      │     Buy Some Bread    │
      └───────────────────────┘
                 │
        ┌─────────────────┐
        │   Return Home   │
        └─────────────────┘
```

Each box contains an instruction, and the arrows show the order in which the instructions are

executed. Well. that looks pretty simple and neat. Now that the flow diagram is drawn on paper we can check it by going through the boxes one by one and asking ourselves: Can the child (or the computer) do this?

Starting with the first box (get dressed) we might think that it's pretty basic. On the other hand, the child may not be able to find his clothes because his naughty sister has thrown them out of the bedroom window. Perhaps we should add an instruction (find clothes) at the beginning of the program.

Go to the kitchen table: well, that looks OK.

Take 50p from purse: what happens if the purse is empty?

Go to the shop: does the child know where the shop is?

Buy some bread: what sort?

Return home: nothing much wrong with that!

Maybe the flow diagram wasn't as good as we thought. Even simple tasks can turn out to be more complicated than we think.

Most of the problems which we've spotted are due to a lack of information and this can be coped with by adding extra instructions at the beginning of the program. These could include:

1. Ask where the shop is
2. Find clothes

When you are writing programs for the ZX-80 it is particularly important to give the computer all the information it needs before it has to carry out a task. For instance, if you ask it to PRINT a variable the computer must know the value of the variable.

Now for some arithmetic.

CHAPTER

6

FINDING THE ANSWERS

FINDING THE ANSWERS

Up to now you've just been getting used to the feel of the computer. Now we'll actually use the computer to do a few sums.

THE LET STATEMENT

Nearly all arithmetic operations are done by using the LET statement. It is of the form

LET variable = expression.

In this case the variable (which can have any name you assign to it) is what you want to find out. The expression describes how you want to define the variable.

For instance

   LET A   = 2+4

variable   expression

means "add 2 to 4 and set A equal to the result (6)".

You can use other variables:

LET A = 2 + B

or

LET A = B + C

One condition: the computer must already know what the values of B and C are before it comes to the LET statement.

The equal sign (=) isn't used in quite the same way it is in ordinary arithmetic or algebra. For instance:

LET J = J +1

Obviously J isn't equal to J + 1. What the LET statement means is:

"Take the existing value of J, add one to it and then set J to this value."

The + sign is called an <u>operator</u> and defines the operation you wish to be performed. The other arithmetic operators are:

-      (minus)

*   (multiply - there is no x sign to avoid confusion with letter x)

/   (divide)

**   (raise to the power)

For the moment we'll leave / and ** out of it.

Here is a program for multiplying two numbers together

10 PRINT "MULTIPLICATION"

20 PRINT "ENTER FIRST NUMBER"

30 INPUT A

40 PRINT "ENTER SECOND NUMBER"

50 INPUT B

60 LETC==A *B

70 PRINT "THE ANSWER IS", C

Notice line 70. It's a PRINT statement with a string between inverted commas, but it's got a C added to it. This form allows you to economize on PRINT statements.

Enter the program and try running it, using two quite small numbers — such as 17 and 25.

The computer will come up with the answer:

THE ANSWER IS 425

Run the program a few more times, increasing the size of the input variables. At about 255 x 129 or just a bit larger you will get another error message — 6/60. The 6 is the code for <u>arithmetic overflow</u> because the answer is larger than 32767, which is the largest number the computer can hold as an integer variable.

Now try editing the program so that it does addition instead - just change the * to a + in line 60. You can also try subtraction and check that you get negative numbers when A is less than B.

All fairly straightforward so far.

Now let's try division. Alter line 60 to

60 LET C == A/B

Run the program for   A (first number) = 24
                      B (second number) = 12

The computer will come up with:

THE ANSWER IS 2

Run the program again, this time for A=18 and
B=12

THE ANSWER IS 1 ??!!

Shouldn't that be 1.5? Well, yes, but the ZX-80 uses integer arithmetic — only whole numbers can be expressed. What the computer does is to do a division normally and the truncate the result towards ZERO. As examples:

ANSWER

-2.58	would become	-2
-0.01	would become	0
-212.1	would become	-212
-5941.98	would become	-5941

This means that you may have to be careful when you use division — it's always fairly accurate when you're dividing a large number by a small number, but may be less accurate when the two numbers are closer together in size.

Luckily there are ways of getting round this problem. For instance, here is a program which will give quotients to 3 decimal places using integer arithmetic only.

10 PRINT "DIVISION PROGRAM"	
20 PRINT "DIVIDEND = ?"	
30 INPUT X	
40 PRINT "DIVISOR = ?"	
50 INPUT Y	
60 LET Z = X/Y	Divides X by Y
70 LET R1 = X-Z*Y	Calculates Remainder
80 LET D1 = 10*R1/Y	Divides 10*remainder by Y to give first decimal
90 LET R2 = 10*R1-D1*Y	Calculates 2nd remainder
100 LET D2 = 10*R2-D2*Y	Divides to give 2nd decimal place
110 LET R3 = 10*R2-D2*Y	Calculates 3rd remainder
120 LET D3 = 10*R3/Y	Last decimal place

130 PRINT "THE ANSWER IS ";Z;".";D1;D2;D3

This program does a long division in exactly the same way as we would do it on paper.

Try running it!

Notice the PRINT list in line 130 — it contains several literal strings and variables separated by semicolons. The semicolons tell the ZX-80 that each thing to be printed must be printed immediately after the preceding item, without spaces between them.

Also notice the LET statements from line 70 on. There's no fiddle — you can use more than one operator per statement. There is a snag, though.

Operations are not necessarily carried out in order from left-to-right.

The order in which they are carried out depends on what sort of operations are present in the statement.

Here is a list of priorities for arithmetic operations:

First   A**B (A to the power B)
Second   —A   (Negation, or multiplication by-1)
Third   A*B
Fourth   A/B
Fifth   A+B or A-B

Now for some experiments

10 REM TEST PROGRAM (REM is a keyword — key Y)
20 PRINT "ENTER A"
30 INPUT A
40 PRINT "ENTER B"
50 INPUT B
60 PRINT "ENTER C"
70 INPUT C
80 LET Z = A + B * C
90 PRINT Z

Try running this with A=1,B=2,C=3 but before you start, use the priority rules to predict what answer the ZX-80 will give: 7 or 9? "

O.K. — run it now.

The answer was 7. First the computer multiplied 2 by 3 to get 6, then it added 1 to get the final answer.

Try a few other combinations by editing statement 80- like

A*B/C

A/B*C   (watch out for truncation!)

A**B   (A to the power B — try using small numbers with C = 0)

-A**B

A*-B

Here's a golden rule: when in doubt either use more than one LET statement and build up that way, or use brackets.

Brackets make life much easier when it comes to complex arithmetic operations.

Take example Z = A/B*C

The normal sequence would be for the computer to evaluate B times C and then divide A by the result.

If we put brackets round A/B thus:

80LETZ=(A/B)*C

then the computer will carry out the operation within the brackets first, even though it has a lower priority.

A more subtle example is:

Z=A*(B/C)

and

Z = A*B/C

At first sight you would think that these would give the same answer.

Well, they do — sometimes!

Try them both using A == 100, B = 25, C = 5 (the answer is 500).

Now try it both ways using A = 100, B = 3, C = 5 (the answer should be 60).

What happened when you used the brackets?

Think about it.

The phantom truncator has struck again!

The computer evaluated 3/5 first and truncated it to ZERO - then it multiplied 100 by ZERO and naturally got ZERO as a result.

All this emphasizes that you have to be very much on the alert when you use integer arithmetic for multiplication and division. Multiplications can cause arithmetic overflow problems (which will cause the program to stop), whilst small numbers used in division may give rise to funny answers which don't give error codes.

If you think that you're going to run into truncation then do multiplications first — on the other hand it may be better to do divisions first to avoid overflow.

That's probably enough arithmetic for now. Let's move on to more interesting things.

CHAPTER

7

DECISIONS, DECISIONS

DECISIONS, DECISIONS

So far we have only considered problems which can be solved by carrying out a list of instructions, starting at the beginning of the list and working steadily down the list until the last instruction is carried out.

The next four chapters deal with ways in which you can make the ZX-80 (or any computer using BASIC) carry out much more complex programs which can perform many tedious tasks.

In chapter 5 we talked briefly of the use of flowcharts for checking that the program steps were in a logical sequence. This can be useful even for simple programs. When it comes to complex programs flowcharts are vital.

Let's consider a problem we face every day — getting up in the morning. There is a flowchart for getting up on the next page.

The program is said to <u>branch</u> at each decision diamond. As you can see, one feature of branches is that they allow some parts of the program to be skipped if they are not neccessary.

Programs can get difficult to understand if there are a lot of branches, and this is where flow diagrams help.

It is a good idea to keep a pad or notebook handy when you are writing programs and an even better one to draw a flow diagram of your program <u>before</u> you start writing it!

Turn to the next chapter to find out how the ZX-80 can take decisions.

GETTING UP

```
        ┌─────────┐
        │  Start  │
        └────┬────┘
             │
        ┌────┴────┐
        │ Wake up │
        └────┬────┘
             │
     ┌───────┴──────┐
     │ Look at clock│
     └───────┬──────┘
             │
          ╱──┴──╲
   No   ╱  Is it  ╲   Yes
 ┌─────◁  time to  ▷──────┐
 │      ╲ get up? ╱       │
 │       ╲──┬──╱          │
 │          │             │
┌┴──────────────┐         │
│ Read a book   │         │
│ until it is   │         │
│ time          │         │
└───────┬───────┘         │
        │                 │
        └────────┬────────┘
                 │
         ┌───────┴───────┐
         │ Get out of bed│
         └───────┬───────┘
                 │
            ┌────┴───┐
            │  Wash  │
            └────┬───┘
                 │
              ╱──┴──╲
        No  ╱  Do I  ╲  Yes
      ┌────◁  need    ▷──────┐
      │     ╲ clean   ╱      │
      │      ╲clothes╱       │
      │       ╲──┬──╱   ┌────┴──────────┐
      │          │      │Get clean      │
      │          │      │clothes        │
      │          │      └────┬──────────┘
      └──────────┼───────────┘
                 │
         ┌───────┴──────┐
         │ Get dressed  │
         └───────┬──────┘
                 │
            ┌────┴───┐
            │   End  │
            └────────┘
```
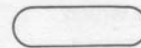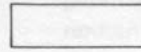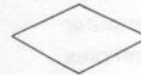
There are three sorts of symbols (or boxes).

```
 ┌─────────┐
(           )     Begin or end.
 └─────────┘

 ┌─────────┐      Processing block —
 │         │      this encloses anything which
 └─────────┘      can be done without making any decision.

