

The Art of Programming the 16K ZX81

M. JAMES & S. M. GEE

```
CL-  
PRINT  
PRINT  
INPUT  
PRINT D(N)  
DTM AT 3,0;"FRACTION"  
0 PRINT DATA F/I  
0 INPUT A$(1)<>"F" AND A$  
70 IF GOTO 1060  
80 THEN PRINT A$  
90 LET T=0  
100 IF A$="I" THEN LET  
110 PRINT AT 4,0;"LOWEST"  
1120 INPUT L  
1130 PRINT "HIGHEST VALUE"  
1140 INPUT H  
1150 PRINT H  
1160 IF H<L THEN GOTO 1120  
1170 PRINT "HIGHEST VALUE"  
1180 GOTO 1120  
1190 FOR I=1 TO N  
1200 LET D(I)=RND#(1)  
1210 IF T=1 THEN LET  
1220 PRINT AT 20,0;  
1230 D(I)  
1240 ROLL
```

ALSO BY THE SAME AUTHORS

BP109 The Art of Programming the 1K ZX81

THE ART OF PROGRAMMING
THE 16K 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.

© 1982 BERNARD BABANI (publishing) LTD

First Published – November 1982

Reprinted – November 1983

British Library Cataloguing in Publication Data

James, M.

The art of programming the 16K ZX81. – (BP114)

1. Sinclair ZX81 (computer) – Programming

I. Title II. Gee, S. M.

001.64'2 QA76.8.S62/

ISBN 0 85934 089 9

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

PREFACE

This book is the sequel to our book about the 1K ZX81, "The Art of Programming the 1K ZX81". While that book laid the foundations for a good understanding of programming to allow you to make the most of your ZX81, this one sets out to help you use your 16K RAM pack and your ZX printer to the full. It concentrates on good programming style and introduces some interesting programs – that are both fun and useful.

Chapter One introduces the 16 RAM pack and the printer. Chapter Two explains how the extra storage space is used and presents a memory test program to check that your new 16K is operational. Some utilities that you'll find useful in writing longer programs are given in Chapter Three. Chapter Four is an interlude from serious applications, presenting five games programs that make the most of the extended graphics capabilities now available to you. The next four chapters deal with writing and debugging large programs, storing them on cassettes and printing out both the programs themselves and their results. They also introduce programs for editing databases and statistical analysis and for financial management and cover text and graphics printing. Chapter Nine takes a further look at randomness, a topic that was introduced in the first book. The final chapter introduces machine code and explains why you might want to use it.

With this book to guide you, we hope that you'll be able to discover just how versatile and powerful your ZX81 is and that you'll realise just how rewarding programming with it can be.

If you would like to be able to run the programs contained in this book without having to type them in yourself from the listings given, you will be pleased to know that they are available on two cassette tapes from Ramsoft, P.O. Box 6, Richmond, North Yorkshire DL10 4HL. Improved versions of programs in our first book, revised to take advantage of the

16K RAM pack are also included on these tapes. For details of how to order them turn to the end of the book.

M. James and S.M. Gee

PLEASE NOTE

The publishers of this book are in no way responsible for the manufacture or supply of these tapes and all enquiries and orders must be sent directly to Ramsoft at the address given.

CONTENTS

| | Page |
|---|------|
| Chapter One, EXTENDING YOUR ZX81 | 1 |
| The 16K RAM pack | 1 |
| New programming considerations | 2 |
| The ZX printer | 3 |
| How to use this book | 4 |
| Chapter Two, AN EXTRA 16K | 5 |
| Memory map and system variables | 5 |
| Manipulating memory locations | 7 |
| More memory locations | 9 |
| A memory test program | 9 |
| Chapter Three, PROGRAMMING UTILITIES | 12 |
| Utility One — Memory use | 12 |
| Utility Two — Variable use | 13 |
| Utility Three — Renumber | 19 |
| Chapter Four, 16K GRAPHICS | 23 |
| Full graphics — Depth charge game | 23 |
| Squash | 27 |
| Screen layout | 30 |
| Screen PEEKing and POKEing | 31 |
| Maze game | 33 |
| Scrolling graphics | 36 |
| Ski run game | 37 |
| Paged graphics | 41 |
| Conclusion | 43 |
| Chapter Five, DESIGNING LARGE PROGRAMS | 44 |
| Designing a program | 44 |
| Using subroutines | 45 |
| An extended RETURN | 49 |
| Collections of programs — Menus | 50 |

| | Page |
|--|------------|
| User-friendly programs | 52 |
| Debugging | 53 |
| Chapter Six, TAPE STORAGE OF DATA | 55 |
| The tape system | 55 |
| Data storage | 59 |
| Statistics program | 61 |
| Chapter Seven, NUMBER FORMATTING | 73 |
| Truncating and rounding | 73 |
| Aligning decimal points | 75 |
| PRINT USING | 77 |
| Interest calculator | 80 |
| Chapter Eight, THE PRINTER | 84 |
| How the printer works | 84 |
| Low resolution printing | 85 |
| High resolution plotting | 88 |
| Plotting a sine curve | 91 |
| A lower case or special character set | 94 |
| Conclusion | 97 |
| Chapter Nine, ADVANCED RANDOMNESS | 98 |
| Continuous events | 98 |
| The Normal Distribution | 100 |
| The CHI Squared Distribution | 101 |
| The Exponential Distribution | 101 |
| The Binomial Distribution | 102 |
| The Poisson Distribution | 102 |
| Monte Carlo integration — finding PI | 102 |
| Problems with random numbers | 105 |
| A business simulation | 107 |
| Sinclair's Triangle | 110 |
| Conclusion | 112 |
| Chapter Ten, MACHINE CODE PROGRAMMING | 113 |
| Why is BASIC so slow? | 114 |
| The characteristics of machine code | 115 |

| | Page |
|---|------|
| The ZX80 | 116 |
| The instructions LD and ADD | 117 |
| A short example program | 118 |
| A second example — reversing the screen | 122 |
| Next steps | 125 |

ACKNOWLEDGEMENTS

We would like to thank Jane Patience for her assistance in the preparation of this manuscript. Our thanks are also due to Sinclair Research for the loan of equipment at very short notice.

EXTENDING YOUR ZX81

Using a ZX81 with only 1K of memory may at first be a challenge, but there is no doubt that the challenge soon changes to frustration. As you learn to write your own programs you discover just how powerful your ZX81 is and the extent of its versatility but you realise just how restrictive having so little user memory is. Many of the features of the ZX81 — its dynamic graphics, for example — are virtually unusable within the 1K provided. Once you've mastered the fundamentals of ZX BASIC and have tried out some of your own ideas for writing programs you'll be itching to increase the scope of your machine. So it's fortunate that with the ZX81 this is so easy to do — you simply buy a 16K RAM pack and push it into the slot at the back of the machine.

The 16K RAM pack

Although there are now other 16K RAM packs on the market which operate in an identical way as far as the user is concerned, the only one that we have tried is the one produced by Sinclair Research themselves. This is the one we therefore recommend to any of you who have not already made your purchase. The pack is a compact unit containing eight 2K RAM chips which connects to the ZX81 via a double-sided edge connector. The only problem with this arrangement is that the connection between the ZX81 and the RAM pack must be maintained at all times. An accidental jog or bump that disturbs the contact will mean you lose the program in memory. We have come across various remedies for this state of affairs, including the extreme one of permanently soldering the unit on. We certainly do not advise such a course of action as it would mean you could not remove it to use some other add-on in its place. The solution that seems to us to be the

simplest and best is one that requires no hardware modifications. It is simply to use some "Blu-tak" or similar adhesive substance to make the temporary connection between the machine and its add-on RAM pack more secure. Use a fairly thick 'sausage' of the adhesive to bridge the gap between the RAM pack and the ZX81. This should be proof against the sort of slight bumps that are bound to occur. It will, of course, be insufficient if you are going to subject your ZX81 to really rough treatment — but then that's not to be encouraged in any case!

When using the printer as well as the RAM pack, the printer interface plugs directly into the ZX81 and the RAM pack plugs into the back of the printer interface. This means being even more careful about maintaining the connection to the RAM pack which is now positioned quite a distance away from the ZX81 itself. However, we have discovered that as long as we arrange the equipment on a *firm, flat surface*, we have no problems at all. Before you start programming test the security of the connection with this simple test — strike a few keys on the keyboard as if you were typing in a program. The junction between the printer interface and the RAM pack *should not flex*. If it does your arrangement is not stable enough. If necessary, try fixing all the equipment, the ZX81, the printer interface and the RAM pack, to a firm base (e.g. a piece of blockboard) with double-sided adhesive strip.

New programming considerations

Getting an extra 16K certainly makes life easier. No longer do you need to employ crafty space-saving tricks to squeeze in your programming ideas. No longer will you need to fret about how to incorporate extra features into your, already overflowing, programs. You can now relax and concentrate on good programming style instead.

You'll find that having an extra 16K makes programming your ZX81 more fun because you can tackle more difficult programs and ones that are quite different in scope. However, if you are going to make full use of its increased power you need more information and new sources of ideas. After all,

suddenly you have sixteen times as much memory available! Your ZX81 could do quite a lot before, so its new potential is really very impressive. Unleashing its capabilities is all a matter of writing programs that make use of them. That's where this book comes in. It aims both to provide an understanding of programming techniques that will use 16K of memory to the full and suggest sufficient ideas to set you off on writing the programs that will really use the new potential of your ZX81.

The ZX printer

If you've not already bought one, it's now time to consider buying a ZX printer. You'll find that you'll need one, both to enable you to write more elaborate programs and to fully appreciate their results. Once your programs grow to a length that fills the screen many times over it is virtually impossible to find the bugs in them by reading them off the screen. Nor is it advisable to write out listings by hand — you are likely to introduce more mistakes this way. The only really practical course is to list them out using the printer to enable you to look at the whole program at once.

The printer also opens up a whole new range of applications for your ZX81. It means that you can keep records — financial statements and accounts, for example — in 'hard copy'. With the 16K RAM pack you can even make the ZX81 capable of high resolution plotting. This means that you can display statistical information of all sorts graphically. Again, once you've obtained the results from such programs you'll want to be able to take them away in order to make full use of them.

The ZX printer has its disadvantages. Compared to conventional printers it is very narrow, which tends to restrict the range of applications to which it can be put, and it uses shiny, grey aluminium paper, which tends to make the output look unattractive — however, it actually photocopies very well, giving a really sharp black image. While copying information from the TV screen the printer makes the image on the screen flicker and shake, and while printing out on paper it is noisy and causes alarming "flashes". Its great advantage however, compared to all other printers yet available, is its price. For a

How to use this book

This book is the sequel to our book about the 1K ZX81, "The Art of Programming the 1K ZX81". In other words, this book goes farther along the road we set out on in that book and we will not go over in any detail the material already covered. Although the final chapter of the 1K book was specifically about special devices to enable you to cram programs into tiny amounts of memory, tricks that you will no longer need, the rest of the book is just as valid for the machine that has been upgraded to 16K. So if you find that this book plunges in at the deep end in any respect try going back to its forebear.

With the addition of 16K your ZX81 is suddenly transformed from a computer that you may have regarded as a toy to one that's in the same league as lots of "serious" micros. In the same way, this book adopts a rather more "serious" approach to its subject. If you find some of the explanations of programming theory and techniques in the next couple of chapters rather heavy going, don't worry. You can still use the programs to good effect and then come back to the explanations after you've written some longer programs of your own. If you want some fun with your new memory capability try skipping to Chapter Four. Equally, there is nothing to stop you going straight to Chapter Seven if you are keen to use your ZX81 for a statistical application. Either way, you might like to use the program suite given in Chapter Three to help you.

Chapter Two

AN EXTRA 16K

You may already know that 1K bytes of memory can be used to store 1024 characters so, by simple arithmetic, you will easily work out that 16K bytes can store 16384 characters. This sounds like a great deal of memory and you may be tempted to ask, "will I ever really need all that storage?" The answer is that although 16384 is a lot of storage it is used in so many ways that it is possible to write a program that consumes any amount of memory. For example, the one line program

```
10 DIM A(50,100)
```

will give an out of memory error (report code 4)! The point is that, while it would take you a long time to type in 16384 characters and use up all the memory, programs have ways of claiming large areas of memory for their own uses.

Memory map and system variables

In general, any memory available to the ZX81 is used in four different ways:

- 1) program storage
- 2) data storage
- 3) display storage
- 4) system storage

The most obvious use of memory is program storage. Every line that you type into the ZX81 is saved in memory and roughly speaking you can say that each line takes as many bytes to store as it took keypresses to enter it. The second use of memory is data storage. Every variable that you create (by using its name) is assigned an area of memory where its value can be stored. As a typical variable will only use 6 bytes there may seem to be plenty of room in 16K unless you start using arrays. Unfortunately, using large arrays of variables is

one of the characteristics of advanced programming!

The final two uses of memory are less obvious and less within the control of the user. The ZX81 maintains an area of memory that is used to generate the screen display. The way that the memory is used differs according to the amount of memory available. In the case of the 16K ZX81, 793 bytes are used no matter what is displayed on the screen. This may seem like a lot of memory to give up to a possibly blank screen but the advantages that this brings are well worth it (see Chapter Four). As well as needing memory for the screen, the ZX81 also needs somewhere that it can use to store 'internal' information such as where your program ends or the position of the current print position on the screen. Memory locations that are used to store such information are called 'system variables'. In any version of the ZX81, 124 bytes are given over to system variables and a knowledge of what these are can be useful.

The best way to see how the ZX81 uses its memory is via a memory map.

| <i>address</i> | <i>use</i> |
|----------------|---------------------------------|
| 16384 | System variables |
| 16509 | Program |
| D_FILE | Display file |
| VARs | Variables |
| E_LINE | Line being typed and Work Space |
| STKBOT | Calculator stack |
| STKEND | Free RAM |
| Stack pointer | Machine stack |

ERR_SP

RAMTOP

32767

| |
|-------------|
| Gosub stack |
| Free RAM |

From this you should be able to see that some of the divisions of the memory occur at fixed addresses and others at variable addresses. For example, the program area always starts at 16509 but the display file starts at a variable position that depends on how long the program is. So that the ZX81 can find out very quickly what the addresses are, they are stored in fixed locations in the 'system variables' area of memory. You can find a complete list of these and other locations in Chapter 28 of your ZX81 manual. The use of these locations is easy enough once you have got over the idea of a pair of memory locations holding the ADDRESS of another memory location. For example, the system variable called D_FILE is stored at location 16396 and 16397 and these two locations are used to store the address of the start of the area of memory used for the TV display. It is important to note that BASIC does not recognise the name D_FILE and if you want to gain access to what is stored in D_FILE you have to use its address.

Manipulating memory locations

The two BASIC statements that allow us to manipulate memory locations are PEEK and POKE. The function

PEEK address

returns the contents of the memory location specified by 'address' and

POKE address, value

stores 'value' in the memory location specified by address. A single memory location can hold a number between 0 and 255 but the address of a ZX81 memory location can lie between 0 and 65534. So, to store a single address we have to use TWO

memory locations. The way that this works is based on the principles of binary numbers but from the point of view of a BASIC programmer this is an unnecessary complication and it is best understood and easier to remember in terms of the decimal numbers that are used in programming statements. The largest number that a single memory location or byte can hold is 255. So if you use a single memory location to count, you can start at zero and count quite happily until you reach 255. If you try to add one to 255 you cannot store the answer 256 in a single memory location but you could store one in another memory location to indicate that you have reached 256 once. You could then carry on counting in the first memory location as if nothing had happened (starting again at zero) until you reach 255 again. In fact, what you are doing is to use the second memory location as a counter of the number of times that you have reached 256. The first memory location counts single things and is known as the 'least significant byte' and the second memory location counts 256s and is known as the 'most significant byte'.

| | |
|-----------------------------------|-----------------------------------|
| first memory location units | second memory location 256s |
|-----------------------------------|-----------------------------------|

least significant

most significant

It should now be obvious how to 'reconstruct' an address from two memory locations. As the first counts units we can just PEEK its value but the second counts 256s so its value must be multiplied by 256 before it is added to the first value. Translating this into BASIC gives:

```
PEEK M + 256 * PEEK(M+1)
```

as the way of finding out what is stored in two memory locations, the address of the first being stored in M. Going the other way is just as easy. If you want to split a number up so that it can be stored in two memory locations, first divide it by 256 to find out how many 256s it contains and store the

result in the most significant byte and then store the remainder in the least significant byte. That is

```
POKE M+1,INT(V/256)
POKE M,V-256*INT(V/256)
```

will store the number in V in the two memory locations M and M+1. Both of these PEEK and POKE methods will be used later on.

More memory locations

The digression about memory addresses interrupted our examination of the memory map before we had considered the final four areas. To resume, the calculator stack is an area of memory that the ZX81 uses to store any temporary results generated while it is doing arithmetic. The size of the calculator stack varies while a program is running according to what calculations are actually being done. The machine stack is different from the other areas of memory in that there is no way to find out where it ends. This information is stored in a location that BASIC has no access to called the 'machine stack pointer', but this doesn't really matter because nothing of any use to a BASIC programmer is stored in this region. The GOSUB stack is used to save the line number that a RETURN statement uses to return control from a subroutine. The final element in the memory map is 'free RAM', in other words, RAM that is available for use by the programmer. The system variable RAMTOP is normally set to the address of the highest RAM location when the ZX81 is first switched on. However it is possible to change the value stored in RAMTOP to reserve some memory for special purposes, such as a machine code subroutine (see Chapter Ten).

A memory test program

As an example of how to use system variables let's consider the problem of writing a program to test the ZX81's memory. The most obvious method of checking that a memory location is working is to store something in it, read it back and see if it

is unchanged. There is a little problem about what to store in the memory location to test it because it is possible for a location to store some values without any trouble but fail with others. The answer is to use the ZX81's random number function RND to generate a wide range of values. Thus our memory test program will start at the beginning of the area of free memory and store a random value in the first location and then read it back to see if it has been stored correctly and will repeat this for every location in the free area. The reason why only the free area is checked is that storing anything in an area of memory that the memory check program was using would cause it to crash! The resulting program is:

```

10 PRINT "+";
20 LET E=PEEK 16386+256*PEEK 16387 -40
30 LET M=PEEK 16412+256*PEEK 16413 +20
40 LET M=M+1
50 IF M>E THEN GOTO 10
60 LET R=INT(RND*256)
70 POKE M,R
80 IF PEEK(M)=R THEN GOTO 40
90 PRINT "ERROR AT ";M;" EXPECT ";R
100 PRINT "ACTUAL ";PEEK M
110 GOTO 40

```

Line 10 starts the program by printing a plus sign to show that the program is working. Lines 20 and 30 are concerned with finding the address of the area of free memory. The variable E is set to the address of the end of the free area by PEEKing the locations corresponding to ERR_SP which gives the address of the end of the GOSUB stack, and subtracting 40 to leave enough space for the machine stack. Notice that there is no way to find out the exact location of the end of the machine stack. The variable M is set to the address of the start of the free area by PEEKing the locations corresponding to STKEND which give the address of the end of the calculator stack and hence the start of the free area and then adding 20. You may think that adding 20 is unnecessary because STKEND gives the exact address of the start of the free area, but as the memory test program runs it carries out some

arithmetic and the calculator stack can increase in size so it is important to allow 20 locations for this growth. Lines 40 to 80 test each location in turn. A random number is generated at line 60 and stored in the memory location under test by line 70. It is then recalled and compared to the original value in line 80. If it is the same then the next memory location is checked by going back to line 40 and adding one to M. If an error is found then an error message is printed by lines 90 and 100 and testing of other memory locations continues. The only other point to notice is that line 50 tests to see if all the free memory has been checked. If it has then another plus is printed and the testing starts over.

If you run the memory check program you might think that it isn't working because although a single plus sign appears on the screen nothing else happens. The reason for this is that it takes about half an hour to test the entire memory! This example shows that speed of operation is going to be one of the problems of writing programs on the ZX81!

PROGRAMMING UTILITIES

A utility is generally understood as a program that helps in the task of writing other programs. In some senses the memory check program described in the last chapter is a utility, in that it tests to see if the machine is working before you write a program. With the 16K of memory installed on the ZX81 we can now consider writing a number of small programs that make life a little easier when writing larger programs. For example, during program development it is often useful to know how much memory is free and how it is being used. It is quite possible to use our knowledge of the memory layout and system variables to write a program that will print out a summary of how memory is used. However, some of the techniques used are quite involved, so you may prefer to skip the explanations of how this and the rest of the utilities described in this chapter work and just use them without worrying about their finer points until you have gained more expertise in programming your ZX81.

The three utilities described below all have line numbers in the 9000 region and each uses a different set of line numbers so that you can have all three of them loaded at the same time. In Chapter Five we will see how to organise them so that they are easier to use as a group rather than as individual programs.

Utility One – Memory use

The memory use utility is a fairly short program which can be used in conjunction with any program you write:

```
9200 PRINT "PROGRAM ";PEEK 16396+256*PEEK
16397-16509
9210 PRINT "DISPLAY ";PEEK 16400+256*PEEK
16401-PEEK 16396-256*PEEK 16397
9220 PRINT "VARIABLES ";PEEK 16404+256*
```

```
PEEK 16405-PEEK 16400-256*PEEK 16401
9230 PRINT "SPARE ";PEEK 16386+256*PEEK
16387-PEEK 16412-256*PEEK 16413
9240 STOP
```

To use this program, load it from tape before starting to write another program and the amount of memory left can be found at any time by typing GOTO 9200. The amount of space taken by a program is calculated by the difference between the address of the beginning of the 'program area', 16509 and the address of the end of the 'program area' stored in D_FILE (16396 and 16397). The amount of memory used by the variables and the display is calculated in a similar fashion. Notice that the memory usage utility creates no extra variables of its own so the amount of memory used to store variables is entirely due to the program being developed. However, the utility does take up space in the 'program area' so the amount of memory used by the program includes the space taken by the utility. The size of the memory left is calculated by finding the difference between the address stored in STKEND and the address stored in ERR_SP. Notice that this does not take into account the memory used by the machine stack and so the reported figure is too large by 5 to 10 locations.

