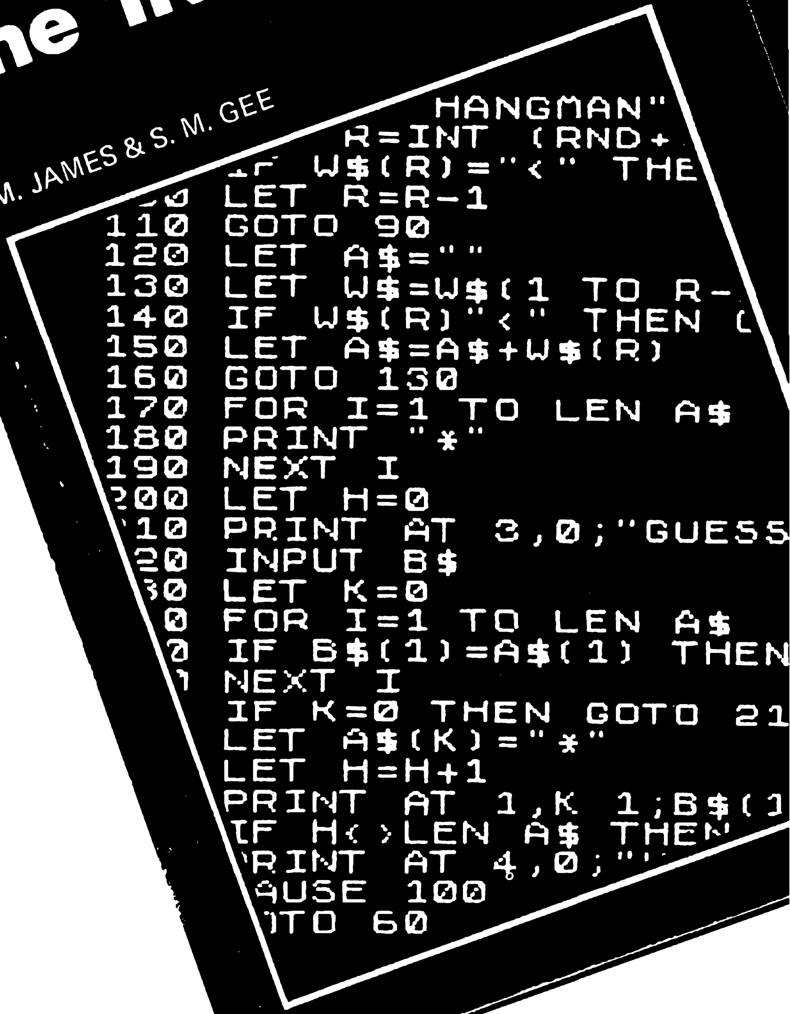
The Art of Programming the 1K ZX81

M. JAMES & S. M. GEE



LE

THE ART OF PROGRAMMING THE 1K ZX81

by
M. JAMES & S. M. GEE

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

Although every care has been taken with the preparation of this book, the publishers or author will not be held responsible in any way for any errors that might occur.

First Published – July 1982

British Library Cataloguing in Publication Data
James, M.
The art of programming the 1K ZX81 – (BP109)
1. Sinclair ZX81 (computer) – Programming
I. Title II. Gee, S.
001.64'24 QA76.8.562/

ISBN 0 85934 084 8

Printed and bound in Great Britain by Cox & Wyman Ltd, Reading

PREFACE

For many people the ZX81 is their first introduction to computing and they often find it difficult to write any interesting or exciting programs. This is not at all surprising since the ZX81 is actually quite a difficult machine to use – even for a fairly experienced programmer. Writing programs that fit into the tiny memory space that the 1K machine has left for them is a challenge and an art.

This book shows you how to use the features of the ZX81 in programs that fit into the 1K machine and are still fun to use. In Chapter Two we explain its random number generator and use it to simulate coin tossing and dice throwing and to play pontoon. There is a good deal of fun to be had, in Chapter Three, from the patterns you can display using the ZX81's graphics and its animated graphics capabilities, explored in Chapter Four, have lots of potential for use in games of skill, such as Lunar Lander and Cannon-ball which are given as complete programs. Chapter Five explains PEEK and POKE and uses them to display large characters. The ZX81's timer is explained in Chapter Six and used for a digital clock, a chess clock and a reaction time game. Chapter Seven is about handling character strings and includes three more ready-to-run programs – Hangman, Coded Messages and a number guessing game. In Chapter Eight there are extra programming hints to help you get even more out of your 1K ZX81.

We hope you'll find that this book rises to the challenge of the ZX81 and that it teaches you enough artful programming to enable you to go on to develop programs of your very own.

M. James and S. M. Gee

CONTENTS

Chapter One

MAKING THE MOST OF YOUR ZX81

The ZX81 really is a remarkably powerful and versatile micro. Considering its size and its price it is in a class of its own. Its smallness and its cheapness mean that it has quickly achieved a popularity that larger more expensive computers cannot yet hope for but they do *not* mean that it is less capable or less interesting. It is a machine to be reckoned with.

Artful programming

The 1K ZX81 is, however, not a particularly easy machine for the beginner to use. It is difficult to write programs that will fit into what little memory is left over once the ZX81's BASIC has taken its share. To write an interesting program, for a game, a simulation, or any other application that will fit into the available space means using tricks and devices that are not really part of introductory programming. This fact alone has probably caused many an aspiring programmer to give up and throw his ZX81 in the bin! After acquiring this book we hope that if your ZX81 is still in the bin you will recover it and make good use of it!

You may have read the BASIC manual that comes with the machine and learned enough BASIC to program but still not be able to produce any of the ideas in your head on the ZX81. The reason for this is that, although you may know BASIC, there is more to programming than knowing BASIC. In the same way there is more to speaking English than knowing how to pronounce a few words.

This book has been written to enable you to get the very most out of the 1K ZX81 (before you contemplate any extras). (A note to ZX80 owners: a lot of the programs will run with only minor modifications on the ZX80 with 8K BASIC ROM.) As mentioned above, in order to squeeze interesting programs into so little space requires artful programming – the programs in this book have been explained in such a way that you learn this art as you go along. In addition there are some special space-saving tricks and devices that are needed for the 1K ZX81 only and are not really anything to do with programming in general. These are explained in the last chapter of the book.

As you have probably gathered this is not an introductory book in the strict sense – it does assume you have already got your ZX81 working. If you are still a complete beginner the book that you should read first is the Sinclair Manual "ZX81 BASIC Programming", which comes with every machine. It provides a good foundation course for the beginner and our book goes on from where it finishes. However this will not stop us from re-explaining some of the more difficult topics to be found in the manual. All that we really require is that you have a very basic grasp of BASIC!

Fun and games

This is a book of games programs. Why just games? There are good reasons for the restriction. The Sinclair Manual provides an explanation of how to use the various routines for calculations and there is also a cassette of educational programs that fit into 1K. Although the ZX81 can be used for business and household management, programs serving this function do not fit into 1K. These are the negative reasons. The positive reason is that games can be designed to use all the capabilities of the ZX81. Also games provide perhaps the best medium for learning, and for learning in an enjoyable way. Games must not be dismissed as nonsense. The dividing line between sheer fun and the serious pursuit of knowledge is a very blurred one as computers have demonstrated. After all, the forerunner of the spaceship simulation games were really used to put men and spacecraft into orbit and land them on the moon.

When you've tried out the games in this book and understood how they work we hope that you will go on to write programs of your own. There are always ways of improving on games so perhaps you'll start by making small modifica-

tions to these programs, then larger ones and then you'll have the confidence to start from scratch and devise your own games. And along the way we hope you will have some fun.

Know your limits

The ZX81 has other limitations apart from the limited memory of the 1K version and these need to be taken into account when deciding what it is reasonable to expect it to do. Its graphics capability is 44 by 64 plotting points. This is by no means "low" resolution graphics but neither is it "high" resolution – compare the Apple II which in "high res" mode has 160 by 270 points. This means that the ZX81 cannot be asked to draw fine pictures, only fairly coarse ones.

The ZX81 may appear to be a bit slow. This is because the CPU has to handle the screen display as well as doing all the calculations involved. This is not true of other micros where this function is performed by dedicated electronics, in many cases a CRT controller chip. In the ZX81 a single chip is responsible for virtually everything so it can be excused for taking some time over operations.

The extent to which you can extend your ZX81 is limited. You can add 16K of RAM and you can attach a printer to make "hard copy" of your programs or their output. You can use a cassette deck to save and load programs but not to save and load data – since there are no commands to facilitate this in the ZX81's BASIC. There is, as yet, no way of attaching disc drives to the ZX81 and even if there were this would hardly seem sensible. For a start, disk drives are very expensive relative to the basic cost of the ZX81 – the very cheapest single disk drive costs twice as much as the ZX81 itself. There is a point beyond which it would be more sensible to sell your ZX81 and start again with a bigger machine.

However, do not be misled into thinking the ZX81 is not expandable. There are already a number of plug-in boards which connect via the slot at the rear of the case. Some of these extra boards allow you to attach others to them so that you can use more than one at a time. Available boards include ones for sound, memory expansion (to 48K) and for colour.

The ZX80

The ZX81 is the successor to Sinclair's first hand-held computer, the ZX80. Brought out in 1980 there were over 50,000 ZX80s sold which means that there must be lots of people still using them. At the same time as the ZX81 became available an 8K BASIC ROM chip was also produced as a replacement for one of the socketed chips in the ZX80. Using this, ZX80 owners are able to upgrade their machines to match many of the sophisticated features of its later counterpart and to make use of the peripherals – including the 16K RAM pack and the printer. One facility that the upgraded ZX80 still lacks though is animated graphics. However even this can be remedied as conversion kits to make the ZX80 produce moving pictures are now available. Except for those in Chapter Four, the programs in this book should work with only minor changes on an upgraded ZX80. However, as we have not tried them out on such a machine, we cannot comment on the sort of results that you might obtain.

Chapter Two

RANDOMNESS

You may think that randomness is a funny place to begin a book about games programming. After all, in the abstract it is rather an esoteric subject. But this book is not about abstract concepts, it is about using the features of the ZX81 to write programs that you can use to play games, and randomness, or chance, is a fundamental component of games of all sorts. For a start there are games of chance – games with cards or dice, then there are games in which the speed with which you react to a chance event counts, and then there are games in which you use your skill to beat an opponent whose decisions you cannot predict in advance. Random numbers are at the heart of all these games.

Pseudo randomness!

What is a random number? Well, we've already hinted at the answer. It is a number which you could not possibly have predicted. The toss of a coin is random, so is the fall of a dice. You cannot know in advance whether the coin will fall as "heads" or as "tails". Neither can you say which face of the dice will fall uppermost and there is no way you can control the outcome. It is the fact that the players cannot influence the result that is the important aspect of a random event as far as game playing is concerned.

You may already be questioning whether a computer can ever produce a random number – after all, it can only output a function of what has previously been input. Well, the sceptics are in this case correct – a computer cannot give you a truly random number. A computer can only produce numbers that it calculates. The most important feature that we use about randomness is that the next outcome, or number, is unpredictable. We can make the computer calculate a set of numbers such that it is very difficult to predict the next number that comes up. Such a set of numbers is said to be "pseudo random".

To clarify matters, a pseudo-random number is one that is not produced by a random event (such as the throw of a dice). It is therefore theoretically predictable, but it can, for all practical purposes, be generated in such a way that no onlooker could ever work out what to expect next. In this sense we could say that the computer generates unpredictable numbers rather than random ones.

A computer generates its random numbers by using a formula and so anybody who has a copy of the formula can predict the next number in the sequence. However, in practice the formula is sufficiently complicated that for the purpose of playing games, where everything happens quickly, you'd need to be a mathematical genius to apply the formula in time.

RND and RAND

The ZX81 uses the function RND to produce pseudo-random numbers in the range 0 to 1. Every time you use the word RND the ZX81 calculates the next number in the sequence. Perhaps now you will not be too worried by the idea of the next random number being calculated! The numbers that RND calculates have one other very important characteristic – every number between 0 and 1 has approximately the same chance of being produced. Another, and more technical, way of saying this, is to say that RND produced "uniformly distributed" pseudo-random numbers between 0 and 1.

Now that you understand what is going on when you use the RND function let's see how to use it. To demonstrate the sort of output you get when you just ask for a random number type:

```
10 PRINT RND
20 GOTO 10
RUN
```

You will get a screen of numbers all lying between 0 and 1. For example

.0011291504
.08581543

Are you struck by a coincidence? Have we just predicted the two numbers at the top of your screen? If not, turn your machine off and on again and then try. Now your screen should display a list starting with those very numbers. The reason this happens is that the sequence of random numbers is generated by the same formula for all our machines – and the formula itself is given in the manual so it's no secret! If you RUN the program more than once without switching off between times the sequence will simply continue from the place it was at before.

The fact that the random sequence is absolutely repeatable is actually very useful for some applications – for example for testing alternative simulations where you want to repeat the same pattern of chance events – but for other purposes it is entirely worthless. Playing games of chance with your ZX81 would soon lose its attraction if it was not for the RAND function. The RAND function gives an instruction to the computer where to start in the sequence. This can either be a set point or it can be a chance point – equivalent to sticking a pin in a list. To pre-determine the starting point you type

```
RAND any number
```

Try for example:

```
10 RAND 35
20 PRINT RND
30 GOTO 20
```

Try this a few times, the sequence is always the same. If you try:

```
10 RAND 35
20 PRINT RND
30 GOTO 10
```

You will find that you keep on printing the same number – the starting point of the sequence given by RAND 35!

To get a different sequence each time use:

```
10 RAND 0
```

When you use RAND 0 what actually happens is that the computer uses the value in an internal counter that counts the number of TV pictures displayed since you switched your machine on (see Chapter Six). Although where the sequence begins is actually related to the time your machine has been switched on, the fact that the counter operates at a speed of 50 counts per second means that it is virtually impossible to predict its position when you type RAND 0. To recap all we have learned so far: the RND function calculates the next random number in the sequence, we can use RAND to set the starting point of the sequence, we can use RAND 0 to set the starting point of the sequence in a way that is about as near to random as we can manage. To prove this try:

```
10 RAND 0
20 PRINT RND
30 GOTO 10
```

This will print out a set of starting points selected by RAND 0.