    ╱╲            Decision diamond —
   ╱  ╲           encloses a decision with branches
   ╲  ╱           to either side depending on
    ╲╱            whether the decision is yes
                  or no.
```
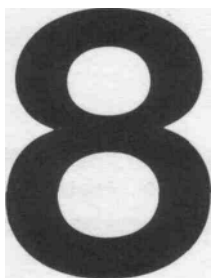
CHAPTER

8

BRANCHING OUT

BRANCHING OUT

Before we go on to real decisions lets have a look at the GO TO statement.

This is of the form

GO TO n where n is <u>normally</u> a line number. It can be a variable, though only the ZX-80 BASIC has the power to do this.

When the ZX-80 comes to a GO TO statement it jumps straight to statement number n and executes that statement. If n is a variable — X, say — the computer will look at the value of X and jump to the statement with that number — assuming that there is. a statement with that number in your program.

A further refinement of the GO TO statement is that n can be any integer expression such as A/10. Here again, this is possible with the ZX-80.

Here is a program which illustrates the use of GOTO.

10 PRINT 'THIS IS STATEMENT 10"
20 GO TO 40
30 PRINT 'THIS IS STATEMENT 30"
40 PRINT "END OF PROGRAM"

Now run the program.

The computer responded with

THIS IS STATEMENT 10
END OF PROGRAM

It skipped statement 30. "That's not much use", you must be saying, "what's the point of - including a perfectly good statement which will never be executed?"

OK

Get back into command mode.

Now, type GO TO 30, and then NEWLINE.
The ZX-80 came back with

THIS IS STATEMENT 30

END OF PROGRAM

This demonstrates two uses of GO TO — first to jump unconditionally to an arbitrary statement (in this case statement 40); second as a command which causes the computer to jump to the desired statement (in this case 30) and start executing the program from there.

THE IF STATEMENT

This is the most powerful control statement in BASIC. It allows the user to incorporate decision-making into his programs.

Here is a program for calculating square roots approximately. It does this by multiplying, a number by itself, starting with 0 and comparing the result with the number whose square root is to be found. If the product is less than the number to be rooted it increases the number, by 1 and tries again.

```
10 PRINT "SQUARE ROOT ROUTINE"
 20 PRINT 'ENTER THE NUMBER TO BE ROOTED"
30 INPUTX
40 LETJ=0
50 LET K = J*J
60 LET D = X-K
70 IF D-0 THENGO TO 110          (Use SHIFT 3 to get THEN)
 80 IFD<0THENGO TO 130           (< is SHIFT N)
90 LET J = J+1
100 GO TO 50
110 PRINT "THE ROOT IS" ; J
120 GO TO 140
130 PRINT " THE ROOT LIES BETWEEN (J-1); "AND ";J
140 STOP
```

See the diagram on the next page.

The form of the I F statement is:

IF (CONDITION TO BE MET) THEN (DO THIS) 70 IF D=0THENGOTO 110

In line 70 the condition to be met was $D = 0$ and if this is so then J is exactly the square root of X, so we want to print THE ROOT IS J. In this case (DOTHIS) is GO TO 110 which is the

number of the PRINT statement we want.

In most BASIC'S the only thing you can do in a IF statement is to GO TO a line number. The ZX-80 lets you do other things as well.

In fact it lets you do almost anything!

Some examples:

70 IF D=0THEN PRINT "NUTS"

70 IF D=0THEN INPUT G

70 IF D=0THEN LET A = 10

FLOWCHART FOR
SQUARE ROOT PROGRAM

Any keyword can follow THEN in an IF statement — even things like LIST or RUN. The results can be rather peculiar if you use commands such as LIST. The examples listed above can be useful, though.

The condition to be met is (in its simplest version) of the form:

(Expression) (relational operator) (expression)

This sounds rather a mouthful!

A relational operator can be:

=   equal to

<   less than

>   greater than

Examples

A =2   A equal to 2

A<10   A less than 10

A+B>C+D   A+B greater than C+D

In these cases. A, 2, 10, A+B and C+D are the expressions.

The expressions need not be a simple integer or integer variable.

The statement   IF A = Z - (Z/B) * B THEN GO TO 10

                             ↑expression

is quite valid, and so is IF A + B > C+D THEN GO TO 20

The condition to be met can be even more complex because logical operators can be used.

For example

 10 IF NOT A =2 THEN GOTO 100

          ↑logical operator

(use SHIFT 1 for NOT)

This NOT operator negates the succeeding condition, so that the ZX-80 will jump to 100 if A does not equal 2.

This program illustrates the use of the AND operator.

10 INPUT A

20 INPUT B

30 INPUT C

40 IF A = 1 AND B = 1 AND C = 1 THEN PRINT "OK"

          ↑   ↑Another logical operator

          ↑

(Use SHIFT 2 for AND)

The condition to be met is that ALL three variables must be 1; if this is so it will print <u>OK</u> You can chain together as many conditions as you like in this way. Try running this program and see if you can make it print <u>OK</u> in any way other than by entering three 1's.

The other logical operator is the OR operator.

Edit the program by changing the AND's to OR's (OR is SHIFT B).

When you run the program now you will find that it prints <u>OK</u> as long as one of the numbers entered is a 1.

You can use brackets to group things together. Consider the following statements:

(a) 40 IF A = 1 OR B = 1 AND C = 1 THEN PRINT "OK"
(b) 40 IF (A = 1 OR B= 1) AND C = 1 THEN PRINT "OK"

If you remember, in chapter 6 we said that some operators had priority over others — multiplications were carried out before divisions or additions, for instance. The same applies to logical operators.

Their priority (in descending order) is

NOT
AND
OR

Thus, in (a) B = 1 AND C = 1 is taken first, so that the computer will print <u>OK</u> if B AND C are 1 but will also print <u>OK</u> if A =1. The brackets in (b) work in the same way as they did for arithmetic operators. Thus the ZX-80 will print <u>OK</u> if C = 1 and <u>either</u> B = 1 or A = 1.

The operators NOT, AND and OR also allow you to produce <u>conditional expressions</u>. For example, if you require

X=3   ifA>B
X = Q+R   if A=B

X = P   if A<B

The obvious way is:

110 IFA>BTHEN LET X = 3

120 IFA=BTHEN LET X = Q+R

130 IF A<BTHEN LET X = P

However you could instead write:

110 LET X=A>B -AND 3 OR A=B AND Q+R OR A<B AND P

To round off this chapter on branching, here is a program which throws a die. The program uses a useful facility, the function

RND(X)

This generates a random number in the range 1 to X. It occurs in line 120 of the program — in this case X = 6


```
 10  PRINT "DIE THROWING"
 20  LET A$ ="●   ●"                    ●is SHIFT A
 30  LET B$= "  ●  "
 40  LET C$= "●    "
 50  LET D$ ="    ●"
 60  LET E$ ="....."
120     LETX=RND(6)
130     PRINT "YOU THREW. .."
140     IF X = 1 THEN GO TO 200
150     IF X= 2 THEN GO TO 300
160     IF X =3 THEN GO TO 400
170     IF X= 4 THEN GO TO 500
180     IF X=5THEN GO TO 600
190     IF X =6 THEN GO TO 700
195     GO TO 1000
```

```
200 PRINT E$
205 PRINT E$
210 PRINT B$
215 PRINT E$
220 PRINT E$
230 GO TO 1000
300 PRINT C$
305 PRINT E$
310 PRINT E$
315 PRINT E$
320 PRINT D$
330 GO TO 1000
400 PRINT D$
405 PRINT E$
410 PRINT B$
415 PRINT E$
420 PRINT C$
430 GO TO 1000
500 PRINT A$
505 PRINT E$
510 PRINT E$
515 PRINT E$
520 PRINT A$
530 GO TO 1000
600 PRINT A$
605 PRINT E$
610 PRINT B$
615 PRINT E$
620 PRINTA$
630 GO TO 1000

700 PRINTA$

705 PRINT E$

710 PRINT A$

715 PRINT E$
```

720 PRINTA$

1000 STOP

It's a bit of a strain to enter this program but it's quite fun to run.

Note that lines 140 to 190 could have been replaced by

140 GOTO(X+»1) * 100

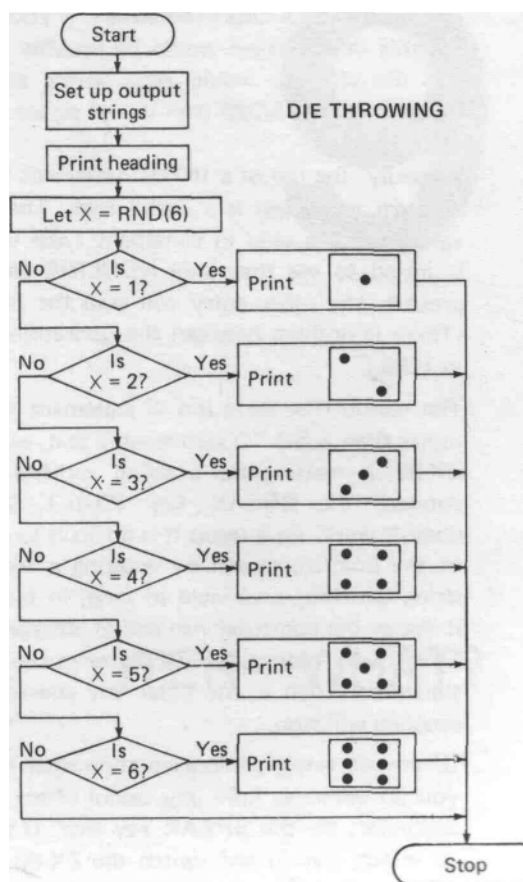If you want to make the program restart itself try editing the program:

1000 PRINT "HIT NEWLINE TO THROW AGAIN"
1100 INPUT X$
1200 CLS   (CLS is on key C)
1300 IF X$= "THEN GOTO 120

DIE THROWING

Two points: CLS clears the screen. If you didn't put this in the screen wouid fill up after 3 goes and the program would stop, giving an error message such as 5/305 (run out of screen at line 305).

Secondly, the use of a INPUT statement to halt program execution is a useful trick. The string variable X$ is used in statement 1300 when it is tested to see that only NEWLINE has been pressed; any other entry will stop the program (There is nothing between the quotation marks in 1300.)

The reason that there is a IF statement at 1300 rather than a GO TO statement is that, while the ZX-80 is waiting for a string variable to be entered, the BREAK key (SHIFT SPACE) doesn't work. As a result it is difficult to get out of the program except by entering a very long string variable, a variable so long, in fact, that it makes the computer run out of storage space. This is very tedious, so it's better to use the IF statement. Then if you enter any character the program will stop.

(If you do ever get into a situation when nothing you do seems to have any useful effect on the computer, try the BREAK key first. If this has no effect, give in and switch the ZX-80 off for a few seconds. You lose the current program, but this may be inevitable if you really have got yourself into a fix.)

CHAPTER

9

ITER - WHAT?

ITER-WHAT?

Very often we come up against problems which involve repeated operations. The square rooting program in chapter 8 was one example of such a problem, and the program was written so that the computer jumped back to line 50, if the test conditions in statements 70 or 80 were not met, and tried again — and again — until the conditions were met.

Programs of this sort are called iterative. By using iterative techniques it's possible to get the computer to do a lot of work with a very short program.

It's easy to make a program iterative — all you have to do is to put a GO TO statement at the end of the program to jump the computer back to the first statement (or any other suitable statement). One snag is that such iterative programs can get stuck in endless loops.

One way to avoid this is to allow the computer to go round only a set number of times. One way of doing this would be to use IF and GO TO statements.

Consider the following program which prints t 152 times.