Utility Two – Variable use

Although the previous utility let us know how much memory was being used by the variables, it didn't give any idea of which variables were taking a lot of space. If you're writing a long program it can very often happen that you forget the names of the variables that you have already used. This means that when you come to creating a new variable you run the risk of using the same name twice. The utility listed below can help with keeping track of the variables that you are using at any point in a program's development. It works by looking at the 'variables area' of memory and listing the name, type and current value (if convenient) of each variable. With our current knowledge of the memory layout, finding the start and end of the 'variables area' is easy. What is more difficult is interpreting

the information that we find there. To do this we need to know a little about how the variables are stored.

The ZX81 recognises six different types of variable although some of these look the same to the user. Each type of variable takes a different number of memory locations to store its value and other information about itself. The one thing that they all share in common, however, is that in each case the first memory location is used to identify the type of variable. To save space, the first location is also used to store the first (and possibly only) letter of the variable name. The way that this is done is quite simple. As the first character of a variable name *must* be a letter, which letter it is can be represented as a code in the range 1 to 26. Then, as there are only six variable types, we can assign each a code as follows:

| code | variable type |
|------|---|
| 2 | String of any length |
| 3 | Simple variable (i.e. single letter name) |
| 4 | Array of numbers |
| 5 | Variable with name longer than one letter |
| 6 | Array of characters |
| 7 | Index variable used in a FOR loop |

Because both pieces of information have a restricted range they can both be stored in a single memory location. Technically, what happens is that the most significant three bits are used to store the type and the least significant five bits are used to store the letter that is the variable's name:

| | | | | | | | | |
|-----|------|---|---|--------|---|---|---|---|
| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | T | T | T | L | L | L | L | L |
| | type | | | letter | | | | |

The most important point to note, however, is that if V0 is the address of the first memory location of a variable then you can separate the type and name using the following two lines:

```
T0=INT(PEEK(V0)/32)
C$=CHR$(PEEK(V0)-T0*32+32)
```

where T0 contains the type and C\$ the letter. This use of the first location makes good sense because, by looking at the first

location, you can decide what the type and name of the variable is and thus how to treat it. This is the strategy adopted by the utility listed below:

```
9300 LET V0=PEEK 16400+256*PEEK 16401
9310 PRINT "VARIABLE";TAB(10);"TYPE";
      TAB (20);"VALUE"
9320 DIM C$(10)
```

```
9330 IF PEEK(V0)=128 THEN STOP
9340 LET T0=INT(PEEK(V0)/32)
9350 LET I0=1
9360 LET C$=""
9370 LET C$(I0)=CHR$(PEEK(V0)-T0*32+32)
```

```
9380 IF T0=3 THEN GOTO 9450
9390 IF T0<>5 THEN GOTO 9480
9400 LET V0=V0+1
9410 LET T0=INT(PEEK(V0)/64)
9420 LET I0=I0+1
9430 LET C$(I0)=CHR$(PEEK(V0)-T0*64)
9440 IF T0=0 THEN GOTO 9400
9450 PRINT C$;TAB(20);VAL C$
9460 LET V0=V0+6
9470 GOTO 9330
```

```
9480 IF T0<>7 THEN GOTO 9520
9490 PRINT C$;TAB(10);"INDEX";TAB(20);VAL C$
9500 LET V0=V0+18
9510 GOTO 9330
```

```
9520 IF T0<>2 THEN GOTO 9640
9530 LET I0=I0+1
9540 LET T0=PEEK(V0+1)+256*PEEK(V0+2)
9550 LET C$(2)="$"
9560 PRINT C$;TAB(10);"LEN";T0;TAB(20);
9570 LET V0=V0+3
9580 IF T0=0 THEN PRINT
9590 IF T0=0 THEN GOTO 9330
```

```

9600 PRINT CHR$(PEEK V0);
9610 LET V0=V0+1
9620 LET T0=T0-1
9630 GOTO 9580

9640 IF T0=6 THEN LET C$(2)="$"
9650 LET I0=0
9660 PRINT C$;TAB(10);"DIM(";
9670 PRINT PEEK(V0+4+I0*2)+256*PEEK(V0+5+
    I0*2);
9680 LET I0=I0+1
9690 IF I0<>PEEK(V0+3) THEN PRINT ",";
9700 IF I0<>PEEK(V0+3) THEN GOTO 9670
9710 LET V0=V0+3+PEEK(V0+1)+256*PEEK(V0
    +2)
9720 PRINT ")"
9730 GOTO 9330

```

Line 9300 sets V0 to the start of the 'variables area' and lines 9340 and 9370 separate the variable type into T0 and the first letter of the name into C\$(1). What happens next depends on the type of the variable, i.e. the value stored in T0.

If T0 is equal to three then we have a simple variable. Line 9380 tests for this condition and, if it is true, control passes to line 9450 where the name of the variable and its value are printed. Printing the variable's name is easy because it is stored in C\$ but printing its value depends on the use of the function VAL. The BASIC function VAL is well worth knowing about because it can be used to work out an arithmetic expression stored in a string variable. Remembering that a single variable name is a special case of an arithmetic expression, we can see that PRINT VAL C\$ will print the contents of the variable whose name is in C\$ — which is exactly what we want to do! After printing this information the only thing that is left to do is to adjust V0 so that it contains the address of the first location of the next variable. Knowing that a simple variable uses five memory locations to store its value (ZX81 manual, page 172), you should be able to see that line 9460 leaves V0 pointing at the start of the next location.

If T0 is equal to five we have to deal with a numerical variable with a name longer than one letter. As we have solved the problem of printing the value stored in a numerical variable once we have its name, the only extra difficulty comes from building up the full name of the variable C\$. This is what happens in lines 9400 to 9440. Each additional character is extracted by line 9430 along with a new type value T0 in line 9410. The full name is built up by adding each character to C\$ and the last character of the name is detected by T0 being not equal to zero. After extracting the last character we can handle the printing and adjustment of V0 in exactly the same way as a simple variable.

If T0 is equal to seven then we have an index variable to deal with. The only difference between an index variable and a simple variable is that it takes 18 memory locations to store its value. You should be able to understand the method employed in lines 9490 to 9500 by comparing it to the way a simple variable is handled.

If T0 is equal to two then we have a string to deal with. As the name of a string can only be a single letter the only thing that we have to do to construct its full name is to add a "\$" sign to C\$ in line 9550. Unfortunately, we cannot use the VAL function to print the value of the string variable because VAL can only evaluate arithmetic expressions. The solution to the problem of printing the string value is to notice that the length of the string is stored in the two memory locations following its name.

| | | | | | | | |
|-----------------|------|---|--------|---|--------|-----|------------|
| memory location | 1 | 2 | 3 | 4 | 5 | ... | length + 3 |
| | name | | length | | string | | |

By PEEKing these two locations (line 9450) we can print the length of the string in line 9560. Using this information we can then PEEK each character of the string in turn and using the CHR\$ function print it on the screen. We know that we have come to the end of the string when the number of characters printed is equal to its length (lines 9580 and 9590).

If T0 is equal to four or six then we have an array of numbers or characters to deal with. The only difference between the way these two are handled is that a "\$" is added

to the single letter name in the case of the string array by line 9640. There is no point in trying to print the values that the array contains because this might produce so much information that it would be overwhelming! What seems to be more useful in the case of an array is to print its dimensions. To do this requires us to look at the way arrays are stored. From Chapter 27 of the ZX81 manual you should be able to see that the third memory location of an array always holds the number of dimensions of the array and the sizes of these dimensions are stored, two locations to each, before the actual values of the array.

| | | | | | | | | | |
|-----------------|------|--------|----------------|---------|---------|---|---|---|-----|
| memory location | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
| | name | length | no. of dims | 1st dim | 2nd dim | | | | |

By PEEKing the fourth location in lines 9690 and 9700 it is possible to decide if the sizes of all the dimensions have been printed. The size of each dimension is obtained by line 9670 and the number of dimensions obtained is recorded in I0 by line 9680. After all this information has been printed V0 is adjusted to the address of the next variable by adding the contents of locations two and three which contain the total length of the array in memory locations (line 9710).

There are only two things left to explain about the way that the program works. Firstly, how does the program know that it has printed all the variables? Quite simply, the last memory location in the 'variables area' contains 128 and this is checked for by line 9330. Secondly, why is C\$ dimensioned? Surely it would be easier to use a string? The reason why C\$ is dimensioned is that strings can change the amount of memory that they take up while a program is running. If this utility used a string then the 'variables area' would be rearranged every time the length of C\$ changed — this would be disastrous for a program trying to analyse the 'variables area'! A string array however always occupies the same amount of space.

You may be feeling that this utility is very difficult to understand but if you look at each section in turn you should be able to make sense of it. If you find you can't, don't let it

stop you from using it! Here is an example of the sort of information it will give you:

```

1 LET A=100
2 LET MAX=80120
3 LET A$="MIKE"
4 FOR I=1 TO 5
5 DIM Z(1,7,9)

```

⋮

| VARIABLE | TYPE | VALUE |
|----------|------------|-------|
| A | | 100 |
| MAX | | 120 |
| A\$ | LEN=4 | MIKE |
| I | INDEX | 1 |
| Z | DIM(1,7,9) | |

⋮

Utility Three — Renumber

The third and final utility is a short renumber program. It is almost impossible to write a program first time round with neatly numbered lines. Something always goes wrong and you have to add out-of-sequence lines. It would be very useful to have a program that could renumber the finished product to give regular line numbers. That is, however, very difficult to do in BASIC because a good renumber program should adjust not only the numbers at the start of each line but any line numbers referred to by GOTO and GOSUB statements. If you simply change all the line numbers but leave the GOTOs and GOSUBs alone then the final program will not work. While it is possible to think of ways in which the GOTOs and GOSUBs could be changed from BASIC it would produce a rather large program. As a sort of compromise, the utility given below renumbers all the lines and gives a list of how the old line numbers have been changed to give the new line numbers. Using this list it is easy to change all the GOTOs and GOSUBs to the new line numbers by using the EDIT command.

Before we can understand the way that the utility works it is necessary to examine the way BASIC statements are stored in the "program area". Chapter 27 of the ZX81 manual supplies the following information:

| | | | |
|---------|---------|------|---------|
| 2 bytes | 2 bytes | text | newline |
|---------|---------|------|---------|

line no. length
 of text
 + newline

As can be seen, the first two locations of the statement contain the line number. The only thing to note is that, unlike all the other two byte numbers that we have looked at, the line number is stored with the most significant byte FIRST. So to PEEK it we must use

`256*PEEK V0 + PEEK (V0+1)`

where V0 contains the address of the first location. To change the line number all we have to do is to POKE the two locations with the new value. This is only complicated by the fact that we have to split the new line number up into two smaller numbers. The easiest way to do this is

most significant byte = `INT(S0/256)`
and
least significant byte = `S0-256*INT(S0/256)`

where S0 is the new line number. After POKEing the new line number the only problem is moving onto the next line. This can be done by PEEKing the second and third locations for the length of the text and adding this to V0. Putting these ideas into a program gives:

```
9800 LET V0=16509
9810 PRINT "START=";
9820 INPUT S0
9830 PRINT S0
9840 PRINT "STEP=";
9850 INPUT I0
```

```
9860 PRINT I0
9870 PRINT
9880 PRINT "OLD";TAB(6);"NEW"
9890 LET L0=256*PEEK V0 + PEEK (V0+1)
9900 IF L0>=9000 THEN STOP
9910 PRINT L0;TAB(6);S0
9920 POKE V0,INT(S0/256)
9930 POKE V0+1,S0-256*INT(S0/256)
9940 LET S0=S0+I0
9950 LET V0=V0+4+PEEK(V0+2)+256*PEEK(V0
+3)
9960 GOTO 9890
```

Line 9800 sets V0 to the start of the "program area" and lines 9810 to 9870 get the line number that the renumbered program should start at (in S0) and the amount that they should go up by in I0. Line 9890 finds the current line number in L0. This is printed along with the new line number at 9910. The new line number is then POKEd into the correct place by lines 9920 and 9930. The utility then proceeds to the next program line by adding the increment I0 to the new line number in 9940 and adjusting V0 to point to the start of the next statement line in 9950. The utility stops when an old line number in the 9000s is detected by line 9900 — after all there is no point in the utility renumbering itself or any of the other utilities!

When you run a program with this utility you will have to supply two pieces of information, the number you want the first line to have, probably 10, and the increment between the lines, also probably 10. Enter the first value when the prompt "START=" appears on the screen and the second after the prompt "STEP=".

```
START=10
STEP=10
OLD      NEW
10       10
11       20
12       30
13       40
14       50
```


This short renumber routine is very useful as it stands but it could form the basis for a more advanced renumber program. For example, you might want to modify it to renumber only part of the program or to tackle the difficult problem of handling GOTOs and GOSUBs.

16K GRAPHICS

You might think that having more memory wouldn't alter the way that you program graphics on your ZX81 but you would be wrong! At the simplest level having more memory means that you can use all of the screen area without being in danger of running out of memory — a full screen needs approximately 700 memory locations and this leaves very little left over from 1K. At the most advanced level the 16K machine organises its screen display in a very different way from the 1K machine and this fact can be used to produce some impressive displays very easily.

Full graphics — Depth charge game

Before dealing with anything new it might be a good idea to write a graphics game based on the techniques introduced in the 1K book. Rather than take any of the programs found in the 1K book and change them so that they use the whole screen, which is left for the reader to do, a brand new program will be used to show how easy it is to write such programs once the 1K limit is removed.

The idea behind the game is to animate two ships — one a surface vessel and the other a submarine — and allow the first one to “drop” a depth charge on the other. If you followed the discussion in the 1K book then you should be able to see how to write this program for yourself. To animate the two ships we could use the general method of plotting a shape at the first position, blanking it out and replotting it moved on a little. However as the ships are going to move horizontally in a straight line we can use a trick to simplify the programming. Try the following program:

```
10 FOR X=0 TO 30
20 PRINT AT 3,X;" *";
```

30 NEXT X

You should see an asterisk move from left to right at the top of the screen. Notice that this program combines the replotting of the asterisk with the unplotting of the previous position by including a blank in the moving "object". In general, it is always possible to use this technique. All you have to do is to ensure that whenever the "object" is printed it includes enough blanks around it to "wipe out" the old version. In practice, this is too difficult unless the object is moving in a straight line. (For another example of this method see "Squash".)

The only other problems with the depth charge game are deciding when the submarine is hit and producing a suitable "explosion" to remove it from the screen. The complete program is listed below. Note that graphics characters are indicated by square brackets around the letter on the key that you should press to produce it. So [A] is the graphics character on the A key and [] is an inverse space i.e. a black block. Whenever you see a character enclosed in square brackets get into "GRAPHICS" mode and then press the shift key before typing in. Note also that lines 230, 430 and 530 need 4 spaces between the double quotes.

```

10 LET MC=0
20 LET HC=0
30 LET F=0
40 LET H=0
50 LET XS=0
60 GOSUB 600
70 LET X=3
80 LET YS=INT(RND*5)+15
90 LET Y=2
100 GOSUB 200
110 PRINT AT 0,0;"HITS=";HC;TAB(10);"MISSES
    =";MC
120 IF F=0 THEN LET XD=2*X
130 IF F=0 THEN LET YD=35
140 IF INKEY$="F" AND F=0 THEN LET F=1

```

```

150 IF F=1 THEN GOSUB 300
160 GOSUB 400
170 IF H=0 THEN GOTO 100
180 GOSUB 500
190 GOTO 30

```

```

200 PRINT AT Y,X;" [6][ ][6][6]";
210 LET X=X+1
220 IF X>25 THEN LET X=0
230 IF X=0 THEN PRINT AT Y,25;"    ";
240 RETURN

```

```

300 UNPLOT XD,YD
310 LET YD=YD-1
320 PLOT XD,YD
330 IF YD=2 THEN LET MC=MC+1
340 IF YD=2 THEN LET F=0
350 IF F=0 THEN UNPLOT XD,YD
360 IF ABS(XD-2*XS-6)>6 THEN RETURN
370 IF YD<>43-2*YS THEN RETURN
380 LET H=1
390 RETURN

```

```

400 PRINT AT YS,XS;" [6][6][W]";
410 LET XS=XS+RND
420 IF XS>27 THEN LET XS=0
430 IF XS=0 THEN PRINT AT YS,27;"    ";
440 RETURN

```

```

500 LET HC=HC+1
510 FOR I=1 TO 20
520 PRINT AT YS,XS;"[A][A][A][A]";
530 PRINT AT YS,XS;"    ";
540 NEXT I
550 RETURN

```

```

600 CLS
610 FOR I=0 TO 31

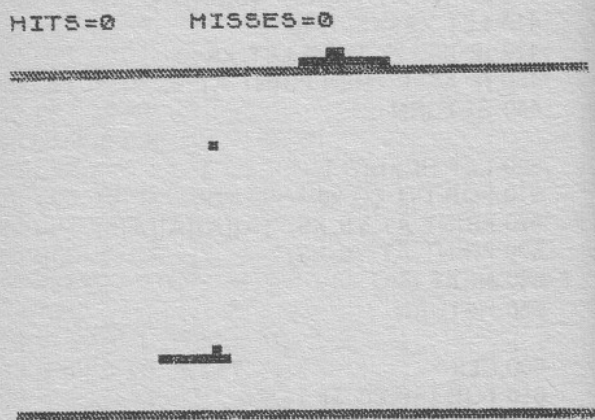
```

```

620 PRINT AT 3,1;"[S]";
630 PRINT AT 21,1;"[D]";
640 NEXT I
650 RETURN

```

The program starts by initialising variables and drawing the sea and sea bed using subroutine 600. The depth of the submarine is selected at random by line 80. Subroutine 200 plots the surface ship at location X,Y and subroutine 400 plots the submarine at location XS,YS. The submarine moves in the same direction as the surface ship and the amount that it moves is random (line 410). The depth charge is released by pressing the "F" key and line 140 checks for this, using the INKEY\$ function. When the depth charge has been dropped the variable F is set to one and the location of the depth charge is then stored in XD,YD. Subroutine 300 is responsible for keeping track of where the depth charge is and keeping it moving. Notice that, as the depth charge is produced using the PLOT/UNPLOT commands and the ships are produced using PRINT AT, there is a problem with comparing co-ordinates. PLOT works with a screen 64 by 44 and PRINT AT works with a 32 by 22 screen. Obviously XD and YD are simply twice the size of a similar X and Y except that YD is measured



from the bottom of the screen and Y is measured from the top! All this makes starting the depth charge off and deciding if it has hit the submarine more difficult than you might expect. Lines 120 and 130 keep XD and YD set to the same location as the surface ship, so that when the depth charge is dropped it starts falling from the current position of the surface ship. Lines 360 and 370 check to see if the depth charge has hit the submarine. Notice that, to compare the two X co-ordinates, it is enough to multiply by two but to compare the two Y co-ordinates you have to subtract $2*YS$ from 43. The rest of the program is fairly straightforward but notice the way that the submarine is "destroyed" in lines 510 to 540, it could be useful in other games!

Squash

As another example of full screen graphics consider the squash program given below:

```

10 LET BALL=0
20 LET BALL=BALL+1
30 CLS
40 LET A=10
50 LET B=10
60 LET V=1
70 LET W=1
80 LET X=10
90 LET Y=15
100 GOSUB 500
110 PRINT BALL
120 GOSUB 200
130 GOSUB 700
140 GOSUB 300
150 IF B<>21 THEN GOTO 120
160 GOTO 20

200 LET A$=INKEY$
210 IF A$="5" THEN LET X=X-1
220 IF A$="8" THEN LET X=X+1

```



```

300 PRINT AT B,A;" ";
310 LET A=A+V
320 LET B=B+W
330 IF A=31 OR A=0 THEN LET V=-V
340 IF B=1 THEN LET W=-W
350 IF B+1=Y THEN GOSUB 400
360 PRINT AT B,A;"[ ]";
370 RETURN

```

```

400 LET R=A-X
410 IF R<1 OR R>3 THEN RETURN
420 LET W=-W
430 RETURN

```

```

500 FOR I=0 TO 31
510 PRINT AT 0,I;"[ ]";
520 NEXT I
530 RETURN

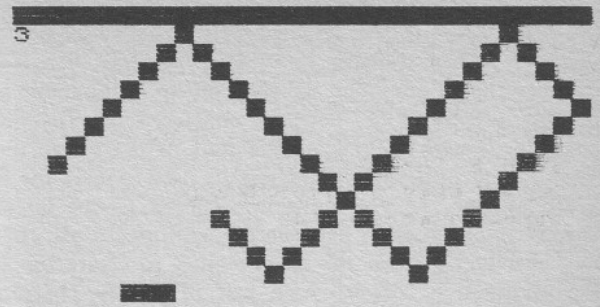
```

```

700 PRINT AT Y,X;"[ ][ ]";
710 RETURN

```

This is a simple application of the moving graphics methods introduced in the 1K book. The program starts by printing the walls of the court using subroutine 500. Subroutine 300 plots the ball at location A,B and keeps track of the ball's horizontal velocity V and vertical velocity W. If the ball hits a "wall", i.e. tries to go off the screen, then the correct velocity is reversed to make it bounce. Subroutine 700 prints the bat at location X,Y. Notice that erasing the old bat is unnecessary because of the blank included at each end of the bat. The only two new features are the way that the bat is moved and the way the ball is made to bounce off the bat. To move the bat all that is necessary is to change the bat's horizontal co-ordinate according to which key is pressed. If the left arrow key is pressed then one is subtracted from X, moving the bat one place to the left. If the right arrow key is pressed then one is



(The routine that blanks out the ball has been temporarily removed in order to produce this illustration.)

added to X, moving the bat one place to the right. All this is done by subroutine 200. No attempt is made to ensure that the bat stays on the screen because once again we come up against the slowness of the ZX81 and to include such a check would slow things down even more. The ball bouncing off the bat is carried out by subroutine 400. If the ball is at the same vertical position as the bat then line 350 calls subroutine 400 to check if it has the correct horizontal position to be bounced, i.e. have its vertical velocity reversed. If the ball goes off the screen at the bottom (detected by line 150) then control passes to line 20 and a new game is started. Notice that the game is played by lines 120 to 140 being repeated over and over again. Line 120 calls the move bat routine, line 130 actually moves the bat and line 140 calls the routines that look after the ball and its bouncing.