Making things happen

On the face of it, the string of numbers that comes up on the screen when we ask for a random number do not seem very useful. So let's take a simple application and see how to get the sort of results we want. Consider tossing a coin. There are two possibilities, "heads" and "tails". How do we simplify the raw output from the RND function to give one of these two answers and to give them fairly, i.e. with equal probability of the coin landing on either "heads" or "tails"? The solution is to split the range of answers into two exactly equal halves. As the range goes from 0 to 1, this is easy. The halfway point is .5

```
|------------|------------|
0           .5            1
```

As the numbers produced by RND are equally likely to fall anywhere on the line, half of them will fall below .5 and half will fall above .5. If we call a number that falls below .5 "heads" and one that falls above .5 "tails", then you can see that we will get as many heads as tails. Translating this into a program gives:

```
10 RAND 0
20 LET R=RND
30 IF R<.5 THEN PRINT "HEADS"
40 IF R>=.5 THEN PRINT "TAILS"
50 GOTO 20
```

At line 10 we randomise the starting point of the random numbers. At line 20 we get a random number into R and at lines 30 and 40 we decide which half of the range it falls in. If it's less than .5 we print "heads", if its greater than or equal to .5 we print "tails' . It's as easy as that! We haven't really written an inspiring coin tossing program so we will return to this problem a little later on.

What if you had a crooked penny? One that said heads threequarters of the time? It's not difficult to see how we would alter the program to give the results that the bad penny would give. Simply change the division of the range into two unequal parts. In general if the probability of getting heads is **P** then:

```
10 RAND 0
20 INPUT P
30 LET R=RND
40 IF R<P THEN PRINT "HEADS"
50 IF R>=P THEN PRINT "TAILS"
60 GOTO 30
```

For tossing a coin all we have to do is allow for two possibilities, each of which occur with the same probability – unless we deliberately alter the odds as in the bad penny example. There are other random situations where there are a larger number of possibilities and where the chances of the different outcomes are not equal to one another.

To the ordinary onlooker, the weather in this country often seems to be a matter of chance – or rather mischance. Let's write a program to see if the weathermen do get it right by forecasting using scientific principles, more often than if they just made an "educated guess". The guess combines a random element with our knowledge of seasonal weather patterns. The following program has been written for spring. In the 100 days of spring (the period from mid-February to mid-June) you might expect 40 sunny days, 30 cloudy days, 20 rainy days and 10 days of snowfall, putting this in terms of probability: sun on .40 of the days cloud on .30 of the days rain on .20 of the days and snow on .10 of them. Let's see how this fits together. Consider a line with 0 at one end and 1 at the other and mark off sections the same in length as the probability of each weather. For example:

```
          .4                .3            .2     .1
|---------------------|-----------------|-----------|-----|
0                    .4                .7          .9     1
```

If we produce random numbers evenly between 0 and 1 the probability of a number falling in any given section of the line is proportional to the length of that section. This is the key to selecting the weathers with the correct probability. The weather condition corresponding to the section in which the random number occurs is the one predicted. So for the number .6712348117 "Cloud" is given.

There is a problem with this. What if the random number is exactly one of the borderline points, say .4, will it be sunny or cloudy? It's not that important which we chose as long as we decide. The correct choice is in fact to give the 0 point to the first section and carry on giving the boundary to the section to the right. What about the point corresponding to 1? Well if you look carefully at the definition of RND you'll find that it gives numbers from 0 up to *but not including* 1. So the point corresponding to 1 need not concern us because it is never selected. The program for weather forecasting should now be obvious:

```
10 RAND 0
20 LET R=RND
30 PRINT "THE WEATHER FORECAST"
```

```
40 IF R<.4 THEN PRINT "SUN"
50 IF R>=.4 AND R<.7 THEN PRINT "CLOUD"
60 IF R>=.7 AND R<.9 THEN PRINT "RAIN"
70 IF R>=.9 THEN PRINT "SNOW"
```

The only difficult part of this program is testing which section the random number falls in but this should be understandable if you refer to the line diagram shown. There are lots of very tricky ways of carrying out the test, some of which save memory and some of which are faster, but the one used above is the easiest to understand and will work on any machine.

Random integers

Dividing up the interval between 0 and 1 is one way of selecting which "event" is going to happen. But it's not the only way. In a case where a given number of events occur with equal probability, there is an alternative – which is to multiply the number output by the computer by the number of possibilities, round it off to a whole number and add one to the answer. In fact this is easier than it sounds. Let's look at the practical example of throwing a dice. We have to choose one of six possibilities. We could use the method of dividing the line into six equal parts but instead let's try the new method. If we multiply RND by six we have a number that lies between 0 and less than 6. If we use the INT function to convert the number to an integer – a whole number – we have a number that lies between 0 and 5. Adding one gives a number between 1 and 6. To see this in action try the following program:

```
10 RAND 0
20 LET R=RND
30 PRINT R
40 LET R=R*6
50 PRINT R
60 LET R=INT (R)
70 PRINT R
80 LET R=R+1
90 PRINT R
```

If you run it a few times you should be able to see what's going on. Of course in practice you would carry out the whole procedure in one statement:

```
10 RAND 0
20 LET R=INT (RND*6)+1
30 PRINT R
40 GOTO 20
```

In general, if you want to produce random numbers between N and M use

```
10 LET R=INT (RND*(M-N+1))+N
```

Usually N is 1 and then this simplifies to

```
10 LET R=INT (RND*M)+1
```

if you put M=6 you get the dice program back again.

It is important to realise that this very easy method of producing random events *only* works if each of the events is equally likely.

Two improved programs

So far we have looked at randomness but we haven't really produced any complete games programs. The reason for this is that randomness is usually found as some part of a bigger program. Even so it is possible to do a better job with the two small programs that we examined earlier.

First let's have a look at the coin tossing program. One of the things that's usually missing from a computer coin tossing is the suspense. A coin is tossed . . . it flys through the air . . . it spins a bit . . . will it be heads . . . or tails . . . finally it stops! A computer tossing simply prints "heads" or "tails" faster than you can blink! Let's try to slow down the selection part of the program to give it an element of suspense. Try:

```
 5 DIM B$(2,5)
10 RAND 0
30 PRINT "DO YOU WANT TO GAMBLE Y/N?"
40 INPUT A$
```

```
50 IF A$<>"Y" THEN STOP
55 CLS
60 PRINT "HEADS OR TAILS (H/T)?";
70 INPUT A$
75 PRINT A$
80 LET R=INT (RND*15)+10
90 LET B$(1)="HEAD"
100 LET B$(2)="TAIL"
110 LET K=0
120 FOR I=1 TO R
125 LET K=NOT K
130 PRINT AT 5,0;B$(K+1)
140 FOR J=1 TO I
150 NEXT J
170 NEXT I
180 IF A$=B$(K+1,1) THEN PRINT "YOU WIN"
190 IF A$<>B$(K+1,1) THEN PRINT "YOU LOSE"
200 GOTO 20
```

e program works on a rather different principle to the ple coin tossing program. Lines 5, 90 and 100 set up a ng array containing the words HEAD and TAIL. The OR loop starting at line 120 and ending at 170 prints one of two words each time through. The statement at line 125 puzzle some readers – NOT K simply changes K to 0 if 1 and 1 if it's 0. It is this "flipping" of K each time ugh the loop that causes heads and tails to be alternatively ted out. If K=0 then line 130 prints "heads" if K=1 it ts "tails". This random element is introduced in line 80 re R is the number of times that the loop is carried out. usly if R is odd, then the final result will be heads; if it's then the result will be tails. The final touch of suspense is d by making the words "heads" and "tails" alternate and more slowly as time goes on by including a delay at lines 140 to 150.

This program is not easy to understand so don't worry too h if you cannot follow all of it. Some of the commands techniques will be described in more detail later on.

Our second improved program is a dice program. The improvement is obvious – we print out the usual dice patterns of dots for each result. We can save some programming by noticing that the pattern for three dots is the same as printing the pattern for two and the pattern for one. Similarly the pattern for four is the same as the pattern for two with two extra dots! And so on with five (four plus one) and six (four plus two extra dots). The resulting program is:

```
 10 RAND 0
 20 LET R=INT (RND*6)+1
 30 GOSUB R*100
 40 INPUT A$
 50 IF A$="S" THEN STOP
 60 CLS
 70 GOTO 20
100 PRINT AT 5,5;"*"
110 RETURN
200 PRINT AT 0,0;"*"
210 PRINT AT 10,10;"*"
220 RETURN
300 GOSUB 100
310 GOTO 200
400 PRINT AT 0,10;"*"
410 PRINT AT 10,0;"*"
420 GOTO 200
500 GOSUB 400
510 GOTO 100
600 PRINT AT 5,0;"*"
610 PRINT AT 5,10;"*"
620 GOTO 400
```

Examples:

```
*     *       *          *     *
   *             *       *     *
*     *            *     *     *
   5             3          6
```


The only clever bit of the program is line 30 which selects subroutine 100 if R is 1, subroutine 200 if R is 2 etc. To use the program, press newline for each throw of the dice and press "S" when you have finished using it.

The trouble with cards

So far we have used random numbers to select which of a number of events would happen. It might seem that we could use the same methods to write programs that play card games. A deck of cards consists of four suits each of 13 cards. There are many ways of using a computer to pick a card. One of the easiest to understand is to simply generate two random numbers, one between 1 and 4 to select the suit, and one between 1 and 13 to select which card. The problem with this method is that if you draw a card – the ace of spades say – there is nothing to stop you from drawing it again! This sort of drawing of cards is the same as drawing a card, noting its value and putting it back in the deck – it is drawing with replacement. The more usual way to draw cards is to deal them out and this is drawing without replacement. You can arrange for this sort of drawing to be programmed but it does take a lot of space – too much in fact for a 1K ZX81 to handle.

The second problem with cards is that anyone who is good at card games will tell you that a lot of the fun comes from working out odds and trying to remember the order in which the cards were dealt. Shuffling is a very inefficient way of rearranging the cards in a deck and if the last time around one card followed another then after shuffling the chances are that it will still follow the same card. It is the use of this fact that makes a good card player. Imagine then a good player's reaction to playing against a computer – there are no cards and the random draw is far too good to allow associations between pairs of cards to remain.

The solution to both the drawing without replacement and the inefficient shuffling problem lies in the computer simulation of a deck of cards. For example if you set up a string containing 52 different symbols – one for each in each suit – then dealing could be carried out by printing each symbol in turn. Randomness could be ensured by the occasional simulated shuffle. For example, for one suit:

```
 10 LET A$=" AH 1H 2H 3H 4H 5H 6H 7H 8H 9H
    10H JH QH KH"
 20 GOSUB 100
 30 LET I=1
 40 PRINT A$(I TO I+2)
 50 LET I=I+3
 60 IF I=13*3+3 THEN STOP
 70 GOTO 40
100 FOR I=1 TO 13
110 LET J=INT (RND*13)
120 LET B$=A$(J*3+1 TO J*3+3)
130 LET A$=A$(1 TO J*3)+A$(J*3+4 TO  )
140 LET A$=A$+B$
150 NEXT I
160 RETURN
```

The array at line 10 represents the suit of cards from AH – Ace of Hearts to KH – King of Hearts. Subroutine 100 carries out a simple shuffling by selecting a card at random and putting it at the end of the deck 13 times. Lines 30 to 70 print out each card in turn. Notice that this is slow and that it results in a not very good shuffle. If you want to, you can call the subroutine again for a more thorough shuffle – insert

```
25 GOSUB 100
```

As you can see by the above example good card games take a lot of memory and are really best left for the 16K ZX81. However, if you're not too worried about shuffling it is possible to program a simple card game such as pontoon.

Pontoon

```
10 LET T=0
20 LET U=T
30 CLS
40 LET A$="YOU"
50 GOSUB 400
```

```
 60 LET T=T+C
 70 IF T>21 THEN GOTO 300
 80 PRINT "YOUR ";T
 90 PRINT "STICK OR TWIST S/T"
100 INPUT A$
110 IF A$="S" THEN GOTO 200
120 CLS
130 GOTO 40
200 CLS
210 PRINT "YOUR TOTAL ";T
220 LET A$="ZX81"
230 GOSUB 400
240 LET U=U+C
250 IF U>21 THEN GOTO 300
260 PRINT "ZX81 TOTAL ";U
270 INPUT A$
280 IF U<T THEN GOTO 200
285 PRINT "ZX81 WINS"
290 GOTO 330
300 IF T>21 THEN PRINT "YOUR"
310 IF U>21 THEN PRINT "ZX81"
320 PRINT "BUST"
330 INPUT A$
340 GOTO 10
400 PRINT A$;" GET ";
405 LET C=INT (RND*13)+1
406 LET A$=" "+STR$ C
410 IF C=1 THEN LET A$="ACE"
420 IF C=11 THEN LET A$="JACK"
430 IF C=12 THEN LET A$="QUEEN"
440 IF C=13 THEN LET A$="KING"
450 PRINT A$
460 RETURN
```

This is a *very* simplified version of pontoon – it has to be to fit into 1K. The card values are ACE=1, JACK=11, QUEEN=12, and KING=13. The cards are drawn without replacement and without any reference to suit – line 405. To allow the ZX81 to draw cards you have to press NEWLINE for every card drawn.

Unequal probabilities – an advanced method

We can use a special feature of the ZX81 to generate a number corresponding to the interval into which a random number falls in the case of unequal sized intervals as well as equal sized ones. As we found earlier, if we want to generate four things with equal probability we can use

```
10 LET R=INT(RND*4)+1
```

but this will not work for unequal probabilities such as those used in the weather program. However:

```
10 LET R=RND
20 LET W= (R>.4) + (R>.7) + (R>.9) +1
30 PRINT W
40 GOTO 10
```

will produce numbers from 1 to 4 with the same (unequal) probabilities as the different weather conditions. It works because the ZX81 "works out" if tests such as R>.4 are true or false and uses 1 to mean true and 0 to mean false. To understand line 20 let's suppose that R is .5; R is bigger than .4 so the first bracket works out to be 1 but R is smaller in all the other tests so the second and third brackets work out to be 0. When you add together all the 1s and 0s – with the extra 1 – you get the answer that W is 2. If you try it for other values of R you can convince yourself that W is the number of the intervals that R falls in (starting at 1).

You could use this method to make the dice or coin tossing program given earlier unfair. But we leave this as a project for you to try for yourself!

Chapter Three

GRAPHICS

What do we mean by graphics? The answer to this question will become clear in this chapter but the most important point to grasp is that, as far as the ZX81 is concerned, graphics are not at all special. Indeed, the computer does not distinguish between text characters and graphics characters. This means that we can handle graphics using the commands we're already familiar with for displaying text on the screen.

Let's examine the ways we can make characters appear on the screen. There are actually two different approaches and both have their uses in graphics applications. The first relies on the PRINT command.

PRINT comma and semicolon

The PRINT command is actually a very versatile one so it's worth spending some time making sure that we understand its finer points. Using quotation marks we can print any character we choose on the screen. We also have the choice of where on the screen to print. Using the semicolon we can place items next to each other on the current line. Using the comma we can place items at the left hand margin or in the middle of the screen. The command that the comma issues is to place the item at the beginning of the next "print zone". There are two zones each having half the width of the screen. The first starts at the first printing position on a line and the second starts at the 16th. For example

```
10 PRINT "FIRST";"SECOND"
20 PRINT "FIRST","SECOND"
```

Notice that the semicolon doesn't even leave a *single* space between the two words. The comma can be used to save some of the ZX81's precious memory,

```
10 PRINT "A","B"
20 PRINT "C"
```

can be written as

```
10 PRINT "A","B","C"
```

because after using the second print field on the current line another comma is still taken to mean "use the next print field" even if this means move to the next line!