```
10 LETJ=1
20  PRINTS
30  IF J = 152 THEN GO TO 60
40  LET J = J + 1
50  GO TO 20
60  STOP
```

First the program sets J (the loop control variable) to 1. Then it prints $. Then it tests to see if J = 152. It isn't and so the computer goes on to line 40 which increments the value of J by 1 to 2. The computer passes on to 50 which tells it to jump back to 20. This process goes on until J = 152 when the IF statement is satisfied and the computer jumps out of the loop to line 60.

The arrows on the program help you to see where the program is going; for short programs this is quicker than drawing a flow diagram and does make it easier to see what the program is doing.

The program took 5 lines (if you don't count the STOP statement, which doesn't actually do anything — it just serves as a target for the GOTO in line 30.

Loops are so useful that you'd have thought that there was an easier way of writing programs using them. Well there is. Turn to chapter 10.

CHAPTER **10**

LOOPING THE LOOP

LOOPING THE LOOP

Because loops are so useful some special statements have been devised to make it easy to use them.

Program for printing $:

```
10 FORJ= 1 TO 152              (use SHIFT 4 for TO)
20 PRINT"$";
60 NEXTJ
```

In chapter 9 it took 5 lines of program to get the same result using IF and GO TO statements.

Note the ; after the PRINT statement. This tells the computer to print each $ immediately after its predecessor. Try running this program. It's a pity that one can't have a dollar bill for each $ sign printed.
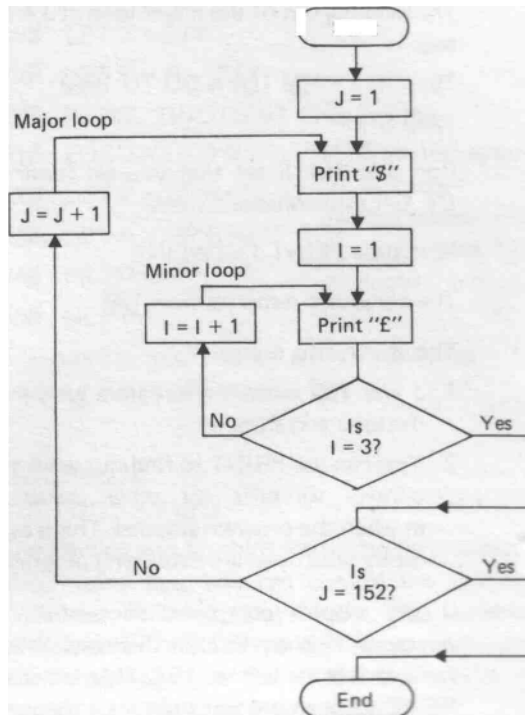
In BASIC these loops are normally called FOR loops. They are also widely known as DO loops (because other languages, such as FORTRAN, use DO instead of FOR).

There is nothing to stop us from putting a loop inside another loop — or putting several loops in:

Add the following lines to the program

```
30 FOR I = 1 TO 3
40 PRINT "£"
50 NEXT I
```

The Flowchart for this program is

You can jump <u>out</u> of a loop at any time. If you do this the loop control variable, will remain at whatever value if had got to when you jumped out. In our case the loop control variables were J for the major loop and I for the minor loop.

Try jumping out of the major loop at J = 100 add:

15 IF J= 100 THEN GO TO 1000
1000 STOP

Run this. You'll see that you get fewer $'s and £'s. Get into command mode.

Now type PRINT J NEWLINE

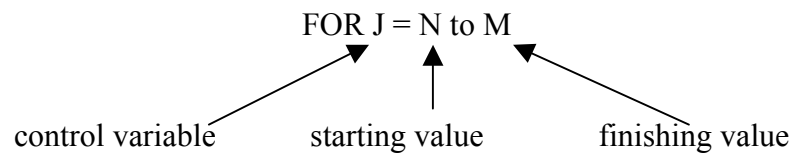The computer responds with <u>100</u>

This shows two things
1. J was 100 when the program jumped out of the loop and stopped.
2. You can use PR I NT to find out what state the control variables (or other variable) were in when the program stopped. This is especially useful when you are debugging programs.

If the loop is completed successfully for all values of J up to 152 (in this case) the control

variable will be left at 153. This is because the NEXT J statement not only tests the value of but also increments J by 1. Some BASIC'S allow you to increment the control variable by other amounts, but this isn't possible on the ZX-80 (It's not a very useful feature anyway.)

All this is quite straightforward.

FOR J = N to M

control variable   starting value   finishing value

defines what variable you want to use, what value it should start at and at what value of control variable you want to leave the loop.

NEXT J reminds the computer that it is in a loop and tells it what variable it should be testing and (in fact) where it should jump back to if the loop isn't finished.

You can start at any value you like — try starting at J = 100. The computer will go round the loop M - N + 1 times.

DON'T EVER jump into a loop unless you have just jumped out of it — if you do the computer will skip the FOR statement and will get very confused because it won't know what the control variable is, what it should start at and what it should finish at. It will do the best it can but **it** will probably stop when it gets to the NEXT statement. The error message will be of the form

1/LINE No. where 1 means "no FOR statement to match this NEXT statement", or 2/LINE No. where 2 means "variable name not found".

You can try this by adding to the program:

5 GO TO 20

and running, and then adding

4 LET J ⁼ 4

and running (you must say RUN, not GO TO 4).

This will jump the ZX-80 into the major loop and it will stop when it gets to statement 60

(NEXT J) giving the error message 2/60 or 1/60.

In chapter 6 there was a program to divide one number by another to give a quotient to 3 decimal places.

Here is a program to give any number of decimal places. Statement 10 is a remark. The computer disregards REM statements when it executes programs. It's often useful to put comments and remarks into programs so that you can remind yourself what the program does when you look at it, often months later!

```
10 REM HIGH PRECISION DIVISION
20 PRINT "HOW MANY DECIMAL PLACES?"
30  INPUT D
40  PRINT "DIVIDEND?"
50  INPUT R
60  PRINT "DIVISOR?"
70  INPUT Y
80  LET Z = R/Y
90  LETR=R-Z*Y

100 PRINT "QUOTIENT IS "; Z; ";.";
```

110 FOR J=1 TO D	(110 to 150 evaluate
120 LET Z = 10* R/Y	successive decimal
130 LETR=10*R-Z*Y	places until the program
140 PRINT Z;	has produced D
150 NEXTJ	decimal places)

Try running this, first of all with dividend = 1,divisor =3, D = 100 (say).

Note line 110

110 FOR J= 1 to D

The starting and finishing values can be variables. This means that you can control the starting value and finishing value of the loop variable as you wish. The starting and finishing values **can** also be underlined expressions of the form D/2, (D+1)*5 or indeed any other arithmetic expression.

110 FOR J = D*10 TO D*200 would be quite legal. However, the name of the control variable must be a single letter.

If you decide to use several FOR loops, one inside the other, be very careful in the way you

go about it

```
        ┌──→ FOR J =      ┌──→ FOR J =
        │  ┌──→ FOR I =   │  ┌→ FOR I =
        │  │  ┌→ FOR K =  │  └── NEXT I
        │  │  └─ NEXT K   │  ┌→ FOR K =
        │  └─── NEXT I    │  └── NEXT K
        └───── NEXT J     └──── NEXT J
```

are both quite OK.

On the other hand

```
        ┌──→ FOR J =
        │  ┌──→ FOR I =
        │  │  ┌→ FOR K =
        │  │  └─ NEXT K
        │  └─── NEXT J
        └───── NEXT I
```

is defiantly not all right and could cause major problems. Another good reason for using flowcharts!.

One final point: because the control variable is effectively tested and incremented by the NEXT statement the main part of the loop will always be executed at least once regardless of the value of the control variable even if you have jumped into the loop.

SUBROUTINES

A subroutine is a sub-program which may be used once or many times by the program (or main program).

Example:
```
10 FORJ= 1 TO 10
20 GO SUB 1000                          (GO SUB is on key V)
30 NEXTJ
40 PRINT "END"

900 GO TO 1200
1000 PRINT "SUBROUTINE EXECUTED"
1100 RETURN                             (RETURN is on key B)
1200 STOP
```

Statement 20 tells the computer to <u>GO TO the subroutine</u> at line 1000.

This prints:

SUBROUTINE EXECUTED

Statement 1100 tells the computer that the subroutine is finished and that it should return to the main program. The computer then jumps to the line <u>immediately following</u> the GO SUB statement and executes that (in this case: NEXT J).

An example of the use of subroutines is given below: The Chinese Ring Puzzle.

As you will see, one subroutine can call another, or even call itself (this is called <u>recursion</u>).

THE CHINESE RINGS PUZZLE

This is a program which will say what moves are required to remove N rings from the T-shaped loop.

The mechanics of this wire puzzle are not important — roughly speaking "one" manipulates, the rings until they all come off the loop.

For an arbitrary number of rings the rules are as follows:

1. Each ring can be either on the loop or off it.
2. Only one ring may be moved (from on to off or vice versa) at a time.

3. The first ring may be moved at any time.

4. The ith ring (i>1) may be moved if and only if:

(a)   All the rings numbered i-2 or lower are off.

(b)   Ring i-1 is on.
. . . the rings higher up the loop (number >i) may be in any state, so:

To remove the first i rings:

1. Remove the first i-2 rings

2. Remove the ith ring

3. Replace the first i-2 rings

4. Remove the first i-1 rings

To replace the first i rings:

1. Replace the first i-1 rings

2. Remove the first i-2 rings

3. Replace the ith ring

4. Replace the first 1-2 rings

CHINESE RINGS (Recursive procedure)

    10 INPUT N

    20 GO SUB 100

    30 STOP

```
100 IF N<1 THEN RETURN
120 LET N=N-2
130 GO SUB 100
140 PRINT N+2; "OFF",
150 GO SUB 500
160 LETN=N+1
170 GO SUB 100
180 LETN=N+1
190 RETURN