You should now be in a position to expand and improve programs such as depth charge and squash and to use them as the basis of your own games. But before we move on to something new, notice how, in the case of the 1K ZX81, program features were left out because of the lack of program space and now, in the 16K machine, features are left out because of the lack of speed!

Screen layout

When the ZX81 is first switched on it checks to see how much memory it has available to it. If it has less than 3½K of memory then a minimum screen is set up. A minimum screen consists initially of 25 "newline" characters. The first newline character is used to indicate the start of the screen display. The subsequent newline characters are used to mark the end of a display line. As you print information to the screen the characters are inserted between the appropriate pairs of newline characters. When the ZX81 comes to create the screen the information between each pair of newline characters is displayed. As each line on the screen must be 32 characters long, if there are less than 32 characters between newlines, the ZX81 substitutes enough "blank" characters to form a full line. It is important to realise that a "blank" is just as much a character as the letter "A" and takes up the same amount of memory if you want to store it. You should now be able to see that there are two ways that a completely blank line can be produced on the screen. Either by storing 32 blank characters between "newlines" or by two "newlines" being next to each other. In the first case the ZX81 would send 32 blanks to the screen because they are stored in memory and in the second case it would send enough blanks to make the line complete. The first method requires 32 extra memory locations and the second requires none! For example, suppose that the first part of the display area of memory looks like this.

```
|NL|NL|H|I|BL|T|H|E|R|E|NL|NL|. . .|NL|
```

Where NL is the code for "newline", BL is the code for "blank" and all other letters should be replaced by their codes e.g. "H" is 45. Then the screen display would be constructed as follows. The first NL simply marks the start of the screen. The second NL marks the end of the first display line so the ZX81 would check to see how many characters had been printed. On finding the answer to be zero it then sends 32 blanks to make a full display line. Following the second NL are eight character codes and these are displayed on the screen

in turn until the next NL is reached when 24 blanks are sent to make another full line and so on to the end of the display.

This method of producing a display may seem complicated but it does save having to use memory to store blank lines. However, if there is sufficient memory, i.e. more than 3½K, the ZX81 does use the simpler, if slightly more wasteful, method of setting up a screen consisting of 24 lines of 32 blanks. The "newline" characters are still used to signal that the display should start a new line so the screen displays of the 1K and 16K ZX81 share the same basic structure. As the screen is used, some of the blanks are replaced by other characters but the screen always stays the same size. There is one exception to this rule, however, and that is the SCROLL command. Issuing a SCROLL command makes the whole screen display move up by one line, the top line being lost. This is done by introducing a blank line at the bottom of the screen consisting of a single "newline" character. So after each SCROLL a short line is introduced to the display. After 22 SCROLLs the screen is back into its 1K form with all the extra blanks removed. To get back to the full screen all that is necessary is to clear the screen using CLS.

Screen PEEKing and POKEing

What we have discovered in the last section is that a 16K screen undisturbed by SCROLL has a fixed layout in memory. The first screen memory location's address is in D_FILE and it contains a "newline" character. The next 32 memory locations contain the character codes of whatever 32 characters appear on the first line of the screen. The next location after that contains another "newline" and so on to the last line of the screen. What this means is that we can easily work out the address of any location on the screen and use it either to change what is displayed or more importantly to discover what is being displayed.

If D is the address of the start of the display file then the memory location that corresponds to the column number stored in X and the row number stored in Y is:

$$D + X + Y * 33 + 1$$

because each row is 32 characters plus one "newline" long and the display area starts with a "newline". Using this formula it is possible to POKE new characters anywhere on the screen including the bottom two lines that are normally reserved for input. Try the following program:

```
10 CLS
20 LET D=PEEK 16396 + 256*PEEK 16397
30 FOR X=0 TO 31
40 FOR Y=0 TO 23
50 LET A=D+X+Y*33+1
60 POKE A,128
70 NEXT Y
80 NEXT X
```

You should see the screen fill with black squares including the two forbidden lines at the bottom. In general you can replace

```
PRINT AT Y,X;A$;
```

where A\$ contains a single character by

```
POKE D+X+Y*33+1,CODE(A$)
```

where D is the address of the start of the display file. The reasons why you might want to POKE characters onto the screen are various but include wanting to use the bottom two lines and needing faster printing.

PEEKing the screen is much more important than POKEing because it provides the only way of finding out what is already on the screen. To find the code of the character at screen location X,Y use:

```
LET C=PEEK(D+X+Y*33+1)
```

Where D is the address of the start of the display file and C is the code of the character. If you're wondering why you would ever want to know what character is at a particular position on the screen then perhaps the next game will serve as a demonstration.

Maze game

The maze game is a ZX81 version of a game found on many a popular micro. The basic idea is that you have control over the movements of an asterisk that starts out in the bottom right hand corner of the screen and the object of the game is to get it into the top left hand corner. Sounds easy doesn't it! The catch is that your way is blocked by a randomly changing pattern of dark squares and the skill is to steer your way as quickly as possible through any openings before they close.

Before looking at the completed program try to think how you might go about writing it. The problem is obviously knowing when your way is blocked. Do you really have to record the X and Y co-ordinate of every dark square on the screen?

```
10 CLS
20 PRINT "DIFFICULTY LEVEL 1-9 ?"
30 INPUT L
40 LET L1=L/10
50 CLS
60 GOSUB 100
70 GOSUB 300
80 GOSUB 500
```

```
100 LET D=PEEK 16396+256*PEEK 16397
110 RETURN
```

```
200 LET C=PEEK(D+X+Y*33+1)
210 RETURN
```

```
300 FAST
310 GOSUB 1000
320 FOR I=1 TO 20*L+40
330 LET X=RND*29+1
340 LET Y=RND*19+1
350 PRINT AT Y,X;"[A]";
360 NEXT I
370 SLOW
```

```

380 PRINT AT 1,1;"$";
390 PRINT AT 20,30;"*";
400 RETURN

```

```

500 LET XC=30
510 LET YC=20
520 LET M=0
530 LET X=XC
540 LET Y=YC
550 LET A$=INKEY$
560 GOSUB 900
570 IF A$="" THEN GOTO 550
580 IF A$="5" THEN LET X=XC-1
590 IF A$="8" THEN LET X=XC+1
600 IF A$="6" THEN LET Y=YC+1
610 IF A$="7" THEN LET Y=YC-1
620 GOSUB 200
630 IF C=13 THEN GOTO 800
640 IF C<>0 THEN GOTO 530
650 PRINT AT YC,XC;" ";
660 LET XC=X
670 LET YC=Y
680 PRINT AT YC,XC;"*";
690 LET M=M+1
700 GOTO 530

```

```

800 CLS
810 PRINT AT 0,2;"YOU HAVE TAKEN ";M;
  " MOVES"
820 PRINT "ANOTHER GAME Y/N"
830 INPUT A$
840 IF A$(1)="Y" THEN RUN
850 IF A$(1)<>"N" THEN GOTO 820
860 STOP

```

```

900 IF RND>L1 THEN RETURN
910 LET C$=""
920 IF RND>.5 THEN LET C$="[A]"

```

```

930 PRINT AT RND*19+1,RND*29+1;C$
940 PRINT AT 1,1;"$";
950 RETURN

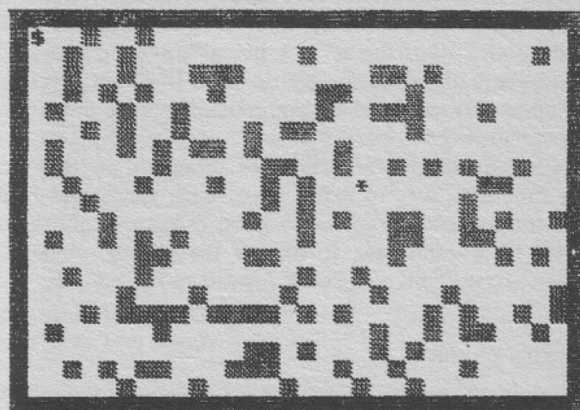
```

```

1000 FOR I=0 TO 31
1010 PRINT AT 0,I;"[ ]";AT 21,I;"[ ]";
1020 NEXT I
1030 FOR I=0 TO 21
1040 PRINT AT I,0;"[ ]";AT I,31;"[ ]";
1050 NEXT I
1060 RETURN

```

The program starts off by asking for the difficulty level, L. This governs how many squares are used to block your way. This information is used by subroutine 300 to construct the initial maze. The details of this are straightforward. First a call to subroutine 1000 draws a border around the maze. Next a PRINT AT is used to print the graphics block [A] at random points on the screen. Notice how FAST and SLOW are used to hide the maze while it is being built and to get things done as quickly as possible. Before leaving the subroutine a \$ sign is printed in the top left hand corner to represent the target position for the asterisk which is printed in the bottom right



hand corner. The game proper begins with a call to subroutine 500. After some initialisation, the arrow keys are checked by line 550 and the values of XC and YC, the present position of the asterisk, are updated into X and Y, the intended position of the asterisk, according to which arrow key has been pressed. The next problem is to decide if the intended position is legal. If it contains only a blank character then it is OK to move the asterisk there. If it contains any other character then the move must be rejected. Subroutine 200 is called at line 620 to PEEK the screen and find out what is at X,Y. The code of the character is returned in C. If the code is 13 then the next move takes the asterisk onto a position that contains a "\$" (the code for a "\$" is 13) and as the only "\$" is in the top left hand corner the game is over and control is passed to line 800 to finish the game. Unless this condition is found, line 640 checks to make sure that C is zero, the code for a blank, if it is anything else then the move is rejected. If it is zero, however, then the old asterisk's position is blanked at line 650 and the new asterisk is printed at line 680. The move count M is incremented and the move logic is repeated by line 700. The only details that we haven't discussed are subroutine 900, that adds and removes blockages, and lines 800 to 860, that print the results of a game and asks if you want to play again. Both of these should be easy to understand. Notice how the use of a PEEK to discover what is in the proposed printing position also stops the asterisk being "driven" off the sides of the screen without having to use extra IF statements. Because the maze is surrounded by a non-blank border the move logic will not allow you to cross it!

After studying the techniques presented in this chapter, you should be able to make use of both PEEK and POKE to produce animated graphics to good effect. As an exercise it would be a good idea to change the squash program given earlier to use PEEK and POKE instead of PRINT AT.

Scrolling graphics

So far we have accumulated a wide range of methods for producing moving graphics but there is still one problem that

is difficult to solve — making many things move at once. In general it is not possible to make more than one or two things move at any speed on the ZX81 using BASIC. However there is one exception to this. Try the following program:

```
10 CLS
20 PRINT AT 21,RND*31;"*";
30 SCROLL
40 GOTO 20
```

You can use the SCROLL commands to move the entire screen one line vertically and the time that this takes is not dependent on how many things are moved! The difficulty with SCROLL graphics is not moving things but making them stay still! If you print something on the top line after each SCROLL then it appears to be stationary in a stream of moving asterisks. Add

```
35 PRINT AT 0,16;"V";
```

to the previous program and you have the start of a "pilot a spaceship through the stars" game!

Ski run game

To show how powerful SCROLL graphics can be try the ski run program given below:

```
10 RAND 0
20 LET B$="◇"
30 CLS
40 DIM C(26,2)
50 GOSUB 5000
60 GOSUB 1000
70 LET T=0
80 LET P=0
90 LET X=16
100 GOSUB 2000
110 GOTO 4000
```

(Please note that in line 20 etc., it is important that you type a "less than" symbol, followed immediately by a "greater than" symbol rather than an "inequality sign".)

```
1000 FOR I=1 TO 25
```

```

1010 LET C(I,1)=INT(RND*5)+5
1020 LET C(I,2)=INT(RND*25)+4
1030 NEXT I
1040 LET C(1,1)=21
1050 RETURN

2000 PRINT AT 21,C(1,2);">";
2010 LET K=1
2020 LET J=0
2030 LET L=0
2040 LET I=2
2050 LET S=0
2060 LET S=S+1
2070 LET T=T+1
2080 SCROLL
2090 GOSUB 3000
2100 IF S<>C(I,1) THEN GOTO 2060
2110 PRINT AT 21,C(I,2);B$(J+1);
2120 LET J=NOT J
2130 LET I=I+1
2140 IF I<26 THEN GOTO 2050
2150 FOR I=1 TO 23
2160 LET T=T+1
2170 SCROLL
2180 GOSUB 3000
2190 NEXT I
2200 RETURN

3000 LET A$=INKEY$
3010 IF A$="5" THEN LET X=X-1
3020 IF A$="8" THEN LET X=X+1
3030 PRINT AT 0,X;"*";
3040 IF T<>C(L+1,1) THEN RETURN
3050 LET T=0
3060 LET L=L+1
3070 LET K=NOT K
3080 IF NOT K AND (X-C(L,2)) > 0 THEN RETURN
3090 IF K AND (X-C(L,2)) < 0 THEN RETURN

```

```

3100 LET P=P+1
3110 PRINT "H"
3120 RETURN

4000 PRINT
4010 PRINT "YOU HIT ";P;" GATES"
4020 PRINT "DO YOU WANT TO TRY AGAIN ?";
4030 INPUT A$
4040 PRINT A$
4050 IF A$(1)="N" THEN STOP
4060 IF A$<>"Y" THEN GOTO 4020
4070 PRINT "SAME COURSE ?";
4080 INPUT A$
4090 PRINT A$
4100 IF A$(1)="Y" THEN GOTO 70
4110 IF A$(1)<>"N" THEN GOTO 4070
4120 GOTO 60

5000 CLS
5010 PRINT
5020 PRINT TAB 8;"S K I  R U N"
5030 PRINT TAB 8;"-----"
5040 PRINT
5050 PRINT "YOU HAVE TO SKI DOWN A COURSE"
5060 PRINT "OF 25 FLAGS."
5070 PRINT
5080 PRINT "YOU MUST PASS TO THE LEFT OF"
5090 PRINT "< AND TO THE RIGHT OF >"
5100 PRINT
5110 PRINT "YOU CAN MOVE TO THE RIGHT AND"
5120 PRINT "LEFT BY PRESSING THE ARROW"
5130 PRINT "(5 AND 8)"
5140 PRINT
5150 PRINT "PRESS ANY KEY TO START"
5160 IF INKEY$="" THEN GOTO 5160
5170 CLS
5180 RETURN

```

Although the idea that lies behind this program is simple enough and based on the idea of SCROLL graphics, it takes some quite tricky "book keeping" to keep track of where everything is on the screen. First, instructions for the game are given by subroutine 5000 and then a random course is constructed by subroutine 1000. This is done by generating two random numbers — the distance, in number of SCROLLs, between each successive gate and its horizontal position. The game starts by a call to subroutine 2000 that prints the first gate and begins to SCROLL the screen toward the skier represented by an asterisk. The difficult part is that, after a certain number of SCROLLs, contained in C(2,1), the next gate is printed and after 21 SCROLLs (remember, there are 22 usable lines on the screen) the first gate reaches the skier. At

S K I R U N

YOU HAVE TO SKI DOWN A COURSE
OF 25 FLAGS.

YOU MUST PASS TO THE LEFT OF
< AND TO THE RIGHT OF >

YOU CAN MOVE TO THE RIGHT AND
LEFT BY PRESSING THE ARROW KEYS
(5 AND 8)

PRESS ANY KEY TO START

> *

<

>

<

this point the position of the skier is checked to see that he is on the correct side of the gate. You should be able to see that following this logic it is possible to work out when each gate passes the skier and thus keep track of the number of gates correctly passed. The details of this are easier to program (lines 2000 to 3120) than they are to explain, so over to you!

Paged graphics

Using our knowledge of the construction of the screen area of memory and the system variables relating to it, we can create a number of additional screens and switch between them using POKE commands. This technique gives the ZX81 the ability to keep a number of "pages" of output, any one of which can be displayed very quickly. The only problem is that it is very easy to make a mistake and end up with a blank screen and an unresponsive ZX81. The solution is to switch off and on again and the only damage that you can do is to your program.

The method is essentially simple but the details make it seem complicated. There are two system variables that are used by the ZX81 to keep track of the screen details. The first, D_FILE, notes where the screen area of memory is; while the second, DF_CC, notes where the next character should be printed. If you set up an area of memory in the full screen format and then change these locations to refer to your new area then the NEW area will be displayed instead of the original screen. While this second screen is being displayed any PRINTs will appear on it but not on the original screen. You can transfer back to the original screen by restoring the addresses that were in D_FILE and DF_CC. The only thing that you have to remember is that when you POKE the new addresses into the two screen variables, you must be in FAST mode. The reason for this is that the two bytes have to be POKEd one after the other and if the ZX81 tried to display the screen between the POKEs then the variables would contain rubbish and the machine would run wild! Using FAST means that the screen isn't displayed during the changing of the addresses. To see this idea in action try the following program. Before you run it make sure it is entered correctly

and *do not break in* while screen two is being displayed!

```
10 LET E=100+PEEK 16412+256*PEEK 16413
20 GOSUB 1000
30 LET C=E
40 LET D=E+1
50 GOSUB 3000
60 PRINT "SCREEN TWO"
70 FOR I=1 TO 100
80 NEXT I
90 LET C=A
100 LET D=B
110 GOSUB 3000
120 PRINT "SCREEN ONE"
130 FOR I=1 TO 100
140 NEXT I
150 LET C=E
160 LET D=E+1
170 GOSUB 3000
180 GOTO 70
```

```
1000 LET I=E
1010 FOR J=1 TO 24
1020 POKE I,118
1030 LET I=I+1
1040 FOR K=1 TO 31
1050 POKE I,0
1060 LET I=I+1
1070 NEXT K
1080 NEXT J
1090 POKE I,118
1100 RETURN
```

```
2000 LET A=PEEK 16396+256*PEEK 16397
2010 LET B=PEEK 16398+256*PEEK 16399
2020 RETURN
```

```
3000 GOSUB 2000
3010 FAST
```

```
3020 POKE 16396,C-INT(C/256)*256
3030 POKE 16397,INT(C/256)
3040 SLOW
3050 POKE 16398,D-INT(D/256)*256
3060 POKE 16399,INT(D/256)
3070 RETURN
```

Subroutine 1000 sets up a new screen starting at the address in E. The new screen consists of 25 "newline" codes with 32 blanks in between. Subroutine 2000 saves the addresses in the screen variables in A and B. Subroutine 3000 POKes the addresses in C and D into the screen variables. The main part of the program begins by finding some free memory and setting up a new screen (lines 10 and 20). Then lines 30 to 60 switch to the new screen and print "SCREEN TWO" on it. After a delay lines 90 to 120 switch back to screen one and print "SCREEN ONE" on it. The two screens are then alternately displayed by lines 70 to 180 (with "SCREEN ONE" being printed on the next line each time through the loop.)

This method of "paged graphics" opens up a wide range of special effects and displays. However it should be used with care as it brings with it the possibility of a program crashing and leaving the machine unresponsive. Because of this danger this technique should be marked "for experts only"!

Conclusion

You can use the techniques described in this chapter to produce some very clever graphics. For example, you could try to combine the methods of SCROLL graphics with PEEK graphics, but beware of the alteration in screen layout! However, after any amount of trickery it is hard to avoid the conclusion that BASIC is just not fast enough to make "arcade" quality games possible, let alone easy. For the solution to this problem we must wait for Chapter Ten.

DESIGNING LARGE PROGRAMS

Writing small programs on the 16K ZX81 or any machine with a reasonable amount of memory is fairly easy. A small program usually does a small number of things and it is possible to keep these in mind at the same time when writing or examining the program. However, the way that you write a small program will not do when you are trying to write a big program. If you try to keep all of the details of a large program in your head you are sure to become confused and this will increase the likelihood of mistakes or bugs. The best way to tackle any programming task is to adopt an ordered programming method and work systematically. This need not take the "fun" out of programming and does improve your final product to a standard that you can be proud of.

Designing a program

You may be familiar with the idea of using a flow chart or flow diagram to describe how a program works. It is often suggested that before you begin to write a program you should draw a flow diagram and that this becomes more important the larger the program. You may even have been told that it is the only way to go about programming and anything else is sloppy. This is, of course, a matter of opinion. The truth is that it is important that you have an overall grasp of the way that your program is going to work but it doesn't have to be in the kind of detail that is necessary to draw a flow diagram. My personal preference is to describe the program in a cross between English and BASIC before beginning to work on the computer. For example, suppose you decide that you would like to write a program that will be useful in teaching arithmetic. The first thing to do is to outline what you would like the program to do in general terms. Your scheme might look like this.

The program should print a question and wait for a reply
 If the reply is correct then inform the user and move on to another problem
 If the reply is incorrect then inform the user and print the same question again

Notice that this description of the program is already a little like BASIC in the way it uses "If" and "Print".

You could start writing the BASIC program from this description or you could write a second more detailed description that explained how the questions are to be constructed etc. This method of writing programs by more and more detailed descriptions is often called "stepwise refinement" because you start out with a vague idea of the program and "work it over" a number of times, each time increasing the accuracy of the description until you have a complete BASIC program. Once you have tried it you will find this a very natural way of working. For example the second "refinement" of the arithmetic program might be:

Select the type of problem i.e. +, -, *, /
 Generate two fairly small random numbers
 Print the question on the screen and wait for a reply
 If the reply is correct then inform the user and move on to another problem
 If the reply is incorrect then inform the user and print the same question again

The point at which you feel that the program is sufficiently defined to get out the computer and start work is up to you. It is an advantage of this method that you can adjust it to suit your level of programming skill and the difficulty of the problem. There is also a subtle psychological advantage to be gained from having something written down on paper, no matter how vague, because getting started on a program can sometimes be the biggest problem!

Using subroutines

Although the stepwise refinement method gives you a way of

working toward an understanding of how a program should be written, it still leaves open the question of how to deal with a very big program. The trouble is that as the level of detail increases you can still be confronted with a vast collection of BASIC instructions and variables. The only sensible way to approach such a collection is to try to break it down into smaller "chunks". It is nearly always possible to split a large program down into sections that each carry out a single identifiable job. For example in the case of the arithmetic test program you could identify three sections:

- the question construction section
- the question presentation section
- the answer evaluation section

Each of these sections can be treated as a little program in its own right and developed more or less separately. From this point of view you should be able to see that the key to the easy production of large programs is to treat them as being nothing more than a collection of smaller programs. These smaller programs may still be too big to be treated in one go and so it may be necessary to break these down into even smaller programs!