There is another important use of the semicolon. If a PRINT statement *ends* with a semicolon then it does not start a new line. For example:

```
10 PRINT "THIS IS";
20 PRINT "ON THE SAME LINE"
```

The main use of this sort of thing is when you have a lot of things to print out but they are generated at different places in the program or by a FOR loop. For example you can fill the screen with a character by:

```
10 FOR I=1 TO 32*21
20 PRINT "A";
30 NEXT I
```

PRINT TAB and AT

Although the careful use of the comma and the semicolon can handle most of our printing problems it is difficult to place something exactly where you want it on a line. This problem can be overcome by use of the TAB function. If you use TAB(N) in a PRINT statement the next thing to be printed will appear at column N on the current line. If the PRINT statement has already gone beyond column N then the next thing to be printed will appear at column N of the *next* line. To show this try the following program:

```
10 PRINT TAB(25);"AB";TAB(25);"AB"
```

Notice that although you could use a comma after the TAB command it wouldn't be useful because it would move the

print position on from where the TAB left it! If you use a value of N bigger than 32, then 32 is subtracted from it until it's in the correct range.

All of the PRINT commands that we have used so far have the limitation that they only allow positioning within the current line but there is a command, PRINT AT, that will let you print anything anywhere.

The AT command is very easy to use –

```
PRINT AT Y,X;"WORD"
```

will print WORD at row Y and column X. If there is already something printed at row Y and column X, it makes no difference, WORD replaces it. The ZX81's screen is 32 characters wide by 22 lines high. The first row is at the top of the screen and is numbered 0. The first column of characters is on the left hand side of the screen and is also numbered zero. This means that X must lie between 0 and 31 and Y must lie between 0 and 21. If X or Y are outside of their proper ranges then an *error* occurs. For a simple example of PRINT AT try:

```
10 PRINT AT 11,16;"*"
```

which will print a star in the middle of the screen. You can use PRINT AT to draw simple shapes on the screen. For example,

```
10 FOR I=0 TO 31
20 PRINT AT 2,I;"*"
30 NEXT I
40 FOR I=0 TO 21
50 PRINT AT I,0;"*"
60 NEXT I
```

will print a horizontal and a vertical line of stars. If you want to see another example of the use of AT go back and look at the improved dice program in Chapter 2. You can use as many ATs in a PRINT statement as you like and this can be used to shorten programs slightly.

Graphics characters

The ZX81 has 22 additional characters that form the basis of its graphics. These extra characters can be entered into PRINT statements exactly like any other characters except that you have to press the "SHIFT" and "GRAPHICS" keys first. This changes the ZX81 into graphics mode which is indicated by the cursor changing to a "G". Because there is a difficulty in reproducing the ZX81 graphics characters in listings, we will indicate graphics characters by square brackets around the letter on the key that you would press to produce it. So [A] is the graphics character that you can see printed on the "A" key on your keyboard.

To prove that graphics characters can be used in PRINT statements try

```
10 PRINT "[A]";
20 GOTO 10
```

which fills the screen with a grey mass.

CHR$

There is another way of producing graphics or any other type of character – the CHR$ function. If you imagine all of the ZX81's characters written out in order, you could pick out a character by saying "the 38th character". This is exactly what the CHR$ function does. CHR$(38) *is* the 38th character in the ZX81's character set. If you want to see all the character set try:

```
10 FOR I=0 TO 255
20 PRINT CHR$(I);
30 FOR J=1 TO 50
40 NEXT J
50 NEXT I
```

You will notice that sometimes nothing is printed – the character is unprintable – and sometimes you get more than you bargain for in the form of a whole word such as COPY or LET. The ZX81 treats all of the BASIC words that you can type in with one keystroke as a *single* character or symbol! The rule is that if you can type it with one key then it's one character.

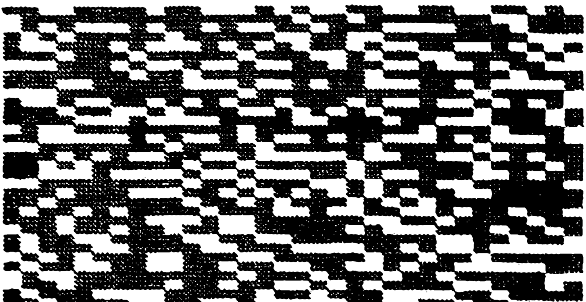
If you are very observant you will have noticed that the graphics characters in positions 0 to 10 come round again but in an inverted form at 128 to 138.

Random patterns

The last section has given us a very important link between numbers and characters. The function CHR$ accepts a number and outputs a character. We already know how to generate random numbers so using CHR$ we can generate random characters – in particular random graphics characters. For example,

```
10 PRINT CHR$((RND*2)+8);
20 GOTO 10
```

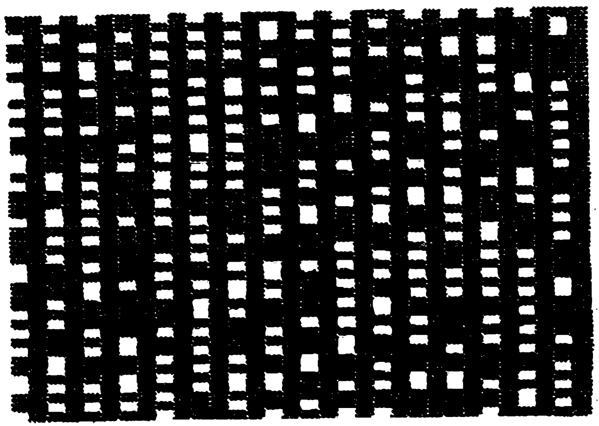
produces a random mixture of graphics characters 8,9 and 10, a typical sample of which can be seen below.



A slightly more interesting pattern can be made by including more characters,

```
10 PRINT CHR$((RND*2)+8);
20 PRINT CHR$((RND*2)+136);
30 GOTO 10
```

which prints a random selection of characters, 8,9,10 and their inverses.



Plotting

If you look at the graphics characters on the top two rows of the keyboard you will see that they take the form of a square divided into four quadrants. Each character has a different arrangement of quadrants coloured black or white. As has been stressed before, these characters can be printed on the screen just like any others. However, if we want to, we can use them to increase the resolution of our graphics screen. By selecting the right character we can make lines and shapes from the smaller quadrants rather than the full character squares. The trouble is that selecting the right character is not at all easy. So the ZX81 provides two commands that are just the job. The command PLOT X,Y will select the correct character to make the quadrant at X,Y white and UNPLOT X,Y will select the correct character to make the quadrant at X,Y black. The only thing that's missing in our description is where the quadrant at X,Y *is*! The ZX81 numbers the horizontal quadrants starting at 0 on the left hand side of the screen just as in the case of the full character positions. You should be able to work out that the last horizontal quadrant is 63. Vertically things are a bit more difficult. The first vertical quadrant is numbered 0 but is at the *bottom* of the screen – the opposite of the first full character position. Again the number of the last vertical quadrant should be obvious – 43.

To recap: by use of the "quadrant" graphics characters we can double the screen resolution. The ZX81 PLOT and UNPLOT command do the selection of characters automatically and give us a "graphics" screen 0 to 63 horizontal and 0 to 43 vertical. The quadrant called 0,0 is in the *bottom* left hand corner.

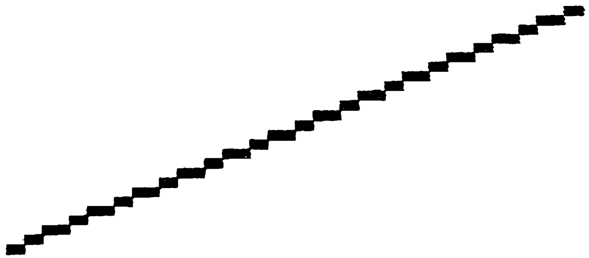
Some simple shapes

Using the PLOT and UNPLOT commands and some simple equations we can draw regular shapes.

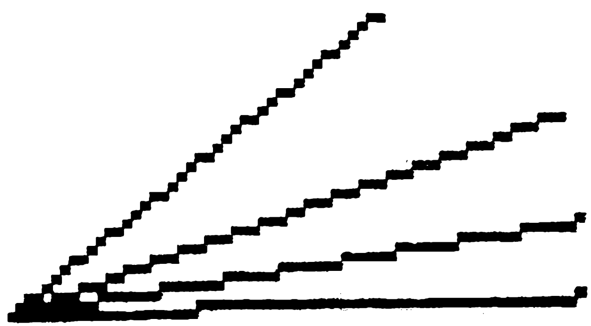
A computer like the Apple has a single command called HPLOT that allows you to draw a straight line between two points. The ZX81 lacks this command but it's not too difficult to make one up! Let's suppose that we want to draw a line from X1,Y1 to X2,Y2. A little bit of maths that need not worry us too much shows that for any point on the line Y=M*X+C, with M=(Y2–Y1)/(X2–X1) and C=Y1–M*X1. But for the moment let's forget about the values of M and C, and just decide *which* two points the line will pass through. The important part is the equation Y=M*X+C. This allows us to plot different lines for different values of M and C. Try the following:

```
10 LET M=.4
20 LET C=10
30 FOR X=0 TO 63
40 PLOT X,M*X+C
50 NEXT X
```

produces the following:



Notice that although it is recognisable as a straight line it's no
a very adequate straight line – the ZX81's graphics are goo
but not that good! To gain confidence, it's a good idea to tr
out the previous program with various values of M and C. Yo
should discover that M alters the slope of the line and C move
it up and down. You should also notice that sometimes yo
get an error message because for some values of X the Y valu
is off the screen.



To return to the problem of drawing a line between tw
points, let's try to draw a line between 10,10 and 20,20.

```
10 LET X1=10
20 LET Y1=10
30 LET X2=20
40 LET Y2=20
50 LET M=(Y2-Y1)/(X2-X1)
```



```
60 LET C=Y1-M*X1
70 FOR X=X1 TO X2
80 PLOT X,M*X+C
90 NEXT X
```

he things to notice about this program is that the FOR loop
line 70 goes from X1 to X2 and that the values of M and C
e worked out only once *before* the line is plotted. A more
ficient way of doing the same thing would be to work out M
d C (using the ZX81 as a calculator) and just setting them to
e result. For example, in the above program M works out to
and C works out to 0 so we could have used,

```
10 FOR X=10 TO 20
20 PLOT X,X
30 NEXT X
```

ich is a lot less demanding of space!

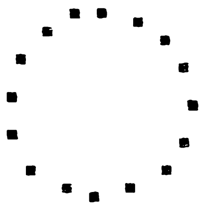
The next simple shape that we need to know how to draw
a circle. Once again our problem is solved by an equation. If
want to draw a circle at X1,Y1 of radius R, then any point
the circle satisfies the two equations Y=R*COS(T)+Y1 and
=R*SIN(T)+X1 for some value of T. It doesn't matter if you
't understand the above equation, you can still make use of
The important point, though, is that if T is given a value
can generate a point on the circle. Try the following
ram:

```
10 LET R=10
20 LET X1=31
30 LET Y1=21
40 LET T=RND*6.283
50 PLOT R*SIN(T)+X1,R*COS(T)+Y1
60 GOTO 40
```

should see a circle appear on the screen in random order.
ave used RND to produce random values of T and hence
om points on the circle. If we want to we can start with
and, by increasing T slowly plot all of the points on the
e. It just so happens that the circle will join up with itself
T reaches the odd value of about 6.283 and this happens

to be twice the value of PI. (There is a deep and very good reason for this but it need not worry us.) We can use the PI key on the ZX81 instead of the rough approximation:

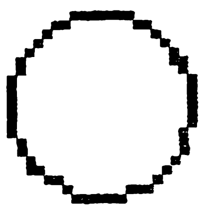
```
10 LET R=10
20 LET X1=31
30 LET Y1=21
40 FOR T=0 TO 2*PI STEP .4
50 PLOT R*SIN(T)+X1,R*COS(T)+Y1
60 NEXT T
```



If you run the above program you should see a circle appe and if you look very carefully you should see the order i which the points are plotted. The distance between each poi is governed by the STEP size. If we reduce the STEP size w can make the points meet and the result is a continuous circl If you change line 40 in the last program to,

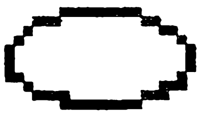
```
40 FOR T=0 TO 2*PI STEP .1
```

you get,



The last simple figure that we shall deal with is the elips This may sound like the sort of shape that you would nev require in games programs but once you realise that the elipse is the shape that you see if you view a circle at an angle its importance becomes obvious. If you think of an elipse as a flattened circle then the way to draw it also becomes obvious:

```
10 LET X1=31
20 LET Y1=21
30 LET R1=10
40 LET R2=5
50 FOR T=0 TO 2*PI STEP .1
60 PLOT R1*SIN(T)+X1,R2*COS(T)+Y1
70 NEXT T
```



To draw an elipse we use the same equation as a circle but use two values for the radius – a horizontal radius R1 and a vertical radius R2. If you alter these values in the program you should be able to produce elipses of different shapes.

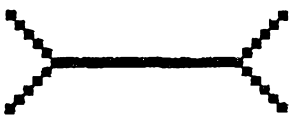
Arrows game

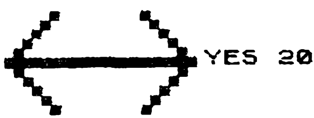
A simple game based on graphics is the arrows game. Two arrows are printed, one with inward pointing ends and one with outward pointing ends (see sample output). The object of the game is to say if the second arrow is the same length, shorter or longer than the first. This sounds easy but because of the well known visual illusion the second arrow always looks shorter than it really is! Try:

```
10 LET L=INT (RND*5+19)
20 LET Y=35
30 FOR I=15 TO 20
40 PLOT I,Y
50 PLOT 60-I,Y
60 PLOT I,60-Y
```

```
 70 PLOT 60-I,60-Y
 80 PLOT I+5,Y-25
 90 PLOT I+5,45-Y
100 PLOT I+L,40-Y
110 PLOT I+L,20-Y
120 LET Y=Y-1
130 NEXT I
140 FOR I=20 TO 40
150 PLOT I,30
160 NEXT I
170 FOR I=20 TO L+20
180 PLOT I,10
190 NEXT I
200 INPUT A$
210 IF A$="E" AND L=20 THEN PRINT "YES ";
220 IF A$="L" AND L>20 THEN PRINT "YES ";
230 IF A$="S" AND L<20 THEN PRINT "YES ";
240 PRINT L
```

If you run the program two arrows will be drawn, the first is always 20 units long. The second is either the same length, shorter or longer depending on the random length L set in line 10. If you think that the two lines are of equal length, then type E, if you think that the second is shorter, type S and if you think that the second is longer, type L. If you're correct the word YES is printed next to the second arrow. Once you've made your guess the true length of the second arrow is printed by line 240.