500 IF N<1 THEN RETURN
520 LET N = N-1
530 GO SUB 500
540 LET N= N-1
550 GO SUB 100
560 PRINT N+2; "ON",
570 GOSUB5(2f0
575 LET N = N+2
580 RETURN
```

Your output for the case N = 4 should look like:
2 OFF 1 OFF 4 OFF 1 ON
2 ON 1 OFF 3 OFF 1 ON
2 OFF 1 OFF

The GO SUB statement can be used in conjunction with a variable, e.g.

10 GO SUB G   whereupon the computer will jump to the subroutine at line whose number is the value of G (if there is such a line).

As before, an expression can also be used.

In general it is not a very good idea to use variables or expressions in this way because the program may alter the variable in a way you hadn't foreseen and thus cause problems.

CHAPTER

11

HOW TO PRINT

HOW TO PRINT

Up to now we have used the PRINT statement as and when it was needed to print out headings and variables. In fact PRINT is a very versatile statement indeed.

The general form of the PRINT statement is

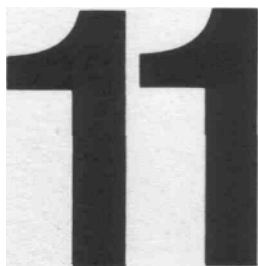PRINT expression ch expression ch expression ch . . . and so on.

ch stands for <u>control character</u> and this can be , or ; or nothing at all (at the end of the statement only).

Expressions can be literal strings in quotes, e.g. "THIS IS A LITERAL STRING" or string variables such as A$

Alternatively they can be integer variables or arithmetic expressions, e.g.

A2 or B*2/C

The object of using control characters is to be able to control the spacing of the line to be printed. We've already come across the use of ; in
PRINT statements.

The program

10 LETX=4
20 PRINT "THE ANSWER IS"; X; "UNITS"

              expression   control   expression

                           characters

Would print:

THE ANSWER IS4UNITS

The semi-colon (;) makes the computer print out the expressions without any spaces between them. If you want spaces you have to include then in the literal strings to be printed.

The comma (,) is used as a tab. Each display line of 32 characters on the screen is divided into 4 <u>fields</u> each of which is 8 characters in length. Each time the computer comes across a comma in a PRINT statement, it starts printing the next expression at the beginning of the next available field.

The effect of this is to provide a display on the screen in 4 columns. If a string variable or literal string is more than 7 characters long and is followed by a comma the computer uses two or more fields in which to print the string and starts the next expression at the beginning of the third field.

You can add extra commas to skip fields:

PRINT "FIELD 1", "FIELD 2", "FIELD 3"

would produce the output
FIELD 1 FIELD2.FIELD3
but
PRINT "FIELD 1",,' FIELD 2". "FIELD 3"

extra comma

would produce

FIELD 1   FIELD2   FIELD3

this field skipped

The statement
PRINT "THIS ISA FIELD",,"FIELD1","FIELD2"

would produce

THIS ISA FIELD   FIELD 1
FIELD2

The computer used 2 fields for the first string, skipped a field and then continued. It runs out of fields on the first line and has to continue in the first field of the second line.

There is no limit to the number of fields you may skip using commas, (Unless you run out of display area, of course.)

If you use a control character right at the end of a PRINT statement the next PRINT statement will obey the control character. For instance

10 LETX=4

20 PRINT "THE ANSWER IS ";

30 PRINT X

will produce the output

THE ANSWER IS 4

If no control character occurs at the end of a line the next PRINT statement will start printing on a new line.

The best way of finding out about PRINT is to experiment with it.

CHAPTER

12

COPING WITH CHARACTERS

COPING WITH CHARACTERS

You will have noticed that, in addition to the normal typewriter characters there are several other characters not usually found on a typewriter. In addition there are some usable characters <u>not</u> written on the keyboard.

Before we go on to investigate what characters <u>can</u> be printed, meet an interesting function:

CHR$(X) where X is a numberJnteger variable or expression.

It is used mainly in conjunction with the PRINT statement.

CHR$(X) means "the character whose code is x"

Try this:

```
10 INPUTX
20 PRINT CHR$(X)
30 GO TO 10
```

What this program does is to ask for a number and then to print out the character whose code is the number entered. The codes lie between 0 and 255.

A useful program for listing the symbols is given below.

```
10 PRINT "ENTER CODE VALUE"
20 INPUT X
30 PRINT X; ". . .";CHR$(X)
40 PRINT
50 LET X = X +1
60 GO TO 30
```

This lists 11 symbols together with their codes. starting with the code entered at the start of the program.

Code	Character	Code	Character
0	Space	128	Inverse Space
1	Null String	129	Inverse Quote
2	Graphics	130	Inverse Graphics
3	Graphics	131	Inverse Graphics
4	Graphics	132	Inverse Graphics
5	Graphics	133	Inverse Graphics
6	Graphics	134	Inverse Graphics
7	Graphics	135	Inverse Graphics
8	Graphics	136	Inverse Graphics
9	Graphics	137	Inverse Graphics
10	Graphics	138	Inverse Graphics
11	Graphics	139	Inverse Graphics
12	£	140	Inverse £
13	$	141	Inverse $
14	:	142	Inverse :
15	?	143	Inverse ?
16	(	144	Inverse (
17	)	145	Inverse)
18	-	146	Inverse -
19	+	147	Inverse +
20	*	148	Inverse *
21	/	149	Inverse /
22	=	150	Inverse =
23	>	151	Inverse >
24	<	152	Inverse <
25		153	Inverse ;
26		154	Inverse ,
27		155	Inverse .
28	0	156	Inverse 0
29	1	157	Inverse 1
30	2	158	Inverse 2
31	3	159	Inverse 3
32	4	160	Inverse 4
33	5	161	Inverse 5
34	6	162	Inverse 6
35	7	163	Inverse 7
36	8	164	Inverse 8
37	9	165	Inverse 9

38	A		166	Inverse A
39	B		167	Inverse B
40	C		168	Inverse C
41	D		169	Inverse D
42	E		170	Inverse E
43	F		171	Inverse F
44	G		172	Inverse G
45	H		173	Inverse H
46	I		174	Inverse I
47	J		175	Inverse J
48	K		176	Inverse K
49	L		177	Inverse L
50	M		178	Inverse M
51	N		179	Inverse N
52	O		180	Inverse O
53	P		181	Inverse P
54	Q		182	Inverse Q
55	R		183	Inverse R
56	S		184	Inverse S
57	T		185	Inverse T
58	U		186	Inverse U
59	V		187	Inverse V
60	W		188	Inverse W
61	X		189	Inverse X
62	Y		190	Inverse Y
63	Z		191	Inverse Z
64	Unused		192	Unused
65	Unused		193	Unused
66	Unused		194	Unused
67	Unused		195	Unused
68	Unused		196	Unused
69	Unused		197	Unused
70	Unused		198	Unused
71	Unused		199	Unused
72	Unused		200	Unused
73	Unused		201	Unused
74	Unused		202	Unused
75	Unused		203	Unused
76	Unused		204	Unused

77	Unused	205	Unused
78	Unused	206	Unused
79	Unused	207	Unused
80	Unused	208	Unused
81	Unused	209	Unused
82	Unused	210	Unused
83	Unused	211	Unused
84	Unused	212	"
85	Unused	213	THEN
86	Unused	214	TO
87	Unused	215	;
88	Unused	216	,
89	Unused	217	)
90	Unused	218	(
91	Unused	219	NOT
92	Unused	220	-
93	Unused	221	+
94	Unused	222	*
95	Unused	223	/
96	Unused	224	AND
97	Unused	225	OR
98	Unused	226	**
99	Unused	227	=
100	Unused	228	>
101	Unused	229	<
102	Unused	230	LIST
103	Unused	231	RETURN
104	Unused	232	CLS
105	Unused	233	DIM
106	Unused	234	SAVE
107	Unused	235	FOR
108	Unused	236	GO TO
109	Unused	237	POKE
110	Unused	238	INPUT
111	Unused	239	RANDOMISE
112	Unused	240	LET
113	Unused	241	?
114	Unused	242	?
115	Unused	243	NEXT

116	Unused	244	PRINT
117	Unused	245	?
118	Unused	246	NEW
119	Unused	247	RUN
120	Unused	248	STOP
121	Unused	249	CONTINUE
122	Unused	250	IF
123	Unused	251	GO SUB
124	Unused	252	LOAD
125	Unused	253	CLEAR
126	Unused	254	REM
127	Unused	255	?

Inverse means that the character appears white on a black background.

Inverse space is a black square, for instance.

The graphics characters are shown on the next page: CHR$ (X) allows us to print any character we want.

SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE
	2		7		130		135
	3		8		131		136
	4		9		132		137
	5		10		133		138
	6		11		134		139

Table of Graphic Symbols

There is a further group of facilities which enables us to handle characters;

These are

(a)   TL$ (string)
—this gives the string minus its first character.

The string can be a literal string inside quotes or a string variable.

10 PRINT TL$ ("ABC")   would give

BC

10   PRINT TL$(G$)   will cut off the first character of string G$

(b) CODE (string)

—this gives the code corresponding to the first character in a string (either a string variable or a literal string)

10 PRINT CODE("ABC") would print 38 which is the code for A.

An example of the way in which these can be used is the following program which accepts a string and prints it out in inverse video.