You might have guessed that the program sections that we identify in larger programs are closely connected with the idea of subroutines. The BASIC commands GOSUB and RETURN can be used to turn a list of other commands into a single program "section". In the case of the arithmetic test program we could assume that we have already written a subroutine starting at line 1000 which constructs a question and another at 2000 that presents the question. The first two lines of our description could now be replaced by pure BASIC

```
10 GOSUB 1000
20 GOSUB 2000
```

This form of the program is particularly easy to understand at a glance. Just think how much more complicated this part of the program would be if you replaced each of the GOSUBs by the lines of BASIC that carried out the same task! You might think that the third part of the program is best

implemented by a subroutine and it most certainly is a possibility. However, because the result of the next section is either to repeat subroutine 2000 if the answer is wrong, or to repeat the whole program if the answer is right, it is easier to write it as part of the program that uses the subroutines. In other words a better way to proceed is:

```
10 GOSUB 1000
20 GOSUB 2000
30 INPUT GUESS
40 IF GUESS=ANSWER THEN GOTO 10
50 PRINT "TRY AGAIN"
60 GOTO 20
```

There is another advantage to using subroutines in this way — you can use them even if you haven't yet written them! If you simply suppose that subroutines 1000 and 2000 can be written you can go ahead and use them to construct the program and write them later on. There is an obvious connection here between the idea of stepwise refinement and this use of subroutines. You can leave the details of the subroutines vague until later. You might then decide that these subroutines could be written using subroutines the details of which can be left vague until later and so on . . . until every subroutine is completely defined.

This method of using as yet unwritten subroutines to complete the program, and treating each of the subroutines in the same way, is known as "modular programming". In practice you should find that it makes programming a more enjoyable and fruitful occupation. Modular programs are easier to understand and far easier to modify than the usual long list of BASIC instructions. You can also re-use subroutines or modules in other programs that need the same operation and so avoid re-inventing the wheel each time.

To finish the arithmetic program all we have to add is:

```
1000 LET A=INT(RND*10)
1010 LET B=INT(RND*10)
1020 LET O$="+-*/"(INT(RND*4)+1)
1030 LET ANSWER=VAL(STR$(A)+O$+STR$(B))
```


1040 RETURN

```
2000 CLS
2010 PRINT AT 10,5;A;" ";O$;" ";B;" = "
2020 PRINT AT 14,5;"WHAT IS THE
    ANSWER ?"
2030 RETURN
```

Notice the use of the "slicing" notation in line 1020 to pick an operation at random and the use of the VAL function in line 1030 to work out the answer.

4 * 6 =

WHAT IS THE ANSWER ?

The BASIC on the ZX81 has an extra feature that can make the use of subroutines in a program even easier to understand. The commands GOSUB and GOTO are not restricted to their most usual forms i.e. GOSUB "linenumber". You can also write things like GOSUB 'expression' e.g. GOSUB 2*3+1 meaning the same thing as GOSUB 7. Now although this facility is not often of any use, a single variable is also an expression, in fact it is the simplest form of expression, so you can write:

```
5 LET GENQUESTION=1000
6 LET PRINTQUESTION=2000
10 GOSUB GENQUESTION
20 GOSUB PRINTQUESTION
```

Using this simple idea you can give subroutines names that not only make your programs easier to read but easier to change.

The main message of this section should be clear. It is simply — *use subroutines!* The next section is a complicated one which can be omitted if you prefer. It deals with a technique which, like the paged graphics in Chapter Four, should be marked "for experts only"!

An extended RETURN

There are cases where it would be nice to use a subroutine but for some reason or another it is easier not to. For example, the third section of the arithmetic test program was not turned into a subroutine because different lines in the main program were to be carried out according to the outcome of the test executed in the section. The reason why it is difficult to branch to a different point in the main program according to what happens in a subroutine, is that subroutines should always end with RETURN and this always executes the instruction following the GOSUB. In most cases you can spot where this sort of tangle will arise before you write the subroutine and then everything is fine. Sometimes, however, you find that what happens in the main program depends on what happens in a subroutine you have just written. In such a case it would be useful if there was some way to save the work that went into writing it without complicating things too much. What we would really like to do is to end a subroutine with a GOTO "linenumber" where the "linenumber" could be changed according to what resulted in the subroutine. You can do this without generating an error as long as you have only called one subroutine at the time. The only unfortunate consequence is that the GOSUB stack area of memory is larger by two locations than it need be and if you keep on calling a subroutine without a RETURN you will eventually use up all the available memory. If you GOTO a subroutine from another subroutine then you are sure to be in trouble when you issue the next RETURN!

What we would really like is a RETURN "linenumber" command that we could use to return from a subroutine to a specified line number. It is important to notice that I am not suggesting that this should be used as a regular feature of BASIC programming, (it is better not to use a subroutine than use a RETURN "linenumber" type of command) only that it is sometimes useful to be able to recover from a mistake! Using our knowledge of the memory layout it is easy to write a few lines of BASIC that will change the address stored on the GOSUB stack so that the next RETURN will take us wherever

we want to go. For example:

```
10 GOSUB 200
20 PRINT "HERE 20"
30 PRINT "HERE 30"
40 STOP
```

```
100 LET S=2+PEEK 16386+256*PEEK 16387
110 POKE S,L-INT(L/256)*256
120 POKE S+1,INT(L/256)
130 RETURN
```

```
200 PRINT "HERE 200"
210 LET L=30
220 GOTO 100
```

If you run this program you will see that first the message "HERE 200" appears on the screen as subroutine 200 is called and then the message "HERE 30". If subroutine 200 finished with a normal RETURN you would expect to see the messages "HERE 20" followed by "HERE 30". What has happened is that the four lines of program 100-130 find the line number that will be used by the next RETURN on the GOSUB stack and replace it by the number stored in L. So line 210 sets L to 30 and line 220 goes to the section of program that returns control to line 30, missing line 20 altogether. The lines of program 100-130 can be used by any subroutine when you want to return to somewhere other than where a normal RETURN would take you.

Collections of programs - Menus

It is very often the case that a large program can be successfully broken down into smaller pieces making development easier. It is also often true that large programs can be made up by joining existing smaller programs together. The smaller programs may not have been conceived of as working together, and indeed they may even have been written by different people, but there is a very easy way of joining them together - the menu. Providing each program occupies a different range

of line numbers, you can type each one in and add a few lines that announce the name of the program and ask what the user wishes to do. For example, the program development utilities described in chapter three can be turned into a single program by adding:

```
9000 CLS
9010 PRINT TAB(5);"U T I L I T I E S"
9020 PRINT AT 5,0
9030 PRINT "SELECT ONE OF-"
9040 PRINT AT 10,0
9050 PRINT TAB 5;"(1) MEMORY USE"
9060 PRINT TAB 5;"(2) VARIABLE LIST"
9070 PRINT TAB 5;"(3) RENUMBER"
9080 PRINT TAB 5;"(4) QUIT"
9090 PRINT AT 20,0;"TYPE REQUIRED NUMBER"
9100 INPUT J0
9110 IF J0<1 OR J0>4 THEN GOTO 9090
9120 IF J0=1 THEN GOSUB 9200
9130 IF J0=2 THEN GOSUB 9300
9140 IF J0=3 THEN GOSUB 9800
9150 IF J0=4 THEN STOP
9160 GOTO 9000
```

You also have to change the "final" lines of each utility to return control to the menu when they have finished i.e. change STOP in line 9240 of utility one, line 9330 in utility two and line 9900 in utility three to RETURN. The advantage of using the menu is that you no longer have to remember the start addresses of the three utilities just type GOTO 9000.

U T I L I T I E S

SELECT ONE OF-

```
(1) MEMORY USE
(2) VARIABLE LIST
(3) RENUMBER
(4) QUIT
```

TYPE REQUIRED NUMBER

User-friendly programs

When you first start writing programs there is an excuse for not worrying too much about the quality of your handiwork. If the program does the job then it is a success. The trouble is that this attitude tends to continue even when you are no longer a beginner. The result is a large collection of programs that no one can use except you. There is nothing more satisfying than finishing a program to such a level that other people can use it and then knowing that other people ARE using it! Programs that are finished in this sense have to be "user-friendly" in other words they have to be kind to even the most inept user. When you write a program you should always have in mind someone who knows very little about computers/programming or even the topic that your program deals with. It is difficult to give a complete list of the points to pay attention to when making a program user-friendly but the following might be helpful:

- Ask clear and un-ambiguous questions, do not use jargon.
- Accept all reasonable answers to each question. For example treat "Y" and "YES" as equivalent.

- Check the answer to each question for validity as it is given. If it is not valid at the very least repeat the question.

- Do not present too much information in one go and use PAUSE together with a "Press any key to continue" message, or a similar method, to allow the user to control when the program should move on.

- Be careful not to fill the ZX81's screen and so stop the program. The best way of avoiding this is to use PRINT AT statements so that repeated printing of questions doesn't accidentally fill the screen. Also use PRINT AT 21,0; print list and SCROLL to print long lists of data.

- Repeat the answer to any questions on the screen on the same line as the question was asked so that the user can see the answer later on.

- Do spend time writing a good title screen including a simple introduction to what the program will do.

It is difficult to write a perfect program but you should try!

For an example of a program that tries to follow all these suggestions, you are referred to the statistics program in the next chapter.

Debugging

One of the most difficult things to learn is the art of debugging a program. It is comparatively easy to learn BASIC or to write programs but how to find out what is wrong with a program that isn't working is something that can take years to learn. The main problem is that debugging is something that is learned by practice rather than by theory. While this is true, it does help to have explained a few possible approaches to debugging and some of the helpful features of the ZX81 in this respect.

Some bugs are easy to find – the program misbehaves, you list it and say "ah ha, there it is"! The real trouble starts when something doesn't go right and you stare at the program and see nothing wrong. What you should do in this situation is not to stare for too long. To find a bug you need information on how the program is running and compare it with your predictions of how it SHOULD be running. When you find a divergence between what does happen and what should happen you are at least nearer to your bug even if you haven't quite found it.

What sort of information on the program's running is relevant to debugging? You may think that there is a huge range of information that has to be considered. In fact there are only two points – the order in which the program lines are carried out and the value stored in each variable. Some versions of BASIC have a TRACE facility which, if used, will print the line number of each line as it's being carried out. ZX81 BASIC doesn't have a TRACE facility but it does have a STOP instruction and a CONTINUE command. If you want to know what is going on at any point in a program all you have to do is to temporarily insert a STOP command. When the program reaches the point in question it will stop with the appropriate report code. The report code includes the line number so if you have a number of STOP commands in a

program you can follow the order of execution. While the program is stopped you can either use utility two or a PRINT statement to discover what is stored in any variable. You can also use a LET statement to change the contents of any variable. In fact the only thing that you cannot do is to add a program line. To resume execution from where it left off simply type CONTINUE. The only problem with this method is that PRINTing variable contents clears the screen, so resuming execution after this can result in an incomplete display.

Finding bugs once you know they are there is simply the application of the method described above. The problem of making sure that a program doesn't have any hidden bugs is the other side of the coin. There is no real solution to the problem apart from using the program a lot with as wide a range of input data as you can manage. There are systematic ways of using different sets of data to make the program take every possible route to the end but even this is not a guarantee that there are no more hidden bugs. One important hint is to get someone else to use the program before you declare it bug-free. It is possible to test a program for days and get it working without any obvious bugs and the first time that someone else uses it they find a bug in the first 30 seconds! The reason for this is that new users tend to use programs in different ways to the programmer who produced them. They tend to test the "easier" parts of the program whereas the programmer tends to test the more "interesting" parts — and bugs are just as likely to be hidden in the straightforward bits you wrote quickly as in the difficult bits that you slaved over.

There is only one attitude towards program bugs that will save your sanity — there is always another bug!

TAPE STORAGE OF DATA

The tape storage facility provided on the ZX81 is obviously good at SAVEing and LOADing programs but there seems to be no way of storing data on tape. However, although there is no direct way of writing data to tape it is possible to write programs that can create and maintain tape files. In this chapter we examine the problems in doing this and write a large statistics program that uses the facility.

The tape system

Probably, one of the first things that you learned to do with your ZX81 was to SAVE and LOAD a program. If you are using a fairly good quality tape recorder then, once you have set the volume and tone controls to the correct levels, you should find that its reliability in use is quite good. Encouraged by this to take the ZX81's tape system seriously it would be natural to try to write some programs that establish data files on tape. The trouble is that there are only two commands that relate to the tape system:

```
SAVE "filename"  
and  
LOAD "filename"
```

whereas other microcomputers have commands such as

```
SAVE A
```

where A is the name of an array that is saved on tape. This omission has often been used as a criticism of the ZX81. However, if you look at exactly what SAVE and LOAD do, you will discover that they are more powerful than they seem.

When you type SAVE an exact copy of all the memory from address 16393 (in the system variable area) to the location whose address is stored in E_LINE is transferred to

the cassette tape. This chunk of memory includes most of the system variables, all of the program, the display file and all of the program variables! You should think of this as a "snapshot" of all the memory locations that matter to a running program. If you break into a program at any point and SAVE it, then as well as the lines of BASIC that make up the program, the current contents of the screen and all the variables are stored on tape. If you then LOAD the program back into the memory then the "snapshot" can be "re-activated" and the program can continue from where it left off! If this is true why is it that every program that you LOAD from tape appears to start from the beginning with no "knowledge" of its past life? The screen is always clear, with no sign of anything printed by an earlier RUN of the program and all the variables are set to zero. The reason for these observations is two-fold. Firstly, whenever you RUN a program the command RUN clears the screen, resets the variable area and starts the program off from the first line. If you want to LOAD a program from tape and restart it then you must avoid RUN. The obvious command to use is CONTINUE. To convince yourself that this works type in the following program:

```
10 FOR I=1 TO 100
20 PRINT AT 21,0;I
30 SCROLL
40 NEXT I
```

If you start it off by using RUN you should see it print integers from 1 up to 100 on the screen. At some convenient point before the program finishes press the BREAK key and notice the last number on the screen. Next prepare your tape recorder ready to SAVE a program and type

SAVE "TEST"

Start the tape running and press "newline". When the SAVE is complete re-wind the tape and switch your ZX81 off to ensure that the old version of the program is lost. Then type

LOAD "TEST"

Start the tape running and press "newline". When the program has finished loading stop the tape recorder but, instead of starting the program with RUN, use CONTINUE. You should be amazed to see the program carry on from where it left off, printing the next number in the list! This is an important and useful discovery because you can now SAVE and CONTINUE a program if it is taking too long to do something and you cannot wait any longer. You can also use GOTO to restart a program at a different place without disturbing the contents of the variables area of memory.

When you try either of these techniques, however, you may be disappointed to notice that although the program CONTINUED from where it left off the screen display wasn't restored to what it contained when you broke into the program's execution. The reason for this is that any BASIC command entered from the keyboard and carried out immediately (i.e. one that doesn't have a line number and isn't part of a program) clears the screen before it has any effect. For example, try typing

LET A=0

When you press "newline" you will see the screen clear. This is unfortunately what happens when you use the SAVE command. Before the memory is stored on tape the screen is cleared and it is this "blank" screen that is restored when you CONTINUE the program.

There is a solution to this "loss of screen" problem. Both SAVE and LOAD can be used as statements within a program and when they are used in this way they do not result in the screen being cleared before they are obeyed. There is also a useful spin-off from using these commands as part of a program. If you LOAD a program that was SAVED as a result of a SAVE statement within a program, then at the end of the LOAD the program will continue from where it left off without the use of a CONTINUE command. To be exact, the first statement to be executed is the one immediately following the SAVE. This facility is immensely useful because it allows us not only to SAVE a partially completed program and restart it including the screen display, but to write self-

running programs! To see that all this actually works type in the following program

```
10 FOR I=1 TO 100
20 PRINT AT 21,0;I
30 SCROLL
40 IF I=20 THEN SAVE "TEST"
50 NEXT I
```

which does the same thing as the previous example apart from the fact that it SAVES itself when "I" reaches 20! If you RUN this program you have to be ready to record the program just a little before the integer count gets to 20. So have the tape recorder ready and start it recording when you first see 15 appear on the screen. When "I" reaches 20 you will see the familiar patterns that tell you that something is being recorded appear on the screen. At the end of this the program continues as if nothing had happened, printing 21,22, and so on. Re-wind the tape and switch your ZX81 off to convince yourself that all trace of the program is lost. If you LOAD the program TEST in the usual way you will find that at the end of the LOAD the program continues from where it left off, including the screen showing the numbers previously printed.

There is one more interesting and useful feature of SAVE and LOAD when used in programs that we haven't yet mentioned. You can use not only a string as the name of a program but a string expression. So the following are all valid:

```
SAVE A$
SAVE "PROG"+STR$(I)
LOAD A$+B$
```

The only restriction on using SAVE within a program is that it must not be used from within a subroutine. The reason for this is that the GOSUB stack area of memory is not SAVED on tape and so when the program is LOADED and restarted the next RETURN command will cause the program to stop and an error message to appear.

Data storage

Now that we know that SAVE stores the contents of the variables area of memory on tape, writing a program that builds and manipulates data files should be easy. As an example consider the problem of writing a statistics program that can be used to enter a list of numbers, edit and generally change them, and then SAVE them on tape for later use. Although this is described as a statistics program the problems encountered and the methods used are similar to storing and maintaining any list of information, including lists of strings.

Following the suggestions presented in the last chapter for designing modular programs using stepwise refinement, the first thing to do is to get an overall outline of the program. We need a subroutine to read numbers into an array, a subroutine to offer the user the chance to change the list and a section to SAVE the list on tape. If the next instruction following the SAVE is a GOTO "the start of a program" then when the program is re-located it will self-start at the beginning but with the data from the previous RUN already in memory. Notice that our file of data is in fact a mixture of data and program. It is the case that the ZX81 can only store data along with the program that uses it. To many people used to other computers this is a strange idea. They are familiar with the idea that programs and data are very different things and should be treated separately. However, if you think about it, the ZX81's approach has a lot to be said for it and it may be just a matter of time before other machines offer such facilities. The advantages of the ZX81's method is that you do not have to remember two file names (one for the program and one for the data) to process some information. The disadvantage is that it is difficult to transfer data from one program to another and even if you want to store only a little data you have to wait while the whole program is SAVED. There are ways around both of these problems involving PEEKs and POKEs and a knowledge of the ZX81's memory layout. For example, you can pass data from one program to another by transferring the entire variables area of memory into free RAM. When you LOAD

a new program, this area of memory is left unaffected and it can then be transferred back into the variables area as the first task of the new program. These are, however, very advanced methods, and as long as we have enough memory, it is possible to do quite a lot without resorting to such techniques.

The statistics program as outlined above will work nicely unless one of the options of the editing subroutines is the addition of fresh data to what already exists. As this is the sort of thing we would like to do it is important to examine what the problem involved is and how we can solve it. Before you can start to enter your list of numbers, the statistics program has to set up an array to receive them. It is obviously important that the array is just the right size to hold all those numbers and no bigger — we don't want to waste time storing un-used array elements on tape. So the first question to ask the user is "*How many values do you want to enter?*" and set up an array of the correct size. Now imagine that all the numbers have been entered and SAVED on tape. How do we increase the size of the array so that some more numbers can be added? While there are clever ways based on a direct manipulation of the variables area of memory to make that array bigger, there is a simple programming solution. If the original number of variables was N and we want to add M extra variables we could create a new array by

`DIM E(N)`

transfer all the current data from the original array, D say, into E and then re-dimension D to the larger size, i.e.

`DIM D(N+M)`

This works on the ZX81 because its dimension command is unlike most other BASIC DIM commands. If you dimension an array that already exists, the ZX81 erases the existing array and replaces it by a brand new version which can be a different size from the original. After creating the larger version of D it is simplicity itself to recopy the old data in E back into the first N elements of D and ask the user for the values of the extra M elements. The thing to remember is to release the

storage used by E by issuing the following command:

`DIM E(1)`

This may seem a silly way of increasing data storage, and it has to be admitted that it can be slow and it does use a lot of memory, but all the other methods used available on the ZX81 are complicated and have problems of their own.

Statistics program

This is the largest program in this book and it illustrates many of the ideas that we have already met and introduces some new ones. Apart from being an example it is also a useful program in its own right and could easily form the basis for an even more comprehensive statistics program.

S T A T I S T I C S

```
(1) ENTER NEW DATA
(2) GENERATE RANDOM DATA
(3) EDIT DATA
(4) SAVE DATA
(5) CALCULATE STATISTICS
(6) PLOT HISTOGRAM
(7) QUIT
```

TYPE REQUIRED NUMBER

Briefly, the program can be used to enter data and calculate a number of simple statistics, the maximum, the minimum, the mean, the range and the standard deviation. In order to examine the data graphically there is a facility to plot a histogram of the frequencies of the data grouped into equal intervals. The option of editing the data is also provided and this is organised as a second "edit menu" and associated subroutines. The editing operations available are: list all or a part of the data; change a single value; delete a subrange of the data; and add extra values. As part of the main menu the edited data can be SAVED on tape for later use. The final option is the generation of random data. This may seem like a strange thing to want to do but it can be used both to test the program and demonstrate it.