Randomness and symmetry

No doubt you have seen the fascinating displays of continuously changing patterns that other computers produce. Well the ZX81 can do the same sort of thing.

Let's start with a truly random pattern:

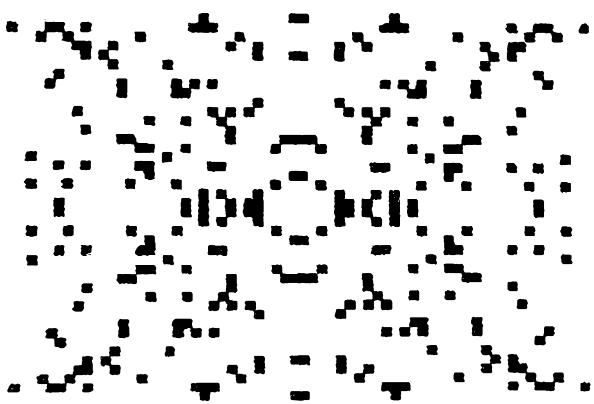
```
10 LET X=RND*63
20 LET Y=RND*43
30 PLOT X,Y
40 GOTO 10
```

The results of this program are interesting but hardly the type of pattern that you could watch for long. The trouble is that the pattern is too random. Interesting and ever changing patterns must use randomness for variety but they must use it in a controlling way. One of the basic organising principles in nature is symmetry and this can be used in computer patterns to introduce order.

On the 1K ZX81 you can only really handle fourfold symmetry but this is quite powerful enough to produce many interesting patterns. Fourfold symmetry is best understood by thinking of the ZX81's screen split into four quarters. If a point is plotted in the first quarter then a symmetrical pattern

will be produced if it is also plotted in the other three regio
If you imagine that the two lines that divide the screen int
four quarters are mirrors then the position of the other thr
points can be thought of as mirror images of the original.
the co-ordinate of the original point in the first quarter is X,
then the co-ordinates of the mirror images are 63–X,
X,43–Y and 63–X,43–Y. The best way to see that this is tr
is to use a piece of graph paper to draw the screen and wo
out the co-ordinates. Using these simple facts we can write
kaleidoscope program:

```
10 LET M=63
20 LET N=43
30 LET X=RND*M/2
40 LET Y=RND*N/2
50 PLOT X,Y
60 PLOT M-X,Y
70 PLOT X,N-Y
80 PLOT M-X,N-Y
90 GOTO 30
```



You may notice a number of strange jumps in the picture
the screen fills. This is a product of the way the ZX81's scr
display works and is nothing to worry about. The only trou
with this program is that it normally ends with an error!

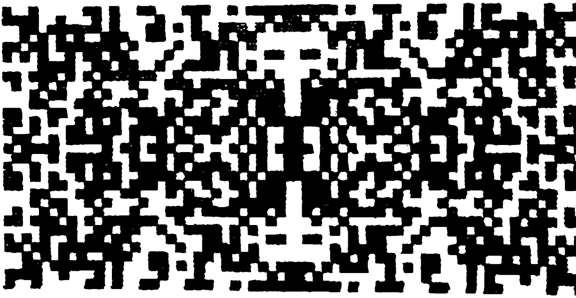
eason for this is that the screen fills slowly to the point at
hich all of the memory is used up. A solution to this problem
s to make the area of the screen that is actually used smaller.
f you change line 20 to LET N=35 then the program will run
ntil the screen is completely filled with plotted points.

Using this basic idea of fourfold symmetry it is possible to
dd other controlling features to make interesting patterns.
or example, it would be nice if the pattern didn't fill up and
at eventually there were both black and white areas. This is
sy, simply alternate a PLOT with an UNPLOT. It also might
interesting if the random changes went in "cycles" starting
the middle and working out. Putting these two ideas
gether gives:

```
 10 LET M=63
 20 LET N=35
 30 FOR X=0 TO M/2
 40 LET Y=RND*N/2
 50 PLOT X,Y
 60 PLOT M-X,Y
 70 PLOT X,N-Y
 80 PLOT M-X,N-Y
 90 LET Y=RND*N/2
100 UNPLOT X,Y
110 UNPLOT M-X,Y
120 UNPLOT X,N-Y
130 UNPLOT M-X,N-Y
140 NEXT X
150 GOTO 30
```

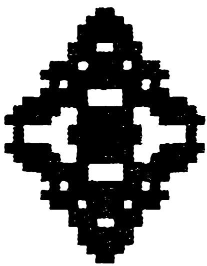
result of this program is difficult to capture in print
use it depends on movement, but a typical output might
like the pattern shown on the next page.

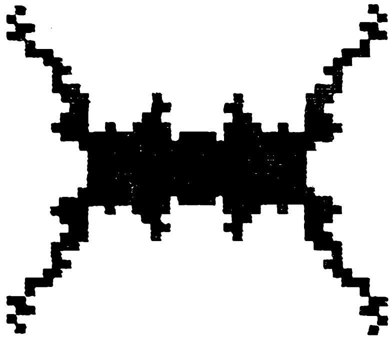
Randomness and symmetry can be used to produce
rns other than the usual "spotty" sort. If we start off
a point in the center of the screen and let it wander
d randomly by adding –1, 0 or 1 to each of its co-
ates we have a fairly interesting random line. But if we
use the fourfold symmetry routine to reflect the line into
quarter, the result is a collection of fascinating shapes,



some samples of which are shown.

```
 10 LET M=63
 20 LET N=35
 30 LET X=M/2
 40 LET Y=N/2
 50 GOSUB 100
 60 LET X=X+RND*2-1
 70 LET Y=Y+RND*2-1
 80 GOTO 50
100 PLOT X,Y
110 PLOT M-X,Y
120 PLOT X,N-Y
130 PLOT M-X,N-Y
140 RETURN
```





We could continue for a lot longer with random patterns but we'll leave the rest of the subject for you to explore for yourselves.

Chapter Four

MOVING GRAPHICS

One of the most rewarding areas of computing is dynamic o moving graphics. It is not at all obvious how you can mov from plotting a single point somewhere on the screen t making a moving display. In fact the transition is not at a difficult.

From flashing to moving

If you plot a single point and then unplot it again you will se a flashing dot. Try the following program:

```
10 PLOT 20,20
20 UNPLOT 20,20
30 GOTO 10
```

You can get the same effect by printing a reversed blan CHR$(128) and then printing at the same place a norm blank, " " or CHR$(0). Try:

```
10 PRINT AT 10,10;CHR$(128);
20 PRINT AT 10,10;CHR$(0);
30 GOTO 10
```

The flashing square produced by the use of the PRINT sta ment is four times bigger than that produced by PLOT a flashes faster because the ZX81 has to do a lot less work PRINT than to PLOT.

In either case, to alter the flashing rate you have to a delay loops. Delay loopS because if you try putting only o FOR loop in, say at line 15, then the time the point is "on increased but not the time it is "off". To lengthen the "o time you also need a FOR loop at line 25. For example try:

```
10 INPUT ONTIME
20 INPUT OFFTIME
```

```
30 PRINT AT 10,10;CHR$(128);
40 FOR I=1 TO ONTIME
50 NEXT I
60 PRINT AT 10,10;CHR$(0);
70 FOR I=1 TO OFFTIME
80 NEXT I
90 GOTO 30
```

…lay loops will be explained in Chapter Six. For now, try …ning the program with different values for "on" time and …ff" time, say 20 or 30.

We now know all there is to know about making things … — but what about move? Well the extension from …ing to moving is easy. If you plot a point and then unplot …nd then plot the point next to it, it looks as though the …int" has "moved". If you keep on repeating the process … point can be made to appear to move continuously. For …mple let's suppose that we want to make a dot move from … side of the screen to the other in a straight line. We know … to describe a straight line from the last chapter but let's … something a little easier first. If the point moves hori…ally then we simply have to increase the X co-ordinate … time we plot. Try the following:

```
10 LET Y=35
20 FOR X=0 TO 63
30 PLOT X,Y
40 UNPLOT X,Y
50 NEXT X
```

…orks in exactly the way we described. Line 30 plots a … line 40 unplots it; then line 30 plots the point next door … on!

…en using this method you have to make sure that every… happens at just the right time to give the impression of …ent at the speed that you want. In this simple example … are two times that matter, the time between plotting and …ing a point and then the time between unplotting the … and plotting the new point. A diagram might help to … this clear:

$\longleftarrow t_1 \longrightarrow\longleftarrow t_2 \longrightarrow$

--- | ---------- | -------------- |

point plotted   point unplotted   next point plotted

The time t_1 is the time that *any* point is displayed for and t_2 i the time that there is *no* point visible on the screen. The tota time, $t_1 + t_2$ is the time it takes to move from one point t another and this governs how fast the point is seen to move What most books on moving graphics don't tell you is wha values t_1 and t_2 should have to produce a smooth display. Th answer is not an easy one and in practice it is normal to chang the program's values of t_1 and t_2 to produce the best possib display. It is easy to see what the values of t_1 and t_2 should b in theory. If you were watching a point moving behind a gri of holes then the time the point would be visible would corres pond to t_1 and the time that the point would be hidden woul correspond to t_2. If the grid of holes were close together an regular you would still be able to see the point "moving" because the human brain tends to interpret a sequence o images as movement. What we are doing with the flashin moving point on the ZX81's screen is to copy the principle o an object hidden behind a grid of holes and rely on the fac that the brain is fooled into seeing movement. The quality o the apparent movement on the screen can be related to ho close we get to copying what is seen through the grid. If th holes are very close together then the time that the point wi be seen will be large and the time that it will be hidden will b small. Put another way t_1 will be much greater than t_2. This i the condition for producing smooth movement on the ZX81' screen. Unfortunately this is not easy to satisfy. The ZX8 takes as long to PLOT as to UNPLOT so t_2 tends to be as lo as t_1. Indeed the method that we are using makes thin worse. We PLOT a point, then UNPLOT it, then do some ca culation before we re-PLOT it. This actually means that t_2 i much longer than t_1. The result of this inbalance in "on" an "off" times is that the moving point tends to "twinkle" o flash as it moves. We could improve on this by PLOTing th point, doing the calculation and then UNPLOTing the ol point and PLOTing the new point. The trouble is that th quires storing the old position and the new position of the lotted point and in a 1K ZX81 this might be too much to add an already big graphics program. To see if the improvement worth it try:

```
10 LET Y=35
20 FOR X=0 TO 62
30 UNPLOT X,Y
40 PLOT X+1,Y
50 NEXT X
```

simpler method of making the movement smoother is to rease t_1 by putting a time-wasting statement, such as 35 T Y=Y, between PLOT and UNPLOT. This is the best thod to use with the ZX81 but if you have a 16K ZX81 might like to try some other methods.

There is something else that we can learn by thinking about ing a moving point through a grid of holes. If the point is ng as a speed S and is invisible for a time t_2 the distance een the holes must be $S*t_2$. This suggests that, as our lem with the ZX81 is that t_2 is too big for a smooth ay, we might be able to do better by increasing the nce between the displayed points! This can be done by ly plotting a point, then unplotting it and *instead* of ng its next door neighbour, plotting a point further . Try the following program:

```
10 LET Y=10
20 FOR I=1 TO 10
30 FOR X=1 TO 31 STEP I
40 PRINT AT Y,X;CHR$ 128
50 PRINT AT Y,X;CHR$ 0
60 NEXT X
70 NEXT I
```

me through the loop 30-60 a point moves across the from left to right. The first time the distance between points is one, then next it is two and so on until the is 10. What is interesting about this example is that, hen the point jumps by 6 or 7 points in one go, the of movement remains and the smoothest movement is d with something around a step size of 2 or 3.

Although this larger step movement is interesting it is no
always useful because many dynamic graphics programs nee
the point to move only one step at a time.

Moving balls and velocity

Now we've seen how to make a point move around the scree
let's consider how to use it in more exciting and interestin
ways. For a start we could plot and replot around the circum-
ference of a circle or a square to make a point move in othe
than straight lines. However, let's look at a more realistic appli
cation. For dynamic games it would be useful to simulate th
movement of a ball. This is best done by defining two veloci
ties with which the ball is moving. At each movement step th
plotted point (or ball) can move a number of places hori
zontally and a number of places vertically. Each step takes th
same amount of time, so we can call the distance it move
horizontally the horizontal velocity and the distance it move
vertically the vertical velocity. Thus at each movement ste
the horizontal velocity is added to the X co-ordinate and th
vertical velocity is added to the Y co-ordinate. Try th
following program:

```
10 LET V=1
20 LET H=1
30 LET X=0
40 LET Y=0
50 PRINT AT Y,X;CHR$ 128
60 PRINT AT Y,X;CHR$ 0
70 LET X=X+H
80 LET Y=Y+V
90 GOTO 50
```

This program moves a ball from the top left of the screen t
the bottom right and then off the screen. Because the ba
shoots off the screen the program ends with an error. Th
obvious thing to do is to let the ball bounce around the edg
of the screen – but how? The answer is surprisingly ea
because we have chosen to use the horizontal and vertic
velocity idea. If the ball meets a vertical wall, i.e. the rig

nd edge of the screen, then it cannot carry on moving in the me horizontal direction. In fact nothing but a complete versal of horizontal velocity will stop it going through the all! The vertical velocity is not affected by meeting a vertical all – why should it be?! So the rule is: when the ball meets a rtical wall reverse the horizontal velocity. Similarly when the l meets a horizontal wall reverse the vertical velocity. Using se two rules we have:

```
 10 LET V=1
 20 LET H=1
 30 LET X=0
 40 LET Y=0
 50 PRINT AT Y,X;CHR$ 128
 60 PRINT AT Y,X;CHR$ 0
 70 LET X=X+H
 80 LET Y=Y+V
 90 IF X=0 OR X=31 THEN LET H=-H
100 IF Y=0 OR Y=16 THEN LET V=-V
110 GOTO 50
```

s a remarkably simple program for the effect it achieves. s 90 and 100 test for the presence of a horizontal or cal wall. If one is found then the appropriate velocity is rsed. (If you haven't already worked it out, reversing a city is the same thing as putting a minus sign in front of Because of their different ways of numbering the screen ions, if you're using PRINT AT statements positive cities take you from the *top* to the *bottom* of the screen, if you're using PLOT statements they go from *bottom* to

ow that you know how to make a ball move and bounce ight seem sensible to try to write a bat and ball type game. all, striking the ball with a bat follows the rules for sing velocity as does the ball striking a wall. If you do try ite this sort of program in a 1K ZX81 you find that you out of memory very quickly and have to reduce the area e screen that's being used to the point that the game isn't nteresting. If you've followed the ideas in the bouncing you should be able to see how to program ball games and

you might even think up a game that is simple enough to fi
into the 1K ZX81.