```
10 PRINT "ENTER YOUR STRING"
20 INPUT G$
30 PRINT G$
40 LET X= CODE (G$)
50 LETX=X+128
60 IF G$=CHR$ (1) THEN GO TO 100
70 PRINT CHR$ (X);
80 LETG$==TL$(G$)
90 GO TO 40
100 STOP
```

Statement:

40 sets X= code of the first character of G$.

50 adds 128 to the code (this gives inverse of the letters, digits and graphics).

60 test the string to see if it is a null string, i.e. has no characters in it. If it is a null string either the input string was a null string to begin with or all the characters have been converted to inverse video and printed.

70 prints the inverse video character.

80 chops off the character which has just been printed and the program then jumps back to 40 and the code for the next character is extracted. The program goes on until the string has been shortened to the null string.

There is a further function which can be useful. This is:

STR$ (variable or number).

This allows an integer number or variable to be treated as a string variable

10 LET G$=STR$(1234)

setsG$= "1234"

    10LETG$=STR$(X)

does the same thing for an integer variable — if X = 4852 then G$ = "4852".

So far we've not said very much about the graphics symbols. These have been designed so as to double the effective resolution of the display, which gives 23 lines of 32 characters each.

Here is a program which plots TWO bar charts on the same display.

10 LET X = 0	Defines
20 PRINT "YZ – AXIS"	variables
30 PRINT "X ="	and
40 FOR I = 1 TO 21	prints
50 LET Y = X	titles;
60 LET Z = 24-X	Generates Y
70 PRINTX.	and X

80 FOR J = 1 TO 20

85 IF J>YANDJ=ZTHEN PRINT CHR$ (3);

90 IF J>Y AND J>Z THEN GO TO 135 95 IF J=Y AND J>Z THEN PRINT CHR$ (11);
100 IF J<Y AND J<Z -THEN PRINT CHR$ (139);

105 IF J<Y AND J=Z THEN PRINT CHR$ (139);

110 IF J<Y AND J>Z THEN PRINT CHR$ (11);

115 IF J=Y AND J<Z THEN PRINT CHR$ (139);
120 IF J>Y AND J<Z THEN PRINT CHR$ (3);

125 IF J=Y AND J=Z THEN PRINT CHR$ (139);

130 NEXTJ

135 PRINT 140 LETX^X+1

150 NEXT I

The program prints Z as black bars, Y as grey bars. Statements 80 to 130 determine what graphic symbol is to be used; ▄,▓,▨ or nothing at all.

This program calculates the graphs of Y = X and Z = 24 — X and plots them in bar chart form. As you will see if you run the program it produces a very clear, unambiguous display. You can use the same sort of trick to achieve greater resolution along the line as well as from line to line.

Lines 85 to 125 decide what character will **be** printed, depending on the relative size of J, Z and Y.

These examples are only scratching the surface when it comes to character manipulation and graphics. The possibilities are almost literally endless.

CHAPTER

13

HELP! OR, WHAT TO DO WHEN DESPERATE

HELP! or "What to do when desperate"

You may occasionally get into difficulties.

These can be divided into two sorts:

(a) Problems with the system
(b) Problems with your programs.

Most problems with the system are likely to occur when you're entering string variables into your programs. One favourite is deleting the quotation marks round the string by accident. When the computer is waiting for you to enter a string variable it prints "**[L]**".

If you enter the wrong string and then use RUBOUT to delete it you may delete one or other of the quotation marks. This will give rise to a **[S]** syntax error symbol. If you see this check to see that both quotes are there.

If you have written a program which calls for a null string to be entered, and the program is recursive, you may find it very difficult to get out of the program back into command mode. Whatever you enter seems to have no effect! When (or if) this happens delete the quotes using RUBOUT $\Rightarrow$ RUBOUT and enter (say) CHR$ (10**6) which causes arithmetic overflow, giving a 6/N error message (when N is the line number of the INPUT statement reading the string).

Generally speaking the BREAK key (SHIFT SPACE) is the first thing to resort to. If the program is caught in an endless loop, or if it is LOADing unsuccessfully, the screen will grey or black for an indefinite period. BREAK will get you back under these conditions.

If BREAK does not work in this situation there is nothing left to do but switch the ZX-80 off for a few seconds and then switch on. You do lose whatever program was in the ZX-80 if you do this.

When it comes to faults in the program it is difficult to offer such specific advice.

Some problems arise if you type an O instead of a 0.

LET J = O would be accepted as a valid program line but would give a 2/N error code

(variable not found) when the program was run. This sort of thing can be very difficult to spot. Similarly S and $ do get confused.

When a program stops unexpectedly or does something peculiar it may be difficult to work out exactly what went wrong. It is possible to carry out a post-mortem by using the immediate PRINT statement to find out what the value of variables (especially loop control variables) was at the time the program stopped. If you type

PR I NT J (or whatever variable you want)

the ZX-80 will print it even after the program has stopped running.

Sometimes there is not room to print the whole of the line number in an error message, particularly with error numbers 4 and 5. In this case usually only the first digit of the line number is printed. For example, error 4 on line 250 may cause the message

4/2

Another useful technique is to put STOP statements into programs at key points. When the program reaches the STOP statement it will stop (of course) and you will be able to see how it has performed up to that point. You can then get back into command mode, type CONTINUE (key T) NEWLINE and the program will continue from the STOP statement.

Luckily the ZX-80 is quite choosy about the program lines it will accept and this eliminates many of the problems which can happen with other BASIC'S.

CHAPTER

14

A RAGBAG OF FUNCTIONS

A RAGBAG OF FUNCTIONS

This chapter covers all those statements and functions that haven't already been dealt with elsewhere.

RND(X)   provides a random number in the range 1 to X

Typical statement using RND:

10 LETJ=RND(X)
                sets J equal to a random number

Every time the ZX-80 executes the function RND it uses a random number generator to generate a pseudo-random number. It is called pseudo-random because the numbers occur in a fixed sequence. However, the sequence is very long and hence appears random.

RANDOMISE-   sets the starting point of the sequence to a number equal
                                to the number of frames supplied to the TV since the
                                machine was turned on (unless POKE is used to alter
                                the count, see below).

RANDOMISE n -   sets the starting point of the sequence to n unless n is 0,
                                when it behaves as RANDOMISE.

The RANDOMISE   statement allows the random sequence to be initialised
                                at any time in the program. Typical forms:

10 RANDOMISE 6
10 RANDOMISE

RANDOMISE n (n ≠ o)   generates a sequence depending only on the value of n
                                (which will be the same from run to run if n is
                                unchanged).

RANDOMISE   generates a different sequence each time.

One highly useful statement is POKE.

This is of the form

POKE A,B   where A is the ADDRESS of a location in store and B
                                       is an expression (the value of which should be less than
                                       256 for sensible results).

Typically A might be the address of one of the two bytes which form the variable which acts
as the frame counter. This is demonstrated in the next example program.

        Thus:

LET X = PEEK (A)   sets X equal to the contents of address A
                                       PEEK (A) is always less than 256 (in range 0 to 255)

The reason that the variables associated with PEEK and POKE should always be less than
256 (255 is maximum) is that everything in the ZX-80 is stored in 8 bit bytes. Bit is short for
Binary digit. The maximum which can be stored in 8 bits is 255, and so all the integer
variables use up 2 bytes, which allows numbers up to 32,767 to be stored.

Each half of a variable is stored in one byte, and every byte has an address of its own.

Here is a program which uses PEEK and POKE to gain access to the TV frame counter.

PEEK/POKE REACTION TIMER

10 FOR I = 1 to 20* RND(100)

20 NEXT I

30 POKE 16414,0

40 POKE 16415,0
50 PRINT "HIT RETURN"
60 INPUTC$
70 LETA= PEEK (16414)
80 LET B= PEEK (16415)

90 PRINT "YOUR REACTION TIME WAS";(B^256+A-4) *20; "MILLISECS"

16414 and 16415 are the addresses of the two halves of a 16 bit number which counts the

frames on the TV — increasing by 1 every 1/50th of a second. Lines 30 to 40 set the count to zero, and the count is stopped when a null string is input to C$. There is a delay on all operations (mainly between pressing the return key and this signal getting to the CPU) of around 60 mS, hence the 4 subtracted from the expression in line 90.

This could have a repeat mechanism tacked on the end, for example:

```
100 PR I NT "DO YOU WANT ANOTHER GO?"
110 PRINT "TYPE Y OR N"
120 INPUT C$
130 IF C$ = "Y" THEN GO TO 10
140 STOP
```

Another function which allows the user **to** communicate directly with the ZX-80 is

USR(A) — this calls a machine-code subroutine at the address A. The value is whatever the routine leaves in HL (a storage register within the central processor of the ZX-80) or, if the subroutine has not altered HL, the result is A

Typical form: LET J = USR(A)

PEEK, POKE and USR(A) are really facilities provided for very experienced users who understand the detailed working of the ZX-80.

CLEAR is a statement which resets all the variables in the program.

Typical form:

```
10 CLEAR
```

(RUN automatically clears the variables every-time a program is run.)

And now, last but not least, DIM. This is of the form

10 DIM A(B) sets up an array A which contains B+1 variables.

Each variable is called an element of array A.

Arrays can have any single-letter name and can have any number of elements (providing that there is enough room to store all the elements).

It is possible, though not a good idea, to have a variable and an array with the same name example:

10 LET A =4
20 DIMA(A)

Each element is referred to by its <u>subscript.</u> For instance

B(0) is the first element of array B
B(2) is the third element of array B
B(N) is the (N+1)th element of array B.

Because you can use any integer expression as a subscript it is possible to process array elements easily and quickly.

Here is an example of the use of arrays for character manipulation.

CHEESE NIBBLER

10 DIMA(10)
20 DIMB(10)
30 DIMC(10)
100 FORJ=1TO10
110 LETA(J)=1
120 LETB(J)==1
130 LETC(J)=1
140 NEXTJ

200 FORJ=1TO10
205 IF NOTA(J)= 1 THEN GO TO 220
210 PRINT "•";
215 GO TO 230
220 PRINT"";
230 NEXTJ
240 PRINT

300 FORJ=1TO10

```
305 IF NOT B(J) = 1 THEN GO TO 320
310 PRINT"""";
315 GO TO 330
320 PRINT"";
330 NEXTJ
340 PRINT

400 FORJ=1T010
410 PRINT "•":
415 GO TO 430
420 PRINT"";
430 NEXTJ
440 PRINT


445 PRINT "HIT NEWLINE TO NIBBLE THE CHEESE-
450 INPUTY$
460 IF NOT Y$="" THEN GO TO 1000
470 CLS


500 LET I = RND(10)
510 LETK=RND(3)
520 IF K = 1 THEN LET A(I) = 0
530 IF K = 2 THEN LET B(I) = 0
540 IN K=3 THEN LET C(I) =0
550 GO TO 200