EDIT DATA

- (1) LIST DATA
- (2) ALTER DATA
- (3) DELETE DATA
- (4) ADD DATA
- (5) RETURN TO MAIN MENU

TYPE REQUIRED NUMBER

LIST STARTING AT? 1
LIST ENDING AT? 10

DATA VALUE 1 = 2
DATA VALUE 2 = 1
DATA VALUE 3 = 5
DATA VALUE 4 = 2
DATA VALUE 5 = 1
DATA VALUE 6 = 5
DATA VALUE 7 = 4
DATA VALUE 8 = 2
DATA VALUE 9 = 3
DATA VALUE 10 = 1

PRESS ANY KEY TO CONTINUE

ALTER WHICH VALUE? 4
CURRENT VALUE = 2
NEW VALUE = 3

PRESS ANY KEY TO CONTINUE

ADD HOW MANY VALUES ?10
WAIT WHILE I MAKE SPACE
NEARLY DONE
DONE - READY FOR EXTRA DATA

DATA VALUE 11 =

To use the program after it has been typed in and SAVED on tape it is first necessary to create some data either by typing it in or by using the random data generation facility. From the main menu it is then possible to choose any of the actions described above as often as necessary. When the time comes to SAVE the data, this should be done by selecting the appropriate option from the main menu. When this is finished the

program will not automatically stop but it is safe to switch the machine off in the knowledge that your data has been stored on tape. When you next want to use the same data simply LOAD the program, using whatever file name you gave it. The program will start automatically and will have the data ready for further analysis.

The complete program is

10 REM ZX81 STATISTICS PROGRAM

```
500 CLS
510 PRINT TAB 5;"STATISTICS"
520 PRINT AT 6,0
530 PRINT "(1) ENTER NEW DATA"
540 PRINT "(2) GENERATE RANDOM DATA"
550 PRINT "(3) EDIT DATA"
560 PRINT "(4) SAVE DATA"
570 PRINT "(5) CALCULATE STATISTICS"
580 PRINT "(6) PLOT HISTOGRAM"
590 PRINT "(7) QUIT"
600 PRINT AT 21,0;"TYPE REQUIRED NUMBER";
610 INPUT SEL
620 IF SEL=1 THEN GOSUB 3000
630 IF SEL=2 THEN GOSUB 1000
640 IF SEL=3 THEN GOSUB 4000
650 IF SEL=4 THEN GOTO 1500
660 IF SEL=5 THEN GOSUB 5500
670 IF SEL=6 THEN GOSUB 6000
680 IF SEL=7 THEN STOP
690 GOTO 500
```

```
1000 CLS
1010 PRINT "RANDOM DATA"
1020 PRINT "HOW MANY VALUES?";
1030 INPUT N
1040 PRINT N
1050 DIM D(N)
1060 PRINT AT 3,0;"FRACTIONAL OR INTEGER
DATA F/I ";
```

```

1070 INPUT A$
1080 IF A$(1) <> "F" AND A$(1) <> "I" THEN
    GOTO 1060
1090 PRINT A$
1100 LET T=0
1110 IF A$="I" THEN LET T=1
1120 PRINT AT 4,0;"LOWEST VALUE ";
1130 INPUT L
1140 PRINT L
1150 PRINT "HIGHEST VALUE ";
1160 INPUT H
1170 PRINT H
1180 IF H>L THEN GOTO 1210
1190 PRINT "HIGHEST<LOWEST"
1200 GOTO 1120
1210 FOR I=1 TO N
1220 LET D(I)=RND*(H-L+T)+L
1230 IF T=1 THEN LET D(I)=INT D(I)
1240 PRINT AT 20,0;"DATA VALUE ";I;" = ";D(I)
1250 SCROLL
1260 NEXT I
1270 GOTO 8900

1500 CLS
1510 PRINT "PLACE A BLANK TAPE INTO"
1520 PRINT "THE RECORDER"
1530 PRINT
1540 PRINT "WHAT DO YOU WANT TO CALL"
1550 PRINT "THE PROGRAM ?";
1560 INPUT A$
1570 PRINT A$
1580 PAUSE 25
1590 PRINT "PRESS PLAY AND RECORD"
1600 PRINT "AND THEN PRESS ANY KEY"
1610 PRINT "ON THE KEYBOARD"
1620 IF INKEY$="" THEN GOTO 1620
1630 PRINT "BYE"
1640 PAUSE 50

```

```

1650 CLS
1660 SAVE A$
1670 GOTO 10

2000 LET M=0
2010 LET S=0
2020 LET L=D(1)
2030 LET H=L
2040 FOR I=1 TO N
2050 LET M=M+D(I)
2060 IF L>D(I) THEN LET L=D(I)
2070 IF H<D(I) THEN LET H=D(I)
2080 NEXT I
2090 LET M=M/N
2100 FOR I=1 TO N
2110 LET S=S+(D(I)-M)*(D(I)-M)
2120 NEXT I
2130 LET S=S/(N-1)
2140 RETURN

2500 CLS
2510 PRINT "NUMBER OF VALUES= ";N
2520 PRINT "MAXIMUM= ";H
2530 PRINT "MINIMUM= ";L
2540 PRINT "RANGE= ";H-L
2550 PRINT "MEAN= ";M
2560 PRINT "VARIANCE= ";S
2570 PRINT "STANDARD DEV.= ";SQR(S)
2580 GOTO 8900

3000 CLS
3010 PRINT "DATA INPUT"
3020 PRINT "HOW MANY VALUES ?";
3030 INPUT N
3040 PRINT N
3050 DIM D(N)
3060 FOR I=1 TO N
3070 PRINT AT 21,0;"VALUE ";I;" = ";

```



```

3080 INPUT D(I)
3090 PRINT D(I)
3100 SCROLL
3110 NEXT I
3120 PRINT "DATA INPUT COMPLETE"
3130 SCROLL
3140 GOTO 8900

4000 CLS
4010 PRINT TAB 5;"E D I T   D A T A"
4020 PRINT AT 5,0
4030 PRINT "(1) LIST DATA"
4040 PRINT "(2) ALTER DATA"
4050 PRINT "(3) DELETE DATA"
4060 PRINT "(4) ADD DATA"
4070 PRINT "(5) RETURN TO MAIN MENU"
4080 PRINT AT 21,0;"TYPE REQUIRED NUMBER"
4090 INPUT ED
4100 IF ED=1 THEN GOSUB 4200
4110 IF ED=2 THEN GOSUB 4500
4120 IF ED=3 THEN GOSUB 4600
4130 IF ED=4 THEN GOSUB 4800
4140 IF ED=5 THEN RETURN
4150 GOTO 4000

4200 CLS
4210 PRINT "LIST STARTING AT ?";
4220 INPUT L
4230 PRINT L
4240 PRINT "LIST ENDING AT (-1 WILL LIST TO
      END) ?";
4250 INPUT H
4260 IF H<0 THEN LET H=N
4270 PRINT H
4280 IF L>H THEN GOTO 4200
4290 IF L>N OR H>N OR L<1 OR H<1 THEN
      GOTO 4200
4300 FOR I=L TO H

```

```

4310 PRINT AT 20,0;"DATA VALUE ";I;" = ";D(I)
4320 SCROLL
4330 IF INT((I-L+1)/20)*20=(I-L+1) THEN GOSUB
      8900
4340 NEXT I
4350 GOTO 8900

4500 CLS
4510 PRINT "ALTER WHICH VALUE ?";
4520 INPUT I
4530 IF I<1 OR I>N THEN GOTO 4500
4540 PRINT I
4550 PRINT "CURRENT VALUE = ";D(I)
4560 PRINT "NEW VALUE = ";
4570 INPUT D(I)
4580 PRINT D(I)
4590 GOTO 8900

4600 CLS
4610 PRINT "DELETE STARTING FROM ";
4620 INPUT L
4630 PRINT L
4640 PRINT "ENDING AT ";
4650 INPUT H
4660 PRINT H
4670 IF H<L THEN GOTO 4600
4680 IF H>N OR H<1 OR L>N OR L<1 THEN
      GOTO 4600
4690 PRINT
4700 PRINT "DELETE FROM ";L;" TO ";H
4710 PRINT "IS THIS OK ?";
4720 INPUT A$
4730 PRINT A$
4740 IF A$(1)<>"Y" THEN RETURN
4750 FOR I=H+1 TO N
4760 LET D(L+I-H-1)=D(I)
4770 NEXT I
4780 LET N=N-H+L-1

```

4790 RETURN

```
4800 CLS
4810 PRINT "HOW MANY EXTRA VALUES ? ";
4820 INPUT M
4830 PRINT M
4840 DIM E(N)
4850 PRINT "WAIT WHILE I MAKE SPACE"
4860 FOR I=1 TO N
4870 LET E(I)=D(I)
4880 NEXT I
4890 PRINT "NEARLY DONE"
4900 DIM D(N+M)
4910 FOR I=1 TO N
4920 LET D(I)=E(I)
4930 NEXT I
4940 PRINT "DONE - READY FOR EXTRA DATA"
4950 DIM E(1)
4960 FOR I=N+1 TO N+M
4970 PRINT AT 21,0;"DATA VALUE ";I;" = ";
4980 INPUT D(I)
4990 PRINT D(I)
5000 SCROLL
5010 NEXT I
5020 PRINT "DATA INPUT COMPLETE"
5030 SCROLL
5040 LET N=N+M
5060 GOTO 8900
```

```
5500 CLS
5510 PRINT "CALCULATING"
5520 GOSUB 2000
5530 GOTO 2500
```

```
6000 CLS
6010 PRINT "HOW MANY BINS ? ";
6020 INPUT M
6030 PRINT M
```

```
6040 PRINT "MAXIMUM VALUE = ";
6050 INPUT H
6060 PRINT H
6070 PRINT "MINIMUM VALUE = ";
6080 INPUT L
6090 PRINT L
6100 IF H<L THEN GOTO 6000
6110 LET D=(H-L)/M
6120 GOSUB 7000
6130 FOR I=1 TO M
6140 PRINT AT 21,0;INT(L*100)/100;TAB 6;
6150 IF H(I)=0 THEN GOTO 6190
6160 FOR J=1 TO H(I)/F*25
6170 PRINT "[ ]";
6180 NEXT J
6190 SCROLL
6200 LET L=L+D
6210 NEXT I
6220 SCROLL
6230 GOTO 8900
```

```
7000 DIM H(M)
7010 FOR I=1 TO N
7020 LET J=(D(I)-L)/(H-L)*M+1
7030 LET J=INT J
7040 IF J<1 OR J>M THEN GOTO 7060
7050 LET H(J)=H(J)+1
7060 NEXT I
7070 LET F=0
7080 FOR I=1 TO M
7090 IF F<H(I) THEN LET F=H(I)
7100 NEXT I
7110 RETURN
```

```
8900 PRINT AT 21,0;"PRESS ANY KEY TO
CONTINUE"
8910 IF INKEY$="" THEN GOTO 8910
8920 RETURN
```

Here are some samples to show you what to expect when you run this program. It is very long so it is impossible to describe its operation in as much detail as usual.

```
NUMBER OF VALUES=40
MAXIMUM=10
MINIMUM=1
RANGE=9
MEAN=8
VARIANCE=18.051282
STANDARD DEV.=4.24868

PRESS ANY KEY TO CONTINUE
```

```
HOW MANY BINS ?10
MAXIMUM VALUE= 11
MINIMUM VALUE= 1
```



```
PRESS ANY KEY TO CONTINUE
```

(NB uses different data set to previous example.)

```
PLACE BLANK TAPE INTO
THE RECORDER

WHAT DO YOU WANT TO CALL
THE PROGRAM ?
TEST

PRESS PLAY AND RECORD
AND THEN PRESS ANY KEY
ON THE KEYBOARD
```

As the program has been constructed using subroutines the following table will help you to find your way around:

| line number | description |
|-------------|------------------------------------|
| 10 | The name of the program |
| 500-690 | The main menu routine |
| 1000-1270 | Generate random data |
| 1500-1670 | SAVE program and data |
| 2000-2140 | Calculate statistics |
| 2500-2580 | Print statistics |
| 3000-3140 | Manual data input |
| 4000-4150 | Secondary edit menu |
| 4200-4350 | Edit option (1) - list data |
| 4500-4590 | Edit option (2) - alter data value |
| 4600-4790 | Edit option (3) - delete data |
| 4800-5060 | Edit option (4) - add data |
| 5500-5530 | Calculate and print statistics |
| 6000-6230 | Plot histogram |
| 7000-7110 | Construct frequency count |
| 8900-8920 | Press any key to continue routine |

Most of these subroutines are easy enough to understand and those that are less than obvious should be easy to understand in the light of the discussion so far. However there are a few points that are worth noticing. In subroutine 1000 notice the random values are generated using the method described in the 1K book. In the case of fractional data a slightly different formula is used. The lines 1500-1670 SAVE the data on tape. They are not used as a subroutine but entered by a GOTO in line 650. The reason for this is that the auto run following SAVE wouldn't work if the program was SAVED from within a subroutine, (see previous section). In subroutine 4200 line 4330 causes the listing to pause after every 20 lines scrolled onto the screen. Subroutine 4800 adds new data to the existing data using the method discussed earlier, but notice the use of messages to stop users thinking that the machine has forgotten them!

The subroutines 6000 and 7000 are new in the sense that we haven't discussed the topic of plotting histograms before. Subroutine 7000 counts the number of data values that fall

into each interval using the equation in line 7020. If you imagine the intervals numbered from left to right starting at one, then the equation gives the number of the interval that any value falls into. This interval number is then used to select the element of H that has one added to it. In this way H(I) keeps count of the number of values falling in interval I. Subroutine 6000 prints the correct number of solid blocks to represent the number stored in H(I) for each value of I. The largest column of blocks that can be printed on the screen is 25. If we make this the length of the longest column of the histogram we print 25/F blocks for each count in H where F is the largest frequency, i.e. for a count of 1 we print 25/F blocks. So for each element of H we should print $H(I)*25/F$ blocks and this is what lines 6160–6180 ensure.

Since this is such a long program, there are many programming details that haven't been mentioned and the best way to become aware of how things work is to try to change or improve the program. Because of its modular design adding extra features is straightforward. Some suggestions for extensions are: add a print out of the frequency table H used in subroutines 6000 and 7000; change the program to handle more than one column of numbers and let the user choose which column to plot or calculate statistics on; and, when you feel confident of the techniques we have discussed in this book, add to this more-than-one-column program routines that will calculate the correlation co-efficient and plot a scatter diagram.

NUMBER FORMATTING

Although it cannot be called a serious deficiency, the ZX81 does lack any facility to control the way that numbers are printed out. If you use PRINT A then you cannot predict or control the number of digits before or after the decimal point and you certainly cannot control the position of the decimal point on the screen. The most you can do is to specify the screen location where the first digit of the number is to appear by using a PRINT TAB or PRINT AT statement. You may be wondering why you might want to go to the trouble of controlling the format that numbers are printed in. The answer is simply to make your print out and screen displays easier to read and less misleading. For example, if you print a number with too many digits after the decimal point you might lead an innocent user into the trap of believing that the calculation is really accurate to that degree. The ZX81 doesn't have the formatting commands that are available in other BASICs but it is possible to write subroutines to provide almost identical features.

Truncating and rounding

The danger of printing too many digits after the decimal point was mentioned in the previous section. It is surprising how often a calculation is carried out on numbers that are INPUT to only a few decimal places and the result then PRINTed to the maximum number of decimal places that the machine can handle. It is a sad fact, however, that calculations performed by a digital computer are not carried out with perfect precision and the final result is always less accurate than the data used in the calculation. The task of deciding how many digits should be printed after the decimal point is a difficult one to give any exact rules for. The best approach to adopt is to print only digits that you feel will be meaningful to the user

of the program. For example, if you are trying to calculate the amount of fuel oil to order to run a heating system for six months then there is little point in arriving at the answer:

NUMBER OF GALLONS OF FUEL NEEDED = 102.34983723

You would be lucky to find anyone who would sell you .3 of a Gallon, let alone .34983723 of a Gallon. It's true that no sensible person would order the exact figure printed by the computer but then again why have the computer print an answer that has to be further processed by a human. More seriously, if a column of fuel amounts was printed out in this way it would be quite difficult to compare figures because of all the spurious digits after the decimal point.

It is fairly easy to limit the number of digits printed after the decimal point using the INT function. Try the following program:

```
10 FOR N=1 TO 10
20 LET V=RND
30 PRINT "DIGITS= ";N;TAB 15;
40 GOSUB 1000
50 PRINT
60 NEXT N
70 STOP
```

```
1000 LET DIG=10**N
1010 PRINT INT (V*DIG)/DIG;
1020 RETURN
```

You should find that the number of digits printed after the decimal increases from 1 to 9. Subroutine 1000, which is responsible for "chopping" off any excess digits can be used by any program. V contains the number to be printed and N is the maximum number of digits to follow the decimal point. The way that the subroutine works is very simple and is best understood via an example. If V is .123 and N is 2 then DIG is $10^{**}2$ i.e. ten squared or 100. Multiplying V by 100 gives 12.3, INT chops off the fractional part giving 12 and dividing by DIG restores the number to its correct magnitude (i.e. .12). Notice that N only governs the maximum number of digits

```
DIGITS=1
DIGITS=2
DIGITS=3
DIGITS=4
DIGITS=5
DIGITS=6
DIGITS=7
DIGITS=8
DIGITS=9
DIGITS=10
```

```
0.8
0.71
0.364
0.3097
0.23228
0.42221
0.6664581
0.98474121
0.85560608
0.17060852
```

that follow the decimal point and it is perfectly feasible that fewer will be printed.

There is an objection to using subroutine 1000 as it stands and that is that is "chops off" the excess digits. This is usually called "truncating" the number, and humans are usually happier with the idea of "rounding" a number. If you want to round a number to a fixed number of decimal places you look at the first digit that would be lost if you truncated the number and add one to the next digit if it is 5 or greater. Thus 0.126 truncated to two decimal places is 0.12 and rounded to two decimal places is 0.13. You can convert subroutine 1000 to round numbers by changing line 1010 to read:

```
1010 PRINT INT (V*DIG+.5)/DIG;
```

If you follow through the calculation using .126 as V and 100 as the value for DIG you should see how it succeeds in rounding the number.

Aligning decimal points

If you were lucky you might have got a neat triangular shaped list of numbers from the example in the previous section but it is more likely that the print out looked more like this —

```
DIGITS= 1
DIGITS= 2
DIGITS= 3
DIGITS= 4
DIGITS= 5
DIGITS= 6
DIGITS= 7
DIGITS= 8
DIGITS= 9
DIGITS= 10
```

```
0.4
0.25
0.127
.0808
.02578
0.223493
.0977434
0.35104859
.068214123
0.12482941
```

To overcome this untidy problem we need a subroutine that will allow us to specify the position of the decimal point as well as the number of digits following. Columns of figures look much better when the decimal point is aligned and this is fairly easy to achieve on the ZX81. Try the following program:

```

10 LET M=7
20 INPUT V
30 PRINT TAB 10;
40 GOSUB 3000
50 LET V=V*100*RND
60 GOTO 30

3000 PRINT "          "(1 TO M-(V>1)*INT(LN
      V/LN 10)+(V<.1));V
3010 RETURN

```

```

      .001
     .0999006653
    2.5245669
   31.572495
  1201.5527
 65293.728
4938575.5

```

This program may look a little complicated, especially line 3000, but its action is easy to understand if you break it down. Lines 10 to 60 simply provide test numbers for subroutine 3000 to format. The variable M sets the "field width" i.e. the number of printing positions before the decimal point and V is the number to be formatted. To make sure that the decimal point is always printed in the same place you have to add a variable number of blanks to the front of the number before it is printed. For example, if the field width is 5 and you are printing a number with 2 digits in front of the decimal point then you have to print three blanks then the number — so to print 22.12 you would print blank|blank|blank|2|2|.1|2|. Obviously if there are N digits in front of the decimal point and the field width is M you have to print M-N blanks to "pad" the number out to M printing positions. This can be done by slicing a string of blanks thus

```
PRINT "          "(1 TO M-N)
```

The only thing left to do is to find a way of calculating N, the number of digits before the decimal point. Try the following program for a range of inputs from 1.0 to 100000.0

```

10 INPUT V
20 PRINT V,INT(LN V/LN 10)
30 GOTO 10

```

You should see that this prints out one less than the number of digits before the decimal point. The way that this works is that LOG V is the power that you have to raise 10 to get V i.e. $V=10^{**}(\text{LOG } V)$. If you chop off the fractional part of LOG V you have one less than the number of digits in front of the decimal point. The only trouble is that the ZX81 doesn't have a LOG (log to the base 10) function, only a LN (log to the base e) function. However using the relationship $\text{LOG } V = \text{LN } V / \text{LN } 10$ solves this problem. We can now write the formatting line as:

```
PRINT "          "(1 TO M-INT(LN V/LN 10));V
```

This is almost the same as line 3000 apart from the terms (V>1) and (V<.1). These are conditional expressions that evaluate to 0 if they are false and 1 if true, and are used to adjust the number of blanks printed to take account of the different way that the ZX81 prints numbers smaller than 1 and smaller than .1.

PRINT USING

Most print formatting problems can be solved using a combination of truncating, rounding or aligning the decimal point. However other versions of BASIC have a very powerful statement PRINT USING that allows a wide range of number formats to be specified. It is possible to write a subroutine that provides some of the capabilities of the PRINT USING command and this would be useful both for converting programs and for new writing new programs.

The format of a number produced by a PRINT USING statement is specified by the use of a "picture" of the number

stored in a string. For example, in most BASICs "###.##" would specify a format of three spaces or digits in front of the decimal point and two digits following i.e. 3.123 printed using this format would be |blank|blank|3|.1|2|. There are many other formatting symbols that can be combined with # to form a "picture" of the number but perhaps the most useful is the "floating" money sign. If you write either a dollar or a pound sign in front of the formatting "picture" the money sign will be printed to the immediate left of the formatted number. For example, "###.###" would format 3.1234 as |blank|blank|3|.1|2|3|. This method of drawing a "picture" of the number is a very easy to use and powerful formatting method. For example, if you don't want a decimal point printed then all you have to do is leave it out of the "picture" i.e. "###". If the number to be printed is too big for the space allocated to it by the "picture" then it is printed unformatted.