Free flight and gravity

The previous section discussed moving a ball around inside
frame and how it could be made to bounce. There is anothe
way in which a ball can move – it can be thrown through th
air. Let's try to find a way of making a ball move under th
influence of gravity.

In outer space, where there is no gravity, a ball set movin
in a particular direction with a particular velocity will carry o
moving in the same direction and at the same velocity fo
ever! (Unless it hits some other object and then it woul
bounce off in the opposite direction at the same velocity lik
the ball in the previous section.) In this sense, the way that w
know how to move a ball at the moment corresponds t
gravity-free movement. Let's write a program that simulates
ball thrown without any gravity.

```
 10 LET V=0
 20 LET H=1
 30 LET X=0
 40 LET Y=0
 50 PRINT AT Y,X;CHR$ 128
 60 PRINT AT Y,X;CHR$ 0
 70 LET X=X+H
 90 LET Y=Y+V
100 GOTO 50
```

If you look at lines 10 and 20 you should be able to see th
the ball is thrown horizontally forward from the top of th
screen. It's rather like pushing a ball off the top of a cliff
only in this case where there is no gravity instead of falling
moves in a straight line, totally unaffected by anything.

If we introduce gravity the difference is that the vertic
velocity changes. For example, if you just release a ball it fa
and its vertical velocity increases as it falls faster and faster.
other words as the ball moves one unit horizontally its vertic
velocity increases by a fixed amount. The value of the fix
amount depends upon how strong gravity is but for o

purpose we can adjust it so that it gives a reasonable result. To see the falling ball add line 80:

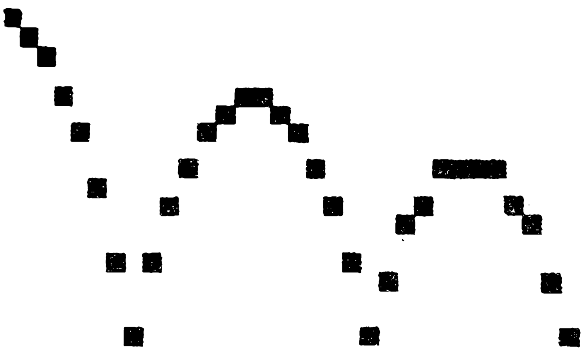
```
80 LET V=V+.1
```

to the "free fall" program. When run, the new program mimics a ball falling in a parabolic curve. The program gives an error as soon as the ball "falls" off the bottom of the screen. If you want to improve the program try subtracting a small amount from the horizontal velocity to allow for wind resistance.

We can combine what we already know about bouncing balls with what we have just discovered about gravity. If we define a horizontal wall at say Y=15 then as the ball reaches it we can apply our previous "bounce" rule and reverse the vertical velocity. The resulting program is:

```
 10 LET V=0
 20 LET H=1
 30 LET X=0
 40 LET Y=0
 50 PRINT AT Y,X;CHR$ 128
 60 PRINT AT Y,X;CHR$ 0
 70 LET X=X+H
 80 LET V=V+.6
 90 LET Y=Y+V
100 IF Y>15 THEN LET V=-V
110 GOTO 50
```

If you remove line 60 then the output looks something like:



By now you should have a good idea how to make a ball d anything that you want it to. Using the horizontal and vertic velocity idea everything is much simpler. If you want to spee up then add something to the appropriate velocity or subtrac it to slow down.

Lunar lander

If we add a few extras to the falling ball program described i the last section we can produce a reasonable lunar landin game. A rocket landing on the moon behaves exactly like falling ball except that it can fire its motors and reduce i vertical velocity. To obtain a reasonable game we have t change from PRINT statements to PLOT statements but this i a minor change. Let H = height, S = speed, F = fuel, BR = bu rate.

```
 10 LET F=1200
 20 LET B=0
 30 LET V=B
 40 LET H=RND*2+1
 50 LET X=V
 60 LET Y=43
 70 PLOT X,Y
 80 GOSUB 170
 90 UNPLOT X,Y
100 LET X=X+H
110 IF X>30 THEN GOSUB 250
120 LET Y=Y-V
130 IF Y>0 THEN GOTO 70
140 IF V>.5 THEN PRINT "**CRASH**"
150 IF V<.5 THEN PRINT "ZX81 HAS LANDED"
160 STOP
170 LET B$=INKEY$
180 IF B$="" THEN GOTO 200
190 LET B=VAL B$*10
200 LET F=F-B
210 IF F<0 THEN LET B=0
220 LET V=V-B/100+.5
```

```
230 PRINT AT 0,0;"H=";INT (11.6*Y);" S=";
    INT (200*V);" F=";F;" BR=";B;" "
240 RETURN
250 LET X=0
260 CLS
270 RETURN
```

The amount of fuel that you start with is set in line 10. If you ant to make the game easier increase the amount of fuel om 1200 to something larger. The rocket starts with a dom horizontal velocity, line 40, and falls under gravity il it hits the ground. As it falls you can burn fuel to reduce rate of descent. Pressing any key between 0 and 9 sets the e at which fuel is burned – the burn rate BR. The burn rate ten times the value of the key pressed. Which key is pressed checked at line 170 by using the INKEY$ statement. Keep ssing the key that you want because it will only affect the ogram once every time the rocket moves. The fuel burned is tracted from the fuel remaining and if you use it all up you fall to the surface. The object of the game is to land with vertical velocity of less than 100 metres per second. If you ve too far to the right the screen is cleared and you start n the far left again. Happy landings!

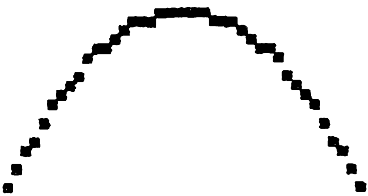
owing in a given direction

far we have found how a ball moves under gravity if thrown zontally from a cliff but many games need a ball to be wn upward. This can be achieved by reversing the Y co-nates in a PRINT AT or simply using a PLOT statement ead. Remember that PLOT 0,0 is the *bottom* left but NT AT 0,0 is the *top* left. Try the following:

```
10 LET X=0
20 LET Y=15
30 LET H=1
40 LET V=2
50 PLOT X,Y
60 UNPLOT X,Y
70 LET X=X+H
```

```
 80 LET V=V-.1
 90 LET Y=Y+V
100 IF Y<15 THEN STOP
110 GOTO 50
```

The initial velocity is H=1 and V=2. At each step the verti velocity is reduced by .1. So the ball first starts moving u quite fast then slows down until it is only moving forwar Then the vertical direction is reversed and the ball starts falli down back to the bottom of the screen. The resulting shape the well known parabola of a thrown object.



Normally we want to throw a ball at a given angle and wi a given force. If we throw the ball with a given force th governs its overall velocity. That is, the harder you throw ball the faster it moves at first. The angle at which you thro it alters the distribution of this overall velocity between t vertical and horizontal parts. For example, if you throw t ball straight up at 90 degrees then the ball moves vertically b not horizontally. As you decrease the angle the ball mo more horizontally and less vertically. If you analyse the sit tion mathematically you will find that, if you throw the b with a force F that produces a total velocity V at an angle then the horizontal velocity is given by V*COS(T) and t vertical velocity is given by V*SIN(T). Using these two start values for horizontal and vertical velocity we can use the sa sort of program to make the ball move under gravity. The o thing that we have to remember is that the ZX81 measu angles in radians. To convert degrees to radians use

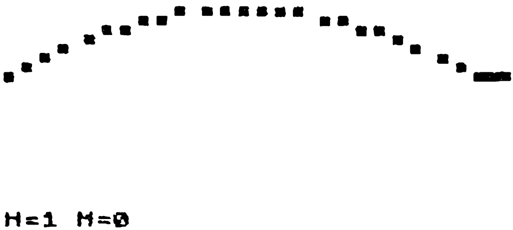
angle in radians = angle in degrees*PI/180

Cannon-ball

Now that we know how to throw something at a given angle and with a given force, we can try to write a shooting game. You have a cannon set at the far left-hand side of the screen and a target randomly placed to the right. You have to specify two numbers – the angle 0–90 degrees and the force of the charge – and try to hit the target. The force of the charge is limited but values around 10 work well. An additional problem is that if you shoot at such an angle that the cannon-ball leaves the screen *anywhere* it is counted as a miss. This means that you have to fire the cannon at a low angle to make sure that you do not shoot off the top of the screen! This restriction also stops you from shooting down any of your own aircraft! You'll find low angle shots more difficult so the limitation actually improves the game.

```
 10 LET B=30
 20 GOSUB 300
 30 INPUT F
 35 LET F=F/4
 40 INPUT T
 45 LET H=F*COS (T*PI/180)
 50 LET V=F*SIN (T*PI/180)
 55 LET X=0
 60 LET Y=B
 65 PLOT X,Y
 70 LET V=V-.1
 75 UNPLOT X,Y
 80 LET X=X+H
 90 LET Y=Y+V
100 IF Y<B THEN GOTO 200
110 IF X>63 OR Y>43 THEN GOTO 200
120 GOTO 65
200 IF X>=P AND X<=P+3 THEN GOTO 250
210 LET M=M+1
220 GOTO 260
250 LET S=S+1
260 PRINT AT 14,0;"H=";S;" M =";M;" "
```

```
270 GOTO 30
300 LET P=RND*10+50
310 FOR I=P TO P+3
320 PLOT I,B
330 NEXT I
340 LET M=0
350 LET S=M
360 RETURN
```



Subroutine 300 plots the target at a random position 3 points wide and initialises M and S the miss and score counters. Lines 30 to 50 input F the force and T the angle. The force is scaled in line 35 to give a reasonable range of velo-cities. The middle section is simply the thrown ball program given earlier in this section but with extra statements to check if the ball has hit the target.

It is possible to write much more complicated programs along these lines, to allow for such things as air resistance and wind direction – but not in a 1K ZX81.

PEEK AND POKE

Two of the most mysterious instructions in the whole of BASIC are PEEK and POKE. The question, "What can I use PEEK and POKE for?" is frequently asked. The answer depends very much on which computer you are using. This chapter gives a brief explanation of what PEEK and POKE do and examples of how they can be used on the ZX81. Do not expect what you learn here about how to use PEEK and POKE to apply to other computers – it almost invariably will not!

What PEEK and POKE do

Of the two instructions, PEEK is the easier to understand and the safer to use. You cannot "crash" the machine with an incorrect use of PEEK but you most certainly can with POKE. Although we have referred to PEEK as an instruction it is more properly called a "function" because it returns a result. (Functions are things like SIN(X) which can be worked out to one number – the result.) PEEK is a special sort of function in that it doesn't "work" anything out it simply "returns" the contents of a particular memory location and converts it to decimal. For example,

```
10 LET A=PEEK 7688
```

will set A to the contents of memory location 7688. There are many things that you will need to know about computers in general and the ZX81 in particular before this example will make very much sense to you. Firstly, you have to know that computers save and retrieve information from numbered memory locations. Each location has a unique number, known as its "address". Secondly you need to know that the amount of information that can be stored at each location is limited. In the case of the ZX81 each memory location can only store

one character. As you probably already know a computer ca
only store zeros or ones in its memory so how does it manag
to store an actual character in a memory location? The answe
is that a group of bits (zeros or ones) can be read as a numbe
For example 0101 is five. It's not important at the mome
that you know how to convert a group of bits, it's sufficie
to know that it can be done. (If you want to find out how, se
"Beginners Guide to Microprocessors and Computing" – BP6
by E. F. Scott.) A group of eight bits – a byte can represe
numbers from 0 (00000000) to 255 (11111111), so any ZX8
memory location can be thought of as holding a number i
this range. At this point you should realise how a character i
stored as a group of 8 bits!

If you look at the back of the ZX81 manual you will find
list of the ZX81 character set. The first column is labelle
"code" and contains numbers starting at 0 and going up t
255. This means that we can either treat the contents of
memory location as a number, 49 say, or as a characte
CHR$(49) which gives L. The most important thing to unde
stand though is that any ZX81 memory location may conta
a number between 0 and 255. If you add PRINT A to th
earlier example you will see that this is true – i.e. A li
between 0 and 255. If you look at the contents of any oth
memory location you will see many different numbers bu
none of them smaller than 0 or greater than 255.

The only other thing that you need to know about usi
PEEK is the range of addresses that you can use. The ZX8
numbers its memory locations starting at 0 and going up to
maximum of 65535. Not all of these memory locations corre
pond to anything in the ZX81; as we shall see later many o
them are unused. One last fact that it is important to know i
that there are two types of memory – RAM and ROM. RA
– Random Access Memory can be used to store and reca
information. ROM – Read Only Memory can only be used t
recall information. A 1K ZX81 has only 1024 memory loc
tions that correspond to RAM but has 8192 memory locatio
that correspond to ROM. This vast quantity of inbuilt info
mation in every ZX81 is used for many different things bu
one of the main uses is to define the rules of the BASIC la

ge. The ROM portion of memory starts at 0 and goes up to 91. The RAM portion starts at 16384 and goes up to 17407 you have 1K of RAM or 32767 if you have 16K. Even in a 16K ZX81 not all of the memory locations are used or ble.

The POKE instruction is easy if you have followed the planation of how PEEK works. POKE allows you to store a e in any RAM memory location. Doing this may destroy ur program if it happens to be stored in a location that you eady use – so take care! The form of POKE is:

```
POKE address,byte
```

or example if you type in (no line numbers because you want e computer to carry out the command at once and you n't want a program in memory that might get in the way!)

```
POKE 17300,33
PRINT PEEK 17300
```

ou should (if your ZX81 is working) see 33 printed out on screen. What you have done is to store a pattern of bits presenting 33 in the memory location whose address is 300 and then printed out its contents. Try the same thing h different data bytes to convince yourself that it works. you try a POKE at addresses that correspond to ROM you n't get very far – for obvious reasons!

ing PEEK to draw big letters

e interesting and useful part of the ROM area of memory is character generator. If you use PRINT "A" somehow or her the ZX81 has to construct a pattern of dots on the een corresponding to the shape of the letter A. To do this it oks the pattern up in a table stored in the ROM region of emory. This table is called the character generator and contins a pattern of dots for the shape of every character that ZX81 can print. A pattern of dots? In the last section we covered that each memory location could only hold a group eight bits, zeros or ones, so how can it store a pattern of ts? The answer is not difficult. If we call a 1 a black dot and

... white dot we can use a group ... bits to repre... single row of eight dots. If we use eight memory locations f... character then we can store eight rows of eight dots to draw character. For example the letter A would be:

pattern of bits	decimal number
0 0 0 0 0 0 0 0	0
0 0 1 1 1 1 0 0	60
0 1 0 0 0 0 1 0	66
0 1 0 0 0 0 1 0	66
0 1 1 1 1 1 1 0	126
0 1 0 0 0 0 1 0	66
0 1 0 0 0 0 1 0	66
0 0 0 0 0 0 0 0	0

If you find the A difficult to see they try colouring in all th ones. The letter is surrounded by zeros to make sure that the is some space around each letter when it's printed. The colum of decimal numbers corresponds to what would be printed o by PEEKing the memory locations where the share of A i stored.