1000   STOP
```

When typing in a program like this where there are several similar lines the EDIT facility is very useful, because you can edit line numbers.

For example; after entering line 210, type:

EDIT

⇐

⇐

RUBOUT

3

NEWLINE

Statements 10 to 30 set up 3 arrays. A, B, and C of 10 elements each.

Statements 100 to 140 set all the elements of all arrays to 1.

200 to 240 examine each element of the array A in turn. If an element is equal to 1 a • is printed. If an element is not 1 a space is printed.

Similarly for 300 to 340 and 400 to 430.

On the first pass all the elements are set to 1 and • is printed all the way through.

445 to 470 print the instruction and if the user then hits NEWLINE the screen is cleared.

500 to 540 select a random element of a random array to be set to 0.

At 550(the program jumps back to print out all the arrays. The element which has been set to 0 is printed as a space — or nibble.

As more nibbles are taken there is a greater chance that the element selected will already be 0 and thus the effective nibble rate slows down as the game progresses.

ABS(n)   this function gives n if n>0, and -n if n<0. n is any integer expression.
e.g.   ABS(5) = 5
        ABS(-5) =5

CHAPTER

15

OVER TO YOU

OVER TO YOU

If you have read all the way through this book, running the programs in it and writing your own, you should be well on the way to becoming a fluent BASIC programmer. Remember, though, that there are many things you can do with ZX-80 BASIC which you can't do using other BASIC'S. Equally, some BASIC'S have features which are not present in ZX-80 BASIC.

The appendices present a concise summary of the error codes and the ZX-80 BASIC. Section 2 of Appendix 2 is written with experienced users in mind, but most of it does not require a detailed knowlege of the working of the central processor.

From now on it's up to you!

APPENDIX

1

ERROR CODES

ERROR CODES

When results are displayed, a code

n/m

is displayed also: n is the error number, m is a line number in the program

Stmt type	Code	meaning
	0	(m = next line that would have been executed) BREAK pressed. (m == —1 or —2) Command successfully completed. (m < 0 or > largest line number in program) GO TO m was executed. (m = largest line number in program) end of program.
NEXT	1	* NEXT <id> where <td> Is not the control variable of an active FOR loop.
Any	2	* variable name not found (any variable being used, or array name of an array element being assigned to).
Any	3	* subscript out of range, or error of any kind while evaluating a subscript.
LET INPUT DIM PRINT	4	* no room to add new variable or to assign a longer string to a string variable or to screen.
PRINT	5	* no more room on **screen.**
Any	6	arithmetic overflow (result > 32767 or < -32768, also result = -32768 in some circumstances.
RETURN	7	* RETURN with no corresponding GOSUB.
INPUT	8	INPUT can only be used in a program, not directly.
STOP	9	STOP statement executed.

* m == line number of offending statement.

CONTINUE is the same as GO TO m, except after code 9 when it is GO TO m+ 1.

APPENDIX

4K BASIC FOR ZX-80

1.ZX-80 user's view,
(a) Program Listing

The user inputs via the keyboard, lines of BASIC for insertion into the program and commands for immediate execution. While he is doing this he sees a display which is divided into two parts: the upper part is a "window" on the program listing, while the lower part displays the line or command he is currently outputting. Normally, the lower part is large enough to hold the whole line (which will take more than one screen tine if it contains more than 32 characters), there is a blank line between the two parts, and the upper part occupies the remainder of the screen. If, however, there is insufficient RAM to hold a disptay of this size (each character on screen occupies 1 byte of RAM*) the upper part of the display will be shrunk line by line until the display file is small enough. When the upper part has disappeared altogether the lower part shrinks character by character.

The computer maintains a "current line number"

* RAM stands for random access memory or store.

for editing the program and the display is always organised so that the line with that number, or the preceding line If no line with that number exists, is on the screen if at all possible. If there is a line with the current number, it is displayed with a symbol consisting of a reverse video **[>]** between its line number and the text of the line; if there is none then the. reverse video **[>]** will not appear.

Three keys are provided for changing the current line number: "⇑" changes it to the line number of the preceding line, "⇓" changes it to the line number of the following line, and "HOME" resets it to zero. If there is no preceding line "⇑" sets it to the line number of the first line in the program; similarly "⇓" will set it to the last line.

There are two other ways which the current line number can change: inserting a line into the program sets it to the line number of that line, and the command "LIST n" will set it to n.

When the current line is off the top of the screen, the window moves up so that it becomes the first line. When it is at or just off the bottom the window moves down a line. If it is well beyond the bottom of the window, the window moves *down so* that it becomes the second line on the screen.

(b) Input area

The lower part of the screen contains the line the user is currently typing in. This line may be a command or a line of program; in the latter case it will begin with a line number (in the range 1 to 9999) and in the former case there should be no number, although zero in practice counts as "no number" here.

Somewhere in the line a "cursor" is displayed. This indicates two things: the position in the line where symbols will be inserted, and whether an unshifted alphabetic key will be treated as a keyword (eg "LIST") or a letter (eg "A"). The cursor is in the form of an inverse video **[K]** for keywords or **[L]** for letters.

Note that this cursor, although displayed in the line and occupying a character position on the screen, does not form part of the line and is ignored by anything interpreting the line.

A second symbol, similar in principle to the cursor, may also be displayed: this is in the form of an inverse video [S] and indicates that the line is not a syntactically correct BASIC statement. It is positioned such that the part to the left of it could be the beginning (or the whole) of a syntactically correct BASIC statement eg if

"20 LET A = B + 5"

 is input from left to right then. **[S]** displayed at the end of

20 LET	and be absent from	2
20 LET A		20
20 LET A =		20 LET A=B
20 LET A=B+		20 LET A=B+5

In most cases the symbol is displayed as far to the right as is consistent with the above description; however there are a few circumstances where this is not quite so, for instance in

    LETA=ASC("X")

although "LET A=ASC" is a syntactically correct statement (ASC here being an integer variable) the **[S]** is not displayed after "ASC" but rather before it. This is because "ASC(. . .)" has already been identified as a function call, but as no built-in function with the name "ASC" is available it is faulted. Having identified it as a function call, however, the computer does not then consider other possible parses.

The following keys are available to alter the input line:

(i) single-character symbols: letters, digits punctuation, etc the symbol is inserted to the left

of the cursor.

(ii) multi-character "tokens": "**", "AND", "OR", "NOT", TO", "THEN" keywords.

Each of these is stored in the computer as a single byte, which, as in (i), is inserted to the left of the cursor. However, they appear on the screen as more than one character. Those that are alphanumeric (ie all except "**") are preceded and followed by a space, the preceding space being omitted (a) at the beginning of the line (b) Where it follows another alphanumeric token. (This rule means that programs appear well-laid-out on the screen without using up scarce RAM space for explicit space characters. Inserting an explicit space character before or after an alphanumeric token always inserts one extra space in the displayed form.)

(iii) "RUBOUT" deletes the symbol or token to the left of the cursor.

(iv) cursor control keys $\Rightarrow$ and $\Leftarrow$ skip the cursor past the next symbol or token to the right and left respectively.

(v) "EDIT" replaces the input line with a copy of the current line from the program. If no line has the current line number, the first line after it is used. If the current line is after the last line in the program, the last line is used. If there are no lines of program at all, then an empty line is used. Note that any existing input line is lost; "EDIT" followed by "NEWLINE" is in fact the quickest way to get rid of an unwanted line, but beware typing "EDIT" in mistake for "NEWLINE"!

(vi) "NEWLINE" is ignored if the inverse video **[S]** symbol is present. Otherwise it marks the end of input and the line or command is submitted to the system (see next section). Note that the whole line is submitted, not just the part to the left of the cursor.

(c) After input
When a line with a nonzero line number is submitted to the system, it is inserted into the program, any existing line with the same number being first deleted. The input area is then cleared. A special case is where the new line consists only of a line number, possibly preceded by spaces: the existing line (if any) is deleted but nothing replaces it and it therefore simply disappears from the listing. The "current line number" is still set to its number, however, so the inverse video **[>]** disappears also (see (a) above). If the line being inserted has one or more spaces after the line number but no other symbols or tokens, it is still inserted in the program and appears in the listing as a line which is blank except for its number; when the program is run such lines are ignored.

A line which has no line number, or which has line number zero, is a "command" and is obeyed immediately. For as long as it takes to obey the command (which for most commands is very brief) the screen is blank, then on completion the upper part of the display contains any output generated and the lower part contains a display of the form

   m/n

where m is a single digit and n is "—2" for most commands.

If m = 0, execution was successful; if m = 9 a STOP command was executed; otherwise m is an error code (see Appendix I).

Where a command (RUN, GO TO, GOSUB, CONTINUE) has caused the program to be entered, n is the line number of the offending instruction if m is an error code (exception: if the error is in a GO TO or GOSUB then n may be the target of the jump), the tine number of the STOP if m = 9,land the line number of the last line in the program if m = 0. Except in the case of m = 0 or m =9, CONTINUE is a jump toline number n (butsee3(c)). If m=9,CONTINUE is a jump to line number n +1.

Sometimes only the first digit of n is displayed because there is no room in the RAM for any more display file. For example beware confusing line number 240, of which only the first digit is displayed, with tine number 2.
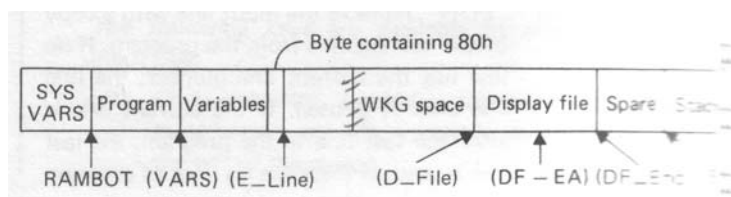
A jump to a line number which is beyond the end of the program, or greater than 9999, or negative, gives m=0, n= the line number jumped to. The commands are described individually in section 3.

2. Computer's view

N.B. 35h means Hexadecimal 35 (35h=3*16+5 =53. 0Ah to 0Fh are the decimal numbers 10 to 15.
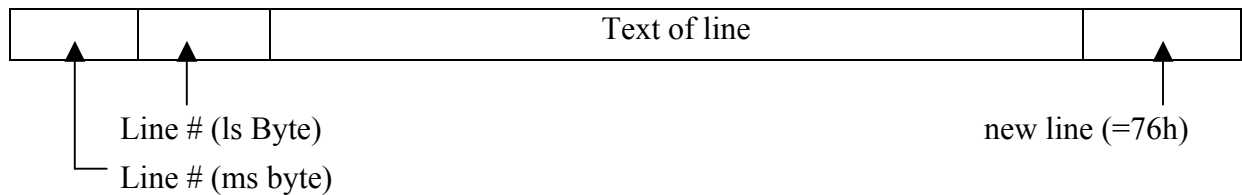
(a)RAM

The contents of the RAM are:

The first area is fixed in size and contains various "system variables" which store various items of information such as the current line number, the line number to which CONTINUE jumps, the seed for the random number generator, etc etc. Those that could possibly be useful