A general PRINT USING subroutine for the ZX81 would be rather long but we can produce a subroutine that will accept a "picture" involving digit positions marked by #, the decimal point and floating money signs. The only change that we have to make to the usual PRINT USING is to change the # to * because the ZX81 doesn't have a # character. The print using subroutine and a small test program is:

```

10 LET U$=" $*****.*"
20 PRINT TAB 9;U$
30 INPUT V
40 GOSUB 2000
50 LET V=V*100*RND
60 PRINT
70 GOTO 20

2000 LET H$=STR$ INT V
2010 LET L$=(STR$(V)) (LEN H$+1 TO)
2020 IF L$<>" " THEN IF L$(1)="." THEN LET L$
    =L$(2 TO )
2030 LET S$=U$(1)
2040 IF U$(1)<>"£" AND U$(1)<>"$" THEN LET
    S$=""

```

```

2050 LET F=0
2060 LET M=0
2070 LET N=0
2080 FOR I=1 TO LEN U$
2090 IF U$(I)="." THEN LET F=1
2100 IF U$(I)="*" AND F=0 THEN LET M=M+1
2110 IF U$(I)="*" AND F=1 THEN LET N=N+1
2120 NEXT I
2130 IF M=0 AND H$="0" THEN LET H$=""
2140 LET H$=S$+H$
2150 IF LEN H$>M THEN GOTO 2170
2160 LET H$="" (1 TO M-LEN H$)+H$
2170 IF F<0 THEN LET H$=H$+"."
2180 LET L$=L$+"00000000000000"
2190 IF N<0 THEN LET H$=H$+L$(1 TO N)
2200 PRINT H$;
2210 RETURN

```

The "picture" format is stored in the string U\$ at line 10. Lines 20-70 simply send test values in V for subroutine 2000 to format. Before any formatting begins the number contained in V is converted into a string by STR\$ and then split into two parts. The digits in front of the decimal point are stored in H\$ by line 2000 and the digits following the decimal point are stored in L\$ by lines 2010-2020. Notice the use of IF ... THEN, IF ... THEN construction in line 2020. This has the same effect as IF ... AND ... THEN but it is needed because in this case the second condition, i.e. L\$(1)=".", can only be worked out if the first condition is true i.e. if L\$<>" ". The expression IF L\$<>" " AND L\$(1) THEN ... will give an error message if L\$ is null because in this case L\$(1) doesn't exist. The variable S\$ is used to hold any floating money sign in the formatting string U\$, if there is no such sign then S\$ is set to the null string (Lines 2030-2040). Lines 2050-2120 count the number of digits before the decimal point (M) and the number of digits after the decimal point (N) in the formatting string U\$. The variable S is zero if no decimal point is found. Line 2020 removes the leading zero if the number is less than one and there is no digit position specified by the

format. Line 2030 adds the money sign, if any to the front of the number. The "padding" blanks are then added to the number in much the same way as described for aligning the decimal point in the last section by lines 2150 and 2160. If a decimal point is required it is added in line 2170 and then the fractional part stored in L\$ is padded with trailing zeros (line 2180) before being truncated to fit into the correct number of digits by line 2190. Finally, line 2200 prints the fully formatted number.

Three samples illustrating different formats:

```

.****
.00001
.00006
.7543
30.9278
2318.9732
54556.0130
3579173.1000
62866000.0000

```

```

*.****
0.00001
0.00006
0.00058
0.2472
23.0431
2067.3304
59288.7480
3023603.1000

```

```

£*****.
£0.01
£0.43
£10.18
£100.24
£9000.67
£9170571.30

```

Interest calculator

As an example of how the formatting subroutine given in the last section can be used, consider the problem of printing a table of the amount of money accumulated by a regular savings plan over a number of years with fixed interest rates.

Apart from being a good example of formatting, this could also serve as a rough guide to comparing saving plans. For simplicity an approximate but straightforward calculation has been used. If you save A pounds per month then you will have A*12 pounds more at the end of each year. If the interest rate is R% per annum then you will earn T*R/100 pounds of interest every year where T is the total amount due to saving and interest at the end of the year.

This program uses subroutine 2000 given in the last section so don't forget to type it in at the end of the new section listed below before you try to run it.

```

10 PRINT TAB 8;"I N T E R E S T"
20 PRINT
30 PRINT "HOW MUCH DO YOU WANT TO SAVE"
40 PRINT "PER MONTH ? ";
50 INPUT A
60 PRINT A
70 PRINT "HOW MANY YEARS ? ";
80 INPUT Y
90 PRINT Y
100 PRINT "ANNUAL INTEREST RATE"
110 PRINT "IN PERCENT ? ";
120 INPUT R
130 PRINT R
140 LET B=12*A
150 PRINT
160 PRINT "YEARLY SAVINGS = ";B
170 LET T=0
180 FOR K=1 TO Y
190 LET T=T+B
200 LET E=T*R/100
210 LET T=T+E
220 PRINT AT 20,0;"SAVINGS AT ";
230 LET U$="*"
240 LET V=K
250 GOSUB 2000
260 PRINT " YEARS";TAB 20;
270 LET U$="£*****.*"

```

```

280 LET V=T
290 GOSUB 2000
300 SCROLL
310 NEXT K
320 STOP

```

Lines 10 to 110 ask for the relevant details of the saving plan. After converting monthly saving to yearly saving (lines 140–160) the program calculates and prints the amount accumulated at the end of each year. Line 190 adds the yearly savings to the total. Line 200 calculates the interest due on this amount and line 210 adds it to the total. This calculation is repeated for each year that the savings plan has to run by the FOR statement at line 180. The formatting subroutine is used to print two numbers — the year number in lines 230–250 and the total amount in lines 270–290. You should be able to see that this provides a much neater set of results than simple printing.

INTEREST

```

HOW MUCH DO YOU WANT TO SAVE
PER MONTH ?50
HOW MANY YEARS ? 20
ANNUAL INTEREST RATE
IN PERCENT ?10

```

YEARLY SAVINGS =600

| | | |
|------------|----------|-----------|
| SAVINGS AT | 1 YEARS | £660.00 |
| SAVINGS AT | 2 YEARS | £1386.00 |
| SAVINGS AT | 3 YEARS | £2184.60 |
| SAVINGS AT | 4 YEARS | £3063.06 |
| SAVINGS AT | 5 YEARS | £4029.36 |
| SAVINGS AT | 6 YEARS | £5092.30 |
| SAVINGS AT | 7 YEARS | £6261.53 |
| SAVINGS AT | 8 YEARS | £7547.68 |
| SAVINGS AT | 9 YEARS | £8962.45 |
| SAVINGS AT | 10 YEARS | £10518.70 |
| SAVINGS AT | 11 YEARS | £12230.57 |
| SAVINGS AT | 12 YEARS | £14113.62 |
| SAVINGS AT | 13 YEARS | £16184.99 |
| SAVINGS AT | 14 YEARS | £18463.46 |
| SAVINGS AT | 15 YEARS | £20969.83 |
| SAVINGS AT | 16 YEARS | £23726.82 |
| SAVINGS AT | 17 YEARS | £26759.50 |
| SAVINGS AT | 18 YEARS | £30095.45 |
| SAVINGS AT | 19 YEARS | £33764.00 |
| SAVINGS AT | 20 YEARS | £37801.50 |

Try running this program with savings of say fifty pounds a month for periods of 25–30 years. You might be surprised at the size of the sum that you have accumulated at the end!

Chapter Eight

THE PRINTER

The ZX81 printer is a remarkable piece of hardware. Not only can it deliver printed copies of whatever is on the screen and your program and data listings, but it can also be used to increase the size and resolution of the graphics screen. Adding a printer to your ZX81, therefore, opens up a whole range of applications.

How the printer works

It may be useful to consider how the printer functions before going on to see what sort of things we can do with it. The ZX81 printer works by evaporating the aluminium coating from the surface of a roll of black paper. Where the aluminium is removed the black shows through and it is this, rather than ink of any sort, that makes the printing stand out black on a silvery background. If you want to test this, take a scrap of printer paper and scrape it with something. As the aluminium comes off you will see the black paper beneath. The ZX81 printer evaporates the aluminium with an electric spark. If you operate it in a slightly darkened room you can see blue flashing sparks just under the tear bar. These are created by a pair of styli (sharp metal points) that are mounted on a motor driven belt. The belt and the paper feed roller are driven continuously while printing and as a result the styli pass over the paper in a series of horizontal lines which move down the paper. At any time during the "scanning" of the paper a high voltage (but safe) pulse of electricity can be applied to the one of the styli to burn a black dot. Thus to build up the shape of a letter the ZX81 has to send pulses at the right time to burn the correct pattern of dots. As we shall see it is possible to not only print the dot pattern for a letter but the dot pattern for any shape. This extra freedom can be used to produce high resolution plots and even a lower case character set using only BASIC.

Low resolution printing

There are three printer commands in BASIC — LLIST, LLPRINT and COPY. The LLIST command can be used in the same way as the LIST command except of course that the listing is produced on the printer instead of the screen. The two most interesting printer commands are LPRINT and COPY.

LPRINT can be used in the same way as the PRINT statement. You can use both TAB and AT to control where the output will be placed on the paper. However, because the printer cannot feed paper backwards, any line information in AT is ignored. For example:

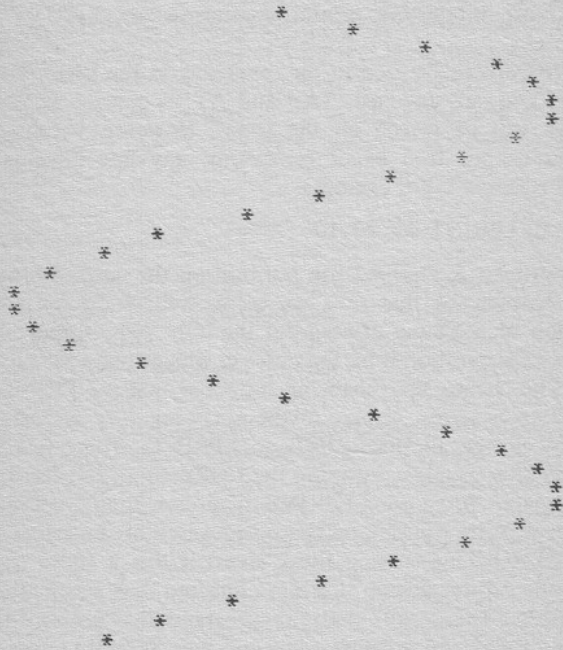
```
10 LPRINT AT 20,10;"X"
```

will print an X at the tenth position on the current printing line. This means that you can easily control the horizontal position of anything printed but the only way of controlling the vertical position is by the order in which things are printed out. This means that changing programs that use PRINT AT to create a screen display might not work very well if you simply change all the PRINTs to LPRINTs. There is a way around this problem by using the COPY command but more of this later. When using LPRINT the printer should be treated as if you were printing on the bottom line of the screen and then SCROLLing. If you think about it this is more or less what the printer does — it prints a line and then automatically SCROLLs. This means that any programs that produce their screen displays by printing a line and then SCROLLing can be converted to produce printer listings by changing the PRINT to LPRINT and taking out the scroll. For example the list of numbers produced by INTEREST or STATISTICS can be transferred to the printer in this way. As another example consider the following program that plots a sine wave on the screen —

```
10 FOR X=0 TO 8*PI STEP .3  
20 LET Y=(SIN(X)+1)*15  
30 PRINT AT 21,Y;"*"
```

```
40 SCROLL
50 NEXT X
```

can be changed to produce the same sine wave on the printer by removing line 40 and changing PRINT to LPRINT.



LPRINT can be used to print any of the ZX81's characters on the printer including the graphics characters and should behave exactly like PRINT. However there is a small bug that occurs when printing numbers in the range .00001 to .0099999999. The leading zeros after the first are printed as non-numeric characters instead of zero! To overcome this problem when using LPRINT it is better to convert all numbers to strings using the STR\$ function. For example instead of

```
10 LPRINT A
```

use

```
10 LPRINT STR$(A)
```

Although LPRINT is a very useful command and is not to be overlooked, especially when you are writing programs that use the printer from scratch, the easiest and most powerful printer command is COPY. COPY will "dump" to the printer an entire screen no matter what it contains. The advantage of this is that you can use PRINT AT or PLOT/UNPLOT to construct a screen as you would normally and then before moving on to the next part of the program save it on the printer. In this sense you can write programs forgetting about the printer's existence until the very moment that you need the print out!

There are two ways of using the COPY command. If at any time during a program you see a screen full of information that you feel would be better printed out, you can stop the program by pressing the BREAK key and enter the COPY command. After the screen has been printed you can restart the program by pressing CONTINUE. The only thing that you have to be careful of is that you must not enter any other commands by accident because this would not only make restarting the program difficult, if not impossible, it would clear the screen and make any further attempt at using COPY futile. A much better way of using COPY is to build it into a program and allow the user the option of copying the screen before the next one is produced. A fact that is often overlooked is that the COPY command can be used as a statement in a program. For example, try the following

```
10 FOR I=1 TO 20
20 PRINT RND
30 NEXT I
40 COPY
50 CLS
60 GOTO 10
```

Lines 10 to 30 print 20 random numbers on the screen and then line 40 COPYs them to the printer. The screen is then

cleared and another 20 numbers printed. Notice that the action of the COPY command is entirely automatic and no user intervention is required.

Using this idea it should be possible to convert programs to use the printer very quickly. Once you have identified the points at which a whole screen is produced simply insert a question "DO YOU WANT TO PRINT THE SCREEN?" and if the answer is "yes" do a COPY. This will provide printer versions of standard output without the need to worry about the LPRINT bug mentioned earlier and is the only way of printing graphics produced using PLOT/UNPLOT.

High resolution plotting

Using a few simple subroutines it is possible to produce high resolution (256 by 256 points) graphs and as many user-defined characters as you want. The basic ideas outlined in the following sections are derived from the demonstration programs in the printer manual. However you should be able to use the subroutines presented here to produce your own programs, not just demonstrations.

To use the printer in high resolution mode it is necessary to modify part of the machine code stored in the ROM. Machine code is treated in more detail in Chapter Ten but for the purposes of this chapter all you need to know is that it is possible to copy the definition of LPRINT from ROM to an area of RAM and then modify it. Before you can do this it is important to reserve an area of memory that BASIC will not try to use. This can be done by altering the address stored in RAMTOP using the following commands:

```
POKE 16389,124
NEW
```

To see high resolution plotting working try the following program which plots random points. (You might find that changing to FAST mode speeds this program up quite a lot.)

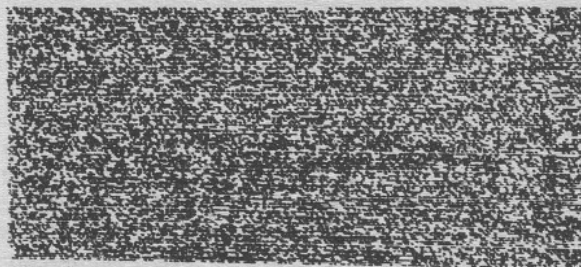
```
10 GOSUB 1000
20 GOSUB 3000
```

```
30 GOSUB 2000
40 GOTO 20
```

```
1000 IF PEEK 16388+256*PEEK 16389=31744
    THEN GOTO 1030
1010 PRINT "MEMORY NOT RESERVED"
1020 STOP
1030 FOR I=0 TO 112
1040 POKE 31744+I,PEEK (2161+I)
1050 NEXT I
1060 POKE 31800,63
1070 POKE 31857,201
1080 RETURN
```

```
2000 FOR H=0 TO 31
2010 POKE 16444+H,H
2020 NEXT H
2030 LET H=USR 31744
2040 RETURN
```

```
3000 FOR I=1 TO 32*8
3010 POKE 32255+I,255*RND
3020 NEXT I
3030 RETURN
```



High resolution pattern (compare with low resolution equivalent on p. 31 of 1K book).

Subroutine 1000 transfers the machine code that defines the way LPRINT works, into the RAM reserved for it. Before this is done lines 1000-1020 check to make sure that RAMTOP has been altered and the memory reserved. The FOR loop (lines 1030-1050) transfers 113 bytes of machine code from 2161 (ROM area) to 31744 (reserved RAM area). Lines 1060 and 1070 POKE the required two changes to the machine code. These two changes make it print the contents of the 256 memory locations from 32256 onwards. Subroutine 2000 is responsible for initiating the printing. Lines 2000 to 2020 set up a character count in the printer buffer starting at 16444. Line 2030 is the line that actually starts the printing by calling the machine code set up by subroutine 1000. This is done via the

USR "address"

function which transfers control to a machine code subroutine starting at "address" in much the same way as GOSUB transfers control to a BASIC subroutine at the stated line number. The only complication is that USR is a function (like SIN or COS) and therefore has to be used in an expression. This is the only reason that line 2030 starts with LET H=. We are not interested in the value stored in H as a result of this assignment, just in getting to the machine code starting at 31744. Subroutine 3000 changes the contents of the area of memory that is printed by subroutine 2000 by POKEing random numbers. (Remember that a memory location can only store numbers between 0 and 255.)

The operation of the program should now be easy to understand. First line 10 calls subroutine 1000 to set up the machine code. Then line 20 calls subroutine 3000 to randomly alter the area of memory to be printed and line 30 calls subroutine 2000 to print it. This is then repeated over and over until you get bored and press "break"! These three stages, set up machine code, set up area of memory to be printed and print it, are the three fundamentals of high resolution printing. Subroutines 1000 and 2000 will be used as they are in all the subsequent programs in this chapter. The only thing that will change is the way the memory is set up before printing.

Plotting a sine curve

The previous example served to introduce the fundamental subroutines 1000 and 2000 but the output could hardly be called useful. To be able to use the high resolution printing facility we must obviously find out how the information POKEd into the memory area by subroutine 3000 controls the pattern of dots produced by the printer. Unfortunately this is where most of the difficulties of high resolution graphics lie. Each time subroutine 2000 is used the equivalent of a whole row of characters is printed out. As each character is formed in a square of eight by eight dots, a whole row is 32x8x8 dots or 2048 dots. This is obviously a lot more than the 256 memory locations that are used to define the pattern of dots that are printed by subroutine 2000. The answer is that each memory location controls the state, i.e. black or white, of eight individual dots. Thus each memory location controls a row of eight dots. You may think that the most obvious arrangement would be for the first 32 memory locations to control the pattern of dots of the first row of the printout. This is not so, the first eight memory locations control the eight rows that make up the first character position, the next eight control the second character position etc.

| | 1st character | 2nd character | ... nth character |
|-------|------------------|------------------|-------------------------|
| Row 1 | memory 1 | memory 9 | |
| 2 | location 2 | location 10 | |
| 3 | 3 | 11 | |
| 4 | 4 | 12 | |
| 5 | 5 | 13 | |
| 6 | 6 | 14 | |
| 7 | 7 | 15 | |
| 8 | 8 | 16 | |

The way that each memory location controls eight bits is slightly more difficult to describe. Depending on which of the

eight points that you want to turn on (i.e. print as black) then you have to POKE one of the following numbers

| Point | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-------|-----|----|----|----|---|---|---|---|
| code | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

So if you wanted to make the point in the top left hand corner black you would POKE 128 into the first memory location. If you wanted to make the third point in the 2nd row of the 2nd character position black then you would POKE 32 into the tenth memory location. If you want to turn on more than one point then you simply POKE the sum of the appropriate codes into the correct memory location. For example, if you wanted to turn on the fifth and second point in the first row of the first character location then you would POKE 64+8, i.e. 72, into the first memory location.

This may seem very complicated but you should be able to see that it is possible to find which memory location and what to POKE into it to turn on any pattern of points. We can summarise the rules given above into a few lines of BASIC. If you imagine the 32 character locations and number the 256 points starting at the left and eight rows starting at the top, then the X(horizontal), Y(vertical) point can be turned on by

```
100 LET C=INT(X/8)
110 LET B=7-X+C*8
120 POKE 32255+C*8+X,2**B
```

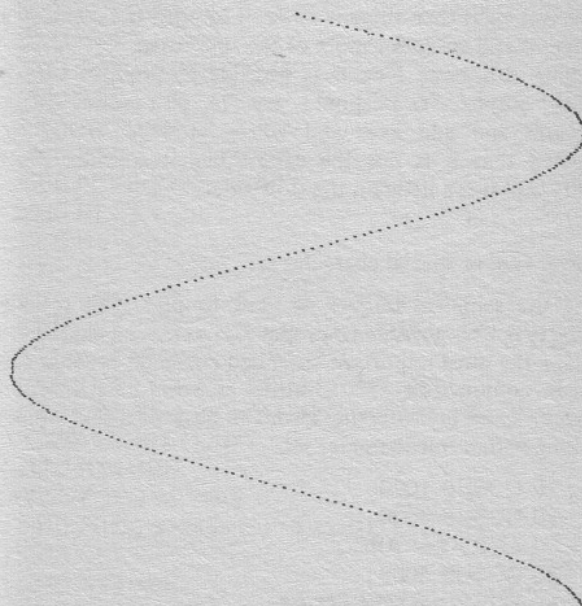
Line 100 calculates the character position and line 110 calculates the bit position within the character position.

We can now make use of all this information to plot a sine curve (or any other equation or data for that matter). Try the following program. Once again it is worth running it in FAST mode:

```
10 LET L=0
20 GOSUB 1000
```

```
30 FOR X=0 TO 8*PI STEP .03
40 LET Y=(SIN(X)+1)*127
50 GOSUB 3000
60 NEXT X
70 STOP
```

```
3000 LET Y=INT Y
3010 LET K=INT(Y/8)
3020 LET R=7-Y+K*8
3030 FOR I=0 TO 31
3040 POKE 32256+I*8+L,0
3050 IF K=I THEN POKE 32256+I*8+L,2**R
3060 NEXT I
3070 LET L=L+1
3080 IF L=8 THEN GOSUB 2000
3090 IF L=8 THEN LET L=0
3100 RETURN
```



The first line (10) sets L, the row counter to zero. When L reaches eight we have plotted eight rows of dots and it is time to call subroutine 2000 to print the section of the graph out (line 3070). Line 20 sets up the machine code subroutine. Lines 20–50 calculate the sine curve from 0 to 8π . The curve is plotted running down the paper so in this case Y, the value of sine, runs across the paper. Y is scaled so that at the lowest value of sine it is 0 and at the largest it is 254. The scaling of the X value is unimportant in this case because each value of Y that is produced is plotted on the next row of dots. Once again it is subroutine 3000 that is responsible for altering the memory to produce the required pattern of dots. For each Y value the character position and dot position within the character position are calculated. The entire row is zeroed (lines 3030 to 3060), to clear the last plotting information, and the new point is POKEd at the correct place (line 3050). The row count is incremented at line 3070. If the row count (L) is not eight then there is no need to print the memory area as there is still space for more of the sine curve.