Knowing where the character generator table is stored i ROM means that we can write a program to use the do patterns to print or plot points to make larger characters. T do this we have to solve a number of problems. We can u PEEK to find the number stored in any location but we ne to know the pattern of ones and zeros. In other words, w have to find a way to reduce a number to its sequence of zer and ones. This is not difficult to do if you understand binar numbers and arithmetic. To avoid getting involved in to much theory we will simply use the following progra without detailed explanation.

```
10 LET A=PEEK 7984
20 FOR I=7 TO 0 STEP -1
30 LET B=A-2*INT(A/2)
40 LET A=INT(A/2)
50 PRINT AT 20,I;B
60 NEXT I
```

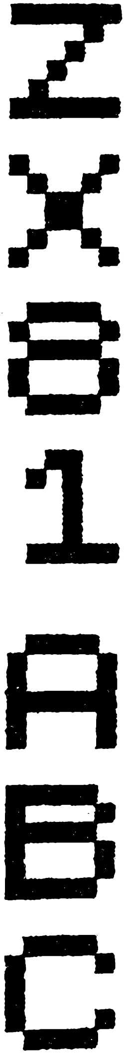
Address 7984 happens to be the start of the eight bytes th of the letter A into the variable called A! Each time rough the FOR loop we extract one bit from A and print it. e first time through we extract the leftmost bit, then the ext leftmost and so on until we have printed all eight bits. The FOR loop goes from 7 to 0 because we want to print the sult from left to right and the index I is used in the PRINT T 0.I. *You can get the whole pattern for A by repeating the* gram eight times, once for each row of the letter A.

```
 10 LET P=7984
 20 FOR J=0 TO 7
 30 LET A=PEEK (P+J)
 40 FOR I=7 TO 0 STEP -1
 50 LET B=A-2*INT (A/2)
 60 LET A=INT (A/2)
 70 PRINT AT 20,I;B
 80 NEXT I
 90 SCROLL
100 NEXT J
```

u should be able to see the dot pattern of the letter A after s program. Now we are nearly home and dry! All we have to is to add some statements to print a blank when B is 0 and black square when B is 1 and we have a large letter A on screen. Add some code to pick out the appropriate part of table for any particular string of letters and we can have ge messages moving up the screen. Try the following:

```
 10 INPUT A$
 20 FOR I=1 TO LEN A$
 30 LET P=7688+(CODE (A$(I))-1)*8
 40 FOR J=0 TO 7
 50 LET A=PEEK (P+J)
 60 FOR K=7 TO 0 STEP -1
 70 PRINT AT 20,K;CHR$ ((A-2*INT (A/2))*128)
 80 LET A=INT (A/2)
 90 NEXT K
100 SCROLL
110 NEXT J
120 NEXT I
```

If you type in any message it will be displayed as a sequence of moving big letters up the lefthand side of the screen.



Line 30 picks out the position in the table of each letter. The table starts at 7688 and the graphics character CHR$(1) is the first. The function CODE is the opposite of CHR$. It takes a letter and returns its position in the table. Each letter takes eight memory locations, so you have to multiply the character code by eight to get to the right place in the table. Line 70 uses the same method introduced in the earlier program to decide if we have a zero or a one, but this time instead of printing zero or one, it prints CHR$(0*128) i.e. a space, or CHR$(1*128) i.e. a black square. If you want to repeat the message forever add

```
130 GOTO 20
```

One problem with our big letter display is that you can only get about three letters to a screen. If we could make the letters slightly smaller we could make it more useful as a moving display. We could make them half the size by using PLOT instead of PRINT. Try the following program:

```
 10 INPUT A$
 20 FOR I=1 TO LEN A$
 30 LET P=7688+(CODE (A$(I))-1)*8
 40 FOR J=0 TO 7
 50 LET ODD=J-INT (J/2)*2
 60 LET A=PEEK (P+J)
 70 FOR K=7 TO 0 STEP -1
 80 LET B=A-2*INT (A/2)
 90 LET A=INT (A/2)
100 IF B=1 THEN PLOT K,NOT ODD
110 NEXT K
120 IF ODD=1 THEN SCROLL
130 NEXT J
140 NEXT I
150 GOTO 20
```

The only difference with plotting instead of printing is that you can fit two rows of plotted points in every print row. If you were to plot a row and then SCROLL to make the display move, you'd discover that each letter was broken up. The solution is to PLOT the first row at Y=1 and the second row at Y=0 and then SCROLL the completed line. To do this you have to introduce an extra variable ODD the tests for the row number being odd (ODD=1) or even (ODD=0). Line 50 works out the correct value for ODD. Line 100 looks a little strange. If B=1 then you need to plot a point, if the row number is even you need PLOT K,1, if the row number is odd you need PLOT K,0. The variable ODD is the wrong way round to let us use PLOT K,ODD (ODD is 0 when the row is even and 1 when the row is odd), but we can reverse it using NOT ODD.

Remember that NOT ODD is 1 if ODD is 0 and 0 if ODD is 1. In the same way line 120 causes the display to scroll after each odd row has been plotted. This is of course exactly what we need.

You could use either of these routines to add large letter outputs to any program. In general though fitting a program into 1K ZX81 is difficult enough without adding features such as big letters! If you have a 16K ZX81 there is no such problem.

Conclusion

This chapter has tried to give some idea of the way that PEEK and POKE work. The example of using PEEK to display large letters is typical of the sorts of things that PEEK and POKE are used for. Notice that, apart from knowing how PEEK works, the example requires knowledge about the machine – i.e. the fact that a character generator exists, where it is and what its format is. These extra pieces of information sometimes become confused with knowing how PEEK and POKE work. If you move to a new machine then the way PEEK and POKE work will remain the same but the big letter program will not work. It might be possible to change it so that it works if you know where the character generator is etc. By now you should be able to understand that there is no answer to the question "what are PEEK and POKE used for?" unless you say which machine you're talking about.

Chapter Six

A SENSE OF TIME

Every computer has a way of keeping time built into it. Some make it easy for a programmer to get at it, others make it nearly impossible. The ZX81 is somewhere in between these two extremes in that it provides a timing command – PAUSE which allows you access to its clock – but for any really useful timing you have to use a PEEK and the occasional POKE. Before going on to examine methods of using time let's see what makes the ZX81 tick.

FAST, SLOW and PAUSE

As we mentioned in Chapter One, the ZX81's microprocessor the Z80 is responsible for maintaining the screen display. A standard television set displays a picture every 1/50th of a second (1/60th of a second in the USA). So the ZX81 has to stop whatever it is doing every 1/50th of a second and display the TV screen. This is of course what slows the ZX81 down when doing calculations. In fact you do have a choice in the matter because the ZX81 has two modes of operation – fast and slow. Slow is the normal mode of operation that the ZX81 adopts when first switched on and in this mode the screen display is continuous. Fast mode is when the ZX81 forgets about displaying the TV screen and gets on with whatever is its main task. You can switch from slow to fast by typing the command FAST and back to slow by typing SLOW! The increase in speed that you get by moving to fast mode is about 15% so it can be well worth while switching modes during a program. The trouble is that fast mode doesn't display any results unless you stop the computer by asking for an INPUT. Even in fast mode the ZX81 will stop and wait for you to type in an answer and while it waits it displays the screen. Let's suppose that rather than display the screen until someone types something in at the keyboard, all we want to do is to

display the screen for a fixed time interval. For this we need a new command. The ZX81 provides the PAUSE command for just such a reason. If you use PAUSE N the machine will stop computing and display N TV frames i.e., it will display the screen for N/50 seconds. (It is a limitation of the PAUSE command that if N is larger than 32767, a pause of 11 minutes, the machine will pause forever!) Using the PAUSE command along with FAST and SLOW it is possible to do some computing, show some results and then go back to the computing without having to ask anyone to push any keys. Take care, though, there is a fault in the ZX81 BASIC which requires every PAUSE in fast mode to be followed by a POKE 16437,255 to avoid destroying your program!

Using PAUSE

Although the intended use of the PAUSE command is to allow a screen to be displayed in the fast mode, it is more often used to provide a fixed time pause in a program running in slow mode. For example try the following program.

```
10 PRINT "TICK"
20 PAUSE 50
30 PRINT "TOCK"
40 PAUSE 50
60 GOTO 10
```

This will print tick/tock on the screen at about one second intervals. There are two things to notice about this very simple program. Firstly, although each PAUSE causes a pause for 1 second (i.e. 50 frames) the time between each tick and tock is longer because the computer takes time to PRINT and GOTO. Secondly, the screen flashes just before each tick or tock is printed. The flash is caused by the ZX81 changing over from pausing and displaying to computing and displaying and there is nothing that can be done about it.

Delay loops

The flashing of the screen following a PAUSE instruction can

destroy the intended effect of your screen display. For th
reason it is often better to use a delay loop rather than
PAUSE command. A delay loop is simply a FOR loop th
does nothing but waste a fixed and known amount of tim
For example:

```
10 LET T=50
20 PRINT "TICK"
30 FOR I=1 TO T
40 NEXT I
50 PRINT "TOCK"
60 FOR I=1 TO T
70 NEXT I
80 GOTO 10
```

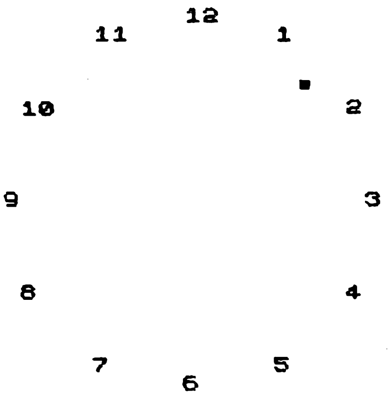
Two delay loops are included in this program. Each give
delay of slightly less than one second and this makes the ti
between each "tick" and "tock" roughly one second. If y
want to check the accuracy and regulate the clock, the b
way is to time a large number of "tick/tocks" and work o
how long each takes. If it's less than a second then increas
and vice versa. Notice that the time delay depends on the ty
of statement used as a delay loop. If you change FOR I=1 T
T to FOR I=1 TO 50, the time that the loop takes will cha
very slightly.

Using the delay loop idea we can improve on the clock f
display given in the Sinclair manual:

```
 10 LET S=24
 20 FOR N=1 TO 12
 30 PRINT AT 10-10*COS (N/6*PI),
    10+10*SIN (N/6*PI);N
 40 NEXT N
 50 LET T=0
 60 LET A=T/30*PI
 70 LET SX=21+18*SIN A
 80 LET SY=22+18*COS A
 90 PLOT SX,SY
100 FOR I=1 TO S
110 NEXT I
```

```
120 UNPLOT SX,SY
130 LET T=T+1
140 GOTO 60
```

The delay loop at lines 100 and 110 replaces a PAUSE statement and the result is a steady display.



The frame counter

As the ZX81 can PAUSE and display a given number of TV frames you may have guessed that somewhere inside it is a memory location that counts the number of frames that have been displayed. In fact there are two memory locations that keep track of the number of frames and these together are known as the FRAME COUNTER. The frame counter is at address 16437 and 16436. The memory location at address 16436 counts the number of frames since the machine was switched on. As the largest number that a single memory location can hold is 255 the number of frames that location 16436 can count is limited. To overcome this problem location 16437 counts the number of times the lower location reaches 255. In other words, the lower location counts frames and the higher location counts every 256 frames – i.e. the lower counter goes from 0 – 255 for every one count of the upper counter. This is very like a traditional clock dial, with the lower counter going "round" every 256 and then moving the upper counter on one.

There is one additional complication – the counters both count down rather than up. In other words, every frame subtracts one from the lower counter and every time the lower counter reaches 0 the upper counter has one subtracted from it. This causes no real trouble as long as we remember that as time goes on the counters get *less*. To see this happening try:

```
10 PRINT PEEK(16436)+256*PEEK(16437)
20 GOTO 10
```

You will see a large number getting smaller all the time. The difference between successive values is the number of frames that the ZX81 displays in between each print. To see the same number in terms of seconds all we have to do is divide by 50. To make the displayed time increase rather than decrease requires two additional actions. First we must set the frame counter to its maximum value using two POKE commands and then we can subtract the PEEKed time value from the maximum value! For example:

```
10 LET P=16436
20 POKE P+1,255
30 POKE P,255
40 PRINT (65535-PEEK(P)-256*PEEK(P+1))/50
50 GOTO 40
```

Notice that it's a good idea to POKE the fastest changing counter last – just as when you set a clock you deal with the hours first, then the minutes and finally the seconds.

Digital clock

There are many ways to turn the ZX81 into a digital clock. One of the easiest is to use the program in the previous section to provide the number of seconds since the machine was switched on. All you have to do is add the current time in seconds, convert the answer to hours, minutes and seconds and display the result. A more interesting method is based on the

tick/tock program. Instead of using the frame counter to keep track of the time why not use it to signal that one second had passed. Try the following program:

```
10 LET P=16437
20 POKE P,50
30 IF PEEK P<>0 THEN GOTO 30
40 POKE P,50
50 PRINT "TICK"
60 GOTO 30
```

Before you decide that there has been a misprint, let me point out that *this program does not work*! The reason why it doesn't work is what interests us. It is difficult to see why the program fails because the idea behind it seems foolproof. At line 20 the lower frame counter is set to 50. At line 30 the value of the lower frame counter is checked to see if it's zero. If it's not then the program checks again. If it is zero then we know that 50 frames have been displayed and one second has passed. The counter is immediately reset and begins to count out the next second while we print "TICK" on the screen. Why doesn't this work? There is certainly plenty of time to get to the IF statement before the count reaches 0 – not even the ZX81 needs a whole second to carry out two lines of BASIC. The trouble lies in the IF statement itself. The IF statement takes longer than 1/50s to complete! This means that when the frame counter changes to 0 the program might only just have finished working out the result of the last PEEK! If you run the program often you might just see one "TICK" printed on the screen because by chance the IF statement happened to read the frame counter just as it reached zero.