with PEEK etc have been documented elsewhere (Appendix 3).

An important subset of the system variables are the five contiguous words labelled VARS to DP _,END which hold pointers into the RAM and define the extent of the remaining areas (apart from the stack).

The program consists of zero or more lines, each of the form

		Text of line		

Line # (ls Byte)   new line (=76h)
Line # (ms byte)

ie beginning with the line number (stored ms byte first contary to the usual practice on Z80's) and ending with a newline. The line number is in the range 1 to 9999 so that the ms 2 bits of the first byte are zeroes. The ms 2 bits of the byte pointed to by (VARS) will not both be zeroes; this gives a simple test for end-of-program.
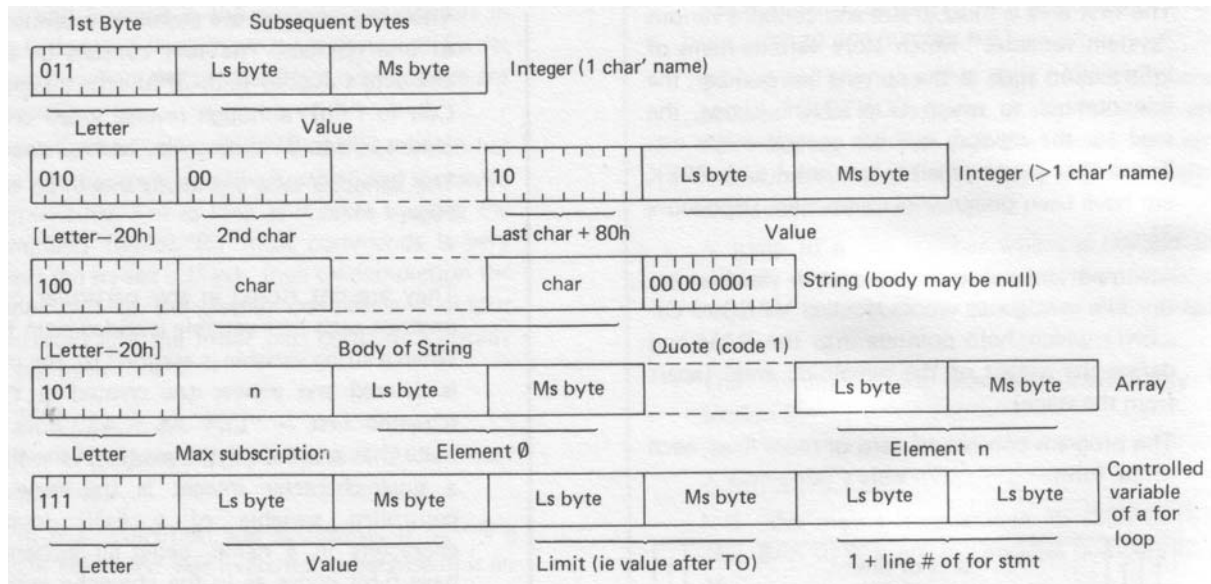
The program lines are stored in ascending order of line number. The text consists of ordinary characters (codes 0 to 3Fh) and tokens (codes CCfh to FFh), although reverse video characters (codes 80h to BFh) have also been allowed for.

The variables take the forms shown on the next page.

They are not stored in any particular order; in practice each new variable is added onto the end. When a string variable is assigned to, the old copy is deleted and a new one created at the end. (Created first - "LET A$ = A$" does work!) Note that apart from the ms bit of the first byte a single-character integer is the same as the controlled variable of a FOR loop. The characters in a name, being all alphanumeric, have 6-bit codes as in the character code table. The first character in a name, being perforce alphabetic (ie in the range 26h to 3Fh) effectively has a 5-bit code.

The "variables" area is terminated by a single byte holding 80h (which can't be the name of a string!).

The working space holds the line being input (or edited, hence "E—LINE") except when statements are being obeyed when it is used for

	1st Byte	Subsequent bytes			
	011	Ls byte	Ms byte	Integer (1 char' name)	
	Letter	Value			

- 010 | 00 | ... | 10 | Ls byte | Ms byte — Integer (>1 char' name)
  - [Letter−20h] | 2nd char | Last char + 80h | Value
- 100 | char | ... | char | 00 00 0001 — String (body may be null)
  - [Letter−20h] | Body of String | Quote (code 1)
- 101 | n | Ls byte | Ms byte | ... | Ls byte | Ms byte — Array
  - Letter | Max subscription | Element Ø | Element n
- 111 | Ls byte | Ms byte | Ls byte | Ms byte | Ls byte | Ls byte — Controlled variable of a for loop
  - Letter | Value | Limit (ie value after TO) | 1 + line # of for stmt

temporary strings (e.g. the results of CHR$ and STR$) and any other similar requirements. The subroutine X—TEMP is called after each statement to clear it out, so there is no need to explicitly release space used for these purposes.

The display file always contains 25 newline characters (hex 76); the first and last bytes are always 76h and in between are 24 lines each of from 0 to 32 (inclusive) characters. (DF—EA) points to the start of the lower part of the screen.

The stack (pointed to by register SP) has at the bottom (high-address end) a stack of 2-byte records. GOSUB adds a record to this stack consisting of 1+ its own line number; RETURN removes a record and jumps to the line number stored therein. The last 2 bytes of RAM contain a value which RETURN recognises as not being a line number.

The expression evaluator (which is also used to check the syntax of expressions) pushes 4 bytes onto the top of the stack for each intermediate result and pops them again when the appropriate operator is found, eg.

A+B*C**D*E+F*G $X=B\verb|^|C^D$

$Y=A+X*E$

stack	A+	B*	C**	X*	Y+	F
		A+	B*	A+		Y+
			A+			

Thus the above expression uses a maximum of 12 bytes of stack.

Parentheses use an additional 6 bytes each, eg

.

$$A+(B*C**D)*E+ F*G$$

would use $12 + 2 \times 6 = 24$ bytes

Apart from these two cases, the stack is only used for subroutine calls and for saving registers.

(b) Actions
The actions taken by the computer in response to the user's keystrokes are as follows.

Each time a symbol or token is inserted into or deleted from the input line, also each time the cursor is moved, this change is put into effect in the input line held in working-space after deleting the lower part of the display file (viz that part from (DF_EA to DF_END) - note that during this period the display file may be incomplete in that less than 25 newline characters are present, although the display file is never allowed to become large enough that there will not be room to add the remaining newline characters). Then the input line is checked to see if it is syntactically correct. The input line contains an inverse video **[K]** at the point where the cursor is; the syntax checker notes the address of the cursor in the variable (P—PTR) and sets variable (X—PTR) to point to the first wrong symbol or to zero if there is none. It also notes whether the cursor should be displayed as **[K]** or an **[L].** Finally the lower part of the display file is rebuilt, inserting the **[S]** symbol and changing the cursor from **[K]** to **[L]** if required, as well as converting tokens into characters.

If there is now insufficient room for the display file the display file area is cleared and the upper part is remade with fewer lines by re-copying from the program stored in the area RAMBOT to (VARS), again converting tokens into characters as part of this process, and the lower part is then output afresh.

When a line is to be inserted into the program, its line number is converted into binary, space is made at the appropriate place by copying everything else up, and the text of the line (from which the cursor has already been deleted) is copied in. The working-space and display file are then re-made, the former now containing just the cursor and a newline.

When a command is executed, it is interpreted in situ in the working-space area. Program lines are of course interpreted in their place in the program.

3. Statements

(a) expressions

Throughout section 3, "n" will be used to represent any space integer expression and "s" to represent any string expression.

String expressions are:

(i) string variable ά$ where ά is any letter (no intervening space allowed). Example: A$. May be used as "dest" for INPUT (see3(b)).

(ii) literal string delimited by quotes, any symbols or tokens other than quote permitted inside the string. Examples "ABC" "++**"

(iii) CHR$(n) the character with code n, or the null string if n=1. (Note: code 212 is a token which is represented by the quote character and this code may be supplied as the parameter to CHR$ to get something which prints as a quote)

(iv) TL$(s) the strind s minus its first character or the null string if s is null or contains only one character

(v) STR$(n) the decimal representation of n: from 1 to 5 digits leading minus sign if negative, no spaces – hence from 1 to 6 characters in all.

(vi) (s) parenthesized string expression.

Integer expressions are:

(i) integer varible first character alphabetic, subsequent characters alphanumeric, no embedded spaces, no limit on length of name. May be used as "dest"for INPUT (see 3(b)).

(ii) literal number decimal, unsigned, value must be <32768 (syntax error otherwise) as many leading zeros as you like. Examples 23   32767   0000032767

(iii) prefix and infix operations

form	priority	notes
n**n	10	"to the power"
-n	9	
n*n	8	
n/n	7	

n+n	6	
n-n	6	n-n behaves identically to n+-n
n=n	5	
n>n	5	
n<n	5	
s=s	5	Value is –1 if "true", 0 if "false"
s>s	5	
s<s	5	
NOT n	4	
n AND n	3	Bitwise Boolean operations
n or n	2	

Ambiguites in parsing operations are resolved by considering the priority of the operators in question: higher priorites bind tighter, equal priorities associate from the left. Example

-A**B+C*D/E*F-G-H

is the same as

(((-(A**B))+((C*D)/(E*F)))-G)-H

(iv) PEEK(n)   the value at address n, being a single byte in the range 0 to 255

(v) CODE(s)   the code for the first character in strings

(vi) RND(n)   a pseudo-random number in the range 1 to n if n>0, in the range 1 to 32767 or -32768 to n-1 if n<0. If n=0 returns 1 always.

(vii) USR(n)   call machine-code sub-routine at address n. Value is whatever the subroutine leaves in HL, or n if it doesn't alter ML.

(viii) ABS(n)   if n<0 then —n else n.

(ix) array element: a(n) Example: A(I-H).
                        May be used as "dest" for INPUT (see3(b)).

(x) (n)   parenthesized integer expression.

Values of string expressions can be of any length and can contain any codes except 1 (the closing quote). Values of integer expressions must be in the range -32768 to +32767; any value outside this range causes a run-time error (number 6). Note that relations yield —1 for "true" and 0for "false" and that

—1 AND n   is the same as n.
0ANDn   is the same as 0
—1 OR n   is the same as—1.

0 OR n   is the same as n.

also

NOT n   is the same as —n—1.

so that for instance

I AND I>0 0R -I AND I<0.

                is the same as ABS (I).

However constructions such as A>B>C do not have the obvious effect, being parsed as

(A>B)>C   ie as

(A>B) AND C<-1 OR (NOT A>B) AND C<0.

(b) Statements

The statements available are:

NEW   re-initialise the computer to the state it has at initial switch-on. Loses all
             program, variables, etc.

LOAD   read system variables, program. and variables, from tape, and re-initialise input
             line and remake display file. Does not affect any GOSUB blocks that may be on
             the stack (although the effect of doing a RETURN is unlikely to be sensible).
             BREAK after data has been found on the tape does the same as NEW:
             BREAK during the lead-in preserves the current program and variables.

SAVE   Write system variables, program, and variables to tape for subsequent reading by
             LOAD. Precedes data with 5 secs of silence; starting the tape before executing
             SAVE (which is recommended .practice) writes frame sync pulses to the tape.

The above can all be used in programs, but are intended to be used as commands, and use in
programs is not particularly sensible.

RUN n   is the same as CLEAR followed by GOTO n.

RUN   is the same as RUN 1.

CONTINUE   is the same as GO TO n, where n is the last number in an end-of-program
               message m/n with m>0 (see 1(c) and 3(c)). However, after message 9/n,
               CONTINUE is the same as GO TO n+1.

REM any text no effect (ie is comment).

IF n THEN   executes the statement unless n statement is zero. (ie unless n is "false").
statement

 INPUTdest   where "dest" can take any of the forms marked above (see 3(a)). Returns to the
               input mode with any output so far produced by the PRINT statement in the
               upper part of the screen, and the input line initialised to contain the cursor alone
               if "dest" is an integer and the cursor inside a pair of quotes if "dest" is a string.
               These quotes can, however, be deleted as the input can be any expression (not
               necessarily a literal). The expression is checked for correct syntax and
               "NEWLINE" is ignored if the **[S]** marker is present. The upper part of the
               screen consists of as many lines as have been written by PRINT and the lower
               part follows on rather than being placed at the bottom of the screen. It is
               possible for the upper part to contain 24 lines, in which case the lower part is
               not visible although the keys all function normally, and, provided the user is
               able to work blind, the input can still be submitted.
               The keys ⇑ and ⇓ have no effect; the effect of "EDIT" is rather comical and not
               very helpful, although it doesn't actually crash the system.