Using this basic idea it is possible to produce very high quality graphs. You should try to plot some different functions and add axes and labels. However it has to be admitted that it is not the easiest high resolution plotting facility and it is a little on the slow side!

A lower case or special character set

Using the same techniques as used to plot high resolution graphs it is also possible to produce an extended character set. Perhaps the most important extended character set that can be used in conjunction with a printer is lower case letters. The program listed below could form the basis of a text processor working with a full character set.

```
10 GOSUB 1000
20 GOSUB 4000
30 LET A$="ABCD"
40 GOSUB 3000
50 FOR P=1 TO LEN A$
```

```
60 LET C=CODE A$(P)-37
70 GOSUB 5000
80 NEXT P
90 GOSUB 2000
100 STOP
```

```
1000 IF PEEK 16388+256*PEEK 16389=31744
    THEN GOTO 1030
1010 PRINT "MEMORY NOT RESERVED"
1020 STOP
1030 FOR I=0 TO 112
1040 POKE 31744+I,PEEK (2161+I)
1050 NEXT I
1060 POKE 31800,63
1070 POKE 31857,201
1080 RETURN
2000 FOR H=0 TO 31
2010 POKE 16444+H,H
2020 NEXT H
2030 LET H=USR 31744
2040 LET L=0
2050 RETURN
3000 FOR I=0 TO 31*8
3010 POKE 32255+I,0
3020 NEXT I
3030 RETURN
4000 DIM L$(26,8)
4010 LET L$(1)=" S[4]W"+CHR$(68)+"W "
4020 LET L$(2)=" RND RND"+CHR$(120)+CHR$(68)+CHR$(68)+CHR$(120)+" "
4030 LET L$(3)=" SRNDRNDRNDS "
4040 LET L$(4)=" [4][4]W"+CHR$(68)+CHR$(68)+CHR$(120)
4999 RETURN
5000 FOR I=1 TO 8
5010 POKE 32255+I+P*8,CODE L$(C,I)
5020 NEXT I
5030 RETURN
```


ADVANCED RANDOMNESS

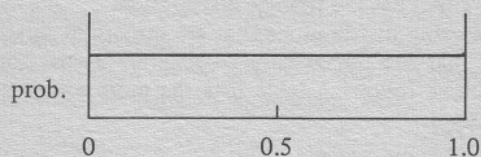
Randomness is a fundamental component of playing games on a computer and we examined some of the ideas involved in "The Art of Programming the 1K ZX81". You may think that after such a full coverage there would be little left to say about the topic of randomness and its uses. In fact, we have hardly scratched the surface of one of the most important areas of computer applications. In the first part of this chapter we will extend some of the ideas introduced in the 1K book and in the second part look at some more uses, both serious and lighthearted, of randomness in computing.

Continuous events

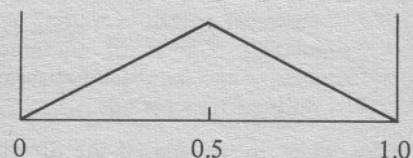
We considered how to generate a random number of discrete events with any given probability in the 1K book but there are situations when we need to select a point along a continuum at random. For example, in a simulation to do with the weather we might want to select temperature at random. Of course this could be done using the methods outlined in the 1K book — divide the range of possible temperatures up into sections, assign each to a value and then pick one of the values at random. However, this is an unnecessary step because the random number generator actually produces numbers that lie on a continuum in the first place. RND is a function that returns a number that is greater than or equal to 0 and less than 1 at random and there are an infinity of such numbers — or would be if the ZX81 did not truncate them in order to display them. So, if we want to predict a continuous event at random then we can use the RND function to do so.

The next point to note is that RND is just one of a whole range of probability generators that we could use. Each one produces numbers over a given range with some probability. The way the probability varies over the range can be plotted

on a graph and such a graph is known as a probability distribution. For example, the probability generator RND produces numbers over the range 0 to 1 and the probability of any number in this interval being produced is constant. A graph of the distribution corresponding to RND would look something like this:



It is not unreasonably called the uniform distribution. Other generators would be associated with different distributions. For example



is a distribution where the probability of producing any number increases as it gets closer to 0.5. This type of distribution may be useful in simulations where a "central" outcome should happen more often than anything else but where there ought to be a chance of something extreme happening occasionally. Consider as an example a game where "sales" of lemonade depend on the temperature. The game's instructions tell you to make your decisions as if it was July in England. On the basis of our typical summer weather you know that the weather is likely to be fairly warm but there is an outside chance of either a heat wave or a really cold snap. The triangle distribution is an appropriate one to use in such a program. The equation to use to produce a random temperature is $2 * M * TRI$ where M is the mean temperature for July and TRI is a new function that generates numbers according to the triangle distribution. (You multiply by 2 as the mid-point of

the distribution is 0.5) There is an obvious problem in that no BASIC has a function called TRI so the best that we can do is to write subroutines that return numbers with the specified distribution. For example, the triangle distribution can be produced by adding together two numbers from a uniform distribution. In other words $LET T=(RND+RND)/2$ produces a number in T that has the triangle distribution.

There are many distributions that are important in serious applications that are not as easy to generate as the triangle distribution. The following table lists the most common and their typical uses:

| <i>Distribution</i> | <i>Typical Use</i> |
|---------------------|--|
| Normal | simulating "naturally" occurring distributions involving measurement error |
| Chi Squared | in statistical sampling experiments |
| Exponential | in economics, simulating time between customer arrivals or waiting periods |
| Binomial | simulating number of equipment failures |
| Poisson | simulating the number of customers arriving during a time period. |

Knowing how or when to use any of the distributions is a matter of understanding the particular problem that you are faced with. An example of when to use the poisson distribution is given later but just in case you need to generate any of the other distributions the following collection of subroutines will prove useful. Don't worry about understanding how they work unless you know something about probability theory. If you would like to look at the "shape" of the distribution associated with any of the following subroutines then you could use it to replace the random data generator in the statistics program and then plot a number of histograms. Remember to substitute appropriate values for the parameters.

The Normal Distribution

As this is such an important distribution two methods are given. The first is based on the Central Limit Theorem. It is an approximation but is good for most purposes.

```
1000 LET Z=0
1010 FOR I=1 TO N
1020 LET Z=Z+RND
1030 NEXT I
1040 LET Z=SQR(3/N)*(2*Z-N)
1050 RETURN
```

The accuracy of the approximation improves as N gets bigger. Suitable values for N are between 20 and 50. The next subroutine is based on the Box-Muller method and should be used only when the highest accuracy is required

```
1000 LET Z=SQR(-2*LN(RND))*COS(PI*RND)
1010 RETURN
```

Both subroutines return Z which has a normal distribution with a mean of zero and a standard deviation of one.

The CHI Squared Distribution

For a distribution with two degrees of freedom use

```
1000 LET X=-2*LOG(RND)
```

for a distribution with an even number of degrees of freedom, $D = 2N$ say, use

```
1000 LET U=1
1010 FOR I=1 TO D
1020 LET U=U*RND
1030 NEXT I
1040 LET X=-2*LOG(U)
```

to generate intermediate degrees of freedom $2N+1$ use

```
1000 LET Y=X+Z*Z
```

where X is CHI squared with $2N$ degrees of freedom and Z is normal with mean zero and standard deviation one.

The Exponential Distribution

The subroutine to substitute is:


```
1000 LET X=-(1/L)*LN(RND)
1010 RETURN
```

where X has the distribution $1 - \exp(-L \cdot X)$.

The Binomial Distribution

In the following subroutine X has the binomial distribution. N is the total number of trials and P is the probability of success.

```
1000 LET X=0
1010 FOR I=1 TO N
1020 IF RND>P THEN GOTO 1040
1030 LET X=X+1
1040 NEXT I
1050 RETURN
```

The Poisson Distribution

You will find this subroutine used in the simulation program presented later in this chapter:

```
1000 LET X=0
1010 LET E=EXP -M
1020 LET P=1
1030 LET P=P*RND
1040 IF P<E THEN RETURN
1050 LET X=X+1
1060 GOTO 1030
```

X is an integer between 0 and infinity and has the distribution $M^X \cdot \exp(-M) / X!$.

Monte Carlo integration — finding PI

No example of a "serious" use of random numbers has been included so far. The major application of random numbers is in the simulation of what happens or what might happen in the real world. An example of this sort of application is given

later but first let's look at a more unusual application. It is possible to avoid a wide range of mathematical calculations by using random methods. Such methods are often called Monte Carlo methods after the famous casino.

Finding the area under a curve specified by an equation is usually done by using integration. Sometimes however it is very difficult to solve this sort of problem using classical mathematics and some other approach has to be found. Integration is based on some very advanced mathematics but if you think about the problem as finding the area under a curve then you can use some very simple methods based on common sense. Consider the problem of integrating x squared between 0 and 1. In other words finding the area below the graph of x squared between zero and one. We could use the usual methods of calculus or numerical integration (i.e. Simpsons rule) but, for the sake of an easy example, let's suppose that these methods are unavailable. If you generate two random numbers, X and Y say, then these can be thought of as defining a point in the unit square, (i.e. X and Y are both positive and less than one). This point will lie either above or below the curve. Now obviously the probability that the point is below the curve depends on the area under the curve and this is the key to Monte Carlo integration. If we generate a sequence of pairs of random numbers and count the proportion of them that fall below the curve this can be taken as an estimate of the area. Notice that it is only an estimate of the area, you cannot guarantee that it is equal to the area. However the estimate improves as you increase the number of random points that you use.

To see Monte Carlo Integration in action try the following program which estimates the area under the curve X squared.

```
10 LET H=0
20 LET N=0
30 LET X=RND
40 LET Y=RND
50 IF Y<X*X THEN LET H=H+1
60 LET N=N+1
70 PRINT AT 21,0;"AREA=";H/N;"N=";N
```

```
80 SCROLL
90 GOTO 30
```

The variable H is used to count the number of points that fall below the curve and N counts the number of points that have been generated. Lines 30 and 40 generate the two random numbers in question and line 50 checks to see if the point is below the curve or not. If it is then one is added to H. The estimate of the area so far is printed at line 70.

```
AREA=1          N=1
AREA=1          N=2
AREA=0.66666667 N=3
AREA=0.5         N=4
AREA=0.4         N=5
AREA=0.33333333 N=6
AREA=0.42857143 N=7
AREA=0.375       N=8
AREA=0.33333333 N=9
AREA=0.3         N=10
AREA=0.27272727 N=11
AREA=0.25        N=12
AREA=0.23076923  N=13
AREA=0.21428571  N=14
AREA=0.2         N=15
AREA=0.1875      N=16
AREA=0.17647059  N=17
AREA=0.16666667  N=18
AREA=0.21052632  N=19
```

If you run this program for any length of time you will see that slowly the estimate of the area settles down to an increasingly accurate number. By using traditional integration the area under the curve is known to be exactly $1/3$ or $.333$ recurring and this can be used to see how well the Monte Carlo Method is doing. You may be a little disappointed to discover how slowly the method works its way to the correct answer. When we ran it for example, at $N=100$ the estimate was $.352$ and the second decimal place was still changing. This would seem to make Monte Carlo Integration of little use and this is true for simple problems such as the above example. However when you change the problem to one involving more than two dimensions, for example, trying to find the volume

under a curved surface, Monte Carlo Integration is one of the best methods we have!

As a final example of Monte Carlo Integration it is interesting to consider finding the area under the curve $1/(X*X+1)$ between 0 and 1. By standard integration this can be shown to be $\pi/4$, so if we use Monte Carlo Integration to find the area we have a technique involving random numbers to calculate π . To calculate π change the following lines in the area program above:

```
50 IF Y<1/(X*X+1) THEN LET H=H+1
70 PRINT AT 21,0;"PI=";4*H/N;"N=";N
```

```

:
PI=3.1894934    N=533
PI=3.1910112    N=534
PI=3.1850467    N=535
PI=3.1865672    N=536
PI=3.1880819    N=537
PI=3.1821561    N=538
PI=3.1836735    N=539
PI=3.1851852    N=540
PI=3.1792976    N=541
PI=3.1734317    N=542
PI=3.174954     N=543
PI=3.1764706    N=544
PI=3.1779817    N=545
PI=3.1794872    N=546
PI=3.1809872    N=547
PI=3.1824818    N=548
PI=3.1839709    N=549
PI=3.1854546    N=550
PI=3.1869329    N=551
PI=3.1884058    N=552
PI=3.1898734    N=553
PI=3.1913357    N=554
:

```

Problems with random numbers

Many of the serious applications of random numbers depend for their success on the "quality" of the random numbers used. This problem hardly arises when playing games, all that we demand of the sequence of numbers in this case is that a human player can treat them as random. However for techniques like Monte Carlo integration the accuracy of the results

depends on the quality of the random number generator. If, for example, the random number generator tended to give numbers close to one more often than numbers close to zero it is easy to see that the estimate of the area would be biased. The subject of testing random number generators is quite a complicated one. Essentially it involves running the random number generator and using statistics to see if the sequence of numbers satisfies the conditions of randomness. For example, the condition that each number should occur with the same probability can be examined by plotting a histogram of a large sample from the generator.

There is a very simple method of improving the output from any random number generator — “shuffling”. The following is a typical shuffling procedure:

- 1 Fill an array A of size N with random numbers from the “suspect” generator.
- 2) Generate an integer random number R say in the range 1 to N and swap the contents of A(1) with A(R).
- 3) Repeat (2) for each element of the array.
- 4) Use the N random numbers in the array as the next N in the sequence.

The following subroutine allows you to use this method in a program:

```

10 DIM A(N)

5000 FOR I=1 TO N
5010 LET A(I)=RND
5020 NEXT I
5030 FOR I=1 TO N
5040 LET T=A(I)
5050 LET R=INT(RND*N)+1
5060 LET A(I)=A(R)
5070 LET A(R)=T
5080 NEXT I
5090 RETURN

```

The general rule is that the more you shuffle the better the random numbers! Of course the trouble is that shuffling takes

a lot of time and if you need a lot of random numbers then it's better to work on the quality of the random number generator instead. Even so, shuffling can make a poor generator very good — so if in doubt shuffle!

A business simulation

As an example of a non-mathematical use of random numbers and distributions consider the following problem. A bread shop that bakes its own bread needs to know how many loaves to bake so that the wastage is kept to a minimum but also that the smallest possible number of customers are disappointed. If we assume that customers come into the shop at the same rate all day we can use the Poisson distribution to simulate the number of potential customers per day. If any event has a constant probability of happening per unit time then you can model it with the Poisson distribution. For example, if you assume that an accident is likely to happen at any time, the number that you see in any given time will have the Poisson distribution.

Returning to the bread shop problem we can suppose that we know the average number of customers per day that come into the shop. This can be found by counting the number of customers (including those turned away after the shop ran out of bread) over a period of time and dividing by the number of days. If the average number of customers is M then the actual number of customers coming into the shop will have a Poisson distribution with mean rate M i.e. the probability of getting X customers in any one day is given by

$$\frac{M^X \text{EXP}(-M)}{X!}$$

To discover the effect of baking S loaves of bread all we have to do is to generate random numbers with a Poisson distribution with mean rate M and see how often the number of customers is more or less than S. This is the method used by the following program:


```

10 LET L=0
20 LET W=0
30 PRINT TAB 8;"S E R V I C E"
40 PRINT AT 5,0
50 PRINT "AVERAGE NUMBER OF CUSTOMERS"
60 PRINT "PER DAY ";
70 INPUT M
80 PRINT M
90 PRINT "HOW MANY CUSTOMERS CAN YOU"
100 PRINT "SERVE IN ONE DAY ? ";
110 INPUT S
120 PRINT S
130 FOR I=1 TO 30
140 GOSUB 1000
150 IF X>S THEN LET L=L+X-S
160 IF X<S THEN LET W=W+S-X
170 PRINT AT 21,0;"DAY ";I;"CUSTOMERS= ";X
180 SCROLL
190 NEXT I
200 PRINT
210 SCROLL
220 PRINT "CUSTOMERS LOST=";L
230 SCROLL
240 PRINT "WASTAGE=";W
250 STOP

```

```

1000 LET X=0
1010 LET E=EXP-M
1020 LET P=1
1030 LET P=P*RND
1040 IF P<E THEN RETURN
1050 LET X=X+1
1060 GOTO 1030

```

Lines 30 to 120 collect the information necessary to start the simulation off. The average number of customers is stored in M and the number of customers that can be served, i.e. the number of loaves of bread baked (assuming one loaf per customer) is stored in S. Lines 130 to 190 simulate the num-

ber of customers coming to the bread shop over a 30 (working) day period. For each day a number of customers is generated by a call to subroutine 1000. You should recognise subroutine 1000 as a Poisson number generator. This number of customers is compared with the number of customers that can be served. If it is greater then the number of customers lost is calculated (line 150) and if it is smaller then the number of wasted loaves is calculated (line 160). At the end of the thirty-day period the total number of customers lost and loaves wasted is printed.

Using this simulation it is possible for the manager of the bread shop to see the effect of baking more or less loaves. Of course, if the profit on a sale is very much larger than the cost of throwing bread away it would be worth baking more bread than you would expect to sell. If you want to you could change the program to ask for the costs associated with both creating waste and having to refuse custom and print out the adjusted profit for any number of loaves baked! However the program as it stands can be used to simulate the behaviour of any business where a fixed number of customers can be served, i.e. a newsagent's.

S E R V I C E

AVERAGE NUMBER OF CUSTOMERS
PER DAY 50

HOW MANY CUSTOMERS CAN YOU
SERVE IN ONE DAY ? 60

```

DAY 1 CUSTOMERS=46
DAY 2 CUSTOMERS=57
DAY 3 CUSTOMERS=41
DAY 4 CUSTOMERS=50
DAY 5 CUSTOMERS=56
DAY 6 CUSTOMERS=48
DAY 7 CUSTOMERS=44
DAY 8 CUSTOMERS=53
DAY 9 CUSTOMERS=54
DAY 10 CUSTOMERS=52
DAY 11 CUSTOMERS=49
DAY 12 CUSTOMERS=48
DAY 13 CUSTOMERS=60
DAY 14 CUSTOMERS=51
DAY 15 CUSTOMERS=61
DAY 16 CUSTOMERS=42
DAY 17 CUSTOMERS=57
DAY 18 CUSTOMERS=45
DAY 19 CUSTOMERS=49

```

```

DAY 20 CUSTOMERS=57
DAY 21 CUSTOMERS=52
DAY 22 CUSTOMERS=52
DAY 23 CUSTOMERS=44
DAY 24 CUSTOMERS=51
DAY 25 CUSTOMERS=39
DAY 26 CUSTOMERS=45
DAY 27 CUSTOMERS=37
DAY 28 CUSTOMERS=41
DAY 29 CUSTOMERS=46
DAY 30 CUSTOMERS=53

```

```

CUSTOMERS LOST=1
WASTAGE=330

```

This example is simple enough for you to be able to see what is going on. In general, business simulations are complicated because they have to take account of the behaviour of human beings. They need to allow for the reactions of the consumers. For example, if on one occasion a person cannot buy a loaf of bread at your shop it is possible that they will take their custom elsewhere on a permanent basis!

Sinclair's Triangle

Although this chapter on randomness is mainly concerned with serious applications it ends on a lighter note! Sinclair's triangle is an entertaining example of a "random walk". An object is said to take a random walk if its movement is in some way governed by probability. One example would be to plot a point at X,Y and then move it by adding a random number to both co-ordinates. The path that many natural phenomena take is a random walk of one sort or another. Examples include gas particles and pollen grains in air. Sinclair's triangle is based on the path that a lightning flash might take.

```

10 LET X=15
20 LET I=0
30 FOR Y=1 TO 21
40 IF RND>.5 THEN GOTO 80
50 LET I=I+1
60 PRINT AT Y,X+I;"[T]"
70 GOTO 100

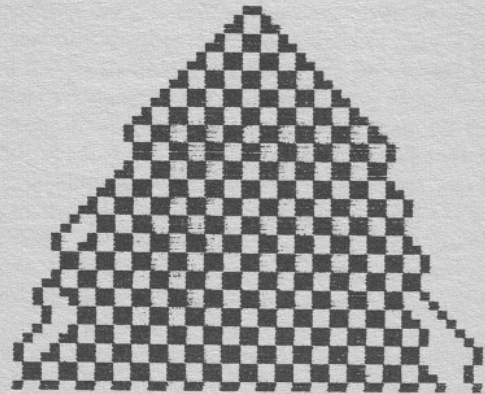
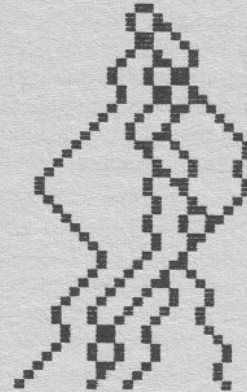
```

```

80 PRINT AT Y,X+I;"[Y]"
90 LET I=I-1
100 NEXT Y
110 GOTO 10

```

Line 10 fixes the Point that the bolt of lightning starts from. Lines 20 to 100 plot points on a random walk from the top of the screen to the bottom. Line 40 decides if the next step should move to the right or the left. Line 50 and 60 move the "lightning" to the right and lines 80 and 90 move it to the left.



If you run the program for a short time you will see a few irregular zig zag lines but if you leave it to run for longer a checkered pattern will slowly be built up. This is triangle-shaped as all the lightning bolts start off from the same fixed point. This routine could be used to produce a laser beam for a "zap-the-alien" type game.

Conclusion

There is a great deal more to randomness than can possibly be covered in a chapter or even two. Even so, we hope that we have included enough to enable you to decide whether simulation is an area of computer application that could be useful to you. If so then look out for "Computer Simulation Models" by Emshoff and Sisson published by Macmillan (1970) or any of the many other books on computer simulation.