If you want to use the frame counter as an internal timer then you have to choose a time interval that is long compared to the length of time that an IF statement takes to execute. The upper frame counter changes only once every 256 frames or about every 5.12 seconds. If you are prepared to have a clock tick only every 5.12 seconds then you can use the upper frame counter in the sort of program that fails using the lower frame counter. Try the following:

```
 10 LET S=0
 20 LET H=20
 30 LET M=39
 40 LET P=16437
 50 POKE P,255
 60 LET A=PEEK P
 70 LET B=A
 80 LET A=PEEK P
 90 IF A=B THEN GOTO 80
100 LET S=S+5.12
110 IF S<60 THEN GOTO 190
120 LET S=S-60
130 LET M=M+1
140 IF M<60 THEN GOTO 190
150 LET M=0
160 LET H=H+1
170 IF H<24 THEN GOTO 190
180 LET H=0
190 SCROLL
200 PRINT AT 0,0;H;AT 0,3;M;AT 0,6;INT S
220 GOTO 70
```

The program works by reading the upper frame counter at li 60 and then reading it again at line 80 and waiting until th difference is one. When this happens 5.12 seconds have pass and the second counter can be updated at line 100 and th time re-displayed by lines 110–200.

If you've got a 16K ZX81 then you might like to add large number display (see Chapter Five) or an alarm cloc facility.

A chess clock

A simple application of the frame counter is a chess clock.

```
10 LET TW=0
20 LET TB=0
30 LET G=0
40 LET P=16436
```

```
 50 PRINT "PRESS ANY KEY TO START"
 60 IF INKEY$="" THEN GOTO 60
 70 CLS
 80 POKE P+1,255
 90 POKE P,255
100 LET T=PEEK (P)+PEEK (P+1)*256
110 LET T=(65535-T)/50
120 IF G=1 THEN PRINT AT 0,15;"BLACK ";INT
    ((T+TB)/60);".";INT ((T+TB)-INT ((T+TB)/
    60)*60);" "
130 IF G=0 THEN PRINT AT 0,0;"WHITE "; INT
    ((T+TW)/60);".";INT ((T+TW)-INT ((T+TW)/
    60)*60);" "
140 IF INKEY$="" THEN GOTO100
150 IF G=0 THEN LET TW=TW+T
160 IF G=1 THEN LET TB=TB+T
170 LET G=NOT G
180 GOTO 80
```

There is nothing really new in this program and you should be able to spot some techniques from earlier chapters. The total move time is kept in TW for white and TB for black. The variable G is 1 if black is playing and 0 if white is. Pressing any key switches players (line 170). The time T since the last switch is added to the correct total play time in lines 150 and 160 for black and white respectively and then the timer is reset at line 80. The only limitation on this chess clock is that any move must take less than 20 minutes otherwise the frame counter reaches 0 and starts counting again.

Reaction time game

Using the lower frame counter it is possible to time events to 1/50 of a second. This makes it just feasible to write a reaction time program. It is important to realise that because of the slowness of ZX81 BASIC the accuracy of the reaction times measured is worse than 1/50th of a second. This is not good enough for any serious purpose but it is fun.

```
 10 PRINT "READY"
 20 FOR I=0 TO RND*50+40
 30 NEXT I
 40 LET P=16436
 50 POKE P+1,255
 60 POKE P,255
 70 PRINT "GO"
 80 IF INKEY$="" THEN GOTO 80
 90 LET T=PEEK (P)+PEEK (P+1)*256
100 PRINT "REACTION TIME=";(65535-T)/50;"SE
110 GOTO 10
```

After a random delay the word "GO" is printed. Pressing any key causes the time to be read and printed out. The program can be made more interesting and accurate by taking the average of 10 reaction time measurements. After you have understood the previous program try:

```
 10 LET S=0
 20 FOR J=1 TO 10
 30 CLS
 40 PRINT "READY"
 50 FOR I=0 TO RND*50+40
 60 NEXT I
 70 LET P=16436
 80 POKE P+1,255
 90 POKE P,255
100 PRINT "GO"
110 IF INKEY$="" THEN GOTO 110
120 LET T=PEEK(P)+PEEK(P+1)*256
130 LET T=(65535-T)/50
140 LET S=S+T
150 NEXT J
160 CLS
170 PRINT "YOUR AVERAGE IS ";S/I
180 IF S/I>.08 THEN PRINT "SLOW**"
190 IF S/I>=.05 AND S/I<=.08 THEN
    PRINT "NOT BAD"
200 IF S/I<.05 THEN PRINT "WELL DONE"
210 IF S/I<.02 THEN PRINT "VERY FAST"
```

The additional lines at the end of the program calculate your score over ten tries and print out an appropriate message. This routine could be used as the basis for a variety of games – but they would all require more memory than a 1K ZX81 has to offer!

Chapter Seven

STRINGS AND WORDS

The ZX81 is very good at manipulating text! The trouble is that text takes a lot of memory so despite being good at it the 1K ZX81 soon runs out of memory. In addition there are a number of problems about using text to play games that all computers share. Some of these problems have been solved but others take us to the limits of our knowledge of computers. They take us into areas of artificial intelligence.

Strings and things

Before starting on the subject of using strings, a quick recap of how the ZX81 handles strings might be a good idea. The ZX81 distinguishes string variables from others by use of a $ sign after a variable name. For example:

```
10 LET A$="NAME"
```

A string can be of any length if it fits into the memory. You can manipulate strings in three ways.

In the first method you can join them together using the + operation. For example:

```
10 LET A$="FIRST NAME"
20 LET B$="LAST NAME"
30 LET A$=A$+B$
40 PRINT A$
```

This program takes the string "FIRST NAME" and the string "LAST NAME" and joins them together to make "FIRST NAMELAST NAME" in the variable A$.

In the second method you can pick out any part of a string using the slicing notation. For example:

```
10 LET A$="ABCDEFG"
20 PRINT A$(2 TO 5)
```

will print the string ABCDEFG from the second letter to the fifth letter i.e. BCDE. You can use the notation A$ (start TO finish) where "start" and "finish" are replaced by numbers to mean – the string in A$ from and including the "start" letter up to and including the "finish" letter. The ZX81 also allows certain short forms of the slicing notation.

```
A$ (  TO n) = A$ (1 TO n)
A$ (n)      = A$ (n TO n) i.e. the nth letter
A$ (n TO  ) = A$ (n TO LEN(A$))
A$ (  TO  ) = A$ (1 TO LEN(A$)) i.e. the whole
                                       string
```

The third method of manipulating strings is very clever indeed. You can change part of a string specified by slicing notation. For example:

```
10 LET A$="ABCDEFG"
20 LET A$(2 TO 3)="12345"
30 PRINT A$
```

will change the string ABCDEFG to A12DEFG. In other words the slicer specifies the part of the string to be changed – the second letter and the third. *It doesn't matter if the string to the right of the equals sign is bigger than the part to be changed – the correct number of characters are used starting from the left. In the example only "12" is used even though the string is "12345".*

The ZX81 has the ability to use multidimensional string arrays but these use up memory *very* quickly and in general are best avoided in a 1K machine. As a simple example of string handling try the following

```
10 INPUT A$
20 LET B$=""
30 FOR I=LEN A$ TO 1 STEP -1
40 LET B$=B$+A$(I)
50 NEXT I
60 PRINT B$
```

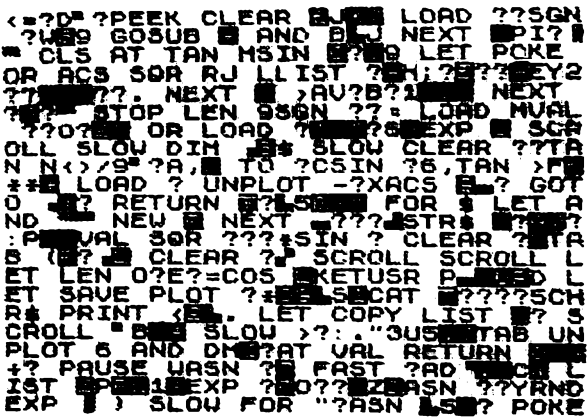
This reads in any string and reverses it. Type in the following sentence "NUF EB NAC MARGORP SIHT" to find out what it says. Notice that the program works by "stripping" do the input to single letters using the slicer notation and building it in the reverse order in line 40. You could use program to send secret messages or simply to teach your to speak backwards!

Random words

We discovered how to use random numbers in Chapter T and how to convert random number into random graphics Chapter Three. Using the same techniques we can gener random characters. Try the following:

```
10 PRINT CHR$(RND*255);
20 GOTO 10
```

You will see the screen fill with random characters somethi like:



The next step is to try to generate random words. If you lo at the screen full of random characters generated by t program you should see some words – like GOTO, RN SCROLL etc. These are of course nothing more than t ZX81's BASIC keywords printed on the keyboard. From t ZX81's point of view these are the same as single characte ey are entered from the keyboard with one keypress and they are stored *in one memory location. The only difference* is that the key words are expanded into a number of letters when printed. Apart from these keywords it is *very* difficult to generate random words on a computer. If you generate random letters (excluding keywords and graphics) you will be very lucky to see any long words! Try:

```
10 PRINT CHR$(RND*25+38);
20 GOTO 10
```

You might pick out a few three or four letter words but not very many when compared to the amount of gibberish produced.

This difficulty with generating random words severely limits the type of word games that a computer can indulge in. For example, if you want to play a number guessing game, the computer can generate a random number and you can try to guess what it is. In the case of a word guessing game the computer has no way of generating a random word so the best that you can do is to input a list of words at the start of the game and program the computer to pick a word at random. If you can get someone else to type in the list of possible words or if the list is so long that you cannot remember all the words then you can use this method to play word games. As the ZX81 cannot store very many words we have no choice but get someone else to type in the list of words.

Hangman

With the restriction described above, i.e. that someone types in a list of words when the player isn't looking it possible to play a simple form of hangman.

```
 10 LET W$="<"
 20 FOR I=1 TO 4
 30 INPUT A$
 40 LET W$=W$+A$+"<"
 50 NEXT I
 60 CLS
 70 PRINT "HANGMAN"
 80 LET R=INT (RND*(LEN W$-1)+1)
 90 IF W$(R)="<" THEN GOTO 120
100 LET R=R-1
110 GOTO 90
120 LET A$=""
130 LET W$=W$(1 TO R-1)+W$(R+1 TO )
140 IF W$(R)="<" THEN GOTO 170
150 LET A$=A$+W$(R)
160 GOTO 130
170 FOR I=1 TO LEN A$
180 PRINT "*";
190 NEXT I
200 LET H=0
210 PRINT AT 3,0;"GUESS A LETTER"
220 INPUT B$
230 LET K=0
240 FOR I=1 TO LEN A$
250 IF B$(1)=A$(I) THEN LET K=I
260 NEXT I
270 IF K=0 THEN GOTO 210
280 LET A$(K)="*"
290 LET H=H+1
```

```
300 PRINT AT 1,K-1;B$(1)
310 IF H<>LEN A$ THEN GOTO 210
320 PRINT AT 4,0;"WELL DONE"
330 PAUSE 100
340 GOTO 60
```

The program starts off (lines 10–60) by asking for someone to type in four words. As each word is typed in it is added to a list of words stored in W$. Each word in the list is separated by a "<". If you want to see this add 45 PRINT W$ to the program. After clearing the screen the program then moves on to the hangman game proper. The first thing to be done is to pick a word from the word list at random. This is carried out by lines 80–160. First a random number smaller than the number of characters in the word list (W$) is generated in line 80. This random number can be thought of as "pointing" to the word that has been selected. We then have to transfer the chosen word to another string variable (A$) and delete it from the word list so that it cannot be picked again. This is done by first moving the pointer R back to the first "<" and then transferring everything from there to the next "<" in W$. This is done by lines 90 – 160. After selecting the target word at random the program goes on to print one "*" for each letter in the word (lines 170 – 190). Then the program waits for you to type in a guess at line 220. The guess is compared with the word in the FOR loop at lines 240–260. If a match is found then the position of the match is saved in the variable K. Lines 270–310 print the correct letter in the correct position in the word and blank out the guessed letter in the target word with a "*" (line 280). Blanking out the letter with a "*" stops it from being picked up as a correct answer in later guesses. If the number of correct guesses is equal to the length of the target word then you must have guessed the whole word – so a congratulations message is printed (line 320) and the next word, if there is one, is picked at random.

This program uses a number of interesting methods and is well worth studying. If you have a 16K ZX81 you could increase the number of possible words to something like 100 and then you could type in a list of words yourself because you are hardly likely to be able to remember all the hundred words after a few games of hangman. You could also try to add some graphics and devise a proper scoring method for the number of tries taken to get the correct answer. Both of these projects are 16K material.

Codes and cyphers

Being good at handling both numbers and text, computers are an obvious tool for anyone interested in codes and cyphers. In the second world war much secret information was discovered by the computer "cracking" coded messages. It is not really possible to use the ZX81 as a code cracker but it can be used as a very good encoding and decoding machine using the properties of the RND and RAND functions described in Chapter Two.

```
 10 PRINT "CODER"
 20 PRINT "WHAT IS YOUR KEY"
 30 INPUT K
 40 RAND K
 50 PRINT "DECODE OR ENCODE (0/1)"
 60 INPUT D
 70 IF D=0 THEN LET D=-1
 80 PRINT "TYPE YOUR MESSAGE"
 90 INPUT A$
100 FOR I=1 TO LEN A$
110 LET A=CODE A$(I)-11
120 IF A$(I)=" " THEN LET A=0
130 LET A=A+D*INT (RND*53)
140 LET A=ABS (A-INT (A/53)*53)
150 IF A=0 THEN LET A=-11
160 PRINT CHR$ (A+11);
170 NEXT I
```

To try the program out decode the following message:

```
VKTIP(/ATL(2L.HK.9:;UB;
```

Run the program and answer 1982 to the question "WHAT IS YOUR KEY". Then answer 0 to the DECODE/ENCODE question and type in the string of code given above – the decoded message will be printed on the screen.

This program uses the fact that by using the RAND function you can get a specific sequence of random numbers! All you have to do to get exactly the same sequence of numbers is to use the same value when defining RAND. Remember that RAND 0 has a special meaning, it sets the start of the random number generator according to the time since the ZX81 was switched on. This would give you an unknown key – one which could not be repeated in order for later decoding. It is therefore the one value that should never be used with this program! The program asks you to input the "KEY" value you have chosen (line 30) and it is used to start the random number generator (line 40). The same key value has to be used for decoding so if you don't know what key was used to code a message then you cannot decode it. The message typed in line 90 is broken down into letters and each letter is turned into a number using the CODE function. By subtracting 11 from the CODE value we avoid getting graphics characters in the output since graphics characters would be difficult to write down and send to someone else. We code space as 0 at line 120. The rest of the coding works by adding a random number between 0 and 53 and then working out the remainder when you divide the result by 53. The remainder when you divide by 53 lies in the same range as the character codes that we started with, i.e. 0 to 52. To print the resulting characters we use CHR$(A+11) again remembering to correct for the space character (line 150). To decode the message the random number between 0 and 53 is subtracted from the code restoring it to its original pre-coded value. The variable D is set to −1 to decode and 1 to encode.