The INPUT statement cannot be used as a command because of the conflict in use of the
working-space; however, in this situation LET can be used instead. Using INPUT as a
command causes error code 8/—2.

If "dest" is an array element and there is an error in evaluating the subscript or the subscript is
out of bounds the error is not reported until after the input value has been submitted.

PRINT <list>
               in which <list> may be:
                   <empty>
                   or   <expr>

or   <list>,<list>

or   <list>; <list>

and <expr> may be s or n.

writes the value of each <expr> to the upper part of the screen, ie in the case of "s" writes the body of the string, converting tokens into characters, and in the case of "n" writes STR$(n). Each comma causes the output to tab to the 9th, 17th, or 25th column on the line of to the 1st column on the next line. A new line is output unless the <list> ends with a comma or semicolon.

LIST n   sets the "current line number" to n and enters "program input" mode without waiting to display any printed output error code (even if there is an error in evaluating n!) It is therefore only really suitable for use as a command.

LIST   is equivalent to LIST 0.

STOP   causes "error" code 9, so that CONTINUE will carry on from the following statement. Useful for displaying results when no input is required.

DIM a(n)   creates an array with name *a* and subscript range 0 to n inclusive. If one already exists, the new space will be reserved but the old array will continue to be used for all accesses. If a variable *a* already exists, it can still be used.

FOR *a* =

n TO n   assigns to *a* (ie an integer with a n TO n single-character name) a FOR block as described in section 2(a). The effect is that a subsequent "NEXT *a*" will increment a and if it is not now greater than the TO value jump to the statement following the FOR. Note that we always enter the body of the loop at least once, and that the association between FOR and NEXT is entirely dynamic, ie not lexical.

GO TO n   jumps to line number n, or to the next line with a line number greater than n, or if neither exists stops showing code 0/n.

POKE n,n   the first n is an address to which the value of the second (modulo 256) is written as a single byte.

RANDOMISE n set the seed of the pseudorandom number generator to n.

RANDOMISE as above with n = number of frames displayed on the TV since initialisation

(modulo 256).

CLEAR   delete all variables.

CLS   clear the upper part of the screen, eg to get rid of a message appropriate to an
              INPUT as in
              20 PRINT 'TYPE THE VALUE FORA".
              30 INPUT A.
              40 CLS.

GOSUB n   as GO TO n, but also push a GOSUB block on the stack so that RETURN will

           jump to the line following this one.

RETURN   pop a GOSUB block off the stack and jump to the line number contained in it.

NEXT *a*   equivalent to LETa=a+ 1.
                IF NOT *a* > (value supplied with
                TO)THEN
                GO TO (line following FOR *a)*

The combined effect of
             m FOR *a* = n-(TO n2 (m is the line number).

and

             NEXT *a*

is of

             m LET *a* = n,
             LET LIMIT*a*=n2

             (assuming LIMIT*a* isn't used elsewhere in the program).

and

             LET*a*=a+1
             IF NOT *a*>LIMIT*a* THEN GO TOm+1.

Note that this does not preclude (i) assignment to 8 (ii) several NEXTs matching one FOR, or
one NEXT matching several FORs.

Note also that FOR - GOSUB - NEXT -RETURN, and FOR I - FOR J - NEXT r ~ NEXT J
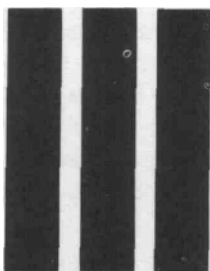are possible, though not very useful.

(c)BREAK

If the BREAK key is found to be pressed at the end of execution of a line, execution does not Follow on to the next line but stops showing 0/n where n is the line number of the next line that would have been executed but for the break-in.

0	space	*64	print as ?	*128	Inverse space	*192	print as ?
1	"	*65	print as ?	*129	Inverse "	*193	print as ?
2	GRAPHICS	*66	print as ?	*130	Inverse GRAPHICS	*194	print as ?
3	GRAPHICS	*67	print as ?	*131	Inverse GRAPHICS	*195	print as ?
4	GRAPHICS	*68	print as ?	*132	Inverse GRAPHICS	*196	print as ?
5	GRAPHICS	*69	print as ?	*133	Inverse GRAPHICS	*197	print as ?
6	GRAPHICS	*70	print as ?	*134	Inverse GRAPHICS	*198	print as ?
7	GRAPHICS	*71	print as ?	*135	Inverse GRAPHICS	*199	print as ?
8	GRAPHICS	*72	print as ?	*136	Inverse GRAPHICS	*200	print as ?
9	GRAPHICS	*73	print as ?	*137	Inverse GRAPHICS	*201	print as ?
10	GRAPHICS	*74	print as ?	*138	Inverse GRAPHICS	*202	print as ?
11	GRAPHICS	*75	print as ?	*139	Inverse GRAPHICS	*203	print as ?
12	£	*76	print as ?	*140	Inverse £	*204	print as ?
13	$	*77	print as ?	*141	Inverse $	*205	print as ?
14	:	*78	print as ?	*142	Inverse:	*206	print as ?
15	?	*79	print as ?	*143	Inverse?	*207	print as ?
*16	(	*80	print as ?	*144	Inverse (	*208	print as ?
*17	)	*81	print as ?	*145	Inverse)	*209	print as ?
*18	-	*82	print as ?	*146	Inverse -	*210	print as ?
*19	+	*83	print as ?	*147	Inverse +	*211	print as ?
*20	*	*84	print as ?	*148	Inverse *	*212	"
*21	/	*85	print as ?	*149	Inverse /	213	THEN
*22	=	*86	print as ?	*150	Inverse =	214	TO
*23	>	*87	print as ?	*151	Inverse >	215	;
*24	<	*88	print as ?	*152	Inverse <	216	,
*25	;	*89	print as ?	*153	Inverse;	217	)
*26	,	*90	print as ?	*154	Inverse,	218	(
27	.	*91	print as ?	*155	Inverse.	219	NOT
28	0	*92	print as ?	*156	Inverse 0	220	-
29	1	*93	print as ?	*157	Inverse 1	221	+
30	2	*94	print as ?	*158	Inverse 2	222	*
31	3	*95	print as ?	*159	Inverse 3	223	/
32	4	*96	print as ?	*160	Inverse 4	224	AND
33	5	*97	print as ?	*161	Inverse 5	225	OR

34	6	*98	print as ?	*162	Inverse 6	226	**
35	7	*99	print as ?	*163	Inverse 7	227	=
36	8	*100	print as ?	*164	Inverse 8	228	<
37	9	*101	print as ?	*165	Inverse 9	229	>
38	A	*102	print as ?	*166	Inverse A	230	LIST
39	B	*103	print as ?	*167	Inverse B	231	RETURN
40	C	*104	print as ?	*168	Inverse C	232	CLS
41	D	*105	print as ?	*169	Inverse D	233	DIM
42	E	*106	print as ?	*170	Inverse E	234	SAVE
43	F	*107	print as ?	*171	Inverse F	235	FOR
44	G	*108	print as ?	*172	Inverse G	236	GO TO
45	H	*109	print as ?	*173	Inverse H	237	POKE
46	I	*110	print as ?	*174	Inverse I	238	INPUT
47	J	*111	print as ?	*175	Inverse J	239	RANDOMISE
48	K	*112	print as ?	*176	Inverse K	240	LET
49	L	*113	print as ?	*177	Inverse L	241	?
50	M	*114	print as ?	*178	Inverse M	242	?
51	N	*115	print as ?	*179	Inverse N	243	NEXT
52	O	*116	print as ?	*180	Inverse O	244	PRINT
53	P	*117	print as ?	*181	Inverse P	245	?
54	Q	*118	print as ?	*182	Inverse Q	246	NEW
55	R	*119	print as ?	*183	Inverse R	247	RUN
56	S	*120	print as ?	*184	Inverse S	248	STOP
57	T	*121	print as ?	*185	Inverse T	249	CONTINUE
58	U	*122	print as ?	*186	Inverse U	250	IF
59	V	*123	print as ?	*187	Inverse V	251	GOSUB
60	W	*124	print as ?	*188	Inverse W	252	LOAD
61	X	*125	print as ?	*189	Inverse X	253	CLEAR
62	Y	*126	print as ?	*190	Inverse Y	254	REM
63	Z	*127	print as ?	*191	Inverse Z	255	?

* not available from the keyboard. Codes 38-63 only available when the cursor is **[L]**, codes 230-255 only when cursor is **[K]**

APPENDIX

SYSTEM VARIABLES

SYSTEM VARIABLES

The contents of the first 40 bytes of RAM are as follows. Some of the variables are 1 byte, and can be POKEd and PEEKed directly. The others are each 2 bytes, and have the low order byte at the given address (n, say) and the high order byte at the next address. Thus to poke value v at address n do

POKE n.v
POKE n+t,v/256

and to PEEK at the value in address n use the expression

PEEK(n) +PEEK(n+1)*256

if the value is known to be in the range 0 to 32767, and something like

LET MSB $^=$ PEEK(n+1)
IF MSB>127 THEN LET MSB = MSB-256.
LET VALUE == PEEK(n) + MSB*256

if it may be negative, ie. if n is 16412 or 16414. The notes at the left hand side of the table have the following meanings:

X   The variable should not be altered by POKE as this could cause the BASIC to
           hang up.

N   POKE will have no lasting effect, as the variable will be rewritten either at the
           end of the POKE statement or the next time "edit" mode is entered (either at the
           end of the run or to get some input).

1 or 2   number of bytes in variable.

U   "unsigned" number in the range 0 to 65535; the BASIC will treat values in the
           range 32768 to 65535 as -32768 to -1 respectively. These are the only variables
           likely to yield negative values.

Notes	Address	Contents
1	16384	less than the "run-time error number": PEEK yields 255 normally, 1 less than the error code if an error has already occurred in this statement, eg PRINT (10^6 AND 0) OR PEEK (16384) prints 5 because 10**6 overflows. Do not POKE any value other than 255 or 0 to 8. POKE 16384,255 does nothing to 8) causes error n+1. Thus

		POKE 16384, 8 is the same as STOP.
X1	16385	sundry flags which control theBASIC system
2	16386	Statement number of current statement: POKE has no effect unless this is the last line in the program
N2	16388	position in RAM of **[K]** or **[L]** cursor last line any input or editing was done
2	16390	Statement number of **[>]** cursor

X2	16392	VARS	
X2	16394	E-LINE	
X2	16396	D-LINE	See section 2a of Appendix 2
X2	16398	DF-EA	
X2	16400	DF-END	

X1	16402	number of lines in lower part of screen, including blank line separating the two parts
2	16403	statement number of first line on screen. The LIST statement sets this, and scrolling alters it
2	16405	address of character or token preceding the [S] marker.
2	16407	statement number to which CONTINUE jumps
N1	16409	sundry flags which control the syntax analysis
N2	16410	address of next item in syntax table (very unlikely to be useful).
U2	16412	seed" for the random number generator. This is set by the RANDOMISE statement, q.v., and updated each time RND is called: see note below.
U2	16414	number of frames displayed since the ZX-80 was switched on (more exactly, the remainder when this is divided by 65536). While a picture is on the screen, this number is incremented 50 times per second in the UK version, 60 times per second in the US version.
N2	16416	address of 1st character of 1st variable name in last LET, INPUT, FOR, NEXT, or DIM statement. Thus in LET ABC= PEEK(16416) ABC is assigned the remainder when the address of the A is divided by 256. (Not likely to be very useful).
N2	16418	value of the last expression or variable: in practice this is the parameter of PEEK so that PEEK(16418) yields 34 and PEEK(16419) yields 64 always
X1	16420	position on line of next line character to be written to screen: 33 = lefthand column, 32 = second from left, etc, up to 2 = righthand column. Also, 1 = 1st column in next line because current line is full, 0 = 1st column on next line because end-of-line has been signalled. Thus after aPRINT not ending in a comma or a semi-colon PEEK(16420) always yields 0. It only yields 33 if the screen is empty (eg after CLS).
X1	16421	position of current line on screen: 23 = top line, 22 = second line down, etc.

X2	16422	address of the character after the closing bracket of the call of PEEK, or of the newline at the end of the POKE statement.

The first line of program begins at address 16424.
The function RND generates a pseudo-random number from the current "seed" as follows:

let n be the seed.

if n is zero, take n = 65536 instead.

let m be the remainder when n*77 is divided
by 65537.

if m is 65536, take m = o instead.

the result of RND(x) is now x*m/65536.

the new seed is m.

The TAB function in printing can be implemented as follows:

```
1000 REM GO SUB; 1050 TABS TO
1001 REM COLUMN I
1010 REM GO SUB 1040 TABS TO
1011 REM COLUMN I OUTPUTTING
1012 REM AT LEAST ONE SPACE
1030
1040 PRINT";
1050 IF I+PEEK(16420) =33 THEN RETURN
1060 IF I = 0 THEN IF PEEK(16420)<2 THEN RETURN
1070 GO TO 1040
```

INDEX