MACHINE CODE PROGRAMMING

As you have probably found out, the ZX81 with its 16K RAM pack is capable of running some very large programs but it doesn't really have the speed necessary to tackle some quite simple tasks such as moving more than one thing about the screen at a time or even to test its own memory. Most of the speed problems can be tolerated by simply waiting a little longer for the answer. After all, a ZX81 costs only a fraction of a large mainframe computer and if it can carry out some of the same jobs you can forgive it for taking a little longer! However sometimes the slowness of the machine, even when in FAST mode, is more than can be tolerated. If it takes minutes to process the answer to a question before the next question is asked the user is likely to have gone away for a cup of coffee before the program is ready! Another example is that, although it is possible to program a space invaders type game in BASIC easily, the result would be the slowest game ever written — and slow motion invaders is no fun. Whatever you are using your ZX81 for, at some point you will run up against the speed barrier.

The surprising thing is that the answer to the problem isn't to go out and buy another more powerful and more expensive micro. The reason for this is that all micro computers, perhaps all computers whatever their size, eventually prove too slow for some task that they are given! A lot can be done to increase the speed of the ZX81 to a point where it is capable of nearly everything that a larger micro would be. The cost of this speed increase is, however, quite high in that you have to abandon BASIC and learn Z80 MACHINE CODE. Despite any arguments to convince you otherwise, machine code *is* more difficult than BASIC (why else would anyone use BASIC!) and it takes a certain amount of dedication to come to terms with it. Now this may sound like advice to avoid machine code like the plague but in fact the rest of this chapter should encourage

you to learn machine code for yourself. The point is that machine code isn't something that you can pick up in an afternoon but it is a rewarding thing to learn.

Why is BASIC so slow?

BASIC is one of the many so-called "high level" languages that you can use to program a computer. Any given computer can often be programmed in a range of such languages e.g. BASIC, FORTRAN, ALGOL etc. Even the ZX81 in theory could be used with other high level languages, it's just that its BASIC is so convenient to use that it's unusual to find anyone wanting to use anything else. The way that one computer can run so many different languages is that each one is translated to a more fundamental language before the program is run. This more fundamental language is usually called machine code and it is the only language that a computer can obey directly. Each different machine has its own machine code language and each machine translates the high level languages available for it into its own personal code. This means that you cannot learn machine code in general but only a specific computer's machine code. The ZX81 has a Z80 microprocessor inside it so the machine code that it uses is Z80 machine code. This is a very good choice for a first machine code to learn because the Z80 is a very popular microprocessor and is used inside many other machines but you should be aware that many well known computers such as APPLE and PET do not use a Z80.

So your ZX81 has to convert your BASIC statements to machine code before they can be carried out. In fact the process is rather more subtle than a direct translation to machine code. There are machines that translate a whole BASIC program to machine code before carrying it out by the use of a program called a "compiler". However these machines are in general more difficult to use than the ZX81 which uses a different technique. What happens inside the ZX81 when you run a program is that each line of BASIC is examined at the moment that it is to be carried out. The keyword is then used to "look up" what is to be done in a list of actions. For

example if the line of BASIC was GOTO 10 then the ZX81 determines that the keyword is GOTO and uses this to look up what to do in a table. The entry in the table for GOTO would contain the machine code equivalent of:

"work out the expression following the GOTO, find the line of BASIC with the same line number and make this the next instruction to be obeyed".

This method of running a BASIC program is known as "interpreting" and the program that carries it out is called an "interpreter". Thus the ZX81 interprets every line of BASIC that you write and this is why BASIC is so slow. Before the action that your BASIC command specifies can happen, the ZX81 has to spend a lot of time working out what your line of BASIC actually means! By contrast, a machine code program is executed immediately without any interpreter and can therefore often run over ten times faster.

The characteristics of machine code

If machine code is so much faster than BASIC why don't we use it more often? The answer to this question has already been briefly mentioned in the introduction to this chapter – it is more difficult to program in machine code than in BASIC. The reason why machine code is more difficult than BASIC is that it is a much simpler language! In BASIC you might write something like `LET A=B+C*2-Z` and rightly expect the answer to be stored in A, but in machine code the only arithmetic operations that you can use are addition and subtraction and these can only be carried out one at a time and on single memory locations! To do "difficult" things like multiplication you have to write subroutines that will split them down into simpler operations. For example, to multiply two numbers together you have to resort to repeated addition.

It is not within the scope of this book to teach you machine code but the rest of this chapter will attempt to give you the "flavour" of machine code programming and an introduction to some of the fundamental ideas involved. The best way to achieve this is via a couple of simple examples. First it

is necessary to examine some of the details of the Z80 microprocessor used in the ZX81.

The Z80

The Z80 microprocessor is one of the fastest and most powerful available today but it is still only capable of very simple operations. It can access any of the memory locations that we have discussed in earlier chapters but it cannot alter them directly. Any operations have to be carried out in special memory locations inside the microprocessor itself. These special memory locations are called "registers" and because there are so few of them they are referred to by names rather than addresses. The Z80 has a number of registers but for the sake of simplicity we will examine and use only four:

- the A register (also known as the accumulator)
- the B register
- the H register and
- the L register

The A register is the most useful register that the Z80 has and is used for most arithmetic operations. It is exactly like an ordinary memory location in that it can only hold numbers between 0 and 255. The B register is a general purpose register that can be used to hold temporary results without having to use external memory. There are fewer operations that can be applied to data stored in the B register and it too is limited to numbers in the range 0 to 255. The H and the L register are also general purpose registers like B but they are most often used in combination — the HL pair. By using two registers together in this way the range of numbers that can be held is considerably increased. In fact the range is large enough to hold the address of any memory location in the ZX81; and this is the main use of the HL pair — as an address register. Many Z80 operations employ the number stored in the HL pair as the address of the memory location that will be used.

All this talk of registers and what they are used for is easier to understand after one or two examples of Z80 instructions.

The instructions LD and ADD

Machine code is recognised by the Z80 in terms of numbers. That is, a machine code program is nothing more than a long list of numbers stored in the computer's memory. (After all what else but a number can you store in a computer's memory!) The trouble is that humans are very poor at reading through and understanding long lists of numbers. So to make machine code easier to use we use convenient symbols instead of the numbers that the computer recognises to represent operations. For example if you want to load the A register from memory location 123 you would use a version of the LD instruction:

LD A,(123)

which you can read as "Load the A register from memory location 123". (It is a Z80 machine code convention that anything in brackets is taken to be an address.) However, a Z80 cannot read this type of code so the instruction has to be coded into numbers. If you look up LD A in a list of Z80 instructions (or by searching appendix A of the ZX81 manual), you will find that its code is 58. This is referred to as the "operation code" or "op code" for LD A from a memory location. The complete coded instruction also requires the address of the memory location to be loaded into A to be written after the op code. So the complete coded instruction is 58,123,0. If you're wondering why there is an additional zero tagged on to the end of the instruction the reason is that two memory locations are used to store the address. Thus a single, very simple Z80 instruction takes three memory locations to store. Other Z80 instructions may take less memory, others take more.

As another example of a Z80 machine code instruction consider the

ADD A,n

instruction where n is a number in the range 0 to 255. This simply adds n to the contents of the A register. That is, if A was already loaded with 3 then after ADD A,5 it would

contain 8. This instruction has an op code of 198 and the number to be added [n] is stored in the next memory location. Notice that as n is restricted to the range 0 to 255 only one memory location is required. Thus the complete coding for ADD A,5 is 198,5 and this only uses two memory locations.

The usual way of writing a machine code program is to use the symbols such as LD and ADD A, to write the entire program and then go through and convert it to a list of numbers. The conversion of the symbols to numbers is such a routine job that you can get a program to do it for you. Such a program is known as an "assembler" and it certainly makes machine code programming easier. Unfortunately the ZX81 doesn't have an assembler as a standard extra although it is possible to write or even buy one.

A short example program

There is no suggestion that after the last two sections you'll know enough about machine code to be able to write a program but you should be able to understand the outline of one. There is a machine code subroutine in the BASIC ROM that starts at memory location 2056 that will print the character whose code is stored in the A register at the current position on the screen. We can use this knowledge to write a very simple program that will fill the screen with any character of our choice.

```
START LD A,21
      CALL 2056
      JP START
```

The first instruction in this program loads the A register with 21, the character code for "+". The second instruction is like a machine code equivalent of GOSUB in that it transfers control to a machine code subroutine starting at 2056. As we already know, the subroutine at 2056 prints the character whose code is stored in the A register. You should be able to see that the result of this CALL is a "+" printed on the screen. The final instruction is a machine code equivalent of GOTO in that it

transfers control to the "START" of the program. (JP is short for JumpP so you can read the last instruction as "JUMP to START".) The workings of this program are not difficult to understand — it will simply print "+" on the screen until it is full and an error code is reported. Normally we have to worry about stopping machine code subroutines after they have finished what they are doing but in this case, for simplicity, we can allow the program to stop itself by causing an error.

We now have a fully specified machine code program. The only things that remain to be done are coding and testing. Coding, if you recall, is simply changing the symbols that humans use to write machine code into the number codes that computers understand. The first two lines of the program can be coded easily.

| <i>instruction</i> | <i>code</i> |
|--------------------|-------------|
| START LD A,21 | 62, 21 |
| CALL 2056 | 205, 8, 8 |

The code for LD A with a number is 62 and the number to be loaded follows the op code i.e. 21. The op code for CALL is 205 and the address of the subroutine follows in the next two memory locations. (i.e. $2056=8+8*256$.) The last instruction presents a problem however, The op code for JP is 195 and the address that it transfers control to is stored in the next two memory locations — the trouble is we don't know the address of the start of the program!

To know the address of the start of the program we have to decide where it is going to be stored in memory. There are a number of places that machine code can be stored in the ZX81 and each has advantages and disadvantages. We have already seen one method in Chapter Eight where machine code was stored in reserved memory above RAMTOP. However in this chapter we will use a slightly more useful method of storing machine code in a REM statement at the beginning of the program. If we want to store a list of numbers in a REM statement all we have to do is type the character corresponding to each number. For example, the first five numbers of our machine code program are 62,21,205,8,8 and these are the character codes of the following characters Y,+,LN,[A],[A].

So we can write a REM statement that holds this machine code as:

```
10 REM Y+LN[A][A]
```

(note that the "LN" is a single character corresponding to the LN function and is entered by pressing the function key and then Z.) We can now code the last line of the program because we can work out the address of the beginning of the program. All BASIC programs start at 16509. There are two bytes for the line number, two bytes for the length of the line plus one byte for the keyword REM, so the code of the first character in the REM is stored at 16514. The last line of the program can now be written as JP 16514 and coded as

```
JP 16514
195,130,64
```

The characters corresponding to 130 and 64 are [W] and RND, however, there is no ZX81 character with a code of 195 so you cannot enter it directly in a REM statement. The solution to this problem is to leave a space in the REM statement and POKE the code into it before the machine code is used. The final REM statement can now be written into a program but we still have to find out how to use it. In Chapter Eight on using the printer the USR function was introduced as a way of calling a machine code subroutine. To recap,

USR address

will transfer control to a machine code subroutine starting at "address". As USR is a function and not a command it has to be used in an assignment statement such as

```
LET A=USR address
```

but for the examples in this chapter we are not worried about the value that is stored in A as a result of the USR function. As we have already worked out the address of the start of the machine code stored in the REM as 16514, the full USR function is

```
LET A=USR 16514
```

Before giving the full BASIC program that uses the machine code it is worth listing exactly what is stored where:

| address | code | character |
|---------|------|-----------|
| 16514 | 62 | Y |
| 16515 | 21 | + |
| 16516 | 205 | LN |
| 16517 | 8 | [A] |
| 16518 | 8 | [A] |
| 16519 | 195 | none |
| 16520 | 130 | [W] |
| 16521 | 64 | RND |

From this table you should be able to see that the op code that doesn't correspond to any character has to be POKEd into address 16519. The complete program is

```
10 REM Y+LN[A][A] [W]RND
20 POKE 16519,195
30 LET A=USR 16514
```

Note that the REM statement has to be entered exactly as written, i.e. the characters such as LN and RND must be entered with one keystroke and the only space in the REM is between [A] and [W].

When you run this program you should see the screen fill very rapidly with plus signs. To compare the speed of this machine code with BASIC add the following lines and then enter GOTO 40:

```
40 PRINT "+";
50 GOTO 40
```

If you would like to check that the machine code is entered correctly you can also add the following which will print the contents of each memory location in the REM:

```
60 FOR I=16514 TO 16521
70 PRINT I,PEEK I
80 NEXT I
```

To use this program type GOTO 60.

Although this example has been very simple, consisting of

only three machine operations, it has taken a long time to produce and a long time to explain! However, you should be able to judge the advantages of machine code from the speed difference between this example and the BASIC equivalent.

A second example — reversing the screen

In this example we will write a machine code subroutine that will change all of the characters displayed on the screen to their inverse form (i.e. white will be changed to black and black to white). Although this is a useful subroutine, it could be used to “flash” the screen during a game for example, its primary purpose is to show how a slightly larger machine code program is written. However the program is so much longer that there just isn't the space to go into as much detail as with the first example. All that can be done is to explain the method used and how each instruction works. The details of coding are given but not explained.

The method used is to add 128 to the character code stored at every screen location. If you look at appendix A in the ZX81 manual you will see that characters and their inverses differ by 128. So adding 128 changes a character to its inverse form. What is less obvious is that adding 128 for a second time will restore the original character code. The reason for this is that if we use the A register to hold the character code and ADD A,128 to add 128 to it then the largest number that the A register can hold is 255. If you add one to 255 stored in the A register is “resets” or “rolls over” to zero and then carries on as normal i.e. no error is generated. This means that if the result of adding 128 is larger than 255 the value stored in A will be the correct answer — 256. For example, if you add 128 to 38 (the code for A) the answer is 166 (the code for inverse A). If you then add 128 to 166 the answer should be 294 but this is larger than 255 so the result stored in the A register is 294–256, namely 38, the number we first started with. In short, adding 128 will change any character to its inverse form and adding it a second time will restore it.

The rest of the method is straightforward. We have to load the address of the start of the screen area and then use this to

load each screen memory location into A, add 128 and store it back into the same location. The only things that we have to be careful about are to avoid adding 128 to the “newline” characters at the end of each screen line and to make sure that we stop after reversing 21 lines of characters. The easiest way to achieve both of these aims is to check the contents of the A register and if it is a “newline” avoid adding 128 to it but alter a record of the number of lines that have been reversed so far. This count of the number of lines is best kept in the B register. If the B register is loaded with 21 at the start of the program and has one subtracted from it each time a “newline” is encountered then after 21 lines it will be zero. So testing for a zero in the B register is all that we have to do to decide when the program is finished! The only other detail is that all machine code programs should return to BASIC by using a RET instruction.

The complete program including details of its coding is:

| <i>address</i> | <i>instruction</i> | <i>code</i> | <i>comments</i> |
|----------------|--------------------|-------------|--|
| 16514 | LD HL,(16396) | 42,12,64 | Load HL with the address of the start of the display file. |
| 16517 | LD B,21 | 06,21 | Load 21 into B to count the number of lines. |
| 16519 | INC HL | 35 | Increment HL, i.e. add one to HL. |
| 16520 | LD A,(HL) | 126 | LOAD A from the address in HL. |
| 16521 | CP 118 | 254,118 | Compare A to 118 which is the code for “newline”. |
| 16523 | JP Z,16532 | 202,148,64 | If A equals 118 then GOTO 16532. |
| 16526 | ADD A,128 | 198,128 | Add 128 to A. |
| 16528 | LD (HL),A | 119 | Store A back in same address. |
| 16529 | JP 16519 | 195,135,64 | GOTO 16519. |
| 16532 | DEC B | 05 | Decrease B by one. |
| 16533 | JP NZ,16519 | 194,135,64 | If B is Not Zero (NZ) GOTO 16519. |
| 16536 | RET | 201 | RETURN to BASIC. |

Turning each of the codes into characters gives the following memory layout:

| <i>address</i> | <i>code</i> | <i>character</i> |
|----------------|-------------|------------------|
| 16514 | 42 | E |
| 16515 | 12 | £ |
| 16516 | 64 | RND |
| 16517 | 06 | [T] |
| 16518 | 21 | + |
| 16519 | 35 | 7 |
| 16520 | 126 | none |
| 16521 | 254 | RETURN |
| 16522 | 118 | none |
| 16523 | 202 | ASN |
| 16524 | 148 | inverse = |
| 16525 | 64 | RND |
| 16526 | 198 | LEN |
| 16527 | 128 | [] |
| 16528 | 119 | none |
| 16529 | 195 | none |
| 16530 | 135 | [3] |
| 16531 | 64 | RND |
| 16532 | 05 | [5] |
| 16533 | 194 | TAB |
| 16534 | 135 | [3] |
| 16535 | 64 | RND |
| 16536 | 201 | TAN |

This machine code can be incorporated into a BASIC program in a REM statement using four POKEs for the codes that do not correspond to characters. To see how the routine works try the following program. Remember Chapter 8, page 96 explains how to enter keywords such as RETURN.

```

10 REM E£RND[T]+7 RETURN ASN [=] RNDLEN
   [ ] [3]RND[5]TAB[3]RNDTAN
20 POKE 16520,126
30 POKE 16522,118
40 POKE 16528,119
50 POKE 16529,195

```

```

60 PRINT "*****"
70 LET A=USR 16514
80 GOTO 70

```

```

500 FOR I=16514 TO 16536
510 PRINT I,PEEK I
520 NEXT I

```

Once again you can check to see what codes are stored where by using lines 500 to 520 by GOTO 500.

Although this is a long example you should be able to understand most of it if you study it carefully.

Next steps

If you have managed to understand some of this chapter and have entered the examples and seen how much faster they are than BASIC then machine code will be the next area of computing that you will want to study. There are a number of books specifically about machine code for the ZX81 and others which deal with Z80 machine code more generally are also relevant.

Whether or not you decide to pursue machine code programming, we hope that this book will enable you to go further with programming your ZX81. With the 16K RAM pack it is capable of quite a remarkable range of applications and is also great fun to use.

ALSO RECOMMENDED

BP109: THE ART OF PROGRAMMING THE 1K ZX81

M. James & S. M. Gee

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 great 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 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.

96 pages

0 85934 084 8

1982

£1.95

BP119: THE ART OF PROGRAMMING THE ZX SPECTRUM

M. James

This book will help you to enjoy all the features of the Sinclair Spectrum. It explains techniques that you can use to write your own programs to use its colour, high-res graphics and sound facilities to the full. The emphasis is on gaining practical experience through programs that are fun to try out. Lots of games programs are included.

Chapter One — Getting to Know Your Spectrum: The Spectrum's special features; Prospects for programming.

Chapter Two — Low-res Colour Graphics: PRINT command, TAB and AT; Graphics characters; User-defined graphics characters; PAPER and INK commands; Using colour; FLASH, BRIGHT, OVER, INVERSE.

Chapter Three — Fun at Random: Pseudo randomness; RND and RANDOMISE; Random events; Random integers; Unequal probabilities; Random graphics.

Chapter Four — High-res Graphics: PLOT and DRAW; CIRCLE.

Chapter Five — Sound: BEEP command; Making music; Sound effects; OUT; Amplifying the sound.

Chapter Six — Moving Graphics: From flashes to moving; Moving balls and velocity; Free flight and gravity.

Chapter Seven — PEEK and POKE: When to use PEEK and POKE; Large screen displays.

Chapter Eight — A Sense of Time: Delay loops; The timer; Clock programs; Reaction time games.

Chapter Nine — Strings: String slicing; Word games; Codes and cyphers.

Chapter Ten — Advanced Graphics: Scrolling and rolling graphics; SCREEN\$ graphics; ATTRibute and POINT.

144 pages

0 85934 094 5

1983

£2.95

CASSETTE TAPES

Ramsoft
P.O. Box 6
Richmond
North Yorkshire
DL10 4HL
England

If you are tired of typing in programs, there is an alternative! The above company can supply, at very reasonable cost, a selection of complete programs taken from this book, on cassette tapes.

TAPE 1 (16K) contains:

Utility Suite and Menu Driver, Machine Code Demo, Depth Charge, Ski Run, Squash and two improved versions of programs that appeared in the 1K book, Pontoon and Random Symmetry.

TAPE 2 (16K) contains:

Statistics, Interest Calculator, Arithmetic Test, Customer Servicing (Poisson distribution) and another two improved programs that appeared in the 1K book, Chess Clock and Hangman.

Also available is

TAPE 3 (1K) which contains twenty programs from *The Art of Programming the 1K ZX81*.

Write directly to the address given above for further details and an order form.

PLEASE NOTE: The Publishers of this book are in no way responsible for the manufacture or supply of these tapes and all enquiries must be sent directly to Ramsoft.

The contents of the tapes may be subject to change without further notice.

Please note overleaf is a list of other titles that are available in our range of Radio, Electronics and Computer Books.

These should be available from all good Booksellers, Radio Component Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the publisher enclosing payment to cover the cost of the book plus adequate postage.

If you would like a complete catalogue of our entire range of Radio, Electronics and Computer Books then please send a Stamped Addressed Envelope to:

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

BERNARD BABANI BP114

The Art of Programming the 16K ZX81

■ This book is the sequel to BP109, *The Art of Programming the 1K ZX81*, and it sets out to help you use your 16K RAM pack and ZX printer to the full. It concentrates on good programming style and introduces some interesting programs that are both fun and useful.

■ Chapter One introduces the 16K RAM pack and the printer. Chapter Two explains how the extra storage space is used and presents a memory test program to check that your new 16K is operational. Chapter Three covers some utilities that you will find useful in writing longer programs. Chapter Four is an interlude from serious applications, presenting four games programs that make the most of the extended graphics capabilities now available to you. Chapters Five to Eight deal with writing and debugging large programs, storing them on cassettes and printing out both programs themselves and their results. These chapters also introduce programs for editing data bases and statistical analysis for financial management and covers text and graphics printing. Chapter Nine takes a look at randomness. Chapter Ten introduces machine code and explains why you might like to use it.

■ With this book to guide you, it is hoped that you will be able to discover just how versatile and powerful your ZX81 is and that you will realise just how rewarding programming with it can be.

ISBN 0 85934 089 9



BERNARD BABANI (publishing) LTD
The Grampians
Shepherds Bush Road
London W6 7NF
England

£2.50