This coding program is short but it is quite good at producing codes that are difficult to crack. If you haven't got the key then it is virtually impossible to read a ZX81 coded message because the characters are not broken into groups by spaces and the same character can represent different characters at different points in the message.

Numbers as words – a number guessing game

If you want to play a number guessing game of the sort that involves guessing individual digits then you have to have a way of matching the guess against the target number and you have to know how to generate a random number with a fixed number of digits. To do this you need to use strings in order to manipulate the individual digits. The number guessing game given below is quite well known. The computer picks a four digit number at random with no repeated digits. You guess a four digit number and the computer tells you how many of the digits in your number are:

a) in the target number
b) in the same place in the target number.

A digit that is in the same place in the target number is called a place and a digit that is in the target number in a different place is a hit. For example if the computer had picked 1234 as the target number and you guessed 2035 then you'd have one place – the three and one hit – the two. The reason for not allowing target numbers with repeated digits is to avoid any difficulty with counting the number of hits.

```
 10 LET A$ =""
 20 RAND 0
 30 FOR I=1 TO 4
 40 LET B$=STR$ INT (RND*10)
 50 FOR J=1 TO LEN A$
 60 IF A$(J)=B$ THEN GOTO 40
 70 NEXT J
 80 LET A$=A$+B$
 90 NEXT I
100 INPUT B$
110 IF LEN B$<>4 THEN GOTO 100
120 LET P=0
130 LET H=P
140 FOR I=1 TO 4
150 IF A$(I)=B$(I) THEN LET P=P+1
160 FOR J=1 TO 4
170 IF A$(J)=B$(I) THEN LET H=H+1
```

```
180 NEXT J
190 NEXT I
200 LET H=H-P
210 PRINT AT 20,0;"P=";P;" H=";H;" G=";B$
220 SCROLL
230 IF A$<>B$ THEN GOTO 100
240 PRINT "CORRECT"
```

The random four digit number is generated in lines 30–90. A random digit (0–9) is generated as a string at line 40 and the digits produced so far are compared with it in lines 50–70. If it's already present we jump back and generate another digit. If it's not already present it is added to the number in line 80. The program waits for a guess at line 100, if it isn't four digits long then it is rejected without comment and the program waits for a correct input. A correct guess is compared for place(s) or hit(s) in lines 140–190. Checking for a place is easy and is done by comparing both numbers, digit by digit in line 150 and adding one to P for every match. Checking for a hit is slightly trickier and requires an extra FOR loop at lines 160 to 180. Each digit in the guess is compared with *every other* digit in the target and one is added to H for every match. This hit count includes digits that are also in the right place so we have to subtract P to get a corrected hit count (line 200). Numbers of places and hits are printed at line 210 and line 230 checks to see if you've guessed the number. In the following sample of output, the computer's number was 9072 and it took the player seven tries to guess it.

```
P=0 H=1 G=1234
P=1 H=0 G=5678
P=0 H=2 G=5690
P=0 H=3 G=0967
P=3 H=0 G=9073
P=3 H=0 G=9071
P=4 H=0 G=9072

CORRECT
```

No instructions or messages are included in the program and the output has been cut down to a minimum to allow the full screen to be used to display the last 20 guesses. If you

have a 16K ZX81 or feel that you could always work out t
number in fewer guesses you could add some PRINT sta
ments to make the program "friendlier". This game can b
addictive, so play with care!

Chapter Eight

HINTS AND TIPS

This chapter is different from all the earlier chapters in that it is specifically about the ZX81! The earlier chapters are about using the ZX81 with general programming ideas that could apply to any machine. This is all very well but if you want to get the best out of a 1K ZX81 then you have to resort to special tricks to use the very meagre amount of memory to the full. All of the programs that we have discussed have been written without the use of any special tricks to make the methods used more obvious. A consequence of this is that many of the programs stop short of doing everything that we would like them to do. Normally it's only PRINT statements for messages about what to do at any point in the program that have been left out. Occasionally though, we have been forced to leave out things like scoring and error detection making the program not as much fun as it could be. After reading this chapter of hints and tips you may be inspired to go back and try to squeeze some extra statements into some of the programs.

Space-saving screen displays

The ZX81 has a very clever way of using memory to produce a screen display. If you display a screen full of characters then you will need approximately 32x22 (704) bytes of memory. On most machines no matter how many characters you are displaying on the screen you always need the same amount of memory. The reason for this is that most machines treat the blank as a standard character. So when you think you are displaying a blank screen you are in fact displaying a screen full of blanks. If this was the way the ZX81 worked you would have very little space to write programs in – 1K is 1024 bytes and a full screen of blanks would leave about 320 bytes for programs!

The way the ZX81 works is to store each line of the screen along with an end of line marker – a NEWLINE character. When the ZX81 is first switched on or when the screen is cleared by a CLS all that is stored in memory are 22 NEWLINE characters. If you print "HELLO" on the second line then the characters "HELLO" are inserted between the second and third NEWLINE character. The ZX81 displays whatever it finds between the NEWLINE characters at the correct place on the screen and then sends enough blanks to make the line the correct length i.e. 32 characters long, before moving to the next line. Using this method saves having to pad each line out to 32 characters with blanks stored in memory.

What this means for the programmer is that the far right hand side of the screen is expensive in memory and the left hand side is cheap! If you print a single character on the far right of an otherwise empty line you store 31 blanks plus the character and a NEWLINE character in memory, i.e. 33 bytes. If you place the single character at the far left of an equally blank line you only store the character and a NEWLINE in memory, i.e. 2 bytes. All of the other blanks are generated by the ZX81 after it passes the NEWLINE character while displaying the screen. This extreme example should convince you that it's better to stay to the left! Otherwise you can use the screen as you require. Leaving blank lines as you move down the screen costs nothing extra in memory. These comments also apply to characters that appear on the screen as the result of a PLOT command.

You can get an idea of the amount of space that you have left for screen display, after typing your program, by the amount that you can LIST on the screen. In most cases the number of characters that you can display on the screen is less than the number that you can LIST on the screen because, during a RUN of the program, memory is taken for new variables. An interesting point is that in between RUNs the memory used for variable storage is NOT freed. This means that the number of characters that you can LIST on the screen is the same as the number of characters you can display on the screen while the program is running, as long as the program doesn't use any extra variables. Another useful point is that, if you free all the memory used by the variables in the previous RUN using the CLEAR command, you can get more program listing on the screen.

A simple trick that is sometimes overlooked when you've run out of memory and can't EDIT a line, is simply to LIST it then clear the screen (CLS) and press the EDIT key. Just because you cleared the screen doesn't mean that you can't EDIT the current line.

Memory-saving numbers

The ZX81 uses a lot of memory to store numbers that look as though they ought to take very little. For example:

```
10 LET A=1
```

This looks as though only one character/byte should be used to store the number 1. In fact the ZX81 takes 7 bytes to store the constant 1. In general every constant will take as many bytes of memory as there are digits, plus a byte for the decimal point if used, plus six more bytes that are used by the machine to store the constant in binary. You can save a lot of memory by using strings to hold constants and the VAL function to convert the strings to numbers when required. For example:

```
10 LET A=VAL "1"
```

The string "1" takes three bytes of storage – two for the quotes and one for the figure one. Although the function VAL looks as though it takes up three bytes of storage, because it is entered by a single key stroke it only uses a single byte. So the constant VAL "1" takes only four bytes while the simpler looking 1 takes seven bytes. This method can be used for any constants. For example:

3.134 takes 11 bytes but VAL "3.134" takes 8 bytes
1235 takes 10 bytes but VAL "1235" takes 7 bytes

Another space-saving trick with constants is to use CODE to give a number from 0 to 255. For example:

```
10 LET A=CODE "Z"
```

is the same thing as

```
10 LET A=63
```

or

```
10 LET A=VAL "63"
```

This first version uses one byte for CODE and three bytes fo "Z" making a total of four bytes. The other two use eight an five respectively. The only trouble with this method is th you cannot use it for numbers bigger than 255 and there mu be a character with the correct character code. Also, th method only saves more memory than the VAL method if th number has two or three digits.

You should also notice that these space-saving constan can be used in other places than LET. For example:

```
10 GOTO VAL "120"
```

or

```
20 IF A=CODE "M" THEN LET A=VAL "0"
```

Two of the most used constants are 0 and 1 and there is especially economical way of obtaining them on the ZX8 Try

```
10 LET A=PI/PI
20 LET B=NOT PI
30 PRINT A,B
```

Line 10 works because PI/PI is one and PI is one keystroke the whole expression only takes 3 bytes. Line 20 is a bit mo difficult to understand, NOT PI is zero because the NOT o any non-zero number is zero. NOT and PI take one byte eac and so the whole expression takes 2 bytes.

The most space-saving and simple method of handlin constants is not to use them! If you are going to use a consta throughout a program assign it to a variable once and use th variable instead. For example:

instead of

```
10 LET A=0
20 LET B=0
```

use

```
10 LET A=0
20 LET B=A
```

Space-saving variables

There are only a few simple things that you can do to save space when using variables. The first hint is not to use them! If you create a variable early in the program and it is no longer needed later on, re-use it rather than create a new variable with a more appropriate name, in order to keep the number of variables that you use to a minimum.

The second simple rule is to keep variable names short. A variable always takes five bytes plus one byte for each letter in the name. For example A takes six bytes but APPLECOUNT takes sixteen bytes.

Variables used in FOR loops – control variables – are especially expensive in memory terms. Each control variable uses 18 bytes so re-using control variables is very worthwhile.

Space-saving strings

Each string variable takes one byte for its name, two bytes to record the length of the string and one byte for every character in the string. The first thing that can be done with strings is to try to minimise the number of characters in them using the keywords on the ZX81's keyboard. For example if you wanted to print two asterisks you could type two asterisks or one raised to the power sign. They both display as "**" but the first uses two bytes and the second only one.

Another useful trick which applies to all constants in a program but is very space-saving when applied to strings is to delete the lines of the program that define them. Try:

```
10 LET A$="1234567890"
20 LET A=100
30 PRINT A$,A
```

If you run this program you will see nothing unexpected. If you then delete lines 10 and 20 (by typing their line numbers) and then type GOTO 30 you will see that, although you hav deleted the lines setting the variables, the PRINT statem still prints out their old values. If you RUN the program, ho ever, you will get an error message. RUNning a program clea all variables and starts everything from scratch. Using GOTO to start a program leaves all the variables as they were the la time the program was run – even if the lines defining th values of the variables have been deleted.

By deleting line 10 you save 21 bytes and still have th string stored in A$ and by deleting line 20 you save 15 byte and still have zero stored in A. The reason for this large savin is that, whenever a constant is assigned to a variable, the ZX8 saves a new copy of the constant under the name of th variable in an area of memory away from the program. The o copy of the constant still exists in the program, so w effectively have two copies of the constant, one of which w never be used again unless the constant is re-assigned to th variable.

The only penalty for this space saving is that you have use GOTO to run the program. To summarise the method:

1) Type in all the program lines that set variables equal constants.
2) RUN the small program
3) Delete each line in turn – do not use NEW which deletes the variables as well.
4) Type in the rest of the program that makes use of th variables set by the previous programs.
5) Do not RUN the program but use GOTO the first li number.

You can SAVE such an incomplete program on ta without any extra trouble because the ZX81 automatica saves all the variables defined by a program. So if you ha

SAVEd an incomplete program you can LOAD it and, as long as you don't use RUN but use GOTO, it will work just as if the deleted lines were there!

Space-saving statements

Each BASIC statement that you use takes two bytes for the line number, two bytes to record the length of the line, one byte to mark the end of the line, plus the space required to store the line that you type. Apart from the extra six bytes every numeric constant takes, you can reckon that every key-stroke that you use to type in the line counts as one byte of storage used. So do not type in unnecessary characters such as plus signs in front of numbers or brackets in functions when they are not required. You can also save space in IF statements by remembering that an IF regards any non-zero value as true and zero as false. So instead of:

```
10 IF A<>0 THEN PRINT A
```

use

```
10 IF A THEN PRINT A
```

and instead of

```
10 IF A=0 THEN PRINT A
```

use

```
10 IF NOT A THEN PRINT A
```

If you're really short of space then remember that every line you start requires a minimum of 5 bytes. So do not start a new line unless you have to. For example:

```
10 PRINT A
20 PRINT B
```

uses 17 bytes but

```
10 PRINT A,,B
```

uses only 10 bytes.

Finally do not use REM statements when you are short of space. They use 6 bytes plus one byte for every keystroke in the remark.

How much space?

The ZX81 uses two areas of RAM to store your program – the program file where the program is stored and the variable file where any variables created by your program are stored.

You can find how much space is being taken by the program file by typing in the following line:

```
PRINT PEEK 16396+256*PEEK 16397-16509
```

To find out the size of the variable file use:

```
PRINT PEEK 16404+256*PEEK 16405-
    PEEK 16400-256*PEEK 16401
```

You could add these lines to the end of your program and find out how much of each file you are using every time you run the program but remember to subtract the space that the two extra lines use up in the program file!

The solution and more problems!

The only real solution to the shortage of space on the 1K ZX81 is to buy the 16K RAM pack. This will save you from the need to use all the devious tricks that we have listed above. If, however, you have got into the habit of saving space you may find yourself continuing to use them even when no longer needed. *But* in computing there is a law which says that you trade off space for speed. Most of these space-saving tricks will actually make your programs run slower. This is a good reason for *not* using them once you have a 16K RAM pack. In fact as soon as you have a 16K RAM pack making your programs run faster is *the* big problem – but that is another story.

Notes

The Art of Programming the 1K ZX81

This book shows you how to use the features of the ZX81 in programs that fit into the 1K machine and are still fun to use. In Chapter Two we explain its random number generator and use it to simulate coin tossing and dice throwing and to play pontoon. There is a good deal of fun to be had, in Chapter Three, from the patterns you can display using the ZX81's graphics. Its animated graphics capabilities, explored in Chapter Four, have lots of potential for use in games of skill, such as Lunar Lander and Cannon-ball which are given as complete programs. Chapter Five explains PEEK and POKE and uses them to display large characters. The ZX81's timer is explained in Chapter Six and used for a digital clock, a chess clock and a reaction time game. Chapter Seven is about handling character strings and includes three more ready-to-run programs — Hangman, Coded Messages and a number guessing game. In Chapter Eight there are extra programming hints to help you get even more out of your 1K ZX81.

We hope that you'll find that this book rises to the challenge of the ZX81 and that it teaches you enough artful programming for you to be able to go on to develop programs of your very own.